Processamento de Linguagens e Compiladores (3º Ano)

Trabalho Prático 2

Relatório de Desenvolvimento

Alexandra Calafate	Gonçalo Magalhães	João Mirra
(a100060)	(a100084)	(a100083)

05/01/2025

Resumo

No contexto da disciplina de Processamento de Linguagens e Compiladores, o segundo projeto prático consiste em criar uma linguagem imperativa personalizada e desenvolver um compilador utilizando os módulos de gramáticas tradutoras disponíveis em Python. Este trabalho inclui a definição de uma gramática que converte a linguagem imperativa em código assembly, utilizando ferramentas como o lex e o yacc do Python. Neste relatório, descrevemos detalhadamente as escolhas realizadas, as produções implementadas na gramática e as etapas seguidas no desenvolvimento do compilador, de forma clara e objetiva.

Conteúdo

Capítulo 1

Introdução

No âmbito da disciplina de Processamento de Linguagens e Compiladores, o professor Pedro Rangel Henriques propôs-nos um projeto de grupo com objetivos fundamentais. Estes incluem o desenvolvimento de competências na escrita eficiente de gramáticas, a capacitação para construir um processador de linguagens com base numa gramática tradutora e, ainda, a criação de um compilador capaz de gerar código para uma máquina de stack virtual.

A linguagem a ser usada neste projeto, será uma linguagem imperativa simples com regras acordadas pelos elementos do grupo.
Com base na gramática independente do contexto (GIC) que definimos, o compilador desenvolvido para a nossa linguagem terá de gerar pseudo-código Assembly para uma VM.

Neste documento apresentamos uma possivel solução para cada um dos problemas propostos, com recurso aos módulos 'Yacc/Lex' do 'PLY/Python'.

Estrutura do Relatório

O relatório está organizado da seguinte forma:
Iniciamos com uma breve introdução, no capítulo 1, onde é descrito o objetivo do trabalho a desenvolver.
No capítulo 2, é apresentado o enunciado do problema proposto.
No capítulo 3, descrevemos a estrutura do nosso trabalho.
No capítulo 4, demonstramos o funcionamento de vários testes realizados pelo grupo.
Por fim, no último capítulo, apresentamos a conclusão e uma análise final do trabalho realizado.

Capítulo 2

Problema Proposto

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 de seleção para o controlo do fluxo de execução.
- efetuar instruções de repetição(cíclicas) para o controlo de 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.

Capítulo 3

Elaboração da Solução

3.1 Organização e estrutura

O nosso trabalho pode ser divido em **4** partes:

- Cosntrução da **GIC** que define a estrutura sintática da liguaguem.
- Construção do analizador léxico, **lexer** .
- Construção do analizador sintático, **parser** .
- Conversão das instruções para código **Assembly** da VM.

Todas as funcionalidades descritas neste capítulo podem ser encontradas no anexo A do documento.

3.2 GIC

A nossa linguagem será gerada pela seguinte Grámatica Independente de Contexto (GIC):

```
## GIC que criamos

Programa : Decls
         | Atrib
         | Decls Corpo
         | Atrib Corpo
         | Corpo

Corpo : Codigo

Codigo : Proc Codigo
       | Proc
       | Decls Codigo
       | Decls
```

```
       | Atrib Codigo
       | Atrib
       | COMENTARIO Codigo
       | COMENTARIO

Decls : Decl PONTOVIRGULA
      | Decl VIRGULA Decls

Proc : IF
     | WHILE
     | PRINTAR

Decl : INT ID
     | MATRIZ INT NUM NUM VIRGULA ID
     | LISTA INT NUM VIRGULA ID PONTOVIRGULA

Atrib : INT ID IGUAL expr PONTOVIRGULA
      | ID IGUAL expr PONTOVIRGULA
      | ALTERA ID IGUAL expr PONTOVIRGULA
      | ALTERA ID LPAREN_RETO expr RPAREN_RETO IGUAL expr PONTOVIRGULA
      | ALTERA ID LPAREN_RETO expr RPAREN_RETO LPAREN_RETO expr RPAREN_RETO IGUAL expr PONTOVIRGU
      | ALTERA ID LPAREN_RETO expr RPAREN_RETO IGUAL LISTA

expr : exprArit
     | exprRel
     | NUM
     | ID
     | INPUT
     | ID LPAREN_RETO expr RPAREN_RETO
     | ID LPAREN_RETO expr RPAREN_RETO LPAREN_RETO expr RPAREN_RETO

exprArit : exprArit PLUS term
         | exprArit MINUS term
         | term

term : term TIMES factor
     | term DIVIDE factor
     | factor

factor : NUM
       | ID

exprRel : expr IGUALIGUAL expr
        | expr DIFERENTE expr
        | expr MENOR expr
        | expr MENOROUIGUAL expr
        | expr MAIOR expr
        | expr MAIOROUIGUAL expr
```

```
        | expr EE expr
        | expr OU expr

IF : SE LPAREN exprRel RPAREN ENTAO Codigo FIM_COND PONTOVIRGULA
   | SE LPAREN exprRel RPAREN ENTAO Codigo SENAO Codigo FIM_COND PONTOVIRGULA

WHILE : ENQUANTO LPAREN exprRel RPAREN FAZ Codigo FIM_ENQUANTO PONTOVIRGULA

PRINTAR : IMPRIMIR LPAREN FRASE RPAREN PONTOVIRGULA
        | IMPRIMIR LPAREN ID RPAREN PONTOVIRGULA

COMENTARIO : COMENTADO
```

3.3 Lexer

Através de expressões regulares, o analisador léxico **lexer** tem a função de identificar os símbolos terminais (*tokens*) da nossa linguagem. Para a sua implementação, utilizámos o módulo 'Lex' do 'PLY/Python'. Os *tokens* e as respetivas expressões regulares da nossa linguagem são os seguintes:

```
tokens = [
    'ID',
    'NUM',
    'INT',

    'PLUS',
    'MINUS',
    'TIMES',
    'DIVIDE',

    'MAIOR',
    'MAIOROUIGUAL',
    'MENOR',
    'MENOROUIGUAL',
    'IGUALIGUAL',
    'DIFERENTE',
    'EE',
    'OU',

    'LPAREN',
    'RPAREN',
    'LPAREN_RETO',
    'RPAREN_RETO',

    'SE',
    'ENTAO',
    'SENAO',
    'FIM_COND',

    'ENQUANTO',
    'FAZ',
    'FIM_ENQUANTO',

    'INPUT',
    'IMPRIMIR',
    'VIRGULA',
    'PONTOVIRGULA',
    'IGUAL',

    'COMENTADO',
```

```
    'FRASE',

    'ALTERA',

    'LISTA',
    'MATRIZ'
]

# Regras dos tokens
t_INT = r'int'
t_ID = r'\w+'

t_PLUS = r'\+'
t_MINUS = r'-'
t_TIMES = r'\*'
t_DIVIDE = r'/'

t_MAIOR = r'>'
t_MAIOROUIGUAL = r'>='
t_MENOR = r'<'
t_MENOROUIGUAL = '<='
t_IGUALIGUAL = '=='
t_DIFERENTE = '!='
t_EE = '\\/'
t_OU = '\\/'

t_LPAREN = r'\('
t_RPAREN = r'\)'
t_LPAREN_RETO = r'\['
t_RPAREN_RETO = r'\]'

t_SE = r'ifse'
t_ENTAO = r'entao'
t_SENAO = r'senao'
t_FIM_COND = r'fim_cond'

t_ENQUANTO = r'enquanto'
t_FAZ = r'faz'
t_FIM_ENQUANTO = r'fim_enquanto'

t_INPUT = r'input'
t_IMPRIMIR = r'imprimir'
t_VIRGULA = r','
t_PONTOVIRGULA = r';'
t_IGUAL = r'='

t_COMENTADO = r'%[^\n]*'
```

```
t_FRASE = r'"[^"]+"'

t_ALTERA = r'altera'

t_LISTA = r'lista'
t_MATRIZ = r'matriz'
```

A implementação do analisador léxico pode ser encontrada no anexo ao documento.

3.4 Parser e geração do código Assembly da VM

O **parser**, ou analisador sintático, tem a função de assegurar que o código escrito na nossa linguagem está corretamente estruturado de acordo com as regras gramaticais predefinidas. Em outras palavras, ele verifica se o código segue a sintaxe estabelecida.

Caso não haja erros sintáticos, o **parser** realiza a transformação do código na nossa linguagem para o código **Assembly** da máquina virtual. Se forem encontrados erros ou anomalias, uma mensagem de erro sintático será exibida ao utilizador.

A implementação deste analisador sintático está completa num dos anexos do nosso relatório, conforme solicitado pelo docente.

3.4.1 Notas sobre declaração de variáveis

Ao gerar o código para fazer a declaração de uma variável sem valor fazemos:

```
PUSHI 0
```

Sendo que variável tem valor default igual a 0.

Por outro lado, para declará-la com valor fazemos:

```
PUSHI <valor>
STOREG <endereco>
```

No caso duma lista de valores fazemos:

```
PUSHN <tamanho>
```

Para Inicializar todos os valores do array como 0.

Seja uma lista de tamanho 2 temos por exemplo:

```
lista int 2, nomelista;
```

Para atribuir valores à lista fazemos:

```
PUSHN <tamanho>
PUSHGP
PUSHI 0
PUSHI <valor1>
STOREN
PUSHGP
```

```
PUSHI 1
PUSHI <valor2>
STOREN
```

Para declarar uma matriz procedemos a:

```
matriz int <tamanho1> <tamanho2>, <nomematriz>;
```

Caso a matriz m seja com formato 2x2, é gerado:

```
PISHN 4
```

Os valores são inicializados a 0.

Se pretendermos modificar os valores duma matriz temos de proceder da seguinte forma:

1. Alteramos uma especifica posição:

   ```
   altera <nomeMatriz>[<pos1>][<pos2>] = <valor>
   ```

Tendo uma matriz 2x2 como exemplo, fazendo a alteração de valores temos:

```
altera m [0][1] com <valor>
```

Gera-se o seguinte:

```
PUSHN 4
PUSHGP
PUSHI 0
PADD
PUSHI 0
PUSHI <valor>
STOREN
```

Capítulo 4

Demonstração do Funcionamento

4.1 Geração e execução de código Assembly

Para escrever instruç˜oes de acordo com as regras gramaticais da linguagem, o utilizador tem 2 opções de execução:

1. Caso esteja na mesma pasta dos restantes ficheiros:

   ```
   >> python yacc.py <ficheiro de input>
   ```

   Por exemplo:

   ```
   >> python yacc.py testeA.plc
   ```

2. Caso esteja numa outra pasta:

   ```
   >> python yacc.py <.\pasta\ficheiro de input>
   ```

   Por exemplo:

   ```
   >> python yacc.py .\testes\testeA.plc
   ```

4.2 Teste 1

Ficheiro de input: 'testeA.plc'

4.2.1 Conteúdo do ficheiro

```
int x,int y, int resultado;

x = 4;
y = 1;

resultado = x+y;

imprimir(resultado);
```

4.2.2 Código assembly gerado

```
PUSHI 0
PUSHI 0
PUSHI 0
START
PUSHI 4
STOREG 3
PUSHI 1
STOREG 4
PUSHG 3
PUSHG 4
ADD
STOREG 5
PUSHG 5
WRITEI
STOP
```

4.2.3 Execução da VM com o código gerado

```
5
```

4.3 Teste 2

Ficheiro de input: 'testeB.plc'.

4.3.1 Conteúdo do ficheiro

```
int x;

x = 4;

enquanto (x > 0) faz
    altera x = x-1;
fim_enquanto;

imprimir(x);
```

4.3.2 Código assembly gerado

```
PUSHI 0
START
PUSHI 4
STOREG 1
l0c: NOP
PUSHG 1
PUSHI 0
SUP
JZ l0f
PUSHG 1
PUSHI 1
SUB
STOREG 1
```

```
JUMP l0c
l0f: NOP
PUSHG 1
WRITEI
STOP
```

4.3.3 Execução da VM com o código gerado

```
0
```

4.4 Teste 3

Ficheiro de input: 'testeC.plc'.

4.4.1 Conteúdo do ficheiro

```
int x;

x = 4;

ifse (x>1) entao
    imprimir("maior");
fim_cond;
```

4.4.2 Código assembly gerado

```
PUSHI 0
START
PUSHI 4
STOREG 1
PUSHG 1
PUSHI 1
SUP
JZ l0
PUSHS "maior"
WRITES
l0: NOP
STOP
```

4.4.3 Execução da VM com o código gerado

```
maior
```

4.5 Teste 4

Ficheiro de input: 'testeD.plc'.

4.5.1 Conteúdo do ficheiro

```
int x;

x = 4;

ifse (x>5) entao
    imprimir("maior");
senao
    imprimir("menor");
fim_cond;
```

4.5.2 Código assembly gerado

```
PUSHI 0
START
PUSHI 4
STOREG 1
PUSHG 1
PUSHI 5
SUP
JZ l0
PUSHS "maior"
WRITES
JUMP l0f
l0: NOP
PUSHS "menor"
WRITES
l0f: NOP
STOP
```

4.5.3 Execução da VM com o código gerado

```
menor
```

4.6 Teste 5

Ficheiro de input: 'testeE.plc'.

4.6.1 Conteúdo do ficheiro

```
int x;

lista int 2, arr;

altera arr[0] = 1;

x = arr[0];

imprimir(x);
```

4.6.2 Código assembly gerado

```
PUSHI 0
START
PUSHN 2
PUSHGP
PUSHI 1
PADD
PUSHI 0
PUSHI 1
STOREN
PUSHGP
PUSHI 1
PADD
PUSHI 0
LOADN
STOREG 3
PUSHG 3
WRITEI
STOP
```

4.6.3 Execução da VM com o código gerado

```
1
```

4.7 Teste 6

Ficheiro de input: 'testeF.plo'.

4.7.1 Conteúdo do ficheiro

```
int x;
int y;

x = 2*4 + 5;
y = 5 + 4*2;

imprimir(x);
imprimir("\n");
imprimir(y);
```

4.7.2 Código assembly gerado

```
PUSHI 0
START
PUSHI 0
PUSHI 2
PUSHI 4
MUL
PUSHI 5
ADD
STOREG 2
NOP
```

```
PUSHI 5
PUSHI 4
PUSHI 2
MUL
ADD
STOREG 3
PUSHG 2
WRITEI
PUSHS "\n"
WRITES
PUSHG 3
WRITEI
STOP
```

4.7.3 Execução da VM com o código gerado

```
13
13
```

Capítulo 5

Conclusão

Ao longo deste projeto, procuramos constantemente aplicar os conhecimentos adquiridos nas aulas (teóricas e práticas), o que nos permitiu aprofundar e solidificar de maneira mais eficaz os conceitos abordados nesta Unidade Curricular.

Consideramos que, de forma geral, conseguimos atingir os objetivos preestabelecidos, o que nos proporcionou uma maior competência na elaboração de gramáticas e no desenvolvimento de compiladores de linguagens. Embora reconheçamos que poderíamos ter implementado uma gramática mais eficiente, acreditamos que a gramática criada atende aos requisitos do trabalho e resolve os problemas da linguagem em questão. Além disso, esta iniciativa contribuiu de forma significativa para ampliar o nosso entendimento sobre a máquina virtual e aprimorar as nossas habilidades na linguagem Assembly.

Em resumo, todo o empenho dedicado à realização deste projeto mostrou-se extremamente benéfico, consolidando as nossas bases e proporcionando uma maior facilidade na abordagem de temas específicos da Unidade Curricular, que podem ser relevantes em uma futura trajetória profissional.

Apêndice A

Código do Programa

A.1 Apêndice

Ficheiro lex.py

```
# lex.py

import ply.lex as lex # Import do ply.lex

# Defini  o dos tokens
tokens = [
    'ID',
    'NUM',
    'INT',

    'PLUS',
    'MINUS',
    'TIMES',
    'DIVIDE',

    'MAIOR',
    'MAIOROUIGUAL',
    'MENOR',
    'MENOROUIGUAL',
    'IGUALIGUAL',
    'DIFERENTE',
    'EE',
    'OU',

    'LPAREN',
    'RPAREN',
    'LPAREN_RETO',
    'RPAREN_RETO',

    'SE',
    'ENTAO',
    'SENAO',
    'FIM_COND',

    'ENQUANTO',
    'FAZ',
    'FIM_ENQUANTO',

```

```
    'INPUT',
    'IMPRIMIR',
    'VIRGULA',
    'PONTOVIRGULA',
    'IGUAL',

    'COMENTADO',

    'FRASE',

    'ALTERA',

    'LISTA',
    'MATRIZ'
]

# Regras dos tokens
t_INT = r'int'
t_ID = r'\w+'

t_PLUS = r'\+'
t_MINUS = r'-'
t_TIMES = r'\*'
t_DIVIDE = r'/'

t_MAIOR = r'>'
t_MAIOROUIGUAL = r'>='
t_MENOR = r'<'
t_MENOROUIGUAL = '<='
t_IGUALIGUAL = '=='
t_DIFERENTE = '!='
t_EE = '\\/'
t_OU = '\\/'

t_LPAREN = r'\('
t_RPAREN = r'\)'
t_LPAREN_RETO = r'\['
t_RPAREN_RETO = r'\]'

t_SE = r'ifse'
t_ENTAO = r'entao'
t_SENAO = r'senao'
t_FIM_COND = r'fim_cond'

t_ENQUANTO = r'enquanto'
t_FAZ = r'faz'
t_FIM_ENQUANTO = r'fim_enquanto'

t_INPUT = r'input'
t_IMPRIMIR = r'imprimir'
t_VIRGULA = r','
t_PONTOVIRGULA = r';'
t_IGUAL = r'='

t_COMENTADO = r'%[^\n]*'

t_FRASE = r'"[^"]+"'

t_ALTERA = r'altera'
```

```
98
99  t_LISTA = r'lista'
100 t_MATRIZ = r'matriz'
101
102 def t_NUM(t):
103     r'\d+'
104     t.value = int(t.value)
105     return t
106
107 # Ignorar espacos em branco e tabulacoes \t
108 t_ignore = ' \t'
109
110 # Tratamento do \n
111 def t_newline(t):
112     r'\n+'
113     t.lexer.lineno += len(t.value)
114
115 # Erro
116 def t_error(t):
117     print(f"Erro no caracter: '{t.value[0]}'")
118     t.lexer.skip(1)
119
120 # Analisador l xico
121 lexer = lex.lex()
```

Apêndice B

Código do Programa

B.1 Apêndice

Ficheiro yacc.py

```
1
2  import ply.yacc as yacc
3  from lex import *   # Import do nosso ficheiro criado
4
5  # Imports para manipula  o de files necess rios
6  import sys
7  import os
8
9  # Programa apenas com declara  es ou atribui  es
10 def p_Programa_Empty(p):
11     '''
12     Programa : Decls
13              | Atrib
14     '''
15     parser.assembly = f'{p[1]}'
16
17 # Programa com corpo p s declara  es e atribui  es
18 def p_Programa(p):
19     '''
20     Programa : Decls Corpo
21              | Atrib Corpo
22     '''
23     parser.assembly = f'{p[1]}START\n{p[2]}STOP\n'
24
25 # Programa -> Corpo
26 def p_Programa_Corpo(p):
27     '''
28     Programa : Corpo
29     '''
30     parser.assembly = f"START\n{p[1]}STOP\n"
31
32 # Corpo -> Codigo
33 def p_Corpo(p):
34     '''
35     Corpo : Codigo
36     '''
37     p[0] = f"{p[1]}"
38
```

```
# Codigo recursivamente
def p_Codigo_Rec(p):
    '''
    Codigo : Proc Codigo
           | Decls Codigo
           | Atrib Codigo
           | COMENTARIO Codigo
    '''
    p[0] = f"{p[1]}{p[2]}"

# Codigo pode ser um processo, atribuicao, declaracao ou um comentario
def p_Codigo(p):
    '''
    Codigo : Proc
           | Atrib
           | Decls
           | COMENTARIO
    '''
    p[0] = f"{p[1]}"

# declaracao unica terminada em ponto e virgula
def p_Decls(p):
    "Decls : Decl PONTOVIRGULA"
    p[0] = f'{p[1]}'

# declara  o recursiva separada por virgulas
def p_DeclsRec(p):
    "Decls : Decl VIRGULA Decls"
    p[0] = f'{p[1]}{p[3]}'

# expr pode ser arit ou relativa
def p_expr_arit(p):
    '''
    expr : exprArit
         | exprRel
    '''
    p[0] = p[1]

# procedimento pode ser um if,while ou um print
def p_Proc(p):
    '''
    Proc : IF
         | WHILE
         | PRINTAR
    '''
    p[0] = p[1]


# Declara  o de uma variavel sem valor
def p_Decl(p):
    "Decl : INT ID"
    varName = p[2]

    #    verificado no dicionario de variaveis se ela ja existe
    if varName not in parser.variaveis:
        parser.variaveis[varName] = (parser.stackPointer, None)
        p[0] = "PUSHI 0\n"                  # Atribuido valor 0 por defini  o
        parser.stackPointer += 1
    else:
```

```
98          parser.exito = False
99          parser.error = f"Vari vel com o nome {varName} j  existe"
100     parser.linhaDeCodigo +=1
101 
102 # Declara  o de uma vari vel com atribui  o de um valor
103 def p_Atrib_expr(p):
104     "Atrib : INT ID IGUAL expr PONTOVIRGULA"
105     varName = p[2]
106 
107     #    verificado no dicionario de variaveis se ela ja existe
108     if varName not in parser.variaveis:
109         value = p[4]
110         parser.variaveis[varName] = (parser.stackPointer, None)
111         p[0] = f"{value}STOREG {parser.stackPointer}\n"
112         parser.stackPointer += 1
113     else:
114         parser.exito = False
115         parser.error = f"Vari vel com o nome {varName} j  existe"
116     parser.linhaDeCodigo +=1
117 
118 # Atribui  o de uma variavel ja declarada
119 def p_Atrib_sem_decl(p):
120     "Atrib : ID IGUAL expr PONTOVIRGULA"
121     varName = p[1]
122 
123     # Verifica se a tal variavel j  existe mesmo
124     if varName in parser.variaveis:
125         value = p[3]
126         parser.variaveis[varName] = (parser.stackPointer, None)
127         p[0] = f"{value}STOREG {parser.stackPointer}\n"
128         parser.stackPointer += 1
129     else:
130         parser.exito = False
131         parser.error = f"Vari vel com o nome {varName} n o existe"
132     parser.linhaDeCodigo +=1
133 
134 
135 # Altera valor de um vari vel
136 def p_alterna_var(p):
137     "Atrib : ALTERA ID IGUAL expr PONTOVIRGULA"
138     varName = p[2]
139 
140     # verifica se a variavel ja existe mesmo
141     if varName in parser.variaveis:
142         p[0] = f"{p[4]}STOREG {parser.variaveis[varName][0]}\n"
143     parser.linhaDeCodigo +=1
144 
145 # expr pode ser um simples valor
146 def p_expr(p):
147     "expr : NUM"
148     p[0] = f"PUSHI {int(p[1])}\n"
149 
150 # expr pode ser uma variavel
151 def p_expr_var(p):
152     "expr : ID"
153     varName = p[1]
154     if varName in parser.variaveis:
155         p[0] = f"PUSHG {parser.variaveis[varName][0]}\n"
156 
```

```
157 # expr pode receber input
158 def p_expr_input(p):
159     "expr : INPUT"
160     p[0] = f"READ\nATOI\n"
161 
162 
163 # Declara lista com tamanho INT
164 def p_DeclLista_Size(p):
165     '''Decl : LISTA INT NUM VIRGULA ID''' # lista int 5, arr;
166     listName = p[5]
167     size = int(p[3])
168 
169     # verifica se esse nome j   n o esta a ser usado noutra variavel
170     if listName not in parser.variaveis:
171         # tamanho obrigatoriamente maior que 0
172         if size > 0:
173             parser.variaveis[listName] = (parser.stackPointer, size)
174             p[0] = f"PUSHN {size}\n"
175             parser.stackPointer += size
176         else:
177             parser.error = f"Imposs vel declarar um array de tamanho {size}"
178             parser.exito = False
179     else:
180         parser.error = (
181             f"Vari vel com o nome {listName} j   definida anteriormente.")
182         parser.exito = False
183     parser.linhaDeCodigo +=1
184 
185 
186 # Altera valor de um indice da lista
187 def p_AlteraLista_elem(p):
188     "Atrib : ALTERA ID LPAREN_RETO expr RPAREN_RETO IGUAL expr PONTOVIRGULA" # altera arr
    [3] = 4;
189     varName = p[2]
190 
191     # garante que a lista esta declarada
192     if varName in parser.variaveis:
193         p[0] = f"PUSHGP\nPUSHI {parser.variaveis[varName][0]}\nPADD\n{p[4]}{p[7]}STOREN\n"
194     else:
195         parser.error = f"Vari vel com o nome {varName} n o definida"
196         parser.exito = False
197     parser.linhaDeCodigo +=1
198 
199 # Fun   o que vai buscar o valor do indice na lista
200 def p_AtribBusca_Lista(p):
201     "expr : ID LPAREN_RETO expr RPAREN_RETO" # x = arr[4];
202     varName = p[1]
203     indice = p[3]
204 
205     # garante que a lista existe
206     if varName in parser.variaveis:
207         p[0] = f"PUSHGP\nPUSHI {parser.variaveis[varName][0]}\nPADD\n{indice}LOADN\n"
208     else:
209         parser.error = (
210             f"Vari vel com o nome {varName} n o definida anteriormente.")
211         parser.exito = False
212     parser.linhaDeCodigo += 1
213 
214 # Declara matriz com tamanho INT INT
```

```
215 def p_DeclMatriz(p):
216     "Decl : MATRIZ INT NUM NUM VIRGULA ID" # matriz int 3 2, mat;
217     listName = p[6]
218     size = int(p[3])
219     size1 = int(p[4])
220 
221     # Garante que o nome escolhido j   n o esta a ser usado
222     if listName not in parser.variaveis:
223         parser.variaveis[listName] = (parser.stackPointer, size, size1)
224         p[0] = f"PUSHN {size*size1}\n"
225         parser.stackPointer += size*size1
226     else:
227         parser.error = (
228             f"Vari vel com o nome {listName} j   definida anteriormente.")
229         parser.exito = False
230     parser.linhaDeCodigo +=1
231 
232 # Fun   o que altera o valor de um indice da matriz por outro
233 def p_AtribMatriz_comExpr(p):
234     "Atrib : ALTERA ID LPAREN_RETO expr RPAREN_RETO LPAREN_RETO expr RPAREN_RETO IGUAL
    expr PONTOVIRGULA" # altera mat[0][1] = 2;
235     matName = p[2]
236     indice1 = p[4]
237     indice2 = p[7]
238     valor = p[10]
239 
240     # Garante que a matriz existe
241     if matName in parser.variaveis:
242         if len(parser.variaveis[matName]) == 3:
243             p[0] = f"PUSHGP\nPUSHI {parser.variaveis[matName][0]}\nPADD\n{indice1}PUSHI {
    parser.variaveis[matName][2]}\nMUL\nPADD\n{indice2}{valor}STOREN\n"
244         else:
245             parser.error = f"Opera   o inv lida, vari vel {matName} n o    uma matriz"
246             parser.exito = False
247     else:
248         parser.error = f"Vari vel  n o declarada anteriormente"
249         parser.exito = False
250     parser.linhaDeCodigo +=1
251 
252 # Fun   o que vai buscar o valor do indice na matriz
253 def p_AtribBusca_Matriz(p):
254     "expr : ID LPAREN_RETO expr RPAREN_RETO LPAREN_RETO expr RPAREN_RETO"
       # para usar tipo x = mat[1][2]
255     varName = p[1]
256     indice1 = p[3]
257     indice2 = p[6]
258 
259     # garante que a matriz existe
260     if varName in parser.variaveis:
261         p[0] = f"PUSHGP\nPUSHI {parser.variaveis[varName][0]}\nPADD\n{indice1}PUSHI {
    parser.variaveis[varName][2]}\nMUL\nPADD\n{indice2}LOADN\n"
262     else:
263         parser.error = f"Vari vel com o nome {varName} n o definida"
264         parser.exito = False
265     parser.linhaDeCodigo +=1
266 
267 # Express o Aritm tica da soma
268 def p_PLUS(p):
269     "exprArit : exprArit PLUS term"
```

```
270     p[0] = f"{p[1]}{p[3]}ADD\n"
271
272 # Express o Aritm tica sub
273 def p_MINUS(p):
274     "exprArit : exprArit MINUS term"
275     p[0] = f"{p[1]}{p[3]}SUB\n"
276
277 # Expressao Atrim tica pode ser um termo apenas
278 def p_expr_arit_term(p):
279     '''exprArit : term'''
280     p[0] = p[1]
281
282 # Express o Aritm tica da mult
283 def p_TIMES(p):
284     "term : term TIMES factor"
285     p[0] = f"{p[1]}{p[3]}MUL\n"
286
287 # Express o Aritm tica da div
288 def p_DIVIDE(p):
289     "term : term DIVIDE factor"
290     p[0] = f"{p[1]}{p[3]}DIV\n"
291
292 # term pode ser um factor
293 def p_term(p):
294     "term : factor"
295     p[0] = p[1]
296
297 # um factor pode ser um numero
298 def p_factor_num(p):
299     '''factor : NUM'''
300     p[0] = f"PUSHI {int(p[1])}\n"
301
302 # um factor tmb pode ser uma variavel
303 def p_factor_id(p):
304     '''factor : ID'''
305     varName = p[1]
306
307     # garante que a var existe
308     if varName in parser.variaveis:
309         p[0] = f"PUSHG {parser.variaveis[varName][0]}\n"
310
311 # Express o Relativa do igual
312 def p_IGUALIGUAL(p):
313     "exprRel : expr IGUALIGUAL expr"
314     p[0] = f"{p[1]}{p[3]}EQUAL\n"
315
316 # Express o Relativa da diff (!=)
317 def p_DIFERENTE(p):
318     "exprRel : expr DIFERENTE expr"
319     p[0] = f"{p[1]}{p[3]}NOT\nEQUAL\n"
320
321 # Express o Relativa menor
322 def p_MENOR(p):
323     "exprRel : expr MENOR expr"
324     p[0] = f"{p[1]}{p[3]}INF\n"
325
326 # Express o Relativa menor ou igual
327 def p_MENOROUIGUAL(p):
328     "exprRel : expr MENOROUIGUAL expr"
```

```
329     p[0] = f"{p[1]}{p[3]}INFEQ\n"
330 
331 # Express o Relativa maior
332 def p_MAIOR(p):
333     "exprRel : expr MAIOR expr"
334     p[0] = f"{p[1]}{p[3]}SUP\n"
335 
336 # Express o Relativa maior ou igual
337 def p_MAIOROUIGUAL(p):
338     "exprRel : expr MAIOROUIGUAL expr"
339     p[0] = f"{p[1]}{p[3]}SUPEQ\n"
340 
341 # Express o Relativa EE (and)
342 def p_EE(p):
343     "exprRel : expr EE expr"
344     p[0] = f"{p[1]}{p[3]}ADD\nPUSHI 2\nEQUAL\n"
345 
346 # Express o Relativa "OU" (or)
347 def p_OU(p):
348     "exprRel : expr OU expr"
349     p[0] = f"{p[1]}{p[3]}ADD\nPUSHI 1\nSUPEQ\n"
350 
351 # Controlo de fluxo (if then)
352 def p_IF_THEN(p):
353     "IF : SE LPAREN exprRel RPAREN ENTAO Codigo FIM_COND PONTOVIRGULA"
354     p[0] = f"{p[3]}JZ l{parser.labels}\n{p[6]}l{parser.labels}: NOP\n"  # NOP n o faz
    nada na vm
355     parser.labels += 1
356     parser.linhaDeCodigo+=1
357 
358 # Controlo de fluxo (if then else)
359 def p_IF_THEN_ELSE(p):
360     "IF : SE LPAREN exprRel RPAREN ENTAO Codigo SENAO Codigo FIM_COND PONTOVIRGULA"
361     p[0] = f"{p[3]}JZ l{parser.labels}\n{p[6]}JUMP l{parser.labels}f\nl{parser.labels}:
    NOP\n{p[8]}l{parser.labels}f: NOP\n" # NOP n o faz nada na vm
362     parser.labels += 1
363     parser.linhaDeCodigo+=1
364 
365 # Ciclo While
366 def p_WHILE(p):
367     "WHILE : ENQUANTO LPAREN exprRel RPAREN FAZ Codigo FIM_ENQUANTO PONTOVIRGULA"
368     p[0] = f'l{parser.labels}c: NOP\n{p[3]}JZ l{parser.labels}f\n{p[6]}JUMP l{parser.
    labels}c\nl{parser.labels}f: NOP\n' # NOP n o faz nada na vm
369     parser.labels += 1
370     parser.linhaDeCodigo+=1
371 
372 # Print duma frase (string)
373 def p_PRINTAR_ID(p):
374     '''PRINTAR : IMPRIMIR LPAREN FRASE RPAREN PONTOVIRGULA'''
375     p[0] = f'PUSHS {p[3]}\nWRITES\n'
376     parser.linhaDeCodigo+=1
377 
378 # print do valor de uma variavel
379 def p_PRINTAR_var(p):
380     '''PRINTAR : IMPRIMIR LPAREN ID RPAREN PONTOVIRGULA'''
381     p[0] = f'PUSHG {parser.variaveis[p[3]][0]}\nWRITEI\n'
382     parser.linhaDeCodigo+=1
383 
384 # Codigo Comentado
```

```
385 def p_COMENTARIO(p):
386     '''COMENTARIO : COMENTADO'''
387     p[0] = "NOP\n" # NOP n o faz nada na vm
388 
389 def p_error(p):
390     print(f"Erro de sintaxe na entrada: {p.value}")
391 
392 
393 ##
394 parser = yacc.yacc()
395 parser.exito = True
396 parser.error = ""
397 parser.assembly = ""         # Codigo pra VM que vai sendo gerado
398 parser.variaveis = {}        # dicionario que armazena as variaveis declaradas
399 parser.stackPointer = 0      # necessario para a gest o da Stack da VM
400 parser.linhaDeCodigo = 0
401 parser.labels = 0
402 
403 assembly = ""            # codigo para a VM final
404 
405 
406 if len(sys.argv) == 2:
407     inputFileName = sys.argv[1]
408     if inputFileName[-4:] == ".plc":            # ficheiro de entrada precisa da extens o
     ".plc"
409         file = open(inputFileName, "r")
410         content = file.read()
411         parser.parse(content)
412         if parser.exito:
413             assembly += parser.assembly         # Atribui  o do codigo para a VM
414             print(parser.variaveis)
415         else:
416             print("Erro:")
417             print(parser.error)
418             print(parser.variaveis)
419             sys.exit()
420         file.close()
421         outputFileName = "a.vm"                 # cria um file chamado "a.vm" para rodar
    na VM
422 
423         # Verifica se o arquivo de sa da j  existe e o remove antes de criar um novo
424         if os.path.exists(outputFileName):
425             os.remove(outputFileName)
426 
427         outputFile = open(outputFileName, "w")
428         outputFile.write(assembly)              # Escrita do codigo para a VM Final
429         outputFile.close()
430 
431         print("File saved successfully")
432 
433     else:
434         print("Invalid file extension")
```