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

Trabalho Prático nº 2

Relatório de Desenvolvimento

Grupo nr. 15

Ruben Silva (a94633)

Carlos Costa (a94543)

15 de janeiro de 2023

Resumo

Neste relatório encontra-se a resolução do Trabalho Prático 2, mais precisamente, o trabalho sobre GIC/GT + Compilador para a unidade curricular Processamento de Linguagens e Compiladores.
O objetivo deste trabalho é a utilização das livrarias do PLY(lex e yacc) e gerar código para a máquina virtual de stack (Virtual Machine).

Conteúdo

Capítulo 1

GIC/GT + Compilador

1.1 Análise do Problema

Neste problema recebemos um ficheiro um pdf com instruções de pseudo-código para rodar numa Máquina Virtual. Será necessário recorrer ao PLY e ao YACC para resolver. O objetivo deste problema é:

- declarar variáveis atómicas do tipo inteiro, com os quais se podem realizar as habituais operações aritméticas, relacionais e lógicas;
- efetuar instruções algorítmicas básicas como a atribuição do valor de expressões numéricas a variáveis;
- ler do standard input e escrever no standard output;
- efetuar instruções de seleção para controlo do fluxo de execução;
- efetuar instruções de repetição (cíclicas) para controlo do fluxo de execução, permitindo o seu aninhamento.

Também foi dado a possibilidade de uma de duas funcionalidades para adicionarmos e escolhemos a seguinte:

- 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).

1.2 Resolução do Problema

Para a resolução deste problema recorremos às seguintes soluções sequenciadas:

i) Criar o lexer;

ii) Criar o parser;

iii) Criar vários ficheiros teste.

1.2.1 Lexer

Esta parte do problema foi resolvido com 3 partes, identificar os literais, os tokens e as funções que encontram os tokens.

```
literals = ['%', '*', '+', '/', '-', '=',
            '(', ')', '.', '<', '>', ',', '{', '}', '[', ']']
```

Figura 1.1: Declaração dos literais

```
tokens = ['INT', 'ID', 'STR', 'NUM',
          'MAIN', 'PRINT', 'READ',
          'IF', 'ELSE',
          'EQUALS', 'GREATERQ', 'LESSERQ',
          'WHILE', 'DO',
          'AND', 'OR', 'NOT']
```

Figura 1.2: Declaração dos tokens

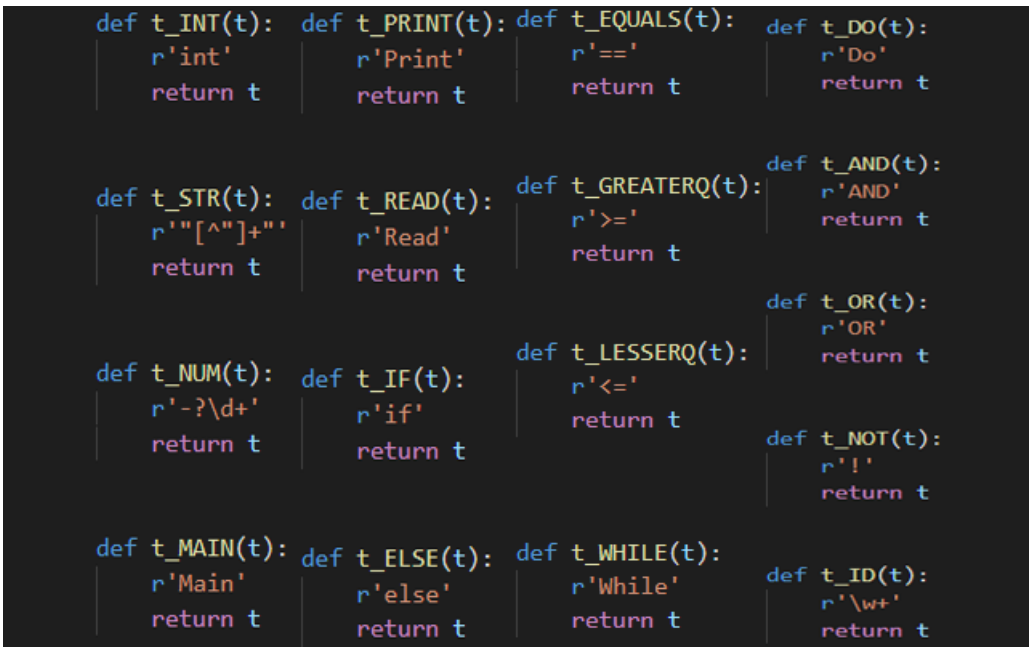

Figura 1.3: Declaração das funções dos tokens

- t_INT - int;
- t_STR - ”string123”, ”ola”, ”@@@@@”;
- t_NUM - -5,2,0,77;
- t_MAIN - Main;
- t_PRINT - Print;
- t_READ - Read;
- t_IF - if;
- t_ELSE - else;
- t_EQUALS - ==;
- t_GREATERQ - ¿=;
- t_LESSERQ - ¡=;
- t_WHILE - While;
- t_DO - Do;
- t_AND - AND;
- t_OR - OR;
- t_NOT - !;
- t_ID - counter, x, var1.

Também é adicionado os tokens responsáveis por ignorar e detetar carateres inválidos, respetivamente:

```
t_ignore = " \t\n"


def t_error(t):
    print('Illegal character: ', t.value[0])
    t.lexer.skip(1)
```

Figura 1.4: ignore e error

1.2.2 Parser

Esta parte do problema foi resolvido com 11 partes:

- Principal;
- Declarations;
- Commands;
- Conditions;
- Expression;
- Term;
- Factors;
- ID/Error;
- Simple Arrays;
- Double Arrays;
- Parser Functions.

Principal

```
def p_program(p):
    "program : '{' MAIN body '}' "
    p[0] = p[3]
# criação do programa {main ... }

def p_body(p):
    "body : declarations commands"
    p[0] = p[1] + 'START\n' + p[2] + 'STOP'
# corpo do programa contém declarações e comandos a realizar
```

Figura 1.5: Principal

Existe 2 funções, 1 responsável pela descrição geral do programa, e a outra responsável pela declaração do corpo da função (respetivamente na imagem).

Declarations

```
def p_empty_declarations(p):
    "declarations : "
    p[0] = ""
# sem declarações

def p_declarations(p):
    "declarations : declaration declarations"
    p[0] = p[1] + p[2]
# especificação de declarações.... uma ou mais declarações
```

Figura 1.6: Excerto 1 das declarations

Nestas 2 funções é permitido ao utilizador fazer múltiplas declarações.

```
def p_int_declaration(p):
    "declaration : INT ID"
    if p[2] in parser.variables:
        parser.success = False
        print("Multiple variable declaration " + p[2])
        sys.exit(0)
    else:
        parser.variables[p[2]] = parser.count
        p[0] = 'PUSHI 0\n'
        parser.count += 1
# declaração de um inteiro sem valor || int ID

def p_int_num_declaration(p):
    "declaration : INT ID '=' NUM"
    if p[2] in parser.variables:
        parser.success = False
        print("Multiple variable declaration " + p[2])
        sys.exit(0)
    else:
        parser.variables[p[2]] = parser.count
        p[0] = "PUSHI "+str(p[4])+"\n"
        parser.count += 1
# declaração de int id com um certo valor || ex: int num1 = 4
```

Figura 1.7: Excerto 2 das declarations

Nestas 2 funções é permitido ao utilizador declarar variáveis com ou sem inicialização.
Não é permitido existir múltiplas declarações da mesma variável.

Commands

```
def p_empty_commands(p):
    "commands : "
    p[0] = ""
# sem comandos

def p_commands(p):
    "commands : command commands"
    p[0] = p[1]+p[2]
# especificação dos comandos 1 ou mais a realizar
```

Figura 1.8: Excerto 1 dos commands

Nestas 2 funções é permitido declarar múltiplos comandos.
Os comandos que serão usados são os seguintes

- Print;
- Read;
- If;
- Atribution;
- While-do.

```
def p_print_command(p):
    "command : cmd_prints"
    p[0] = p[1]
# especificação do cmd_print (será desenvolvido as várias alternativas de print)

def p_cmd_prints_all(p):
    "cmd_prints : PRINT '(' cmd_print prints ')' "
    p[0] = str(p[3])+str(p[4])
# cmd_print função geral

def p_cmd_prints(p):
    "prints : '+' cmd_print prints "
    p[0] = p[2] + p[3]
# desenvolvimento de cmd_print

def p_empty_cmd_prints(p):
    "prints :"
    p[0] = ""
# cmd_print vazio

def p_print_id_command(p):
    "cmd_print : Id"
    p[0] = p[1][1]+"WRITEI\n"
# comando para dar print a um Id

def p_print_str_command(p):
    "cmd_print : STR"
    p[0] = "PUSHS " + p[1] + "\n" + "WRITES\n"
# comando para dar print a uma STR
```

Figura 1.9: Excerto 2 dos commands (Print)

Neste conjunto de funções é criada o comando "Print", que permite-nos dar print, ou a variáveis ou a strings.

```
def p_read_command(p):
    "command : cmd_read"
    p[0] = p[1]


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

Figura 1.10: Excerto 3 dos commands (Read)

Neste conjunto de funções é criada o comando "Read", que permite-nos ler o input do utilizador.

```
def p_if_command(p):
    "command : cmd_if"
    p[0] = p[1]

def p_cmd_if(p):
    "cmd_if : IF condition '{' commands '}'"
    p[0] = str(p[2])+"JZ IF" + str(parser.label) + "\n" + \
        str(p[4])+"IF"+str(parser.label)+":\n"

    parser.label += 1

def p_if_else_command(p):
    "command : cmd_if_else"
    p[0] = p[1]

def p_cmd_if_else(p):
    "cmd_if_else : IF condition '{' commands '}' ELSE '{' commands '}'"
    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
```

Figura 1.11: Excerto 4 dos commands (If)

Neste conjunto de funções é criada os comandos: "If"e "If Else".

```
def p_atribution_command(p):
    "command : attribution"
    p[0] = p[1]


def p_cmd_atb(p):
    "attribution : Id '=' exp"
    p[0] = p[3] + p[1][0]
# igualdade de um Id para uma expressão || resultado = num1 + num2 || resultado = num1 - 2 etc..
```

Figura 1.12: Excerto 5 dos commands (Atribution)

Neste conjunto de funções é criada o comando responsável por atribuir expressões às variáveis.

```
def p_do_while_command(p):
    "command : cmd_while"
    p[0] = p[1]


def p_cmd_while(p):
    "cmd_while : WHILE condition DO '{' commands '}'"
    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
```

Figura 1.13: Excerto 6 dos commands (While-do)

Neste conjunto de funções é criada o comando: While-do

Conditions

```
def p_condition(p):
    "condition : '(' context ')'"
    p[0] = p[2]

def p_condition_Neg(p):
    "condition : NOT '(' context ')'"
    p[0] = str(p[3])+"NOT\n"

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

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

def p_condition_EQUALS(p):
    "context : exp EQUALS exp"
    p[0] = str(p[1])+str(p[3])+"EQUAL\n"

def p_condition_GREATER(p):
    "context : exp '>' exp"
    p[0] = str(p[1])+str(p[3])+"SUP\n"

def p_condition_LESSER(p):
    "context : exp '<' exp"
    p[0] = str(p[1])+str(p[3])+"INF\n"

def p_condition_GREATERQ(p):
    "context : exp GREATERQ exp"
    p[0] = str(p[1])+str(p[3])+"SUPEQ\n"

def p_condition_LESSERQ(p):
    "context : exp LESSERQ exp"
    p[0] = str(p[1])+str(p[3])+"INFEQ\n"

def p_condition_exp(p):
    "context : exp"
    p[0] = str(p[1])

def p_context(p):
    "context : condition"
    p[0] = p[1]
```

Figura 1.14: Conditions

Este conjunto de funções possuem as funções para gerar o essencial para as relações lógicas. Existe:

- Negação - 'Not';
- And - 'AND';
- Or - 'OR';
- Igualdade - 'EQUALS';
- Maior - ≫'
- Menor - ≪'
- Maior ou igual - 'GREATERQ';
- Menor ou igual - 'LESSERQ'.

Expressions

```
def p_plus_expression(p):
    "exp : exp '+' term"
    p[0] = p[1] + p[3] + "ADD\n"

def p_minus_expression(p):
    "exp : exp '-' term"
    p[0] = p[1] + p[3] + "SUB\n"

def p_expression(p):
    "exp : term"
    p[0] = p[1]
```

Figura 1.15: Expressions

Estas funções permite a existência de aritmética simples.

Terms

```
def p_division_term(p):
    "term : term '/' factor"
    p[0] = p[1] + p[3] + "DIV\n"

def p_multiplication_term(p):
    "term : term '*' factor"
    p[0] = p[1] + p[3] + "MUL\n"

def p_mod_term(p):
    "term : term '%' factor"
    p[0] = p[1] + p[3] + "MOD\n"

def p_term(p):
    "term : factor"
    p[0] = p[1]
```

Figura 1.16: Terms

Estas funções expandem a definição do "term"e permite a divisão, a multiplicação e o "MOD".

Factors

```
def p_num_factor(p):
    "factor : NUM"
    p[0] = "PUSHI "+p[1]+"\n"

def p_id_factor(p):
    "factor : Id"
    p[0] = p[1][1]
```

Figura 1.17: Factors

Estas funções permite tornar, tanto um número, como um id, num fator que tem como objetivo ser usado nas contas

ID/Error

```
def p_Id(p):
    "Id : ID"
    if p[1] not in parser.variables:
        parser.success = False
        print("Variable not declared: " + p[1])
        sys.exit(0)
    else:
        p[0] = ("STOREG " + str(parser.variables[p[1]])+"\n", "PUSHG " +
                str(parser.variables[p[1]])+"\n", p[1])
# definição Id = ID

def p_error(p):
    print("Syntax error!")
    parser.success = False
    sys.exit(0)
# definição de erro de sintaxe
```

Figura 1.18: ID/Error

A função id é responsável por 3 coisas, detetar se todas as variáveis foram declaradas (pois não faz sentido existir variáveis que não foram declaradas), dá store na stack e nas variáveis do sistema (que é as variáveis do parser).

Simple Arrays

```
def p_def_array_values(p):
    "values : NUM ',' values"
    p[0] = "PUSHI "+p[1]+"\n"+p[3]
    parser.arraycount += 1


def p_def_array_num(p):
    "values : NUM"
    p[0] = "PUSHI "+p[1]+"\n"
    parser.arraycount += 1


def p_def_empty_array(p):
    "values :"
    p[0] = ""
```

Figura 1.19: Excerto 1 dos Arrays Simples

Estas 3 funções são responsáveis por definir o termo "values"que é importante para os arrays. Pois é com este termo que podemos dar valores a cada index do array.

```
def p_array_declaration(p):
    "declaration : INT ID '[' NUM ']'"
    if p[2] in parser.variables:
        parser.success = False
        print("Multiple variable declaration " + p[2])
        sys.exit(0)
    else:

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

def p_array_num_declaration(p):
    "declaration : INT ID '[' NUM ']' '=' values"
    if p[2] in parser.variables:
        parser.success = False
        print("Multiple variable declaration " + p[2])
        sys.exit(0)

    elif (parser.arraycount) != int(p[4]):
        parser.success = False
        print("Index out of range -> variable: " + p[2])
        sys.exit(0)
    else:
        parser.variables[p[2]] = parser.count
        p[0] = p[7]
        parser.count += int(p[4])
        parser.arraycount = 0
```

Figura 1.20: Excerto 2 dos Arrays Simples

Estas 2 funções são responsáveis por declarar os arrays, com ou sem inicialização. Caso que seja com inicialização, é necessário garantir que não que seja escrito for do array.

```
def p_print_command_id_Array(p):
    "cmd_print : ID_Array"
    p[0] = p[1][0]+p[1][1] + "LOADN\n"+"WRITEI\n"

def p_read_id_array_command(p):
    "cmd_read : READ ID_Array"
    p[0] = p[1][0]+p[1][1] + "READ\nATOI\nSTOREN\n"
```

Figura 1.21: Excerto 3 dos Arrays Simples

Funções que permitem que a função "Read"e "Print"sejam utilizadas nos arrays simples também.

```
def p_id_array_factor(p):
    "factor : ID_Array"
    p[0] = (p[1][0]+p[1][1]+"LOADN\n", p[1][2])

def p_array_exp_command(p):
    "command : ID_Array '=' exp"
    p[0] = p[1][0]+p[1][1]+p[3]+"STOREN\n"

def p_id_array(p):
    "ID_Array : ID '[' factor ']'"
    if (p[1] not in parser.variables):
        parser.success = False
        print("Multiple variable declaration " + p[1])
        sys.exit(0)
    else:
        p[0] = ("PUSHGP\nPUSHI " + str(parser.variables[p[1]])+"\nPADD\n", p[3])
```

Figura 1.22: Excerto 4 dos Arrays Simples

A primeira função é responsável por permitir a utilização do array como um factor, a segunda é responsável por permitir que o array receba uma expressão. E for fim, é a função para atribuir o id ao array.

Double Arrays

```
def p_double_values(p):
    "d_values : '[' values ']' d_values"
    p[0] = p[2] + p[4]
    parser.darraycount += 1

def p_empty_double_values(p):
    "d_values :"
    p[0] = ""

def d_values_values(p):
    "d_values : '[' values ']'"
    p[0] = p[2]
    parser.darraycount += 1
```

Figura 1.23: Excerto 1 dos Arrays Duplos

Estas 3 funções são responsáveis por definir o termo "d_values"que é importante para os arrays duplos. Pois é com este termo que podemos dar valores a cada index de cada array.

```
def p_double_array_declaration(p):
    "declaration : INT ID '[' NUM ']' '[' NUM ']'"
    if p[2] in parser.variables:
        parser.success = False
        print("Multiple variable declaration " + p[2])
        sys.exit(0)
    else:
        parser.variables[p[2]] = parser.count
        p[0] = "PUSHN " + str(int(p[4])*int(p[7])) + "\n"
        parser.count += (int(p[4]) * int(p[7]))

def p_double_array_num_declaration(p):
    "declaration : INT ID '[' NUM ']' '[' NUM ']' '=' d_values "
    if p[2] in parser.variables:
        parser.success = False
        print("Multiple variable declaration " + p[2])
        sys.exit(0)
    elif (parser.arraycount != int(p[4])*int(p[7])) or (parser.darraycount != int(p[4])):
        parser.success = False
        print("Index out of range -> variable: " + p[2])
        sys.exit(0)
    else:
        parser.variables[p[2]] = parser.count
        p[0] = p[10]
        parser.count += (int(p[4]) * int(p[7]))
        parser.size[p[2]] = int(p[7])
        parser.arraycount = 0
        parser.darraycount = 0
```

Figura 1.24: Excerto 2 dos Arrays Duplos

Estas 2 funções são responsáveis por declarar os arrays duplos, com ou sem inicialização. Caso que seja com inicialização, é necessário garantir que não que seja escrito for dos arrays.

```
def p_print_command_double_array(p):
    "cmd_print : ID_Double_Array"
    p[0] = p[1][0]+p[1][1] + "LOADN\n"+"WRITEI\n"

def p_read_command_double_array(p):
    "cmd_read : ID_Double_Array"
    p[0] = p[1][0]+p[1][1] + "READ\nATOI\nSTOREN\n"
```

Figura 1.25: Excerto 3 dos Arrays Duplos

Funções que permitem que a função "Read"e "Print"sejam utilizadas nos arrays duplos também.

```
def p_read_command_double_array(p):
    "cmd_read : ID_Double_Array"
    p[0] = p[1][0]+p[1][1] + "READ\nATOI\nSTOREN\n"


def p_id_double_array_factor(p):
    "factor : ID_Double_Array"
    p[0] = (p[1][0]+p[1][1]+"LOADN\n")


def p_id_double_array(p):
    "ID_Double_Array : ID '[' factor ']' '[' factor ']'"
    if (p[1] not in parser.variables):
        parser.success = False
        print("Multiple variable declaration " + p[1])
        sys.exit(0)
    else:
        p[0] = ("PUSHGP\nPUSHI " + str(parser.variables[p[1]]) +
                "\nPADD\n", p[3] + "PUSHI " + str(parser.size[p[1]]) + "\nMUL\n" + p[6] + "ADD\n")
```

Figura 1.26: Excerto 4 dos Arrays Duplos

A primeira função é para permitir que o array duplo receba uma expressão, a segunda é responsável por permitir a utilização do array duplo como um factor. E for fim, é a função para atribuir o id ao array duplo.

Parser Functions

```
parser = yacc.yacc()
parser.variables = {}
parser.success = True

parser.count = 0
parser.label = 0
parser.loop = 0
parser.arraycount = 0
parser.darraycount = 0

fIn = input('FileInput: ')
fOut = input('FileOutput: ')

with open(fIn, 'r') as file:
    code = file.read()
out = parser.parse(code)
with open(fOut, 'w') as output:
    output.write(str(out))

print(parser.variables)
```

Figura 1.27: Parser

Por fim, é utilizado o parser do yacc para compilar e criar o pseudo-código para executarmos na Máquina Virtual

1.3 Exemplos

1.3.1 Exemplo 1

```
{
    Main

    int num1
    int num2
    int num3
    int i = 0
    int resultado
    int array[4]

    Print ("Armazenar valores de todas as operacoes basicas num array de dois num\n\n")
    Print ("Digite o primeiro num: \n ")
    Read num1
    Print ("Digite o segundo num: \n")
    Read num2


    While !(num3 == 4) Do {

        if (num3 == 0){
            resultado = num1 + num2
            }
        if (num3 == 1){
            resultado = num1 - num2
            }
        if (num3 == 2){
            resultado = num1 * num2
            }
        if (num3 == 3){
            resultado = num1 / num2
            }
        array[num3] = resultado
        num3 = num3 +1
    }
    Print ("Valores do Array: \n")
    While !(i == 4) Do {
        Print(array[i])
        Print("\n")
        i = i +1
    }
}
```

1.3.2 Exemplo 2

```
{

    Main
    int valor
    int i = 0
    int j = 0
    int matriz[3][3] = [1,2,3][4,5,6][7,1,9]
    int counter = 0

    Print ("Procurar um valor na matriz e contar quantas vezes ele aparece\n")
    Print("Digite o valor a procurar: \n")
    Read valor

    While (i < 3) Do {
        While ( j < 3) Do {
        if (valor == matriz[i][j]){

            valor = 1
            counter = counter + 1
            j = j + 1
        }
        else {
            j = j + 1
        }

        }
        i = i +1
        j = 0
    }

    if (valor == 1){

        Print("Valor encontrado: ")
        Print(valor)
        Print("\n numero de vezes encontrado: ")
        Print(counter)

    } else{

        Print("Valor nao encontrado\n")
    }

}
```

1.4 Pseudo-Código Gerado

1.4.1 Pseudo-Código - Exemplo 1

```
PUSHI 0
PUSHI 0
PUSHI 0
PUSHI 0
PUSHI 0
PUSHN 4
START
PUSHS "Armazenar valores de todas as operacoes basicas num array de dois num\n\n"
WRITES
PUSHS "Digite o primeiro num: \n "
WRITES
READ
ATOI
STOREG 0
PUSHS "Digite o segundo num: \n"
WRITES
READ
ATOI
STOREG 1
WHILE0:
PUSHG 2
PUSHI 4
EQUAL
NOT
JZ ENDWHILE0
PUSHG 2
PUSHI 0
EQUAL
JZ IF0
PUSHG 0
PUSHG 1
ADD
STOREG 4
IF0:
PUSHG 2
PUSHI 1
EQUAL
JZ IF1
PUSHG 0
PUSHG 1
SUB
STOREG 4
IF1:
PUSHG 2
```

```
PUSHI 2
EQUAL
JZ IF2
PUSHG 0
PUSHG 1
MUL
STOREG 4
IF2:
PUSHG 2
PUSHI 3
EQUAL
JZ IF3
PUSHG 0
PUSHG 1
DIV
STOREG 4
IF3:
PUSHGP
PUSHI 5
PADD
PUSHG 2
PUSHG 4
STOREN
PUSHG 2
PUSHI 1
ADD
STOREG 2
JUMP WHILE0
ENDWHILE0:
PUSHS "Valores do Array: \n"
WRITES
WHILE1:
PUSHG 3
PUSHI 4
EQUAL
NOT
JZ ENDWHILE1
PUSHGP
PUSHI 5
PADD
PUSHG 3
LOADN
WRITEI
PUSHS "\n"
WRITES
PUSHG 3
PUSHI 1
ADD
```

```
STOREG 3
JUMP WHILE1
ENDWHILE1:
STOP
```

1.4.2 Pseudo-Código - Exemplo 2

```
PUSHI 0
PUSHI 0
PUSHI 0
PUSHI 1
PUSHI 2
PUSHI 3
PUSHI 4
PUSHI 5
PUSHI 6
PUSHI 7
PUSHI 1
PUSHI 9
PUSHI 0
START
PUSHS "Procurar um valor na matriz e contar quantas vezes ele aparece\n"
WRITES
PUSHS "Digite o valor a procurar: \n"
WRITES
READ
ATOI
STOREG 0
WHILE1:
PUSHG 1
PUSHI 3
INF
JZ ENDWHILE1
WHILE0:
PUSHG 2
PUSHI 3
INF
JZ ENDWHILE0
PUSHG 0
PUSHGP
PUSHI 3
PADD
PUSHG 1
PUSHI 3
MUL
PUSHG 2
ADD
LOADN
EQUAL
JZ IF0
PUSHI 1
STOREG 0
PUSHG 12
```

```
PUSHI 1
ADD
STOREG 12
PUSHG 2
PUSHI 1
ADD
STOREG 2
JUMP IFEND0
IF0:
PUSHG 2
PUSHI 1
ADD
STOREG 2
IFEND0:
JUMP WHILE0
ENDWHILE0:
PUSHG 1
PUSHI 1
ADD
STOREG 1
PUSHI 0
STOREG 2
JUMP WHILE1
ENDWHILE1:
PUSHG 0
PUSHI 1
EQUAL
JZ IF1
PUSHS "Valor encontrado: "
WRITES
PUSHG 0
WRITEI
PUSHS "\n numero de vezes encontrado: "
WRITES
PUSHG 12
WRITEI
JUMP IFEND1
IF1:
PUSHS "Valor nao encontrado\n"
WRITES
IFEND1:
STOP
```

1.5 Código-Fonte

1.5.1 tpLexer.py

```
import ply.lex as lex

literals = ['%', '*', '+', '/', '-', '=',
            '(', ')', '.', '<', '>', ',', '{', '}', '[', ']']

tokens = ['INT', 'ID', 'STR', 'NUM',
          'MAIN', 'PRINT', 'READ',
          'IF', 'ELSE',
          'EQUALS', 'GREATERQ', 'LESSERQ',
          'WHILE', 'DO',
          'AND', 'OR', 'NOT']

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

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

def t_NUM(t):
    r'-?\d+'
    return t

def t_MAIN(t):
    r'Main'
    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_EQUALS(t):
    r'=='
    return t

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

def t_LESSERQ(t):
    r'<='
    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_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()
```

1.5.2 tpParser.py

```
import ply.yacc as yacc
import sys
from tp_lexer import tokens


def p_program(p):
    "program : '{' MAIN body '}' "
    p[0] = p[3]
# criação do programa {main ... }


def p_body(p):
    "body : declarations commands"
    p[0] = p[1] + 'START\n' + p[2] + 'STOP'
# corpo do programa contém declarações e comandos a realizar

# region DECLARATIONS FUNCTIONS


def p_empty_declarations(p):
    "declarations : "
    p[0] = ""
# sem declarações


def p_declarations(p):
    "declarations : declaration declarations"
    p[0] = p[1] + p[2]
# especificação de declarações.... uma ou mais declarações


def p_int_declaration(p):
    "declaration : INT ID"
    if p[2] in parser.variables:
        parser.success = False
        print("Multiple variable declaration " + p[2])
        sys.exit(0)
    else:
        parser.variables[p[2]] = parser.count
        p[0] = 'PUSHI 0\n'
        parser.count += 1
# declaração de um inteiro sem valor || int ID


def p_int_num_declaration(p):
    "declaration : INT ID '=' NUM"
```

```
    if p[2] in parser.variables:
        parser.success = False
        print("Multiple variable declaration " + p[2])
        sys.exit(0)
    else:
        parser.variables[p[2]] = parser.count
        p[0] = "PUSHI "+str(p[4])+"\n"
        parser.count += 1
# declaração de int id com um certo valor || ex: int num1 = 4

# endregion

# region ALL COMMANDS FUNCTIONS


def p_empty_commands(p):
    "commands : "
    p[0] = ""
# sem comandos


def p_commands(p):
    "commands : command commands"
    p[0] = p[1]+p[2]
# especificação dos comandos 1 ou mais a realizar

# region PRINT COMMAND FUNCTIONS


def p_print_command(p):
    "command : cmd_prints"
    p[0] = p[1]
# especificação do cmd_print (será desenvolvido as várias alternativas de print)


def p_cmd_prints_all(p):
    "cmd_prints : PRINT '(' cmd_print prints ')' "
    p[0] = str(p[3])+str(p[4])
# cmd_print função geral


def p_cmd_prints(p):
    "prints : '+' cmd_print prints "
    p[0] = p[2] + p[3]
# desenvolvimento de cmd_print


def p_empty_cmd_prints(p):
```

```
    "prints :"
    p[0] = ""
# cmd_print vazio


def p_print_id_command(p):
    "cmd_print : Id"
    p[0] = p[1][1]+"WRITEI\n"
# comando para dar print a um Id


def p_print_str_command(p):
    "cmd_print : STR"
    p[0] = "PUSHS " + p[1] + "\n" + "WRITES\n"
# comando para dar print a uma STR

# endregion

# region READ COMMAND FUNCTIONS


def p_read_command(p):
    "command : cmd_read"
    p[0] = p[1]


def p_read_id_command(p):
    "cmd_read : READ Id"
    p[0] = "READ\nATOI\n" + p[2][0]


# endregion

# region IF COMMAND // IF ELSE COMMAND FUNCTIONS


def p_if_command(p):
    "command : cmd_if"
    p[0] = p[1]


def p_cmd_if(p):
    "cmd_if : IF condition '{' commands '}'"
    p[0] = str(p[2])+"JZ IF" + str(parser.label) + "\n" + \
        str(p[4])+"IF"+str(parser.label)+":\n"

    parser.label += 1
```

```
def p_if_else_command(p):
    "command : cmd_if_else"
    p[0] = p[1]


def p_cmd_if_else(p):
    "cmd_if_else : IF condition '{' commands '}' ELSE '{' commands '}'"
    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


# endregion

def p_atribution_command(p):
    "command : attribution"
    p[0] = p[1]


def p_cmd_atb(p):
    "attribution : Id '=' exp"
    p[0] = p[3] + p[1][0]
# igualdade de um Id para uma expressão || resultado = num1 + num2 || resultado = num1 - 2 etc..

# region WHILE DO FUNCTIONS


def p_do_while_command(p):
    "command : cmd_while"
    p[0] = p[1]


def p_cmd_while(p):
    "cmd_while : WHILE condition DO '{' commands '}'"
    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

# endregion

# endregion


# region CONDITIONS FUNCTIONS

def p_condition(p):
```

```
    "condition : '(' context ')'"
    p[0] = p[2]


def p_condition_Neg(p):
    "condition : NOT '(' context ')'"
    p[0] = str(p[3])+"NOT\n"


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


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


def p_condition_EQUALS(p):
    "context : exp EQUALS exp"
    p[0] = str(p[1])+str(p[3])+"EQUAL\n"


def p_condition_GREATER(p):
    "context : exp '>' exp"
    p[0] = str(p[1])+str(p[3])+"SUP\n"


def p_condition_LESSER(p):
    "context : exp '<' exp"
    p[0] = str(p[1])+str(p[3])+"INF\n"


def p_condition_GREATERQ(p):
    "context : exp GREATERQ exp"
    p[0] = str(p[1])+str(p[3])+"SUPEQ\n"


def p_condition_LESSERQ(p):
    "context : exp LESSERQ exp"
    p[0] = str(p[1])+str(p[3])+"INFEQ\n"


def p_context(p):
    "context : condition"
    p[0] = p[1]
```

```
# endregion

# region EXPRESSION FUNCTIONS


def p_plus_expression(p):
    "exp : exp '+' term"
    p[0] = p[1] + p[3] + "ADD\n"


def p_minus_expression(p):
    "exp : exp '-' term"
    p[0] = p[1] + p[3] + "SUB\n"


def p_expression(p):
    "exp : term"
    p[0] = p[1]

# endregion

# region TERM FUNCTIONS


def p_division_term(p):
    "term : term '/' factor"
    p[0] = p[1] + p[3] + "DIV\n"


def p_multiplication_term(p):
    "term : term '*' factor"
    p[0] = p[1] + p[3] + "MUL\n"


def p_mod_term(p):
    "term : term '%' factor"
    p[0] = p[1] + p[3] + "MOD\n"


def p_term(p):
    "term : factor"
    p[0] = p[1]
# endregion

# region FACTORS FUNCTIONS


def p_num_factor(p):
```

```
    "factor : NUM"
    p[0] = "PUSHI "+p[1]+"\n"


def p_id_factor(p):
    "factor : Id"
    p[0] = p[1][1]

# endregion


def p_Id(p):
    "Id : ID"
    if p[1] not in parser.variables:
        parser.success = False
        print("Variable not declared: " + p[1])
        sys.exit(0)
    else:
        p[0] = ("STOREG " + str(parser.variables[p[1]])+"\n", "PUSHG " +
                str(parser.variables[p[1]])+"\n", p[1])
# definição Id = ID


def p_error(p):
    print("Syntax error!")
    parser.success = False
    sys.exit(0)
# definição de erro de sintaxe

# region ALL FUNCTIONS RELATED TO SIMPLE ARRAYS


def p_def_array_values(p):
    "values : NUM ',' values"
    p[0] = "PUSHI "+p[1]+"\n"+p[3]
    parser.arraycount += 1


def p_def_array_num(p):
    "values : NUM"
    p[0] = "PUSHI "+p[1]+"\n"
    parser.arraycount += 1


def p_def_empty_array(p):
    "values :"
    p[0] = ""
```

```
def p_array_declaration(p):
    "declaration : INT ID '[' NUM ']'"
    if p[2] in parser.variables:
        parser.success = False
        print("Multiple variable declaration " + p[2])
        sys.exit(0)
    else:

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


def p_array_num_declaration(p):
    "declaration : INT ID '[' NUM ']' '=' values"
    if p[2] in parser.variables:
        parser.success = False
        print("Multiple variable declaration " + p[2])
        sys.exit(0)

    elif (parser.arraycount) != int(p[4]):
        parser.success = False
        print("Index out of range -> variable: " + p[2])
        sys.exit(0)
    else:
        parser.variables[p[2]] = parser.count
        p[0] = p[7]
        parser.count += int(p[4])
        parser.arraycount = 0


def p_print_command_id_Array(p):
    "cmd_print : ID_Array"
    p[0] = p[1][0]+p[1][1] + "LOADN\n"+"WRITEI\n"


def p_read_id_array_command(p):
    "cmd_read : READ ID_Array"
    p[0] = p[1][0]+p[1][1] + "READ\nATOI\nSTOREN\n"


def p_id_array_factor(p):
    "factor : ID_Array"
    p[0] = (p[1][0]+p[1][1]+"LOADN\n", p[1][2])


def p_condition_exp(p):
```

```
    "context : exp"
    p[0] = str(p[1])


def p_array_exp_command(p):
    "command : ID_Array '=' exp"
    p[0] = p[1][0]+p[1][1]+p[3]+"STOREN\n"


def p_id_array(p):
    "ID_Array : ID '[' factor ']'"
    if (p[1] not in parser.variables):
        parser.success = False
        print("Multiple variable declaration " + p[1])
        sys.exit(0)
    else:
        p[0] = ("PUSHGP\nPUSHI " + str(parser.variables[p[1]])+"\nPADD\n", p[3])

        # endregion


# region ALL FUNCTIONS RELATED TO DOUBLE ARRAYS

def p_double_values(p):
    "d_values : '[' values ']' d_values"
    p[0] = p[2] + p[4]
    parser.darraycount += 1


def p_empty_double_values(p):
    "d_values :"
    p[0] = ""


def d_values_values(p):
    "d_values : '[' values ']'"
    p[0] = p[2]
    parser.darraycount += 1


def p_double_array_declaration(p):
    "declaration : INT ID '[' NUM ']' '[' NUM ']'"
    if p[2] in parser.variables:
        parser.success = False
        print("Multiple variable declaration " + p[2])
        sys.exit(0)
    else:
        parser.variables[p[2]] = parser.count
```

```
        p[0] = "PUSHN " + str(int(p[4])*int(p[7])) + "\n"
        parser.count += (int(p[4]) * int(p[7]))

def p_double_array_num_declaration(p):
    "declaration : INT ID '[' NUM ']' '[' NUM ']' '=' d_values "
    if p[2] in parser.variables:
        parser.success = False
        print("Multiple variable declaration " + p[2])
        sys.exit(0)
    elif (parser.arraycount != int(p[4])*int(p[7])) or (parser.darraycount != int(p[4])):
        parser.success = False
        print("Index out of range -> variable: " + p[2])
        sys.exit(0)
    else:
        parser.variables[p[2]] = parser.count
        p[0] = p[10]
        parser.count += (int(p[4]) * int(p[7]))
        parser.size[p[2]] = int(p[7])
        parser.arraycount = 0
        parser.darraycount = 0


def p_double_array_exp_command(p):
    "command : ID_Double_Array '=' exp"
    p[0] = p[1][0]+p[1][1]+p[3]+"STOREN\n"


def p_print_command_double_array(p):
    "cmd_print : ID_Double_Array"
    p[0] = p[1][0]+p[1][1] + "LOADN\n"+"WRITEI\n"


def p_read_command_double_array(p):
    "cmd_read : ID_Double_Array"
    p[0] = p[1][0]+p[1][1] + "READ\nATOI\nSTOREN\n"


def p_id_double_array_factor(p):
    "factor : ID_Double_Array"
    p[0] = (p[1][0]+p[1][1]+"LOADN\n")


def p_id_double_array(p):
    "ID_Double_Array : ID '[' factor ']' '[' factor ']'"
    if (p[1] not in parser.variables):
        parser.success = False
        print("Multiple variable declaration " + p[1])
        sys.exit(0)
```

```
    else:
        p[0] = ("PUSHGP\nPUSHI " + str(parser.variables[p[1]]) +
                "\nPADD\n", p[3] + "PUSHI " + str(parser.size[p[1]]) + "\nMUL\n" + p[6] + "ADD\n"


# endregion

parser = yacc.yacc()
parser.variables = {}
parser.success = True

parser.count = 0
parser.label = 0
parser.loop = 0
parser.size = {}
parser.arraycount = 0
parser.darraycount = 0

fIn = input('FileInput: ')
fOut = input('FileOutput: ')

with open(fIn, 'r') as file:
    code = file.read()
out = parser.parse(code)
with open(fOut, 'w') as output:
    output.write(str(out))

print(parser.variables)
```