Processamento de Linguagens (3º ano de Curso)

Trabalho Prático 2

Relatório de Desenvolvimento

Carlos Preto
(a89587)

Pedro Veloso
(a89557)

30 de maio de 2021

Resumo

O segundo trabalho da disciplina de Processamento de Linguagens consiste na criação de uma linguagem de programação, bem como o seu compilador.
Neste relatório pretende-se discutir as soluções encontradas para a realização do trabalho.

Conteúdo

Capítulo 1

Introdução

Com o desenvolvimento do trabalho pretende-se criar um compilador que transforme uma linguagem de programação, criada pelo grupo de trabalho, em código *assembly* que pode ser executado pela máquina virtual fornecida.
O primeiro passo realizado foi definir um analisador léxico, recorrendo à livraria *lex* da biblioteca *ply* do *python*. Posteriormente, desenvolveu-se recorrendo ao *yacc* um analisador sintático e semântico, que a partir da lista de *tokens* fornecida pelo analisador léxico obtém o significado.
Para a linguagem criada permitiu-se realizar atribuições de valores ou operações matemáticas às variáveis previamente declaradas, bem como realizar instruções condicionais ou ciclos de instruções. Para realizar estas operações é possível declarar *arrays* de inteiros com 1 ou mais dimensões ou variáveis atómicas do tipo inteiro.
Por fim, para a linguagem criada desenvolveu-se programas que resolvem pequenos problemas, como por exemplo: multiplicações de matrizes, inversão de uma lista ou determinar o menor valor de um determinado conjunto.

Capítulo 2

Analisador léxico

2.1 Tokens

O analisador léxico têm como função traduzir a sequência de caracteres de entrada num conjunto pré-determinado de símbolos léxicos. Os símbolos léxicos identificados na linguagem correspondem aos *tokens* apresentados de seguida.

2.1.1 SE

A expressão regular que encapsula a instanciação do *token* SE é a seguinte.

```
1            def t_SE(t):
2                r'(?i:SE)'
3                return t
```

2.1.2 ENQUANTO

A expressão regular que identifica o *token* ENQUANTO é a seguinte.

```
1            t_ENQUANTO = r'(?i:ENQUANTO)'
```

2.1.3 LER e ESCREVER

Os *tokens* LER e ESCREVER são identificados recorrendo às seguintes expressões regulares.

```
1            t_LER = r'(?i:LER)'
2            t_ESCREVER = r'(?i:ESCREVER)'
```

2.1.4 INTDECL e LINTDECL

As expressões regulares que identificam os *tokens* utilizados na declaração de variáveis, são as seguintes.

```
t_INTDECL = r'(?i:INT)'
t_LINTDECL = r'(?i:LINT)'
```

2.1.5 ID

De modo a identificar as os nomes das variáveis, declarou-se o *token* ID. É importante que a declaração deste *token* apareça depois dos *tokens* já referidos, de modo a que as palavras reservadas não sejam identificadas como ID's.

```
t_ID = r'\w+'
```

2.1.6 INT

Para identificar um inteiro declara-se a seguinte expressão regular.

```
def t_INT(t):
    r'\d+'
    t.value = int(t.value)
    return t
```

Contudo, ao identificar este *token* regista-se o seu valor no formato de um inteiro.

2.1.7 STRING

Uma string corresponde a um qualquer conjunto de caracteres entre aspas, por isso criou-se a seguinte expressão regular capaz de identificar o *token* STRING.

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

2.1.8 Literals

Por fim definiu-se a seguinte lista com símbolos literais.

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

Capítulo 3

Compilador

3.1 Gramática

3.1.1 Símbolos terminais

Os símbolos terminais representam um conjunto de símbolos que podem aparecer como entrada ou saída de uma produção. Para os símbolos terminais não é possível derivar-se nenhuma unidade. Desta forma, os símbolos terminais considerados para a criação da gramática são representados pelos 9 *tokens*, juntamente com a lista de símbolos literais identificados, no capítulo anterior.

3.1.2 Símbolos não terminais

Os símbolos não terminais representam o conjunto de símbolos que se derivam em uma ou mais unidades. Para a gramática criada considera-se como símbolos não terminais os 16 símbolos seguintes:

```
N = {
    programa,
    ZonaDeclaracao,
    Declaracoes,
    Declaracao,
    Array,
    ZonaInstrucoes,
    Instrucoes,
    Instrucao,
    if,
    else,
    ciclo,
    condicao,
    expressao,
    termo,
    fator,
    ListArrayExp
}
```

3.1.3 Axioma

O axioma representa a raiz da árvore de derivação, sendo por isso a primeira produção. O axioma considerado para a gramática desenvolvida é:

```
S = programa
```

3.1.4 Produções

De modo a determinar os símbolos que podem ser substituídos por outros é necessário determinar as produções da gramática. Assim para a linguagem criada, considerou-se a existência das 45 produções seguintes:

```
P = {
    P1 programa => ZonaDeclaracao ZonaInstrucoes
    P2 ZonaDeclaracao : €
    P3 ZonaDeclaracao : Declaracoes
    P4 Declaracoes : Declaracao
    P5 Declaracoes : Declaracoes Declaracao
    P6 Declaracao : INTDECL ID ';'
    P7 Declaracao : LINTDECL ID Array ';'
    P8 Array : '[' INT ']'
    P9 Array : Array '[' INT ']'
    P10 ZonaInstrucoes : €
    P11 ZonaInstrucoes : Instrucoes
    P12 Instrucoes : Instrucao
    P13 Instrucoes : Instrucoes Instrucao
    P14 Instrucao : ID '<' '-' expressao ';'
    P15 Instrucao : ID ListArrayExp '<' '-' expressao ';'
    P16 Instrucao : SE '(' condicao ')' '{' if '}'
    P17 Instrucao : SE '(' condicao ')' '{' if '}' '{' else '}'
    P18 if : ZonaInstrucoes
    P19 else : ZonaInstrucoes
    P20 Instrucao : ENQUANTO '(' condicao ')' '{' ciclo '}'
    P21 ciclo : ZonaInstrucoes
    P22 Instrucao : LER '(' ID ')' ';'
    P23 Instrucao : LER '(' ID ListArrayExp ')' ';'
    P24 Instrucao : ESCREVER '(' expressao ')' ';'
    P25 Instrucao : ESCREVER '(' STRING ')' ';'
    P26 Instrucao : ESCREVER '(' STRING ',' expressao ')' ';'
    P27 condicao : expressao '<' expressao
    P28 condicao : expressao '>' expressao
    P29 condicao : expressao '<' '=' expressao
    P30 condicao : expressao '>' '=' expressao
    P31 condicao : expressao '=' expressao
    P32 condicao : expressao '!' '=' expressao
    P33 expressao : expressao '+' termo
    P34 expressao : expressao '-' termo
    P35 expressao : termo
```

```
        P36 termo : termo '*' fator
        P37 termo : termo '%' fator
        P38 termo : termo '/' fator
        P39 termo : fator
        P40 fator : INT
        P41 fator : ID
        P42 fator : ID ListArrayExp
        P43 fator : '(' expressao ')'
        P44 ListArrayExp : '[' expressao ']'
        P45 ListArrayExp : ListArrayExp '[' expressao ']'
    }
```

Observando as produções acima apresentadas, percebe-se que um programa (axioma da gramática) é constituído por duas secções, a secção onde se declara as variáveis e a secção onde se explicitam as instruções do programa, apresentadas de seguida.

Declaração de variáveis

Nesta secção do código do programa deve ser declaradas todas as variáveis que irão ser usadas no futuro. Na linguagem desenvolvida existem dois tipos de variáveis: variáveis atómicas ou *arrays* (tipo inteiro) de N dimensões. Assim, para declarar um inteiro deve ser colocado a etiqueta 'INT' e para declarar um *array* deve ser utilizado a etiqueta 'LINT', tal como se observa no exemplo seguinte.

Listing 3.1: Exemplo de declarações de variáveis

```
1    LINT m1[2][2];   // matriz 2x2
2    LINT m2[2][5];   // matriz 2x5
3    LINT l[2];       // array com dimensao 2
4    int i;           // variavel atomica
5    INT j;           // variavel atomica
```

Por fim, referencia-se que um programa criado com a linguagem desenvolvida pode não declarar variáveis, tal como observado na produção P2. Contudo, caso não sejam declaradas variáveis o compilador apresenta um *WARNING* a evidenciar essa ocorrência.

Instruções

Nesta secção são descritas todas as instruções que o algoritmo criado deve executar. Ao observar as produções acima descritas verifica-se que existe a possibilidade de se realizar atribuições de valores (dados por um número atómico ou através de uma expressão matemática) a variáveis previamente declaradas ou realizar instruções de controlo de fluxo. Para além dessas instruções também é possível ler valores inteiros introduzidos pelo utilizador em *run time*, ou escrever valores de variáveis no ecrã, tal como se observa no exemplo seguinte.

Listing 3.2: Exemplo de instruções

```
1    ENQUANTO(i<n) {
2        ESCREVER("Introduza um valor: ");
3        LER(temp);
4
```

```
5           SE (temp%2!=0){
6                num <- num + 1;
7                ESCREVER("E impar (", temp);
8                ESCREVER(")\n");
9           }
10
11          i <- i+1;
12      }
```

Observando o exemplo e a gramática criada para a linguagem, percebe-se que:

- Para realizar a operação de atribuição segue-se o seguinte padrão 'x ¡- y', que implica guardar o valor de y na variável x.
- Para realizar a operação de escrita no ecrã do utilizador deve-se utilizar a etiqueta 'ESCREVER' seguido de uma variável de onde o valor é extraído ou de uma string.
- Para realizar a leitura de valores inteiros introduzidos pelo utilizador via teclado, utiliza-se a etiqueta 'LER' seguida da variável onde o valor será armazenado.
- Para realizar um bloco condicional utiliza-se a etiqueta 'SE' seguida da condição de entrada. A acompanhar deve ser declarado dois blocos de instruções, onde o primeiro corresponde ao caso em que a condição introduzida é verdadeira e o segundo corresponde ao caso em que a condição é falsa. Contudo, se não se pretender executar operações no caso da condição falhar o seu bloco de código correspondente pode ser omitido.
- Para realizar ciclos deve-se utilizar a etiqueta 'ENQUANTO' seguida da condição de entrada no ciclo. Esta inicialização deve ser acompanhada pelo bloco de instruções referente ao ciclo.

3.2 Tabela de identificadores

Para possibilitar a construção do código, correspondente ao desenvolvido, em *assembly* pelo compilador é necessário criar em tempo de compilação uma tabela de identificadores. Esta tabela é criada recorrendo aos dicionários definidos em *python*. O uso de dicionários permite atribuir valores a uma chave, garantindo o acesso dos mesmos em tempo constante. Assim para o compilador criado desenvolveu-se uma tabela que associa aos identificadores de cada variável, à sua posição na *stack*, ao tipo da variável e a uma lista com o tamanho de cada dimensão do *array* (no caso de ser uma variável atómica utiliza-se uma lista vazia). Assim a tabela criada para o exemplo 3.2 é a seguinte: Recorrendo a esta tabela é possível prever situações de

Tabela 3.1: Exemplo de uma tabela de identificadores

Identificador	Informação
m1	(0, 'LINT', [2,2])
m2	(4, 'LINT', [2,2])
l	(8, 'LINT', [2])
i	(10, 'INT', [])
j	(11, 'INT', [])

erro. Assim, sempre que um utilizador atribua valores a variáveis que não tenham sido declaradas ou sempre

que um utilizador utilize *arrays* com mais dimensões que as declaradas é possível ao compilador declarar a existência desses erros. Contudo, para além das situações de erro também é possível identificar situações que apesar de ser possível realizar poderão causar conflitos recorrendo à sinalização de um *WARNING*.

Capítulo 4

Exemplos de programas obtidos

4.1 Verificar se 4 números podem ser os lados de um quadrado

Para a realização de um programa que testa se 4 coordenadas podem representar os 4 lados de um quadrado desenvolveu-se o código seguinte.

```
1     INT x1;
2     INT x2;
3     INT x3;
4     INT x4;
5
6     INT y1;
7     INT y2;
8     INT y3;
9     INT y4;
10
11    ESCREVER("Considere o seguinte quadrado:\n");
12    ESCREVER("\n");
13    ESCREVER("A ------ B\n");
14    ESCREVER("|        |\n");
15    ESCREVER("C ------ D\n");
16
17    ESCREVER("Introduza o valor da coordenada Ax: ");
18    LER(x1);
19    ESCREVER("Introduza o valor da coordenada Ay: ");
20    LER(y1);
21
22    ESCREVER("Introduza o valor da coordenada Bx: ");
23    LER(x2);
24    ESCREVER("Introduza o valor da coordenada By: ");
25    LER(y2);
26
27    ESCREVER("Introduza o valor da coordenada Cx: ");
28    LER(x3);
29    ESCREVER("Introduza o valor da coordenada Cy: ");
30    LER(y3);
31
32    ESCREVER("Introduza o valor da coordenada Dx: ");
```

```
33      LER(x4);
34      ESCREVER("Introduza o valor da coordenada Dy: ");
35      LER(y4);
36
37      SE (y1 = y2){
38          SE (x1 = x3){
39              SE (y3 = y4){
40                  SE (x2 = x4){
41                      ESCREVER("E um quadrado!\n");
42                  }{
43                      ESCREVER("Nao e um quadrado!\n");
44                  }
45              }{
46                  ESCREVER("Nao e um quadrado!\n");
47              }
48          }{
49              ESCREVER("Nao e um quadrado!\n");
50          }
51      }{
52          ESCREVER("Nao e um quadrado!\n");
53      }
```

Ao analisar o código é possível perceber que para determinar a solução foram declaradas 8 variáveis atómicas que permitem registar as coordenadas de x e y dos 4 pontos correspondentes aos lados do triângulo. Posteriormente são lidos do teclado 8 valores que irão corresponder às coordenadas, que para determinarem um quadrado, cada conjunto de 2 pontos deve ter coordenadas em x ou y iguais. Para este exemplo não se considera que um quadrado pode estar rotacionado ou que os pontos introduzidos não correspondem a um outro lado do triângulo, que não o pedido.

4.2 Leitura de um inteiro N, seguida da leitura de N números e posterior escrita do menor deles

Para realizar a leitura de N números deve ser lido, em primeiro lugar, o valor que N irá tomar. Posteriormente realiza-se um ciclo que irá executar enquanto uma variável (iniciada a a zero) seja menor que o valor de N (previamente lido). Em cada iteração do ciclo é só verificar se o número lido é menor que o valor já registado como menor (a variável que executa este papel deve ser inicializada com um valor alto de modo a que, na primeira iteração qualquer valor lido substitua o valor desta variável). No final deve-se escrever no ecrã do utilizador o menor valor introduzido. Este algoritmo encontra-se descrito de seguida.

```
1       INT n;
2       INT temp;
3       INT i;
4       INT menor;
5
6       menor <- 9999;
7
8       ESCREVER("Introduza a quantidade de numeros a ler: ");
9       LER(n);
10
11      ENQUANTO(i<n) {
12          ESCREVER("Introduza um valor: ");
```

```
13          LER(temp);
14
15          SE(temp<menor){
16               menor <- temp;
17          }
18
19          i <- i+1;
20      }
21
22      ESCREVER("O menor valor introduzido foi: ", menor);
23      ESCREVER("\n");
```

4.3 Leitura de um número N e acompanhado da leitura de N números, calculando no final o seu produtório

Uma vez mais, em primeiro lugar deve ser lido o valor que corresponde à variável N e posteriormente realizar um ciclo em que o seu corpo irá ser executado N vezes. A cada iteração do ciclo deve-se multiplicar um valor lido por o valor registado na variável, iniciada a 1 (elemento neutro da multiplicação), que acumula o resultado. Este processo pode ser observado no código apresentado de seguida.

```
1       INT n;
2       INT i;
3       INT prod;
4       INT temp;
5
6       prod <- 1;
7
8       ESCREVER("Introduza a quantidade de numeros a ler: ");
9       LER(n);
10
11      ENQUANTO(i<n) {
12          ESCREVER("Introduza um valor: ");
13          LER(temp);
14
15          prod <- prod * temp;
16
17          i <- i+1;
18      }
19
20      ESCREVER("O produto dos valores introduzidos e: ", prod);
21      ESCREVER("\n");
```

4.4 Contagem e impressão dos números ímpares de uma sequência de números naturais

Para a resolução deste problema deve ser lida a quantidade de iterações que o ciclo seguinte irá realizar, o que implica definir a quantidade de números a considerar para a resolução total do problema apresentado. Assim, sempre que se lê um número deve-se realizar a verificação do veracidade da paridade desse mesmo

número. Caso o número seja ímpar escreve-se uma mensagem no ecrã a indicar a ocorrência e incrementa-se um contador, que no final irá conter o número de vezes que se introduziu um valor ímpar.
A realização do algoritmo descrito em cima é apresentada de seguida, contudo, poderá existir mais do que uma forma de abordar o problema.

```
1     INT n;
2     INT i;
3     INT num;
4     INT temp;
5
6     ESCREVER("Introduza a quantidade de numeros a ler: ");
7     LER(n);
8
9     ENQUANTO(i<n) {
10        ESCREVER("Introduza um valor: ");
11        LER(temp);
12
13        SE (temp%2!=0){
14            num <- num + 1;
15            ESCREVER("E impar (", temp);
16            ESCREVER(")\n");
17        }
18
19        i <- i+1;
20    }
21
22    ESCREVER("\n");
23    ESCREVER("Introduziu ", num);
24    ESCREVER(" numeros impares\n");
```

4.5 Leitura e armazenamento de N números num array; imprimir os valores por ordem inversa.

Uma possibilidade para escrever todos valores lidos e armazenados num *array* de forma inversa é realizar a escrita desses mesmos valores a começar no último elemento do *array* e a acabar no índice 0. Contudo, existem mais possibilidades, como por exemplo registar os elementos de forma inversa num outro *array*.
Um exemplo para este problema é a seguinte abordagem apresentada.

```
1     LINT l[10];
2     INT i;
3     INT temp;
4
5     ENQUANTO(i < 10) {
6         ESCREVER("Introduza um valor: ");
7         LER(l[i]);
8
9         i <- i+1;
10    }
11
12    i <- 9;
```

```
13
14      ESCREVER("lista: [");
15      ENQUANTO(i >= 0) {
16          ESCREVER(l[i]);
17          ESCREVER(",");
18          i <- i-1;
19      }
20      ESCREVER("]\n");
```

4.6 Produto de duas matrizes

Para a realização de um algoritmo que realiza a multiplicação de duas matrizes, retirou-se, da *Internet*, um excerto de código escrito na linguagem C e adaptou-se para a linguagem criada. Assim obteve-se o seguinte algoritmo que multiplica duas matrizes 2x2, sendo uma matriz constituída por apenas elementos com valor 1 e uma outra matriz com apenas o valor 2, tal como se apresenta de seguida.

```
1       LINT m1[2][2];
2       LINT m2[2][2];
3       LINT res[2][2];
4       int i;
5       INT j;
6       INT x;
7       INT aux;
8
9       m1[0][0] <- 1;
10      m1[0][1] <- 1;
11      m1[1][0] <- 1;
12      m1[1][1] <- 1;
13
14      m2[0][0] <- 2;
15      m2[0][1] <- 2;
16      m2[1][0] <- 2;
17      m2[1][1] <- 2;
18
19      ENQUANTO(i<2){
20          j <- 0;
21          ENQUANTO(j<2){
22              x <- 0;
23              ENQUANTO(x<2){
24                  aux <- aux + m1[i][x] * m2[x][j];
25                  x <- x + 1;
26              }
27              res[i][j] <- aux;
28              aux <- 0;
29              j <- j + 1;
30          }
31          i <- i + 1;
32      }
33
34      i <- 0;
35      ENQUANTO(i<2){
```

```
36          j <- 0;
37          ENQUANTO(j<2){
38              ESCREVER(res[i][j]);
39              ESCREVER(" ");
40              j <- j + 1;
41          }
42          ESCREVER("\n");
43          i <- i + 1;
44      }
```

4.7 Ordenar um conjunto de elementos

Para ordenar um conjunto de elementos, uma vez mais recorreu-se a um excerto de código escrito em C, que depois de adaptado permitiu realizar o algoritmo de *insertion sort* na linguagem criada. O algoritmo desenvolvido encontra-se descrito de seguida.

```
1       INT i;
2       INT j;
3       INT count;
4       INT temp;
5       LINT number[25];
6       INT flag;
7
8       ESCREVER("Insira o quantidade de numeros a introduzir: ");
9       LER(count);
10
11      ENQUANTO(i<count){
12          ESCREVER("Introduza o ", (i+1));
13          ESCREVER(" numero: ");
14          LER(number[i]);
15          i <- i+1;
16      }
17
18      i <- 1;
19
20      ENQUANTO(i<count){
21          temp <- number[i];
22          j <- i-1;
23
24          SE(temp<number[j]){
25           SE(j>=0){
26                  flag <- 1;
27              }{
28                  flag <- 0;
29              }
30          }{
31              flag <- 0;
32          }
33
34          ENQUANTO(flag = 1){
35              number[j+1] <- number[j];
```

```
36              j <- j-1;
37
38              SE(temp<number[j]){
39               SE(j>=0){
40                      flag <- 1;
41                  }{
42                  flag <- 0;
43                  }
44              }{
45                  flag <- 0;
46              }
47          }
48          number[j+1] <- temp;
49          i <- i+1;
50      }
51
52      ESCREVER("Ordem dos elementos ordenados:\n");
53
54      i <- 0;
55
56      ENQUANTO(i<count){
57          ESCREVER(number[i]);
58          ESCREVER(" -> ");
59          i <- i+1;
60      }
61      ESCREVER("\n");
```

Capítulo 5

Conclusão

Com a realização do trabalho criou-se uma linguagem de programação que recorrendo à máquina virtual permite a execução de algoritmos.
Na linguagem de programação criada considerou-se a existência de dois tipos de variáveis: as variáveis atómicas inteiras (INT) e os *arrays* de inteiros (LINT). O uso das variáveis, em conjunto com operações de leitura do teclado, escrita no ecrã, ciclos e instruções condicionais permite a criação de algoritmos como por exemplo a multiplicação de matrizes, inversão de uma lista de inteiros ou ordenação de valores (*insertion sort*).
Para a realização da linguagem foi necessário construir um analisador léxico que em conjunto da gramática apresentada neste relatório permite construir o compilador da linguagem.
Por fim, admite-se que a linguagem de programação criada é simplista, uma vez que apenas admite um tipo de dados (tipo inteiro), sendo por isso fundamental no futuro adicionar a possibilidade de declarar e manipular outros tipos, como por exemplo tipos com virgula flutuante ou caracteres. Para além disto, deve-se implementar num futuro a possibilidade de realizar outros tipos de ciclos e realizar a declaração de funções.

Apêndice A

Código do Programa

Lista-se a seguir o CÓDIGO desenvolvido para o analisador léxico.

```
import ply.lex as lex

tokens = [
    'INT',
    'ID',
    'STRING',
    'SE',
    'ENQUANTO',
    'LER',
    'ESCREVER',
    'INTDECL',
    'LINTDECL'
]

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

def t_SE(t):
    r'(?i:SE)'
    return t

t_INTDECL = r'(?i:INT)'

t_LINTDECL = r'(?i:LINT)'

t_ENQUANTO = r'(?i:ENQUANTO)'

t_LER = r'(?i:LER)'

t_ESCREVER = r'(?i:ESCREVER)'

t_ID = r'\w+'

def t_INT(t):
```

```
    r'\d+'
    t.value = int(t.value)
    return t

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

t_ignore = ' \n\t'

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

lexer = lex.lex()
```

O CÓDIGO desenvolvido recorrendo ao yacc é o seguinte.

```
import ply.yacc as yacc
import math

from compilador_lex import tokens

def p_programa(p):
    "programa : ZonaDeclaracao ZonaInstrucoes"
    p[0] = p[1] + 'START\n' + p[2] + 'STOP'


#Zona de declaracoes
def p_declaracao_vazia(p):
    "ZonaDeclaracao : "
    p[0] = ""

def p_declaracao_notvazia(p):
    "ZonaDeclaracao : Declaracoes"
    p[0] = p[1]

def p_declaracoes_paragem(p):
    "Declaracoes : Declaracao"
    p[0] = p[1]

def p_declaracoes_rec(p):
    "Declaracoes : Declaracoes Declaracao"
    p[0] = p[1] + p[2]


#Declarar um inteiro
```

```
def p_declaracaoID(p):
    "Declaracao : INTDECL ID ';' "
    x = p.parser.numVar
    p.parser.numVar += 1
    p.parser.variaveis[p[2]] = (x,'INT',[])
    res = "PUSHI 0\n"
    p[0] = res

def p_declaracaoList(p):
    "Declaracao : LINTDECL ID Array ';' "
    x = p.parser.numVar
    espaco = math.prod(p[3])
    p.parser.numVar += espaco
    p.parser.variaveis[p[2]] = (x,'LINT',p[3])

    p[0] = f'PUSHN {espaco}\n'

def p_arraySingle(p):
    "Array : '[' INT ']'"
    p[0] = [p[2]]

def p_arrayNotSingle(p):
    "Array : Array '[' INT ']'"
    p[0] = p[1] + [p[3]]

# Zona de instrucoes
def p_zonaInstrucoesVazia(p):
    "ZonaInstrucoes : "
    p[0] = ""

def p_zonaInstrucoesNotVazia(p):
    "ZonaInstrucoes : Instrucoes"
    p.parser.temInst = True
    p[0] = p[1]

def p_InstrucoesParagem(p):
    "Instrucoes : Instrucao"
    p[0] = p[1]

def p_Instrucoes(p):
    "Instrucoes : Instrucoes Instrucao"
    p[0] = p[1] + p[2]


#Atribuicoes
def p_AtribuicaoInt(p):
    "Instrucao : ID '<' '-' expressao ';' "
    if(p[1] in p.parser.variaveis):
```

```
        (pos,tipo,ig2) = p.parser.variaveis[p[1]]
        if(tipo != 'INT'):
            print(f'WARNING: Atribuir valor a um array sem declarar a posição.
                    A usar: {p[1]}[0]...')
        p[0] = p[4] + f'STOREG {pos}\n'
    else:
        print(f'ERRO: Variável "{p[1]}" não declarada.')
        p.parser.sucesso = False
        p[0] = "\n"

def p_AtribuicaoListEXP(p):
    "Instrucao : ID ListArrayExp '<' '-' expressao ';' "
    if(p[1] in p.parser.variaveis):
        (pos,tipo,tam) = p.parser.variaveis[p[1]]

        if(tipo == 'LINT'):

            if(len(p[2]) == len(tam)):
                l = p[2]
                temp = ""

                if(len(l) != 1):
                    i = 0
                    while(i < (len(l)-1)):
                        temp += l[i] + f'PUSHI {tam[i]}\n' + 'MUL\n'
                        if((i+1)<((len(l)-1))):
                            i += 1
                            temp += l[i] + f'PUSHI {tam[i]}\n' + 'MUL\n' + 'ADD\n'
                        i += 1
                    temp += l[len(l)-1] + 'ADD\n'
                else:
                    temp = l[0]

                p[0] = 'PUSHGP\n' + f'PUSHI {pos}\n' + 'PADD\n' + temp + p[5] +'STOREN\n'
            else:
                print(f'ERRO: Variavel "{p[1]}" apenas contém {len(tam)} dimensões
                        (contra as {len(p[2])} dimensões a usar)')
                p.parser.sucesso = False
                p[0] = "\n"
        else:
            print('WARNING: Atribuição de índices a uma variável INT. IGNORANDO o índice.')
            p[0] = p[5] + f'STOREG {pos}\n'
    else:
        print(f'ERRO: Variável "{p[1]}" não declarada.')
        p.parser.sucesso = False
        p[0] = "\n"

# IF
```

```
def p_InstrucaoIF(p):
    "Instrucao : SE '(' condicao ')' '{' if '}' "
    num = p.parser.inst
    p[0] = p[3] + f'JZ endIF{num}\n' + p[6] + f'endIF{num}: nop\n'
    p.parser.inst += 1

def p_InstrucaoIFElse(p):
    "Instrucao : SE '(' condicao ')' '{' if '}' '{' else '}' "
    num = p.parser.inst
    p[0] = p[3] + f'JZ else{num}\n' + p[6] + f'JUMP endIf{num}\n'
                + f'else{num}: nop\n' + p[9] + f'endIf{num}: nop\n'
    p.parser.inst += 1

def p_if(p):
    "if : ZonaInstrucoes"
    p[0] = p[1]

def p_else(p):
    "else : ZonaInstrucoes"
    p[0] = p[1]


#while
def p_InstrucaoENQUANTO(p):
    "Instrucao : ENQUANTO '(' condicao ')' '{' ciclo '}' "
    num = p.parser.inst
    p[0] = f'testa{num}: nop\n' + p[3] + f'JZ saiciclo{num}\n' + p[6]
                                + f'JUMP testa{num}\n' + f'saiciclo{num}: nop\n'
    p.parser.inst += 1

def p_ciclo(p):
    "ciclo : ZonaInstrucoes"
    p[0] = p[1]


#Ler do input
def p_InstrucaoLER(p):
    "Instrucao : LER '(' ID ')' ';' "
    res = "\n"
    if(p[3] not in p.parser.variaveis):
        print(f'ERRO: Variavel "{p[3]}" não declarada.')
        p.parser.sucesso = False
        p[0] = "\n"
    else:
        (pos,tipo,ig2) = p.parser.variaveis[p[3]]
        if(tipo!='INT'):
            print(f'WARNING: Atribuir valor a um array sem declarar a posição.
                            A usar: {p[3]}[0]...')
```

```
        res = 'READ\n' + 'ATOI\n' + f'STOREG {pos}\n'
    p[0] = res

def p_InstrucaoLERarray(p):
    "Instrucao : LER '(' ID ListArrayExp ')' ';' "

    if(p[3] in p.parser.variaveis):
        (pos,tipo,tam) = p.parser.variaveis[p[3]]

        if(tipo == 'LINT'):

            if(len(p[4]) == len(tam)):
                l = p[4]
                temp = ""

                if(len(l) != 1):
                    i = 0
                    while(i < (len(l)-1)):
                        temp += l[i] + f'PUSHI {tam[i]}\n' + 'MUL\n'
                        if((i+1)<((len(l)-1))):
                            i += 1
                            temp += l[i] + f'PUSHI {tam[i]}\n' + 'MUL\n' + 'ADD\n'
                        i += 1
                    temp += l[len(l)-1] + 'ADD\n'
                else:
                    temp = l[0]

                p[0] = 'PUSHGP\n' + f'PUSHI {pos}\n' + 'PADD\n' + temp + 'READ\n'
                                + 'ATOI\n' +'STOREN\n'
            else:
                print(f'ERRO: Variável "{p[3]}" apenas contém {len(tam)} dimensões
                        (contra as {len(p[4])} dimensões a usar)')
                p.parser.sucesso = False
                p[0] = "\n"
        else:
            print('WARNING: Atribuição de índices a uma variável INT. IGNORANDO o índice.')
            p[0] = 'READ\n' + 'ATOI\n' + f'STOREG {pos}\n'
    else:
        print(f'ERRO: Variável "{p[3]}" não declarada.')
        p.parser.sucesso = False
        p[0] = "\n"

#Escrever no output
def p_InstrucaoESCREVERExp(p):
    "Instrucao : ESCREVER '(' expressao ')' ';' "
    p[0] = p[3] + 'WRITEI\n'

def p_InstrucaoESCREVERS(p):
```

```
    "Instrucao : ESCREVER '(' STRING ')' ';' "
    p[0] = f'PUSHS {p[3]}\n' + 'WRITES\n'

def p_InstrucaoESCREVERSV(p):
    "Instrucao : ESCREVER '(' STRING ',' expressao ')' ';' "
    p[0] = f'PUSHS {p[3]}\n' + 'WRITES\n' + p[5] + 'WRITEI\n'

# Condicoes
def p_condicao_menor(p):
    "condicao : expressao '<' expressao"
    p[0] = p[1] + p[3] + 'INF\n'

def p_condicao_maior(p):
    "condicao : expressao '>' expressao"
    p[0] = p[1] + p[3] + 'SUP\n'

def p_condicao_menorI(p):
    "condicao : expressao '<' '=' expressao"
    p[0] = p[1] + p[4] + 'INFEQ\n'

def p_condicao_maiorI(p):
    "condicao : expressao '>' '=' expressao"
    p[0] = p[1] + p[4] + 'SUPEQ\n'

def p_condicao_igual(p):
    "condicao : expressao '=' expressao"
    p[0] = p[1] + p[3] + 'EQUAL\n'

def p_condicao_diff(p):
    "condicao : expressao '!' '=' expressao"
    p[0] = 'PUSHI 1\n' + p[1] + p[4] + 'EQUAL\n' + 'SUB\n'


#operacoes matematicas
def p_expressao_plus(p):
    "expressao : expressao '+' termo"
    p[0] = p[1] + p[3] + 'ADD\n'

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

def p_expressao(p):
    "expressao : termo"
    p[0] = p[1]

def p_termo_vezes(p):
    "termo : termo '*' fator"
```

```
    p[0] = p[1] + p[3] + 'MUL\n'

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

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

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

def p_fator1(p):
    "fator : INT"
    p[0] = f'PUSHI {p[1]}\n'

def p_fator2(p):
    "fator : ID"
    res = "\n"
    if(p[1] not in p.parser.variaveis):
        print(f'ERRO: Variavel "{p[1]}" não declarada.')
        p.parser.sucesso = False
        p[0] = "\n"
    else:
        (pos,tipo,ig2) = p.parser.variaveis[p[1]]
        if(tipo!='INT'):
            print(f'WARNING: Atribuir valor a um array sem declarar a posição.
                             A usar: {p[1]}[0]...')
        res = f'PUSHG {pos}\n'

    p[0] = res

def p_fatorL(p):
    "fator : ID ListArrayExp "
    if(p[1] in p.parser.variaveis):
        (pos,tipo,tam) = p.parser.variaveis[p[1]]

        if(tipo == 'LINT'):

            if(len(p[2]) == len(tam)):
                l = p[2]
                temp = ""

                if(len(l) != 1):
                    i = 0
                    while(i < (len(l)-1)):
```

```
                        temp += l[i] + f'PUSHI {tam[i]}\n' + 'MUL\n'
                        if((i+1)<((len(l)-1))):
                            i += 1
                            temp += l[i] + f'PUSHI {tam[i]}\n' + 'MUL\n' + 'ADD\n'
                        i += 1
                    temp += l[len(l)-1] + 'ADD\n'
                else:
                    temp = l[0]

                p[0] = 'PUSHGP\n' + f'PUSHI {pos}\n' + 'PADD\n' + temp + 'LOADN\n'
            else:
                print(f'ERRO: Variavel "{p[1]}" apenas contém {len(tam)} dimensões
                        (contra as {len(p[2])} dimensões a usar)')
                p.parser.sucesso = False
                p[0] = "\n"
        else:
            print('WARNING: Atribuição de índices a uma variável INT. IGNORANDO o índice.')
            res = f'PUSHG {pos}\n'
    else:
        print(f'ERRO: Variável "{p[1]}" não declarada.')
        p.parser.sucesso = False
        p[0] = "\n"


def p_fatorExp(p):
    "fator : '(' expressao ')'"
    p[0] = p[2]

def p_ListArrayExpStop(p):
    "ListArrayExp : '[' expressao ']' "
    p[0] = [p[2]]

def p_ListArrayExp(p):
    "ListArrayExp : ListArrayExp '[' expressao ']' "
    p[0] = p[1] + [p[3]]

# error
def p_error(p):
    parser.sucesso = False
    print('ERRO: Syntax error -> ',p)


##############################################################################

parser = yacc.yacc()

parser.inst = 0
parser.numVar = 0
```

```
parser.variaveis = {}
parser.sucesso = True
parser.temInst = False

import sys
import re
param = sys.argv[1:]

if(len(param)!=0):
    fich = param[0]

    outList = re.split(r'\.',fich)
    outCount = 0
    outName = ""
    while(outCount < (len(outList)-1)):
        outName += outList[outCount] + '.'
        outCount += 1
    outName = outName + 'vm'

    f = open(fich, encoding='UTF-8')
    r = f.read()
    res = parser.parse(r)
    f.close()

    if(len(parser.variaveis) == 0):
        print('WARNING: Não foram declaradas variaveis.')

    if(parser.temInst == False):
        print('WARNING: Não foram declaradas instruções para o programa.')

    if(parser.sucesso == False):
        pass
    else:
        output = open(outName,'w', encoding='UTF-8')
        output.write(res)
        output.close()
else:
    print('ERRO: Nenhum ficheiro introduzido.')
    pass
```