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

Trabalho Prático 2

Relatório de Desenvolvimento

Eduardo Diogo Costa Soares
(A95917)

Paulo André Alegre Pinto
(A97391)

10 de janeiro de 2024

Resumo

O presente relatório aborda a elaboração de um compilador destinado à implementação de uma linguagem original. Este projeto é parte integrante do segundo trabalho prático da disciplina de Processamento de Linguagens e Compiladores, no qual foram utilizados os módulos *lex* e *ply* em *Python*. O compilador desenvolvido engloba um analisador léxico e um analisador sintático, ambos configurados para aplicar as regras gramaticais previamente definidas. O resultado desse processo é a geração de um conjunto de instruções destinadas à máquina virtual EWVM, que, por sua vez, é equivalente ao programa original escrito na linguagem criada para o projeto. O relatório detalha o processo de desenvolvimento, desde a conceção das regras gramaticais até a produção final de código executável na máquina virtual mencionada.

Conteúdo

1 Introdução **3**

2 Análise do problema **4**
2.1 Enunciado . . . 4
2.2 Análise do Enunciado . . . 5

3 Desenho conceptual do compilador **6**

4 Linguagem ISTINC **8**
4.1 Estrutura do programa . . . 8
4.2 Declarações e Atribuições . . . 8
4.3 Operações aritméticas, lógicas e condicionais . . . 8
4.4 Estruturas de controlo de execução . . . 9
4.5 Instruções de repetição . . . 10
4.6 Arrays . . . 10
4.7 Comentários . . . 10

5 Codificação e Testes **11**
5.1 Lexer . . . 11
5.2 Parser . . . 12
5.2.1 Estrutura Geral . . . 12
5.2.2 Declaração de Variáveis . . . 13
5.2.3 Corpo do programa . . . 19
5.3 Alternativas, Decisões e Problemas de Implementação . . . 24
5.4 Testes realizados e Resultados . . . 24
5.4.1 *Swap* . . . 24
5.4.2 *Uso de IF e ELSE* . . . 25
5.4.3 Declaração de um *array* multi-dimencional . . . 26
5.4.4 Exemplo de um *array* e um *while* . . . 29
5.4.5 Exemplo da utilização de variáveis *STRING* e várias condições interligadas . . . 31
5.4.6 Exemplo de Aninhamento de *IfElse* e leitura de *Input* . . . 31
5.4.7 Exemplos de erros . . . 33
5.5 Gramática da Linguagem . . . 35

6 Conclusão **37**

A Código do Programa **38**
A.1 Lex 38
A.2 Parser 42

Capítulo 1

Introdução

A comunicação eficaz entre humanos e máquinas demanda a presença de linguagens estruturadas, as quais são passíveis de análise sintática. Nesse contexto, ocorre o processo de parsing, no qual o texto de origem é desmembrado em unidades mais básicas e significativas. Antecedendo esse procedimento, encontra-se a análise léxica, na qual sequências de caracteres são transformadas em tokens. Este último estágio envolve a identificação e categorização de elementos léxicos, preparando assim o terreno para a subsequente análise sintática. Esses processos são fundamentais para a compreensão e interpretação de comandos e instruções fornecidos pelos usuários, possibilitando uma interação eficiente entre seres humanos e sistemas.

Das duas opções do enunciado optamos por implementar *Arrays* com 1 e 2 dimensões.

Estrutura do Relatório

- 2 **Análise do problema**
- 3 **Desenho conceptual do compilador**
- 4 **Linguagem ISTINC**
- 5 **Codificação**
- 6 **Conclusão**

Capítulo 2

Análise do problema

2.1 Enunciado

Pretende-se que comece por definir uma linguagem de programação imperativa simples, a seu gosto. Apenas deve ter em consideração que essa linguagem terá de permitir:

- *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 *condicionais* para controlo do fluxo de execução;
- *efetuar* instruções cílicas para controlo do fluxo de execução, permitindo o seu aninhamento.
  Note que deve implementar pelo menos o ciclo **while-do**, **repeat-until** ou **for-do**.

Adicionalmente deve ainda suportar, à sua escolha, uma das duas funcionalidades seguintes:

- *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);
- *definir e invocar subprogramas* sem parâmetros mas que possam retornar um resultado do tipo inteiro.

Como é da praxe neste tipo de linguagens, as variáveis deverão ser declaradas no início do programa e não pode haver re-declarações, nem utilizações sem declaração prévia. Se nada for explicitado, o valor da variável após a declaração é 0 (zero). Desenvolva, então, um compilador para essa linguagem com base na GIC criada acima e com recurso aos módulos Yacc/ Lex do PLY/Python. O compilador deve gerar pseudo-código, Assembly da Máquina Virtual VM.
Muito Importante:
Para a entrega do TP deve preparar um conjunto de testes (programas-fonte escritos na sua linguagem) e mostrar o código *Assembly* gerado bem como o programa a correr na máquina virtual *VM*.

2.2 Análise do Enunciado

No desenvolvimento da nossa linguagem, definimos uma gramática independente de contexto (*GIC*) e um compilador em *python* que gera um código *Assembly* que é executado por uma máquina de *stack* virtual(VM). Neste trabalho todosos pontos de objetivo foram cumpridos, o nosso programa consegue:

- declarar variáveis do tipo inteiro que permitem realizar as operações de aritmética, relacionamento e lógica, para além disso, como *extra* também conseguimos declarar variáveis do tipo booleano e string;
- efetuar instruções algorítmicas básicas;
- ler do stdin e escrever no stdout da máquina virtual;
- efetuar instruções condicionais para controlo do fluxo de execução;
- efetuar inestruções cíclicas equivalnetes aos ciclos *while-do* e *do-while*.

No ponto adicional do enunciado preferimos manusear variáveis do tipo *array* de inteiros de 1 e 2 dimensões que permitem indexação de números inteiros. Como pedido fazemos as instruções de declaração no início do programa e só depois desenvolvemos o corpo do mesmo, para além disso foram desenvolvidas declarações sem atribuição de valor que automaticamente atribui o valor *0*, e *TRUE* para variáveis.

Capítulo 3

Desenho conceptual do compilador

O objetivo de todo o compilador é converter um programa fonte dado para um programa objeto que uma máquina conseguirá por si converter para um programa executável. Neste caso o nosso compilador irá produzir um programa objeto orientado para a máquina virtual EWVM reconhecer. Há principalmente duas fases essenciais de um compilador, a fase de análise e a fase de síntese.
Dado um programa fonte na linguagem de programação criada esta é analisada lexicamente pelo nosso analisador léxico (usando ply.lex), sintaticamente pelo nosso analisador sintático (usando ply.yacc) e para obter uma linguagem coesa em cada passo da análise são tratados vários tipos de erros ao longo da análise.
Feitas as análises do programa fonte de entrada é feita a tradução do programa fonte para o programa objeto, constituído pelas várias isntruções que fornece a máquina virtual EWVM. Através do parser ascendente criado e com os devidos tratamentos de erros consoante a nossa gramática criada obtemos por fim o desejado programa objeto.

Na página a seguir é apresentado um desenho de como é composta a estrutura do compilador.

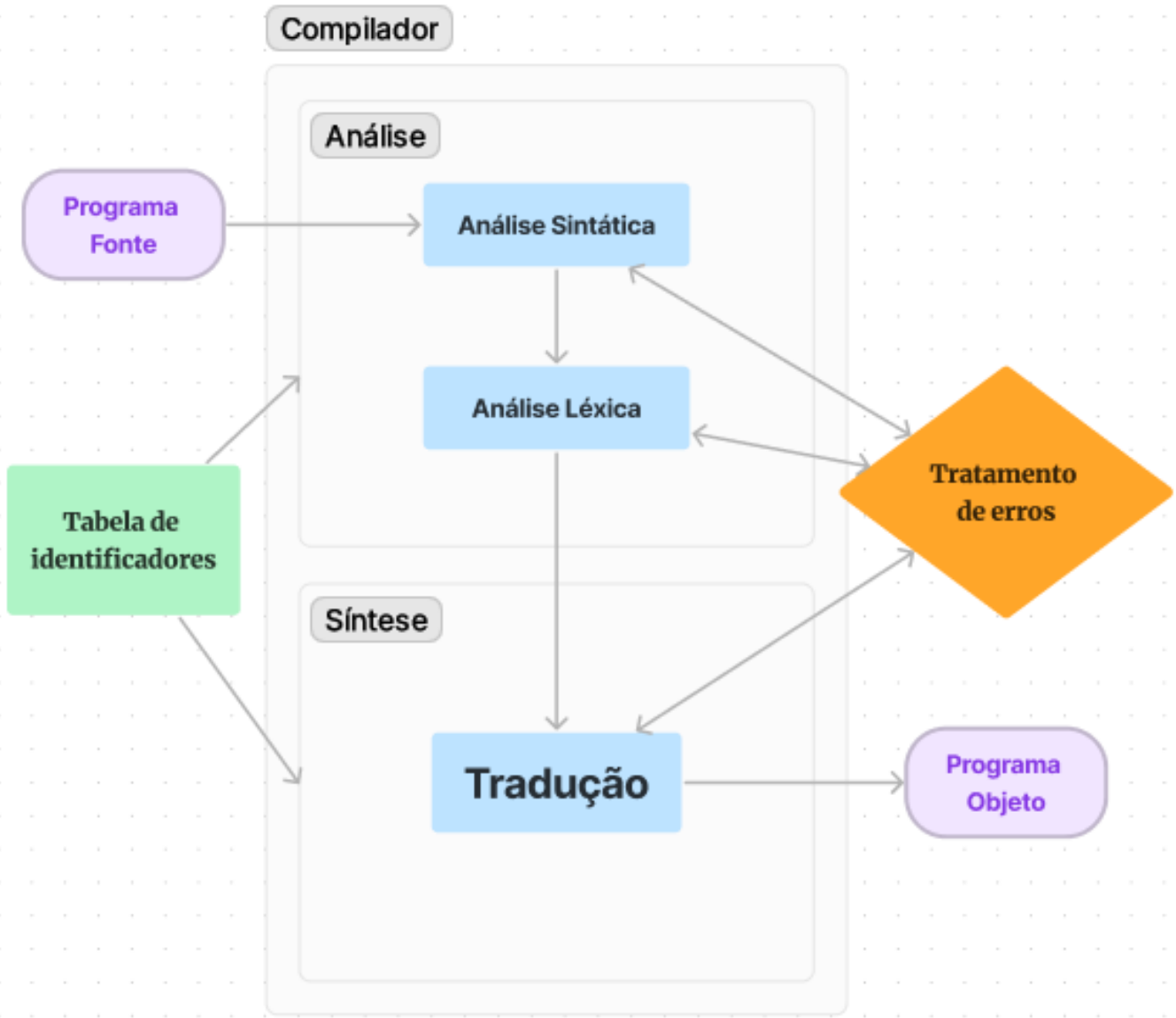

Figura 3.1: Estrutura do compilador

Capítulo 4

Linguagem ISTINC

Para a linguagem criada, batizamo-la de **ISTINC** ("I Swear This Is Not C). A sua sintaxe é inspirada por **C** com poucas diferenças. **ISTINC** opera sobre inteiros e floats.

4.1 Estrutura do programa

Qualquer programa escrito em **ISTINC** deve seguir a seguinte estrutura: As declarações de variáveis só podem ser feitas no início programa. Todos os comandos devem terminar com ';'. Todas as condições IF são acompanhadas obrigatoriamente de uma condição ELSE.

4.2 Declarações e Atribuições

A declaração de variáveis é feita da mesma forma que em **C**. Inicialmente é indicado o tipo de dados e seguidamente o nome da variável, para atribuir um valor pode ser no mesmo comando ou não.
O valor *DEFAULT* para inteiros é 0, para arrays é 0 para cada elemento, para bools é *TRUE* e para strings é (string vazia).

```
1 INT a;
2 INT b,c,d;
3 INT x=3,y,z=2;
4 a = 5;
5 INT c;
6 ARRAY x [2];
7 ARRAY y [3] = {1,2,3};
8 BOOL vf = FALSE;
9 STRING p = "palavras";
```

Listing 4.1: Exemplo de Declaração e Atribuição

4.3 Operações aritméticas, lógicas e condicionais

ISTINC é capaz de realizar as operações aritméticas habituais, a adição (+), subtração (-), divisão (/), multiplicação (*). Além disso estão implementadas as operações lógicas, *AND*, *OR* e *NOT*. Para mais, a linguagem contém também as operações usuais de relação como maior (>), maior ou igual (>=), menor (<), menor ou igual (<=), igual (==) e diferente (!=).

```
1
2 a + b
3
4 a - b
5
6 a / b
7
8 a * b
9
10 (a) AND (b)
11
12 (a) OR (b)
13
14 NOT (a)
15
16 a >= b
17 a <= b
18 a > b
19 a < b
20
21 a == b
22
23 a != b
```

Listing 4.2: Exemplo de Operações

Note-se que quando feitas duas condições interligadas por um operador lógico *AND* ou *OR* as condições necessitam de estar entre parênteses.

4.4 Estruturas de controlo de execução

Em termos de estruturas de controlo, **ISTINC** possui a estrutura mais usual, *if*, escrito de modo totalmente análogo àquele de **C**, com a diferença que o *else* é obrigatório e os tokens *if* e *else* são em maiúsculas.

```
1
2 IF ( a != 2) {
3     a = a + 1;
4 } ELSE {
5     (...);
6 }
7
8 IF (a == 2) {
9     PRINT("acabou");
10 } ELSE {
11
12 }
```

Listing 4.3: Exemplo estruturas de controlo

4.5 Instruções de repetição

Como estrutura de repetição foi implementado o *JUMPTO* acompanhado de uma marca *MARK* e uma condição, estes dois tokens e a condição em conjunto no fundo funcionam como o *while* da linguagem C, e o *TOJUMP*, que equivale ao *dowhile* do C.

```
?? tojump ??
MARK{
    x = y + 1;
    y = y + 1;
} IF( x == 10 ) JUMP;

?? jumpto ??
MARK IF( x < 5 ){
    x = 2 * y;
    y = y + x / 2;
} JUMP;
```

Listing 4.4: Exemplo While

4.6 Arrays

É possível declarar e manusear variáveis estruturada do tipo *array* de 1 e 2 dimensões. Como só é suportado inteiros para os arrays, as suas declarações são chamadas de *ARRAY*. Dentro do '[]', temos de indicar o tamanho daquela dimensão. Se quisermos aceder a um índice do array de 1 dimensão colocamos o nome da varável *ARRAY* seguido do índice desejado entre parênteses, para 2 dimensões de forma equivalente mas acendendo à linha e à coluna desejada.

```
ARRAY x[2];
x(0) = 12;

ARRAY y[5][2];
x(3)(1) = 42;
```

Listing 4.5: Exemplo de Arrays

4.7 Comentários

É importante, para uma linguagem de programação, possuir comentários, pois permite a explicação do código junto deste, tal como permite manter excertos de códigos dentro do ficheiro que não se pretendem compilar. Podemos fazer comentários através de um "??"no início e outro "??"no fim.

```
?? Exemplo de comentario de uma linha ??
```

Listing 4.6: Exemplo de Comentários

Capítulo 5

Codificação e Testes

O nosso trabalho está contido em dois ficheiros, PLC_TP2_Lexico.py(*Analisador Léxico*) e parserGerador(*Analisador Sintático* e *Analisador Semântico*).

A invocação do programa faz-se através de uma linha de comandos a partir do *python* e do ficheiro *yacc.py*, o output desse programa é a conversão da linguagem ISTINC para a usada na Virtual Machine.

NOTA: Devido à utilização do *print* para a escrita dos erros, estes irão ser também escritos.

De seguida, deve-se introduzir o output na virtual machine.

5.1 Lexer

O código do ficheiro **PLC_TP2_Lexico** encontra-se em A.1. Nele, foram definidos os *tokens* necessários.

```
1 def t_VAR(t):
2     r"[a-z]\w*"
3     return t
```

Listing 5.1: Definição do dicionário *reserved* e do terminal **ID**

Pela definição de **VAR**, pode-se notar que as variáveis da linguagem devem começar por uma letra minúscula e apenas depois podem ser seguidas por outro tipos de carateres habituais.

Por fim, também se definiu a regra para comentários, encapsulando qualquer letra entre ocorrências de *??*.

```
1 def t_COMMENT(t):
2     r'\?\?.*\?\?'
3     return t
```

Listing 5.2: Definição do *token* **COMMENT**

5.2 Parser

O ficheiro *yacc.py*, que contém o código do *Parser* e que gera o código assembly encontra-se em A.2.

5.2.1 Estrutura Geral

A nossa linguagem obriga a que as declarações sejam feitas no início de qualquer programa e só depois é que vem o corpo da função, ou seja, após as declarações o programa irá dar um erro se escrever qualquer tipo de dado para declarar variáveis *INT*, *BOOL*, *ARRAY* ou *STRING*.

```
1  def p_axioma(p):
2      'axioma : programa'
3      p[0] = p[1]
4      for x in p[0]:
5          if (x != None and x !=''):
6              print(x,end="")
7      if(parser.nPops > 0):
8          print(f"POP{parser.nPops}")
9          parser.nPops = 0
10
11 def p_programa(p):
12     'programa : declaracoes corpo'
13     p[0] = elementosLista([p[1]]) + elementosLista([p[2]])
```

Listing 5.3: Produções a partir do Axioma

Aqui dá-se uso a uma variável subjacente ao *parser* *parser.nPops*, que por cada elemento na stack no final de um programa esta dá o número de pops necessários para a limpeza da stack. Esta variável como podemos ver no código do parser é incrementada uma ou várias vezes quando se declaram variáveis.

Para além disso é usada a função *elementosLista* que transforma um array de arrays em um único array com todos os elementos das várias listas, isto porque é guardada a informação de declarações num array e se o programa tiver várias declarações os elementos serão guardados em vários arrays:

```
1  def elementosLista(p): #Dividir elementos de p[2]
2      s = []
3      for elemento in p:
4          if isinstance(elemento, list):
5              s.extend(elementosLista(elemento)) # Chama recursivamente a funcao para
       listas aninhadas
6          else:
7              s.append(elemento)
8      return s
```

Listing 5.4: Função auxiliar elementosLista

5.2.2 Declaração de Variáveis

As declarações nas nossas regras seguem três variações, uma que permite escrever várias declarações que por suas vez faz uso de uma outra regra *declaracao* que permite fazer várias declarações numa só invocação de tipo (*por exemplo: INT x , y , z;*), uma vazia que trava esse processo para não entrar em ciclo e uma para fazer comentários:

```
#----------------declaracoes-----------------------------------------#

def p_declaracoes_declaracoesDeclaracao(p):
    'declaracoes : declaracoes declaracao'
    p[0] = [p[1]] + [p[2]]
    p[0] = elementosLista(p[0])

def p_declaracoes_empty(p):
    'declaracoes : '

#--------------------------------------------------------------------#
#----------------declaracao------------------------------------------#

def p_declaracao_seqDecl(p):
    'declaracao : tipo seqDecl PV'
    parser.tipos = str(p[1])
    p[0] = str(p[2])

#--------------------------------------------------------------------#
#----------------SEQDECL---------------------------------------------#

def p_seqDecl_atrVIRGseqDecl(p):
    'seqDecl : atr VIRG seqDecl'
    p[0] = p[1] + str(p[3])
    parser.tipos = ""

def p_seqDecl_atr(p):
    'seqDecl : atr'
    p[0] = p[1]
```

Listing 5.5: Declarações no parser

Como visto no código total do parser, o tipo deriva nos vários tipos que temos em mão (*INT*,*BOOL*,*STRING*,*INT* e *ARRAY*).
A declaração de variáveis deriva em atribuições, estas têm duas possibilidades:

- São declaradas sem atribuição (uso de valores *DEFAULT**);
- São declaradas e atribuídas um valor;

*Valores *DEFAULT*:

- *0 para variáveis do tipo INT*
- *para variáveis do tipo STRING*
- *TRUE para variáveis do tipo BOOL*
- *[0,0,...,0] para variáveis do tipo ARRAY*

Ao declarar uma variável sem atribuição de valor usamos três variáveis subjacentes ao *parser* para controlo da geração de código *assembly*:

- *parser.tipos* que nos indica que tipo de variável está a ser declarada;
- *parser.posicao* que guarda a posição para onde tem que ser "empurrada"a próxima variável na stack
- *parser.nPops* que, como referido anteriormente, incrementa sempre que é feita uma declaração

Ao declarar uma variável também é feito tratamento de erros, saber se esta foi declarada com um *tipo* antes ou se esta já foi declarada. Para isso cada variável, quando declarada é armazenada num dicionário específico para cada tipo de dados onde a chave é o nome da variável e o seu valor a posição onde se encontra na stack para futuramente se puder usar o valor se neecssário:

- *parser.inteiros* para variáveis INT;
- *parser.strings* para variáveis STRING;
- *parser.bools* para variáveis BOOL;
- *parser.arrays* para variáveis ARRAY com 1 dimensão;
- *parser.matrizes* para variáveis ARRAY com 2 dimensões;

As variáveis ARRAY são guardadas de maneira diferente das outras, caso esta tenha 1 dimensão é guardado um tuplo da posição em que se encontra na stack e o seu tamanho, caso tenha 2 dimensões é guardado a posição na stack e os dois parâmetros de dimensão (número de linhas e colunas).

```
def p_atr_VAR(p):
    'atr : VAR'
    if (notInConjunto(p[1])):
        match parser.tipos:
            case "INT":
                parser.inteiros[p[1]] = parser.posicao
                p[0] = f"PUSHI0\n"
                parser.posicao += 1
                parser.nPops += 1
            case "STRING":
                parser.strings[p[1]] = parser.posicao
                p[0] = f"PUSHS \"\"\n"
                parser.posicao += 1
                parser.nPops += 1
            case "BOOL":#default TRUE
                parser.bools[p[1]] = parser.posicao
                p[0] = f"PUSHI1\n"
                parser.posicao += 1
                parser.nPops += 1
            case "" :
                print("ERRO: Nao atribuido o tipo")
    else:
        print(f"ERRO: A variavel {p[1]} ja foi declarada")
        exit()

def p_atr_VAROPATOperacao(p):
```

```
27      'atr : VAR OPAT operacao'
28      if(not notInConjunto(p[1])):
29          print(f"ERRO: A variavel {p[1]} ja foi declarada!")
30          exit()
31      if(parser.tipos != "STRING" and parser.tipos != "BOOL" and parser.tipos != "INT")
    :
32          print("ERRO: Esse tipo de dados nao existe.")
33          exit()
34      if (parser.tipos == "INT" and notInConjunto(p[1])):
35          parser.inteiros[p[1]] = parser.posicao #Associa o numero a variavel no
    dicionario
36      if (parser.tipos == "STRING" and notInConjunto(p[1])):
37          parser.strings[p[1]] = parser.posicao
38          p[0] = p[3]
39      if (parser.tipos == "BOOL" and notInConjunto(p[1])):
40          parser.bools[p[1]] = parser.posicao
41          p[0] = p[3]
42      p[0] = f"{p[3]}"
43      parser.posicao += 1
44      parser.nPops +=1
45
46  def p_atr_VARARRAY(p):
47      'atr : VAR PRE NUMINT PRD'
48      if (parser.tipos == 'ARRAY' and notInConjunto(p[1])):
49          tamanho = int(p[3])
50          parser.arrays[p[1]] = (parser.posicao,tamanho)
51          p[0] = 'PUSHN'+str(tamanho) +'\n'
52          parser.posicao += tamanho
53          parser.nPops += tamanho
54      elif (parser.tipos [0] != 'ARRAY'):
55          print(f"ERRO: A variavel {p[1]} nao e do tipo ARRAY.")
56          exit()
57      elif(not notInConjunto(p[1])):
58          print(f"ERRO: Variavel {p[1]} ja declarada anteriormente")
59          exit()
60
61  def p_atr_VARARRAYcomConteudo(p):
62      'atr : VAR PRE NUMINT PRD OPAT ABRIR_CH seqNumInt FECHAR_CH'
63      if(not notInConjunto(p[1])):
64          print(f"ERRO: Variavel {p[1]} ja foi declarada anteriormente")
65          exit()
66      if(parser.nArray == int(p[3])): #Conteudo com numero certo de elementos
67          if (parser.tipos == 'ARRAY' and notInConjunto(p[1])):
68              tamanho = int(p[3])
69              parser.arrays[p[1]] = (parser.posicao,tamanho)
70              p[0] = p[7]
71              parser.posicao += tamanho
72              parser.nPops += tamanho
73      else:
74          print(f"ERRO: A variavel {p[1]} tem limite {int(p[3])} e foram colocados {
    parser.nArray} elementos!")
75          exit()
76
77  def p_atr_matrizDefault(p):
```

```
78      'atr : VAR PRE NUMINT PRD PRE NUMINT PRD'
79      if (parser.tipos == 'ARRAY' and notInConjunto(p[1])):
80          tamanho = int(p[3]) * int(p[6])
81          #Guarda a posicao de onde comeca na stack e os parametros de tamanho
82          parser.matrizes[p[1]] = (parser.posicao,int(p[3]),int(p[6]))
83          p[0] = f"PUSHN{tamanho}\n"
84          parser.posicao += tamanho
85          parser.nPops += tamanho
86      elif (parser.tipos [0] != 'ARRAY'):
87          print(f"ERRO: A variavel {p[1]} nao e do tipo ARRAY.")
88          exit()
89      elif(not notInConjunto(p[1])):
90          print(f"ERRO: Variavel {p[1]} ja foi declarada anteriormente")
91          exit()
92
93  def p_atr_matrizComConteudo(p):
94      'atr : VAR PRE NUMINT PRD PRE NUMINT PRD OPAT ABRIR_CH seqNumInt FECHAR_CH'
95      if(not notInConjunto(p[1])):
96          print(f"ERRO: Variavel {p[1]} ja foi declarada anteriormente")
97          exit()
98      if(parser.nArray == int(p[3])): #Conteudo com numero certo de elementos
99          if (parser.tipos == 'ARRAY' and notInConjunto(p[1])):
100             tamanho = int(p[3]) * int(p[6])
101             parser.matrizes[p[1]] = (parser.posicao,int(p[3]),int(p[6]))
102             p[0] = p[10]
103             parser.posicao += tamanho
104             parser.nPops += tamanho
105     else:
106         print(f"ERRO: A variavel {p[1]} tem limite {int(p[3])} e foram colocados {
    parser.nArray} elementos!")
107         exit()
```

Listing 5.6: Declaração de variáveis sem valores atribuídos

São usadas as variáveis auxiliares do *parser*, *parser.tipos* que guarda o tipo que está a ser declarado para tratamento de erros para quando uma variável não coincide com o tipo atirbuído e também a variável *parser.nArray*, usado para quando é atribuído uma sequência de inteiros maior que o tamanho declarado do array. Para além disso também note-se que para verificar se uma variável não está já declarada usamos uma função auxiliar *notInConjunto*:

```
1       def notInConjunto(p):
2       res = True
3       if (p in parser.inteiros): res = False
4       elif(p in parser.strings): res = False
5       elif(p in parser.bools): res = False
6       elif(p in parser.arrays): res = False
7       elif(p in parser.matrizes): res = False
8       return res
```

Listing 5.7: Função auxiliar notInConjunto

Como anteriormente é primeiramente verificado se a variável passada já foi declarada e se o tipo a ser declarado é o correto e só depois dessa verificação é que é empurrado na stack o valor pretendido na posição correta e guardada a posição na stack no devido dicionário do parser.
Note-se que é possível fazer uma operação inteira para atribuir um valor inteiro a uma variável, a regra *operacao* será analisada mais à frente. Como referido anteriormente as variáveis de tipo *ARRAY* são guardadas no dicionário de forma diferente.
Para declarar conteúdo de um array a regra é criar uma sequência de números inteiros usando parênteses retos e colocando-os por ordem os números equivalente à linguagem **C**.

```
def p_seqNumInt_NUMINT(p):
    'seqNumInt : NUMINT'
    parser.nArray += 1
    p[0] = f"PUSHI{p[1]}\n"

def p_seqNumInt_NumMaisNum(p):
    'seqNumInt : NUMINT VIRG seqNumInt'
    parser.nArray += 1
    p[0] = f"PUSHI{p[1]}\n" + p[3]
```

Listing 5.8: Implementação da regra seqNumInt

Operacao

A regra *operacao* é usada para quando é necessário, como o nome sugere, fazer uma operação, inteira, booleana ou para ler *input*, esta pode usar tanto número inteiros quanto variáveis inteiras nas várias operações. Na nossa linguagem *ISTINC* estas são feitas apenas nas atribuições de valor a variáveis, seja nas declarações seja no corpo da função para alterar o valor da variável. A função desta regra na nossa GIC é apenas de:

- **Verificar**: se é usada alguma variável na operação e ver se esta foi declarada;
- **Empurrar**: os devidos números para a stack e fazer as operações na ordem correta

Na nossa GIC a *operacao* é derivada várias vezes de maneira a evitar conflitos *shift-reduce* e *reduce-reduce* implementando:

```
#------------------------------------------------------------#
#-----------------OPERACAO-----------------------------------#

def p_operacao_termo(p):
    'operacao : termo'
    p[0] = p[1]

def p_operacao_adicao(p):
    'operacao : operacao OPAD termo'
    p[0] = p[3] + p[1] + "ADD\n"

def p_operacao_subtracao(p):
    'operacao : operacao OPSUB termo'
    p[0] = p[3] + p[1] + "SUB\n"

#------------------------------------------------------------#
#-----------------TERMO--------------------------------------#

```

```
19 def p_termo_fator(p):
20     'termo : fator'
21     p[0] = p[1]
22 
23 def p_termo_mult(p):
24     'termo : termo OPMUL fator'
25     p[0] = p[1] + p[3] + 'MUL\n'
26 
27 def p_termo_DIV(p):
28     'termo : termo OPDIV fator'
29     p[0] = p[1] + p[3] + 'DIV\n'
30 
31 #------------------------------------------------------------------#
32 #-----------------FATOR--------------------------------------------#
33 
34 def p_fator_NUMINT(p):
35     'fator : NUMINT'
36     p[0] = f"PUSHI{p[1]}\n"
37 
38 def p_fator_VAR(p):
39     'fator : VAR'
40     if (p[1] in parser.inteiros):
41         p[0]= f"PUSHG {parser.inteiros[p[1]]}\n"
42     elif (p[1] in parser.bools):
43         p[0]= f"PUSHG {parser.bools[p[1]]}\n"
44     elif (p[1] in parser.strings):
45         p[0]= f" PUSHG {parser.strings[p[1]]}\n"
46     elif (p[1] in parser.arrays): #GUARDAR A QUANTIDADE DE WRITES A FAZER NO PRINT
47         i = 0
48         a = parser.arrays[p[1]][i]
49         s = ""
50         while(i<parser.arrays[p[1]][1]):
51             s += f"PUSHG {a}\n"
52             i+=1
53             a+=1
54         p[0] = s
55     elif(p[1] in parser.matrizes):#GUARDAR A QUANTIDADE DE WRITES A FAZER NO PRINT
56         i1 = 0
57         i2 = 1
58         #O tamanho e dado pelo nmr_linhas * nmr_colunas
59         tamanho = parser.matrizes[p[1]][1] * parser.matrizes[p[1]][2]
60         # a = posicao do array na stack
61         a = parser.matrizes[p[1]][0]
62         s = ""
63         while(i1<tamanho):
64             s += f"PUSHG {a}\n"
65             a += 1
66             i1 += 1
67             i2 += 1
68         p[0] = s
69 
70 def p_fator_VARARRAY(p):
71     'fator : VAR PCE NUMINT PCD'
72     p[0] = f"PUSHG{parser.arrays[p[1]][0]+p[3]}\n"
```

```
73
74 def p_fator_VARMATRIZ(p):
75     'fator : VAR PCE NUMINT PCD PCE NUMINT PCD'
76     pm = parser.matrizes[p[1]][0] + (parser.matrizes[p[1]][2] * p[3]) + p[6]
77     p[0] = f"PUSHG{pm}\n"
78
79 def p_fator_bool(p):
80     'fator : bool'
81     p[0] = p[1]
82
83 def p_fator_PAL(p):
84     'fator : PAL'
85     p[0] = f" PUSHS {p[1]}\n"
86
87 def p_fator_NOTcondicao(p):
88     'fator : NOT PCE condicao PCD'
89     p[0] = p[3] + "NOT\n"
90
91 def p_fator_lerInput(p):
92     'fator : LER PCE PCD'
93     p[0] = "READ\nATOI\n"
```

Listing 5.9: Implementação da Regra operacao

Mais uma vez é usada a função auxiliar notInConjunto para verificar se uma variável já foi ou não declarada.

5.2.3 Corpo do programa

Feitas as declarações o programa transita para uma fase onde se desenvolve o corpo do programa, nesta secção a linguagem *ISTINC* permite alterar valores de variáveis com as operações apresentadas, criar ciclos com as condições criadas, operações de controlo de fluxo e imprimir qualquer valor ou variável (exceto do tipo ARRAY, apenas dá para imprimir indíces).

```
1  #------------------------------------------------------------------------#
2  #----------------corpo-----------------------------------------------------#
3
4  def p_corpo_instrucao(p):
5      'corpo : corpo instrucao'
6      p[0] = [p[1]] + [p[2]]
7      p[0] = elementosLista(p[0])
8      s = ""
9      for elemento in p[0]:
10         if elemento != None:
11             s+=elemento
12     p[0] = s
13
14 def p_corpo_VAZIO(p):
15     'corpo : '
16     p[0] = ""
17
18 #------------------------------------------------------------------------#
19 #----------------instrucao-------------------------------------------------#
20
```

```
21 def p_instrucao_ifelsestatement(p):
22     'instrucao : ifelsestatement'
23     p[0] =  p[1]
24 
25 def p_instrucao_print(p):
26     'instrucao : print PV'
27     p[0] = p[1]
28 
29 def p_instrucao_VARopatOPERACAO(p):
30     'instrucao : VAR OPAT operacao PV'
31     if(not notInConjunto(p[1])):
32         if(p[1] in parser.inteiros): #Variavel declarada mas nao inteira
33             p[0] = p[3] + f"STOREG{parser.inteiros[p[1]]}\n"
34         elif(p[1] in parser.bools):
35             p[0] = p[3] + f"STOREG{parser.bools[p[1]]}\n"
36         elif(p[1] in parser.strings):
37             p[0] = p[3] + f"STOREG{parser.strings[p[1]]}\n"
38 
39 def p_instrucao_mudarindexarray(p):
40     'instrucao : VAR PCE NUMINT PCD OPAT operacao PV'
41     if(not notInConjunto(p[1])):
42         if(p[1] not in parser.arrays): #Variavel declarada mas nao do tipo ARRAY
43             print(f"ERRO: A variavel {p[1]} nao e do tipo ARRAY")
44             exit()
45         else:#Se for variavel declarada do tipo certo
46             p[0] = p[6] + f"STOREG{parser.arrays[p[1]][0]+p[3]}\n"
47     else:
48         print(f"ERRO: A variavel {p[1]} nao foi declarada")
49 
50 def p_instrucao_mudarindexMatriz(p):
51     'instrucao : VAR PCE NUMINT PCD PCE NUMINT PCD OPAT operacao PV '
52     if(not notInConjunto(p[1])):
53         if(p[1] not in parser.matrizes): #Variavel declarada mas nao do tipo ARRAY
54             print(f"ERRO: A variavel {p[1]} nao e do tipo ARRAY")
55             exit()
56         else:#Se for variavel declarada do tipo certo
57             #Se quiser mudar o valores de x[z1][z2]  da matriz x[y1][y2] y1 linhas e
    y2 colunas
58             # tenho que aceder na stack a posicao:
59             # posicao da matriz na stack + (y2 * z1) + z2
60             pm = parser.matrizes[p[1]][0] + (parser.matrizes[p[1]][2] * p[3]) + p[6]
61             p[0] = p[9] + f"STOREG{pm}\n"
62     else:
63         print(f"ERRO: A variavel {p[1]} nao foi declarada")
64 
65 def p_instrucao_tojumpPV(p):
66     'instrucao : tojump PV'
67     p[0] = p[1]
68 
69 def p_instrucao_jumptoPV(p):
70     'instrucao : jumpto PV'
71     p[0] = p[1]
```

Listing 5.10: Derivação das várias regras do corpo do programa

Note-se que o corpo deriva em uma quantidade finita de instruções, na regra *corpo* são eliminadas possíveis ocorrências vazias das instruções e com a função auxiliar *elementosLista* para torná-la num array de elementos singulares (ou seja, não um array de arrays).

Como é de notar o corpo deriva em uma série de instruções que fazem uso de outras regras:

- *ifelsestatement*
- *print*
- *tojump*
- *jumpto*
- *operacaoInt*
- *comment*

Que iremos ver com mais detalhe à parte das regras *operacaoInt* e *comment* que já foram revistas

IfElse

A lógica da estrutura de controlo de fluxo IfElse provém da manipulação de saltos e de *labels* no pseudo-código máquina. Como estratégia nos saltos do programa na linguagem da EWVM começamos por avaliar a condição, se esta for falsa usamos o comando *JZ label* para saltar imediatamente para o corpo do *else*, após o *JZ label* colocamos o corpo do *if* já que se for verdadeira a condição não vai saltar para o corpo do *else* e assim execute o corpo do *if*. Após o corpo do *if*, como não queremos executar o corpo do *else* saltamos para uma label para lá do corpo do *else* para não a executarmos. Nesta regra também é guardada a posição dos controlos de fluxo na stack com uma variável do parser *parser.posicaoIf*, isto para, se for da vontade do programador, poder fazer vários controlos de fluxo ao longo do seu programa.
Como referido anteriormente, a linguagem ISTINC obriga a que seja acrescentado sempre em conjunto com uma condição *if* uma condição *else*, caso não se deseje fazer nada na condição *else* é possível apenas abrir e fechar chavetas sem qualquer conteúdo no meio.

```
1 def p_ifselsetatement(p):
2     'ifelsestatement : IF PCE condicao PCD ABRIR_CH corpo FECHAR_CH ELSE ABRIR_CH
    corpo FECHAR_CH'
3     if_ = parser.posicaoIf
4     p[0] = f"{p[3]}\nJZ elselabel{if_}\n{p[6]}JUMP iflabel{if_}\nelselabel{if_}:\n{p
    [10]}iflabel{if_}:\n"
5     parser.posicaoIf += 1
```

Listing 5.11: Produções subjacentes a e *IfElse*

Ciclos dowhile e whiledo

De formas semelhante às instruções de controlo de fluxo, no pseudo-código da EVWM para o *jumpto* (equivalente ao *whiledo*) criamos uma label com a posição do *parser.posicaoCiclo* testamos a condição e se esta falhar saltamos para uma label *endwhile* para não executar o corpo do while senão executamo-lo e saltamos para a label que criamos no início do ciclo.
De maneira semelhante a regra *tojump* utiliza a mesma estratégia de pseudo-código mas primeiro executa uma iteração do corpo do while.

```
#-------------------------------------------------------------------------#
#-----------------tojump--------------------------------------------------#
#dowhile
def p_tojump_MARKABRIRCHcorpoFECHARCHif(p):
    'tojump : MARK ABRIR_CH corpo FECHAR_CH IF PCE condicao PCD JUMP'
    cic = parser.posicaoCiclo
    p[0] = p[3]+f"labelwhile{cic}:\n"+p[7]+f"JZ endwhile{cic}\n"+p[3]+f"JUMP
    labelwhile{cic}\n"+f"endwhile{cic}:\n"
    parser.posicaoCiclo += 1

#-------------------------------------------------------------------------#
#-----------------jumpto--------------------------------------------------#
#whiledo
def p_jumpto_MARKifCondcorpoJump(p):
    'jumpto : MARK IF PCE condicao PCD ABRIR_CH corpo FECHAR_CH JUMP'
    cic = parser.posicaoCiclo
    p[0] = f"labelwhile{cic}:\n"+p[4]+f"JZ endwhile{cic}\n"+p[7]+f"JUMP labelwhile{
    cic}\n"+f"endwhile{cic}:\n"
    parser.posicaoCiclo += 1
```

Listing 5.12: Produções correspondentes ao *while*

Condições

Foi mencionado nas nossas operações de IfElse e de ciclos a regra *condicao*, esta pode ser uma sequência de condições interligadas de um operador lógico, podem ser negadas e podem ser usadas variáveis do tipo *BOOL*, já que a regra de *operacao* consegue derivar em variáveis booleanas:

```
def p_condicao_operacao(p):
    'condicao : operacao'
    p[0] = p[1]

def p_condicao_igualigual(p):
    'condicao : operacao IGUAL operacao'
    p[0] = p[1] + p[3] + "EQUAL\n"

def p_condicao_DIFF(p):
    'condicao : operacao DIFF operacao'
    p[0] = p[1] + p[3] + "EQUAL\n" + "NOT\n"

def p_condicao_MENOR(p):
    'condicao : operacao LESS operacao'
    p[0] = p[1] + p[3] + "INF\n"

def p_condicao_MAIOR(p):
```

```
18      'condicao : operacao GREATER operacao'
19      p[0] = p[1] + p[3] + "SUP\n"
20
21  def p_condicao_MENORouIGUAL(p):
22      'condicao : operacao LEQ operacao'
23      p[0] = p[1] + p[3] + "INFEQ\n"
24
25  def p_condicao_MAIORouIGUAL(p):
26      'condicao : operacao GEQ operacao'
27      p[0] = p[1] + p[3] + "SUPEQ\n"
28
29  def p_condicao_OU(p):
30      'condicao : PCE condicao PCD OR PCE condicao PCD'
31      p[0] = p[2] + p[6] + "OR\n"
32
33  def p_condicao_AND(p):
34      'condicao : PCE condicao PCD AND PCE condicao PCD'
35      p[0] = p[2] + p[6] + "AND\n"
```

Listing 5.13: Produções da regra *condicao*

As produções *oplog* e *opcond* são deriváveis respetivamente nos operadores lógicos e nos operadores condicionais:

- Conjunção(AND) e disjunção(OR);
- Igualdade($==$), diferença($!=$);
- menor($<$), maior($>$), maior ou igual ($\geq$)$oumenorouigual$($\leq$).

Prints

Como dito anteriormente, a linguagem *ISTINC* tem a capacidade de imprimir variáveis e valores do tipo *INT*, *STRING* e *BOOL*, para além disso os índices dos arrays e das matrizes, operações e ainda condições. Como na nossa gramática as condições derivam em operações, as operações em termos e termos em fatores é possível escrever no parser a regra apenas da seguinte forma:

```
1  def p_print(p):
2      'print : PRINT PCE condicao PCD'
3      padrao_string = r'\sPUSH[SG].*'
4      if (re.match(padrao_string,p[3])):
5          p[0] = f"{p[3]}WRITES\n"
6      else:
7          p[0] = f"{p[3]}WRITEI\n"
```

Listing 5.14: Produções da regra print

É de notar duas coisas

- que, ao contrário do que queríamos, o print de uma variável booleana não retorne os tipos *TRUE* e *FALSE* definidos mas retorna respetivamente *1* e *0*.
- e que, para diferenciar um print de uma variável ou valor inteiro de uma variável ou valor string colocámos quando a código da máquina uma instrução *PUSHS* com um espaço () antes da instrução para diferenciar um *PUSHG* de uma variável inteira e uma string.

5.3 Alternativas, Decisões e Problemas de Implementação

A nossa solução foi concebida permitindo uma rápida, facilitada e flexível expansão futura por parte do utilizador. Decidimos que não existe nenhum tipo de *RETURN* que interrompa o programa e que retorne um valor, uma alternativa seria fazer um *IfElse* com um *Else* vazio para substituir a instrução *RETURN* equivalente à linguagem C. O nosso parser *yacc* lê um ficheiros com a linguagem *ISTINC* e dá o resultado em código EVWM no *stdout*. Note-se que para ler do stdin precisamos apenas de no código do parser (no final) alterar o processo de leitura de ficheiro por um de leitura do *stdin*.

5.4 Testes realizados e Resultados

5.4.1 *Swap*

```
1 INT x = 10,y = 5,temp;
2
3 temp = y;
4 y = x;
5 x = temp;
6
7 PRINT("x = "); PRINT(x);
8 PRINT("\n");
9 PRINT("y = "); PRINT(y);
```

Listing 5.15: Teste de *Swap*

Este teste demonstra, não só a atribuição de valores *DEFAULT* na variável temp mas de valores inteiros dados, assim como a declaração de três variáveis seguidas e também a sua utilização e alteração ao longo do corpo do programa. Além disso mostra como é implementado o *PRINT*.

```
1  PUSHI10
2  PUSHI5
3  PUSHI0
4  PUSHG 1
5  STOREG2
6  PUSHG 0
7  STOREG1
8  PUSHG 2
9  STOREG0
10   PUSHS "x = "
11 WRITES
12 PUSHG 0
13 WRITEI
14   PUSHS "\n"
15 WRITES
16   PUSHS "y = "
17 WRITES
18 PUSHG 1
19 WRITEI
20 POP3
```

Listing 5.16: Código do teste anterior.

5.4.2 *Uso de IF e ELSE*

```
BOOL b = TRUE;
INT x = 10;

IF (b){
    x = 5;
    b = FALSE;
}
ELSE{
    x = 20;
}
PRINT("x = "); PRINT(x);
PRINT("\n");
PRINT("b = "); PRINT(b);
```

Listing 5.17: Exemplo de utilização de *IfElse*

Este é um teste do *IfElse* em que é mostrado que b pode conter valores booleanos e pode ser usado em condições assim como pode ser alterado ao longo do programa. Além disso, como referido anteriormente, o valor escrito de b é 0 em vez de *FALSE*.

```
PUSHI1
PUSHI10
PUSHG 0

JZ elselabel0
PUSHI5
STOREG1
PUSHI0
STOREG0
JUMP iflabel0
elselabel0:
PUSHI20
STOREG1
iflabel0:
 PUSHS "x = "
WRITES
PUSHG 1
WRITEI
 PUSHS "\n"
WRITES
 PUSHS "b = "
WRITES
PUSHG 0
WRITEI
POP2
```

Listing 5.18: Código do teste *IfElse*

5.4.3 Declaração de um *array* multi-dimencional

Neste teste é feita uma declaração *DEFAULT* e um acesso a um array com duas dimensões. Esta matriz em específico é a representação de um grafo por uma matriz de adjacências, note-se que podemos alterar o valor das posições da matriz, podendo assim representar as arestas:

- 0 -> 1
- 1 -> 2
- 3 -> 4
- 4 -> 0
- 0 -> 2

e após isso é feito também o print na máquina EWVM da matriz.

```
ARRAY matrizAdj[5][5];

matrizAdj(0)(1) = 1;
matrizAdj(1)(2) = 1;
matrizAdj(3)(4) = 1;
matrizAdj(4)(0) = 1;
matrizAdj(0)(2) = 1;

??Print da primeira linha??
PRINT(matrizAdj(0)(0)); PRINT (" ");
PRINT(matrizAdj(0)(1)); PRINT (" ");
PRINT(matrizAdj(0)(2)); PRINT (" ");
PRINT(matrizAdj(0)(3)); PRINT (" ");
PRINT(matrizAdj(0)(4)); PRINT (" ");
PRINT("\n");
??Print da segunda linha??
PRINT(matrizAdj(1)(0)); PRINT (" ");
PRINT(matrizAdj(1)(1)); PRINT (" ");
PRINT(matrizAdj(1)(2)); PRINT (" ");
PRINT(matrizAdj(1)(3)); PRINT (" ");
PRINT(matrizAdj(1)(4)); PRINT (" ");
PRINT("\n");
??Print da terceira linha??
PRINT(matrizAdj(2)(0)); PRINT (" ");
PRINT(matrizAdj(2)(1)); PRINT (" ");
PRINT(matrizAdj(2)(2)); PRINT (" ");
PRINT(matrizAdj(2)(3)); PRINT (" ");
PRINT(matrizAdj(2)(4)); PRINT (" ");
PRINT("\n");
??Print da quarta linha??
PRINT(matrizAdj(3)(0)); PRINT (" ");
PRINT(matrizAdj(3)(1)); PRINT (" ");
PRINT(matrizAdj(3)(2)); PRINT (" ");
PRINT(matrizAdj(3)(3)); PRINT (" ");
PRINT(matrizAdj(3)(4)); PRINT (" ");
PRINT("\n");
??Print da quinta linha??
```

```
38 PRINT(matrizAdj(4)(0)); PRINT (" ");
39 PRINT(matrizAdj(4)(1)); PRINT (" ");
40 PRINT(matrizAdj(4)(2)); PRINT (" ");
41 PRINT(matrizAdj(4)(3)); PRINT (" ");
42 PRINT(matrizAdj(4)(4)); PRINT (" ");
43 PRINT("\n");
```

Listing 5.19: Teste While e Array

Note-se que noutras linguagens seria possível fazer um ciclo que imprimi-se cada elemento da matriz, porém na linguagem ISTINC não é possível aceder à posição de um array através de variáveis. Dá como resultado o pseudo-código:

```
1  PUSHN25
2  PUSHI1
3  STOREG1
4  PUSHI1
5  STOREG7
6  PUSHI1
7  STOREG19
8  PUSHI1
9  STOREG20
10 PUSHI1
11 STOREG2
12 PUSHG0
13 WRITEI
14   PUSHS " "
15 WRITES
16 PUSHG1
17 WRITEI
18   PUSHS " "
19 WRITES
20 PUSHG2
21 WRITEI
22   PUSHS " "
23 WRITES
24 PUSHG3
25 WRITEI
26   PUSHS " "
27 WRITES
28 PUSHG4
29 WRITEI
30   PUSHS " "
31 WRITES
32   PUSHS "\n"
33 WRITES
34 PUSHG5
35 WRITEI
36   PUSHS " "
37 WRITES
38 PUSHG6
39 WRITEI
40   PUSHS " "
41 WRITES
42 PUSHG7
```

```
43 WRITEI
44   PUSHS "   "
45 WRITES
46 PUSHG8
47 WRITEI
48   PUSHS "   "
49 WRITES
50 PUSHG9
51 WRITEI
52   PUSHS "   "
53 WRITES
54   PUSHS "\n"
55 WRITES
56 PUSHG10
57 WRITEI
58   PUSHS "   "
59 WRITES
60 PUSHG11
61 WRITEI
62   PUSHS "   "
63 WRITES
64 PUSHG12
65 WRITEI
66   PUSHS "   "
67 WRITES
68 PUSHG13
69 WRITEI
70   PUSHS "   "
71 WRITES
72 PUSHG14
73 WRITEI
74   PUSHS "   "
75 WRITES
76   PUSHS "\n"
77 WRITES
78 PUSHG15
79 WRITEI
80   PUSHS "   "
81 WRITES
82 PUSHG16
83 WRITEI
84   PUSHS "   "
85 WRITES
86 PUSHG17
87 WRITEI
88   PUSHS "   "
89 WRITES
90 PUSHG18
91 WRITEI
92   PUSHS "   "
93 WRITES
94 PUSHG19
95 WRITEI
96   PUSHS "   "
```

```
97 WRITES
98  PUSHS "\n"
99 WRITES
100 PUSHG20
101 WRITEI
102  PUSHS " "
103 WRITES
104 PUSHG21
105 WRITEI
106  PUSHS " "
107 WRITES
108 PUSHG22
109 WRITEI
110  PUSHS " "
111 WRITES
112 PUSHG23
113 WRITEI
114  PUSHS " "
115 WRITES
116 PUSHG24
117 WRITEI
118  PUSHS " "
119 WRITES
120  PUSHS "\n"
121 WRITES
122 POP25
```

Listing 5.20: Teste de matriz

5.4.4 Exemplo de um *array* e um *while*

Este é um exemplo simples de um ciclo *while* implementado com a regra JUMPTO que incrementa uma variável i até 4 e mudar os índices do array para esse mesmo valor.

```
1  INT i;
2  ARRAY ar[4];
3  PRINT("Array antes das insercoes: \n");
4  PRINT(ar(0));PRINT(ar(1));PRINT(ar(2));PRINT(ar(3));
5  PRINT("\n");
6  MARK IF (i<4) {
7      i = i + 1;
8  } JUMP;
9  ar(0) = i;
10 ar(1) = i;
11 ar(2) = i;
12 ar(3) = i;
13 PRINT("Array depois das insercoes: \n");
14 PRINT(ar(0));PRINT(ar(1));PRINT(ar(2));PRINT(ar(3));
```

Listing 5.21: Teste While

```
1 PUSHI0
2 PUSHN4
3  PUSHS "Array antes das insercoes: \n"
```

```
4  WRITES
5  PUSHG1
6  WRITEI
7  PUSHG2
8  WRITEI
9  PUSHG3
10 WRITEI
11 PUSHG4
12 WRITEI
13  PUSHS "\n"
14 WRITES
15 labelwhile0:
16 PUSHG 0
17 PUSHI4
18 INF
19 JZ endwhile0
20 PUSHI1
21 PUSHG 0
22 ADD
23 STOREG0
24 JUMP labelwhile0
25 endwhile0:
26 PUSHG 0
27 STOREG1
28 PUSHG 0
29 STOREG2
30 PUSHG 0
31 STOREG3
32 PUSHG 0
33 STOREG4
34  PUSHS "Array depois das insercoes: \n"
35 WRITES
36 PUSHG1
37 WRITEI
38 PUSHG2
39 WRITEI
40 PUSHG3
41 WRITEI
42 PUSHG4
43 WRITEI
44 POP5
```

Listing 5.22: pseudo-código máquina do While e Array

5.4.5 Exemplo da utilização de variáveis *STRING* e várias condições interligadas

Neste exemplo são executados dois comandos, um para demonstrar que o *PRINT* consegue imprimir variáveis declaradas do tipo *STRING* e um que mostra que é possível interligar várias condições com os operadores lógicos definidos no parser.

```
INT x = 0;
STRING s = "Ola Mundo!";
IF((x==0) AND ((x<1) OR (x>1))){
    PRINT(s);
}
ELSE{}
```

Listing 5.23: Teste de variável STRING e várias condições interligadas

```
PUSHI0
 PUSHS "Ola Mundo!"
PUSHG 0
PUSHI0
EQUAL
PUSHG 0
PUSHI1
INF
PUSHG 0
PUSHI1
SUP
OR
AND

JZ elselabel0
 PUSHG 1
WRITES
JUMP iflabel0
elselabel0:
iflabel0:
POP2
```

Listing 5.24: pseudo-código máquina do teste anterior

5.4.6 Exemplo de Aninhamento de *IfElse* e leitura de *Input*

```
INT numero = LER();
IF (numero >= 0) {
    IF(numero == 0) {
        PRINT("O numero inserido e zero.\n");
    }ELSE{
        PRINT("O numero inserido e positivo.\n");
    }
}ELSE{
    PRINT("O numero inserido e negativo.\n");
}
```

Listing 5.25: Teste de aninhamento de condicionais e leitura de *Input*

Este teste tem em consideração a declaração de variáveis com o mesmo identificador em âmbitos diferentes e as considerações que se tem de ter para que não haja corrupção da *stack*.

```
READ
ATOI
PUSHG 0
PUSHI0
SUPEQ

JZ elselabel1
PUSHG 0
PUSHI0
EQUAL

JZ elselabel0
 PUSHS "O numero inserido e zero.\n"
WRITES
JUMP iflabel0
elselabel0:
 PUSHS "O numero inserido e positivo.\n"
WRITES
iflabel0:
JUMP iflabel1
elselabel1:
 PUSHS "O numero inserido e negativo.\n"
WRITES
iflabel1:
POP1
```

Listing 5.26: Pseudo-código máquina do teste do aninhamento e leitura de *Input*

5.4.7 Exemplos de erros

Erro de variáveis já declaradas

A nossa linguagem, através de dicionários consegue armazenar todas as variáveis declaradas e as suas posições na stack. Sendo assim, caso haja uma tentativa de declarar alguma variável já declarada o nosso programa retorna uma mensagem de erro:

```
INT x = 3;
INT y = 2;
INT x;

x = x + y;

PRINT(x);
```

Listing 5.27: Erro de declaração de variáveis

Dá a mensagem de erro:

```
    ERRO: A variavel x ja foi declarada
```

Listing 5.28: Mensagem de erro produzida a partir do bloco anterior

Erro de variáveis não correspondentes ao seu tipo

Para além disso a linguagem, sempre que encontrar um tipo não coincidente aquele que lhe está a ser atribuído este retorna uma mensagem de erro, seja na fase inicial de declarações seja no corpo:

```
INT x = 3;
INT y = "ABC";

y = 5;

PRINT(y);
```

Listing 5.29: Erro de náo correspodência de tipo e valor na declaração

Retorna a mensagem de erro:

```
    ERRO: A variavel nao e do tipo STRING.
```

Listing 5.30: Mensagem de erro produzida a partir do bloco anterior

E o mesmo para quando é atribuído um valor não correspondente ao seu declarado na fase do corpo do programa:

```
INT x;
STRING y;

y = 1;
```

Retorna a mensagem de erro:

```
    ERRO: A Variavel nao e do tipo STRING.
```

Erro de atribuição de um conjunto maior que a capacidade do array

Quando um array é declarado com capacidade C e é lhe atribuído uma sequência de valores com maior número que C obtemos um erro:

```
ARRAY x[4] = {1,2,3,4,5};
```

Retorna a mensagem de erro:

```
    ERRO: A variavel x tem limite 4 e foram colocados 5 elementos!
```

Note-se que as atribuições de array não podem ser feitas no corpo da função, por isso o erro que retorna se houver uma tentativa de atribuição de um conjunto de inteiros a um array no corpo da função será retornado um erro de sintaxe.

5.5 Gramática da Linguagem

```
axioma : programa

programa : declaracoes corpo

declaracoes      : declaracoes declaracao
                 |

declaracao       : tipo seqDecl ';'
                 |

seqDecl          : atr ',' seqDecl
                 | atr

atr              : VAR
                 | VAR '[' NUMINT ']'
                 | VAR '[' NUMINT ']' '=' '{' seqNumInt '}'
                 | VAR '[' NUMINT ']' '[' NUMINT ']'
                 | VAR '[' NUMINT ']' '[' NUMINT ']' '=' '{' seqNumInt '}'
                 | VAR '=' operacaoInt
                 | VAR '=' PAL
                 | VAR '=' bool

seqNumInt : NUMINT ',' seqNumInt
          | NUMINT

corpo : corpo instrucao

instrucao    : ifelsestatement
             | print ';'
             | VAR '=' operacao ';'
             | VAR '(' NUMINT ')' '=' operacao';'
             | VAR '(' NUMINT ')' '(' NUMINT ')' '=' operacao ';'
             | tojump ';'
             | jumpto ';'

operacao : termo
             | operacao '+' termo
             | operacao '-' termo

termo  :  fator
           | termo  '*'  fator
           | termo '/'   fator

fator  :  NUMINT
             | VAR
             | VAR '(' NUMINT ')'
             | VAR '(' NUMINT ')' '(' NUMINT ')'
             | bool
             | PAL
             | NOT '(' condicao ')'
             | LER '(' ')'

```

```
53 ifelsestatement : IF '(' condicao ')' '{' corpo '}' ELSE '{' corpo '}'
54
55 condicao     : operacao
56            | operacao '==' operacao
57            | operacao '!=' operacao
58            | operacao '<=' operacao
59            | operacao '>=' operacao
60            | operacao '<' operacao
61            | operacao '>' operacao
62            | '(' condicao ')' 'AND' '(' condicao ')'
63            | '(' condicao ')' 'OR' '(' condicao ')'
64
65 print :   PRINT '(' condicao ')'
66
67 tojump: MARK '{' corpo '}' IF '(' condicao ')' JUMP
68
69 jumpto: MARK IF '(' condicao ')' '{' corpo '}' JUMP
70
71 tipo     : INT | STRING | BOOL | ARRAY
72
73 bool     : TRUE | FALSE
```

Listing 5.31: Gramática

Capítulo 6

Conclusão

Este trabalho aborda a criação de uma linguagem de programação bem como a implementação de um compilador da mesma, através da construção da sua Gramática, de um Analisador Léxico e de um Sintático.

Neste relatório explicamos as decisões de design feitas ao longo do projeto bem como detalhes da implementação das mesmas.

Em retrospetiva, obtivemos uma linguagem elegante e liberal que implementa, na nossa opinião, bons aspetos de que se pede de uma linguagem de programação.

Consideramos que a linguagem é, no entanto, algo simplista e propomos como trabalho futuro os seguintes aspetos:

- Permitir criar variáveis do tipo *FLOAT* para além das inteiras, isso seria possível adicionando mais um tipo de dados e fazer um tratamento equivalente às variáveis do tipo inteiro.
- Permitir indexar um array ou matriz através de variáveis inteiras e não apenas números inteiros, isso seria possível modificando a técnica de armazenamento de variáveis na stack na VM *EWVM* de maneira a que conseguíssemos extrair o valor da variável inteira na stack;
- Dar print de arrays e matrizes;
- Dar print de variáveis booleanas de valores *TRUE*, *FALSE* em vez de *1* e *0* respetivamente.

Para terminar gostávamos de mencionar que este trabalho foi criado de maneira complexa, bem estruturada e, na nossa opinião, bem manuseada em termos de tempo, recursos e conhecimentos. Após este trabalho conseguimos afirmar com certeza que percebemos minimamente a implementação física de um compilador.

Apêndice A

Código do Programa

A.1 Lex

```
import ply.lex as lex
import sys

tokens = (
    'INT',       # Declarar inteiro
    'STRING',    # Declarar string
    'PAL',       # Palavra string
    'ARRAY',     # Declarar array
    'VAR',       # Vari vel que cont m o valor
    'NUMINT',    # N mero inteiro
    'VIRG',      # ,
    'PCE',       # (
    'PCD',       # )
    'PRE',       # [
    'PRD',       # ]
    'ABRIR_CH', # {
    'FECHAR_CH',# }
    'PV',        # ;
    'OPAT',      # =
    'OPAD',      # +
    'OPSUB',     # -
    'OPDIV',     # /
    'OPMUL',     # *
    'IGUAL',     # ==
    'DIFF',      # !=
    'LEQ',       # <=
    'LESS',      # <
    'GEQ',       # >=
    'GREATER',   # >
    'FALSE',     # false
    'TRUE',      # true
    'IF',        # if
    'ELSE',      # else
    "AND",       # E l gico
    "OR",        # OU l gico
    'JUMP',      # JUMP
```

```
37      'BOOL',       # TIPO BOOL
38      'NOT',        # NOT l gico
39      'MARK',       # MARK
40      'PRINT',      # PRINT
41      'COMMENT',    # ??...??
42      'LER'         # READ do EWVM
43  )
44
45  #OP      = + | - | / | *
46  #TIPO    = INT|BOOL|ARRAY|STRING
47  #BOOL    = TRUE | FALSE
48  #OPLOG   = AND | OR
49  #OPCOND = '==' | '!=' | '<' | '<=' | '>' | '>='
50
51  def t_COMMENT(t):
52      r'\?\?(.*)?\?\?|\?\*([\s\S]*?)\*?'
53      pass
54
55  t_OPSUB = r"\-"
56  t_PCE = r"\("
57  t_PCD = r"\)"
58  t_ABRIR_CH = r"\{"
59  t_FECHAR_CH = r"\}"
60  t_PV = r"\;"
61  t_IGUAL = r"\=\="
62  t_OPAT = r"\="
63  t_OPAD = r"\+"
64  t_OPDIV = r"\/"
65  t_OPMUL = r"\*"
66  t_DIFF = r"\!\="
67  t_LEQ = r"\<\="
68  t_LESS = r"\<"
69  t_GEQ = r"\>\="
70  t_GREATER = r"\>"
71  t_PRE = r'\['
72  t_PRD = r'\]'
73
74  def t_LER(t):
75      r'LER'
76      return t
77
78  def t_PRINT(t):
79      r"PRINT"
80      return t
81
82  def t_NOTHING(t):
83      r"NOTHING"
84      return t
85
86  def t_NOT(t):
87      r"NOT"
88      return t
89
90  def t_BOOL(t):
```

```
91      r"BOOL"
92      return t
93
94  def t_NUMFLOAT(t):
95      r"-?\d+\.\d+"
96      t.value = float(t.value)
97      return t
98
99  def t_NUMINT(t):
100     r"-?\d+"
101     t.value = int(t.value)
102     return t
103
104 def t_PAL(t):
105     r'"([^"]*)"'
106     return t
107
108 def t_INT(t):
109     r"INT"
110     return t
111
112 def t_FLOAT(t):
113     r"FLOAT"
114     return t
115
116 def t_STRING(t):
117     r"STRING"
118     return t
119
120 def t_ARRAY(t):
121     r"ARRAY"
122     return t
123
124 def t_VIRG(t):
125     r","
126     return t
127
128 def t_FALSE(t):
129     r"FALSE"
130     return t
131
132 def t_TRUE(t):
133     r"TRUE"
134     return t
135
136 def t_IF(t):
137     r"IF"
138     return t
139
140 def t_ELSE(t):
141     r"ELSE"
142     return t
143
144 def t_RETURN(t):
```

```
145     r"RETURN"
146     return t
147 
148 def t_AND(t):
149     r"AND"
150     return t
151 
152 def t_OR(t):
153     r"OR"
154     return t
155 
156 def t_JUMP(t):
157     r"JUMP"
158     return t
159 
160 def t_VAR(t):
161     r"[a-z][a-zA-Z0-9_]*"
162     return t
163 
164 def t_MARK(t):
165     r'MARK'
166     return t
167 
168 t_ignore = ' \t\n'
169 
170 def t_error(t):
171     print(f"Carater ilegal '{t.value[0]}' na linha {t.lineno}, posi  o {t.lexpos}")
172     t.lexer.skip(1)
173 
174 lexer = lex.lex()
175 
176 '''programa = """INT i;
177 ARRAY ar[4];
178 
179 MARK IF (i<4) {
180     i = i + (1);
181 } JUMP
182 ar(0) = i; ??COMENTARIO??
183 ar(1) = i; ?? OUTRO COMENTARIO ??
184 ar(2) = i;
185 ar(3) = i;"""
186 
187 lexer.input(programa)
188 
189 while tok := lexer.token():
190     print(tok)
191 
192 print("\nFim da An lise l xica\n")'''
```

Listing A.1: Código pertencente ao lex

A.2 Parser

```
import ply.yacc as yacc
import sys
import re
from PLC_TP2_Lexico import tokens

def notInConjunto(p):
    res = True
    if (p in parser.inteiros): res = False
    elif(p in parser.strings): res = False
    elif(p in parser.bools): res = False
    elif(p in parser.arrays): res = False
    elif(p in parser.matrizes): res = False
    return res

def elementosLista(p): #Dividir elementos de p[2]
    s = []
    for elemento in p:
        if isinstance(elemento, list):
            s.extend(elementosLista(elemento)) # Chama recursivamente a funcao para listas aninhadas
        else:
            s.append(elemento)
    return s
#Axioma
def p_axioma(p):
    'axioma : programa'
    p[0] = p[1]
    for x in p[0]:
        if (x != None and x !=''):
            print(x,end="")
    if(parser.nPops > 0):
        print(f"POP{parser.nPops}")
        parser.nPops = 0

def p_programa(p):
    'programa : declaracoes corpo'
    p[0] = elementosLista([p[1]]) + elementosLista([p[2]])

#---------------declaracoes-----------------------#

def p_declaracoes_declaracoesDeclaracao(p):
    'declaracoes : declaracoes declaracao'
    p[0] = [p[1]] + [p[2]]
    p[0] = elementosLista(p[0])

def p_declaracoes_empty(p):
    'declaracoes : '

#---------------declaracao-----------------------------#

def p_declaracao_seqDecl(p):
    'declaracao : tipo seqDecl PV'
```

```
52      parser.tipos = str(p[1])
53      p[0] = str(p[2])
54
55  #---------------SEQDECL-----------------------------------#
56
57  def p_seqDecl_atrVIRGseqDecl(p):
58      'seqDecl : atr VIRG seqDecl'
59      p[0] = p[1] + str(p[3])
60      parser.tipos = ""
61
62  def p_seqDecl_atr(p):
63      'seqDecl : atr'
64      p[0] = p[1]
65
66
67  #---------------atr---------------------------------#
68
69  def p_atr_VAR(p):
70      'atr : VAR'
71      if (notInConjunto(p[1])):
72          match parser.tipos:
73              case "INT":
74                  parser.inteiros[p[1]] = parser.posicao
75                  p[0] = f"PUSHI0\n"
76                  parser.posicao += 1
77                  parser.nPops += 1
78              case "STRING":
79                  parser.strings[p[1]] = parser.posicao
80                  p[0] = f"PUSHS \"\"\n"
81                  parser.posicao += 1
82                  parser.nPops += 1
83              case "BOOL":#default TRUE
84                  parser.bools[p[1]] = parser.posicao
85                  p[0] = f"PUSHI1\n"
86                  parser.posicao += 1
87                  parser.nPops += 1
88              case "" :
89                  print("ERRO: Nao atribuido o tipo")
90      else:
91          print(f"ERRO: A variavel {p[1]} ja foi declarada")
92          exit()
93
94  def p_atr_VAROPAToperacao(p):
95      'atr : VAR OPAT operacao'
96      if(not notInConjunto(p[1])):
97          print(f"ERRO: A variavel {p[1]} ja foi declarada!")
98          exit()
99      if(parser.tipos != "STRING" and parser.tipos != "BOOL" and parser.tipos != "INT")
    :
100         print("ERRO: Esse tipo de dados nao existe.")
101         exit()
102     if (parser.tipos == "INT" and notInConjunto(p[1])):
103         parser.inteiros[p[1]] = parser.posicao #Associa o numero a variavel no
    dicionario
```

```
104     if (parser.tipos == "STRING" and notInConjunto(p[1])):
105         parser.strings[p[1]] = parser.posicao
106         p[0] = p[3]
107     if (parser.tipos == "BOOL" and notInConjunto(p[1])):
108         parser.bools[p[1]] = parser.posicao
109         p[0] = p[3]
110     p[0] = f"{p[3]}"
111     parser.posicao += 1
112     parser.nPops +=1
113 
114 def p_atr_VARARRAY(p):
115     'atr : VAR PRE NUMINT PRD'
116     if (parser.tipos == 'ARRAY' and notInConjunto(p[1])):
117         tamanho = int(p[3])
118         parser.arrays[p[1]] = (parser.posicao,tamanho)
119         p[0] = 'PUSHN'+str(tamanho) +'\n'
120         parser.posicao += tamanho
121         parser.nPops += tamanho
122     elif (parser.tipos [0] != 'ARRAY'):
123         print(f"ERRO: A variavel {p[1]} nao e do tipo ARRAY.")
124         exit()
125     elif(not notInConjunto(p[1])):
126         print(f"ERRO: Variavel {p[1]} ja declarada anteriormente")
127         exit()
128 
129 def p_atr_VARARRAYcomConteudo(p):
130     'atr : VAR PRE NUMINT PRD OPAT ABRIR_CH seqNumInt FECHAR_CH'
131     if(not notInConjunto(p[1])):
132         print(f"ERRO: Variavel {p[1]} ja foi declarada anteriormente")
133         exit()
134     if(parser.nArray == int(p[3])): #Conteudo com numero certo de elementos
135         if (parser.tipos == 'ARRAY' and notInConjunto(p[1])):
136             tamanho = int(p[3])
137             parser.arrays[p[1]] = (parser.posicao,tamanho)
138             p[0] = p[7]
139             parser.posicao += tamanho
140             parser.nPops += tamanho
141     else:
142         print(f"ERRO: A variavel {p[1]} tem limite {int(p[3])} e foram colocados {
    parser.nArray} elementos!")
143         exit()
144 
145 def p_atr_matrizDefault(p):
146     'atr : VAR PRE NUMINT PRD PRE NUMINT PRD'
147     if (parser.tipos == 'ARRAY' and notInConjunto(p[1])):
148         tamanho = int(p[3]) * int(p[6])
149         #Guarda a posicao de onde comeca na stack e os parametros de tamanho
150         parser.matrizes[p[1]] = (parser.posicao,int(p[3]),int(p[6]))
151         p[0] = f"PUSHN{tamanho}\n"
152         parser.posicao += tamanho
153         parser.nPops += tamanho
154     elif (parser.tipos [0] != 'ARRAY'):
155         print(f"ERRO: A variavel {p[1]} nao e do tipo ARRAY.")
156         exit()
```

```
157     elif(not notInConjunto(p[1])):
158         print(f"ERRO: Variavel {p[1]} ja foi declarada anteriormente")
159         exit()
160 
161 def p_atr_matrizComConteudo(p):
162     'atr : VAR PRE NUMINT PRD PRE NUMINT PRD OPAT ABRIR_CH seqNumInt FECHAR_CH'
163     if(not notInConjunto(p[1])):
164         print(f"ERRO: Variavel {p[1]} ja foi declarada anteriormente")
165         exit()
166     if(parser.nArray == int(p[3])): #Conteudo com n mero certo de elementos
167         if (parser.tipos == 'ARRAY' and notInConjunto(p[1])):
168             tamanho = int(p[3]) * int(p[6])
169             parser.matrizes[p[1]] = (parser.posicao,int(p[3]),int(p[6]))
170             p[0] = p[10]
171             parser.posicao += tamanho
172             parser.nPops += tamanho
173     else:
174         print(f"ERRO: A variavel {p[1]} tem limite {int(p[3])} e foram colocados {
    parser.nArray} elementos!")
175         exit()
176 
177 
178 #---------------seqNumInt-------------------------------#
179 
180 def p_seqNumInt_NUMINT(p):
181     'seqNumInt : NUMINT'
182     parser.nArray += 1
183     p[0] = f"PUSHI{p[1]}\n"
184 
185 def p_seqNumInt_NumMaisNum(p):
186     'seqNumInt : NUMINT VIRG seqNumInt'
187     parser.nArray += 1
188     p[0] = f"PUSHI{p[1]}\n" + p[3]
189 
190 #---------------corpo-----------------------------------#
191 
192 def p_corpo_instrucao(p):
193     'corpo : corpo instrucao'
194     parser.tipos = "DECLDONE"
195     p[0] = [p[1]] + [p[2]]
196     p[0] = elementosLista(p[0])
197     s = ""
198     for elemento in p[0]:
199         if elemento != None:
200             s+=elemento
201     p[0] = s
202 
203 def p_corpo_VAZIO(p):
204     'corpo : '
205     parser.tipos = "DECLDONE"
206     p[0] = ""
207 
208 #---------------instrucao-------------------------------#
209 
```

```
210 def p_instrucao_ifelsestatement(p):
211     'instrucao : ifelsestatement'
212     p[0] =  p[1]
213 
214 def p_instrucao_print(p):
215     'instrucao : print PV'
216     p[0] = p[1]
217 
218 def p_instrucao_VARopatOPERACAO(p):
219     'instrucao : VAR OPAT operacao PV'
220     if(not notInConjunto(p[1])):
221         if(p[1] in parser.inteiros): #Variavel declarada mas nao inteira
222             padrao_string = r'\sPUSH[SG].*'
223             if (re.match(r'PUSHI.*',p[3])):
224                 p[0] = p[3] + f"STOREG{parser.inteiros[p[1]]}\n"
225             else:
226                 print("ERRO: A Varivel nao e do tipo INT.")
227                 exit()
228             p[0] = p[3] + f"STOREG{parser.inteiros[p[1]]}\n"
229         elif(p[1] in parser.bools):
230             padrao_bool = r'\s\sPUSH[IG].*'
231             if (re.match(padrao_bool,p[3])):
232                 p[0] = p[3] + f"STOREG{parser.bools[p[1]]}\n"
233             else:
234                 print("ERRO: A Varivel nao e do tipo BOOL.")
235                 exit()
236         elif(p[1] in parser.strings):
237             padrao_string = r'\sPUSH[SG].*'
238             if (re.match(padrao_string,p[3])):
239                 p[0] = p[3] + f"STOREG{parser.strings[p[1]]}\n"
240             else:
241                 print("ERRO: A Varivel nao e do tipo STRING.")
242                 exit()
243     elif (notInConjunto(p[1])):
244         print("ERRO: Varivel nao declarada corretamente.")
245         exit()
246 
247 def p_instrucao_mudarindexarray(p):
248     'instrucao : VAR PCE NUMINT PCD OPAT operacao PV'
249     if(not notInConjunto(p[1])):
250         if(p[1] not in parser.arrays): #Variavel declarada mas nao do tipo ARRAY
251             print(f"ERRO: A variavel {p[1]} nao e do tipo ARRAY")
252             exit()
253         else:#Se for variavel declarada do tipo certo
254             p[0] = p[6] + f"STOREG{parser.arrays[p[1]][0]+p[3]}\n"
255     else:
256         print(f"ERRO: A variavel {p[1]} nao foi declarada")
257 
258 def p_instrucao_mudarindexMatriz(p):
259     'instrucao : VAR PCE NUMINT PCD PCE NUMINT PCD OPAT operacao PV '
260     if(not notInConjunto(p[1])):
261         if(p[1] not in parser.matrizes): #Variavel declarada mas nao do tipo ARRAY
262             print(f"ERRO: A variavel {p[1]} nao e do tipo ARRAY")
263             exit()
```

```
264         else:#Se for variavel declarada do tipo certo
265             #Se quiser mudar o valores de x[z1][z2]  da matriz x[y1][y2] y1 linhas e
    y2 colunas
266             # tenho que aceder na stack a posicao:
267             # posicao da matriz na stack + (y2 * z1) + z2
268             pm = parser.matrizes[p[1]][0] + (parser.matrizes[p[1]][2] * p[3]) + p[6]
269             p[0] = p[9] + f"STOREG{pm}\n"
270     else:
271         print(f"ERRO: A variavel {p[1]} nao foi declarada")
272 
273 def p_instrucao_tojumpPV(p):
274     'instrucao : tojump PV'
275     p[0] = p[1]
276 
277 def p_instrucao_jumptoPV(p):
278     'instrucao : jumpto PV'
279     p[0] = p[1]
280 
281 
282 #---------------operacao-------------------------#
283 
284 def p_operacao_termo(p):
285     'operacao : termo'
286     p[0] = p[1]
287 
288 def p_operacao_adicao(p):
289     'operacao : operacao OPAD termo'
290     p[0] = p[3] + p[1] + "ADD\n"
291 
292 def p_operacao_subtracao(p):
293     'operacao : operacao OPSUB termo'
294     p[0] = p[3] + p[1] + "SUB\n"
295 
296 
297 #---------------TERMO-----------------------------------#
298 
299 def p_termo_fator(p):
300     'termo : fator'
301     p[0] = p[1]
302 
303 def p_termo_mult(p):
304     'termo : termo OPMUL fator'
305     p[0] = p[1] + p[3] + 'MUL\n'
306 
307 def p_termo_DIV(p):
308     'termo : termo OPDIV fator'
309     p[0] = p[1] + p[3] + 'DIV\n'
310 
311 
312 #---------------FATOR-----------------------------------#
313 
314 def p_fator_NUMINT(p):
315     'fator : NUMINT'
316     p[0] = f"PUSHI{p[1]}\n"
```

```
317
318 def p_fator_VAR(p):
319     'fator : VAR'
320     if (p[1] in parser.inteiros):
321         p[0]= f"PUSHG {parser.inteiros[p[1]]}\n"
322     elif (p[1] in parser.bools):
323         p[0]= f"PUSHG {parser.bools[p[1]]}\n"
324     elif (p[1] in parser.strings):
325         p[0]= f" PUSHG {parser.strings[p[1]]}\n"
326     elif (p[1] in parser.arrays): #GUARDAR A QUANTIDADE DE WRITES A FAZER NO PRINT
327         i = 0
328         a = parser.arrays[p[1]][i]
329         s = ""
330         while(i<parser.arrays[p[1]][1]):
331             s += f"PUSHG {a}\n"
332             i+=1
333             a+=1
334         p[0] = s
335     elif(p[1] in parser.matrizes):#GUARDAR A QUANTIDADE DE WRITES A FAZER NO PRINT
336         i1 = 0
337         i2 = 1
338         #O tamanho e dado pelo nmr_linhas * nmr_colunas
339         tamanho = parser.matrizes[p[1]][1] * parser.matrizes[p[1]][2]
340         # a = posicao do array na stack
341         a = parser.matrizes[p[1]][0]
342         s = ""
343         while(i1<tamanho):
344             s += f"PUSHG {a}\n"
345             a += 1
346             i1 += 1
347             i2 += 1
348         p[0] = s
349
350 def p_fator_VARARRAY(p):
351     'fator : VAR PCE NUMINT PCD'
352     if (parser.tipos == "ARRAY" or parser.tipos == "DECLDONE"):
353         p[0] = f"PUSHG{parser.arrays[p[1]][0]+p[3]}\n"
354     else:
355         print("ERRO: A varivel nao e do tipo ARRAY.")
356         exit()
357
358 def p_fator_VARMATRIZ(p):
359     'fator : VAR PCE NUMINT PCD PCE NUMINT PCD'
360     if (parser.tipos == "ARRAY" or parser.tipos == "DECLDONE"):
361         pm = parser.matrizes[p[1]][0] + (parser.matrizes[p[1]][2] * p[3]) + p[6]
362         p[0] = f"PUSHG{pm}\n"
363     else:
364         print("ERRO: A varivel nao e do tipo ARRAY.")
365         exit()
366
367 def p_fator_bool(p):
368     'fator : bool'
369     if (parser.tipos == "BOOL" or parser.tipos == "DECLDONE"):
370         p[0] = p[1]
```

```
371     else:
372         print("ERRO: A varivel nao e do tipo BOOL.")
373         exit()
374 
375 def p_fator_PAL(p):
376     'fator : PAL'
377     if (parser.tipos == "STRING" or parser.tipos == "DECLDONE"):
378         p[0] = f" PUSHS {p[1]}\n"
379     else:
380         print("ERRO: A varivel nao e do tipo STRING.")
381         exit()
382 
383 def p_fator_NOTcondicao(p):
384     'fator : NOT PCE condicao PCD'
385     p[0] = p[3] + "NOT\n"
386 
387 def p_fator_lerInput(p):
388     'fator : LER PCE PCD'
389     p[0] = "READ\nATOI\n"
390 
391 
392 #---------------IFELSESTATEMENT---------------------#
393 
394 def p_ifselsetatement(p):
395     'ifelsestatement : IF PCE condicao PCD ABRIR_CH corpo FECHAR_CH ELSE ABRIR_CH
    corpo FECHAR_CH'
396     if_ = parser.posicaoIf
397     p[0] = f"{p[3]}\nJZ elselabel{if_}\n{p[6]}JUMP iflabel{if_}\nelselabel{if_}:\n{p
    [10]}iflabel{if_}:\n"
398     parser.posicaoIf += 1
399 
400 
401 #---------------condicao---------------------------#
402 def p_condicao_operacao(p):
403     'condicao : operacao'
404     p[0] = p[1]
405 
406 def p_condicao_igualigual(p):
407     'condicao : operacao IGUAL operacao'
408     p[0] = p[1] + p[3] + "EQUAL\n"
409 
410 def p_condicao_DIFF(p):
411     'condicao : operacao DIFF operacao'
412     p[0] = p[1] + p[3] + "EQUAL\n" + "NOT\n"
413 
414 def p_condicao_MENOR(p):
415     'condicao : operacao LESS operacao'
416     p[0] = p[1] + p[3] + "INF\n"
417 
418 def p_condicao_MAIOR(p):
419     'condicao : operacao GREATER operacao'
420     p[0] = p[1] + p[3] + "SUP\n"
421 
422 def p_condicao_MENORouIGUAL(p):
```

```
423     'condicao : operacao LEQ operacao'
424     p[0] = p[1] + p[3] + "INFEQ\n"
425 
426 def p_condicao_MAIORouIGUAL(p):
427     'condicao : operacao GEQ operacao'
428     p[0] = p[1] + p[3] + "SUPEQ\n"
429 
430 def p_condicao_OU(p):
431     'condicao : PCE condicao PCD OR PCE condicao PCD'
432     p[0] = p[2] + p[6] + "OR\n"
433 
434 def p_condicao_AND(p):
435     'condicao : PCE condicao PCD AND PCE condicao PCD'
436     p[0] = p[2] + p[6] + "AND\n"
437 
438 
439 #---------------tojump-----------------------------------#
440 
441 #dowhile
442 def p_tojump_MARKABRIRCHcorpoFECHARCHif(p):
443     'tojump : MARK ABRIR_CH corpo FECHAR_CH IF PCE condicao PCD JUMP'
444     cic = parser.posicaoCiclo
445     p[0] = p[3]+f"labelwhile{cic}:\n"+p[7]+f"JZ endwhile{cic}\n"+p[3]+f"JUMP
    labelwhile{cic}\n"+f"endwhile{cic}:\n"
446     parser.posicaoCiclo += 1
447 
448 
449 #---------------jumpto-----------------------------------#
450 
451 #whiledo
452 def p_jumpto_MARKifCondcorpoJump(p):
453     'jumpto : MARK IF PCE condicao PCD ABRIR_CH corpo FECHAR_CH JUMP'
454     cic = parser.posicaoCiclo
455     p[0] = f"labelwhile{cic}:\n"+p[4]+f"JZ endwhile{cic}\n"+p[7]+f"JUMP labelwhile{
    cic}\n"+f"endwhile{cic}:\n"
456     parser.posicaoCiclo += 1
457 
458 
459 #---------------PRINT-----------------------------------#
460 def p_print(p):
461     'print : PRINT PCE condicao PCD'
462     padrao_string = r'\sPUSH[SG].*'
463     if (re.match(padrao_string,p[3])):
464         p[0] = f"{p[3]}WRITES\n"
465     else:
466         p[0] = f"{p[3]}WRITEI\n"
467 
468 #---------------bool-----------------------------------#
469 
470 def p_bool_TRUE(p):
471     'bool : TRUE'
472     p[0] = "  PUSHI1\n"
473 
474 def p_bool_FALSE(p):
```

```
475     'bool : FALSE'
476     p[0] = "  PUSHI0\n"
477 
478 
479 #----------------------tipo---------------------------#
480 
481 def p_tipo_INT(p):
482     'tipo : INT'
483     parser.tipos = 'INT'
484     p[0] = ""
485 
486 def p_tipo_STRING(p):
487     'tipo : STRING'
488     parser.tipos = 'STRING'
489     p[0] = ""
490 
491 def p_tipo_BOOL(p):
492     'tipo : BOOL'
493     parser.tipos = 'BOOL'
494     p[0] = ""
495 
496 def p_tipo_ARRAY(p):
497     'tipo : ARRAY'
498     parser.tipos = 'ARRAY'
499     p[0] = "" #N inteiros
500 
501 #----------------------main---------------------------#
502 def p_error(p):
503     parser.success = False
504     print('Syntax error!',p)
505 
506 #inicio do parsing
507 parser = yacc.yacc() #Transforma automato num parser
508 parser.success = True
509 
510 parser.nPops = 0            #Para limpar a stack
511 parser.posicaoIf = 0        #Guarda contagem de labels
512 parser.posicaoCiclo = 0     #Guarda contagem de labels
513 parser.posicao = 0      #Guardar posicoes na stack para declaracoes
514 parser.nArray = 0       #Saber o tamanho do array ao declarar
515 parser.tipos    = ""    #Para saber se o tipo e o declarado
516 parser.inteiros = {}    #Guardar posicoes das variaveis declaradas INT
517 parser.strings  = {}    #Guardar posicoes das variaveis declaradas STRING
518 parser.bools    = {}    #Guardar posicoes das variaveis declaradas BOOL
519 parser.arrays   = {}    #Guardar posicoes das variaveis declaradas ARRAY
520 parser.matrizes = {}    #Guardar posicoes das variaveis declaradas ARRAY (matriz)
521 parser.nWrites = 0
522 #Exemplo swap
523 #file = open("exemploswap.txt",'r')
524 #Exemplo IfElse
525 #file = open("exemploIfElse.txt",'r')
526 #Exemplo Declaracao de um array multi-dimensional
527 #file = open("exemploarrayMultiDim.txt",'r')
528 #Exemplo Declaracao de um array e um while
```

```
529 #file = open("exemploarrayWhile.txt",'r')
530 #Exemplo Utilizacao de variaveis STRING e varias condicoes interligadas
531 #file = open("exemploVarStringCond.txt",'r')
532 #Exemplo Aninhamento de IfElse e leitura de Input
533 #file = open("exemploIfElseInput.txt",'r')
534 #Exemplo Erro de declara o de variveis j declaradas
535 #file = open("exemploErroDeclaracao.txt",'r')
536 #Exemplo Erro de tipo no correspondente declaracao
537 #file = open("exemploErroNaoCorrespondenciaDecl.txt",'r')
538 #Exemplo Erro de tipo no correspondente corpo
539 #file = open("exemploErroNaoCorrespondenciaCorpo.txt",'r')
540 #Exemplo Erro de tipo no conjunto atribu do ao array maior que a sua capacidade
541 #file = open("exemploErroConjDemais.txt",'r')
542 #parser.parse(file.read())
```

Listing A.2: Código pertencente ao Parser