



Universidade do Minho
Escola de Engenharia

Processamento de Linguagens

Trabalho Prático 2

João Pedro da Santa Guedes A89588
Luís Pedro Oliveira de Castro Vieira A89601



A89588



A89601

30 de maio de 2021

Conteúdo

| | | |
|----------|--|-----------|
| 1 | Introdução | 2 |
| 2 | Estrutura do Relatório | 3 |
| 3 | Descrição do Problema | 4 |
| 4 | Formulação do Trabalho Prático | 5 |
| 4.1 | Linguagem Definida | 5 |
| 4.2 | Tokens | 6 |
| 4.3 | Gramática | 7 |
| 4.3.1 | Programa Principal | 7 |
| 4.3.2 | Bloco de Declarações | 8 |
| 4.3.3 | Bloco de Instruções | 9 |
| 4.4 | Expressões Aritméticas e Gerais | 12 |
| 5 | Controlo de Erros | 15 |
| 6 | Programas de Teste | 16 |
| 6.1 | Lados do Quadrado | 17 |
| 6.2 | Menor de N Inteiros | 19 |
| 6.3 | Produtório de N Números | 20 |
| 6.4 | Contabilização e Impressão dos Números Ímpares Naturais de uma Sequência | 21 |
| 6.5 | Leitura, Armazenamento e Impressão por Ordem Inversa dos Números de um Array | 22 |
| 7 | Conclusão | 24 |
| 8 | Apêndicas | 25 |
| 8.1 | vim_tokens.py | 25 |
| 8.2 | vim_yacc.py | 27 |

Listings

| | | |
|----|---|----|
| 1 | Exemplo da Linguagem Implementada | 6 |
| 2 | Programa Principal | 7 |
| 3 | Bloco de Declarações | 8 |
| 4 | Declaração de variáveis | 8 |
| 5 | Bloco de Instruções | 9 |
| 6 | Gramática para Instruções de Leitura/Escrita/Atribuição | 10 |
| 7 | Atribuição a Arrays de 1 ou 2 dimensões | 10 |
| 8 | CondicionaI IF | 11 |
| 9 | CondicionaI IF then ELSE | 11 |
| 10 | Condicionais | 11 |
| 11 | Ciclo repeat-until | 12 |
| 12 | Ciclo while-do | 12 |
| 13 | Expressões Aritméticas Básicas e Variáveis | 12 |
| 14 | Lados do Quadrado na Linguagem Definida | 17 |
| 15 | Código Máquina Gerado | 17 |
| 16 | Menor de N números na Linguagem Definida | 19 |
| 17 | Código Máquina Gerado | 19 |
| 18 | Produtórios de N Números na Linguagem Definida | 20 |
| 19 | Código Máquina Gerado | 20 |
| 20 | Contabilização e Impressão dos Números Ímpares na Linguagem Definida | 21 |
| 21 | Leitura de Inteiros num Array e Impressão Inversa na Linguagem Definida | 22 |
| 22 | Código Máquina Gerado | 22 |
| 23 | Código presente no Lex | 25 |
| 24 | Código do Parser | 27 |

1 Introdução

No âmbito da Unidade Curricular de Processamento de Linguagens, foi proposto ao grupo, como forma de desenvolver e demonstrar os seus conhecimentos sobre aquilo que tem vindo a ser alvo de atenção na UC, a realização de um exercício que prevalece na capacidade de escrever gramáticas, quer independentes do contexto (GIC) , quer tradutoras (GT). O problema tem como objetivo o desenvolvimento de um compilador que seja capaz de gerar código para uma máquina de stack virtual.

De modo a concretizarmos este problema demonstraremos também o nosso conhecimento com o uso de *geradores de compiladores* baseados em gramáticas tradutoras, o ***Yacc*** do **PLY** do **Python**, bem como o *gerador de analisadores léxicos* **Lex**, também ele do **PLY** do **Python**.

Assim, no presente relatório, passaremos a descrever o processo de resolução do problema, bem como explicar o código implementado para o mesmo.

2 Estrutura do Relatório

- A primeira parte corresponde à introdução, que irá retratar qual o objetivo do trabalho.
- A segunda parte corresponde ao presente tópico onde será explicado como está organizado este documento, referindo os capítulos existentes e explicando o conteúdo de cada um.
- A terceira parte corresponde à análise e especificação. É neste capítulo que é feita uma análise detalhada do problema proposto de modo a se poder especificar o desenvolvimento do mesmo.
- No quarto capítulo será apresentado e explicado o raciocínio por trás da resolução do problema, usando como auxiliar o código implementado.
- Na quinta parte serão apresentados exemplos do programa através de escritas de código.
- No sexto capítulo, irá constar a conclusão que irá conter uma síntese do que foi retratado ao longo do documento, a conclusão e uma análise crítica do trabalho realizado.
- No sétimo e último capítulo constará o código produzido durante a resolução do trabalho.

3 Descrição do Problema

No presente trabalho pretende-se que, através da definição de uma linguagem de programação imperativa simples, a nosso gosto, sejamos capazes de, fazendo uso das ferramentas *Lex* e *Yacc* do **PLY**, gerar pseudo-código, Assembly da Máquina Virtual VM disponibilizada pela equipa docente.

Devemos ter em consideração que a linguagem estabelecida deverá ser capaz de concretizar os seguintes pontos:

- 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 *standart input* e escrever no *standart output*.
- Efetuar *instruções condicionais* para controlo do fluxo de execução.
- Efetuar *instruções cíclicas* para controlo do fluxo de execução, permitindo o seu aninhamento. No caso do nosso grupo, devemos ser capazes de implementar pelo meno o ciclo ***repeat-until***.

Para além dos pontos acima mencionados, deveremos ainda ser capazes de implementar uma das duas funcionalidades seguintes, ficando a nosso critério a escolha:

- Declarar e manusear variáveis estruturadas do tipo ***array*** (a 1 ou 2 dimensões) ***de inteiro***, 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.

4 Formulação do Trabalho Prático

Tendo já em conta o trabalho a ser implementado é agora importante ter em conta outros fatores mencionados no enunciado tais como:

- As variáveis deverão ser declaradas no início do programa, não podendo haver re-declarações, nem utilizações sem declaração prévia.
- Se nada for explicitado, o valor da variável após a declaração é 0 (zero).

Para além disso tínhamos também que tomar uma decisão relativamente à funcionalidade adicional a implementar. Depois de uma discussão sobre o assunto entre os elementos do grupo, chegámos à conclusão que seria de maior interesse sermos capazes de **declarar e manusear arrays de inteiros (a 1 ou 2 dimensões)**.

Tendo agora o necessário para a realização do trabalho, passaremos a explicar nos tópicos que se seguem o raciocínio e decisões por nós tomadas ao longo do trabalho.

4.1 Linguagem Definida

Enquanto grupo debatemos várias possibilidades de linguagens a adotar, tendo sempre em consideração que deveria ser algo simples e fácil de entender.

Acabámos por adotar a linguagem que a seguir apresentaremos, representada através das instruções:

- *DECL* e *ENDDECL* - delimitam o bloco de declarações de variáveis.
- *int* - declaração de um inteiro.
- *int[]* - declaração de um array de inteiros de uma dimensão.
- *int[][]* - declaração de um array de inteiros de duas dimensões.
- *INSTR* e *ENDINSTR* - delimitam o bloco de instruções.
- *IF* e *ENDIF* - delimitam um bloco condicional.
- *ELSE* e *ENDELSE* - delimitam um bloco alternativo ao bloco condicional.
- *REPEAT* e *UNTIL* - delimitam o ciclo *repeat-until*.
- *WHILE* e *DO* - delimitam o ciclo *while-do*.
- *print()* - imprime no standart output.
- *input()* - lê do standart input.
- *AND* - representa a operação lógica 'E', tal como em Python.
- *OR* - representa a operação lógica 'OU', tal como em Python.

- *NOT* - representa a negação de uma condição, tal como em Python.

De notar que quer no bloco de declarações, quer no bloco de instruções para que a sintaxe seja correta, deve-se terminar quer as declarações, quer as instruções com ';' (ponto e vírgula).

Tendo tudo isto em conta, um exemplo da utilização da nossa linguagem para, por exemplo, a declaração de uma variável e posterior atribuição de um valor, bem como a sua escrita no standart output, seria:

```
1 DECL
2     int a;
3 ENDDECL
4 INSTR
5     a=3;
6     print(a);
7 ENDINSTR
```

Listing 1: Exemplo da Linguagem Implementada

Estando a linguagem por nós definida, passamos agora à implementação da *gerador de analisadores léxicos* e da *gramática tradutora*.

4.2 Tokens

De modo a identificar-mos e sermos capazes de representar com eficiência as diferentes funcionalidades e blocos dentro do nosso programa é necessário definirmos alguns tokens, literais e palavras reservadas, de modo a não gerar alguns conflitos com certas definições de outros tokens.

Assim, no trabalho fazemos uso dos seguintes literais:

- '(' e ')'
- '[' e ']'
- ','
- '.,'

Dos seguintes tokens:

- ID - estando definido como '[a-zA-Z_][a-zA-Z0-9_]*'
- ADD - estando definido como '\+'
- SUB - estando definido como '\-'
- MUL - estando definido como '*'
- DIV - estando definido como '/'
- MOD - estando definido como '%'

- EQUIVALENT - estando definido como '=='
- DIFFERENT - estando definido como '!='
- GREATEREQUAL - estando definido como '>='
- GREATER - estando definido como '>'
- LESSEREQUAL - estando definido como '<='
- LESSER - estando definido como '<'
- EQUALS - estando definido como '='
- NUM - estando definido como '?\d+'

Os tokens encontram-se definidos por esta ordem dada a precedência dos mesmos.

Como mencionado anteriormente, existem certas palavras reservadas por causa de conflitos existentes com outros tokens, nomeadamente com o ID, que dada a sua definição, o parser poderia captar primeiro como um ID ao invés do token que desejamos. Isto poderia ser também tratado com a alteração da precedência, mas o uso de palavras reservadas facilita e agiliza o processo.

As palavras reservadas são as que constam na definição da linguagem imperativa implementada pelo grupo.

4.3 Gramática

Já estando os tokens definidos e apresentados, passaremos de seguida a demonstrar a nossa gramática e como o código da nossa linguagem é traduzido para o código máquina VM. De maneira a melhor entender a forma como se encontra estruturada, a gramática será apresentada em diferentes secções que considerámos críticas e essenciais ao trabalho.

É importante notar que se trata de uma gramática LR, ou seja, Bottom-Up, de forma a podermos usufruir ao máximo das funcionalidades do PLY.

4.3.1 Programa Principal

Apesar de se tratar de somente uma produção, é nesta que estão estabelecidos os conteúdos que serão escritos em código máquina a ser interpretado pela VM. Assim temos que:

```

1 def p_Programa(p):
2     "Programa : Decl Instr"
3     p[0] = f'{p[1]}start\n{p[2]}stop'
```

Listing 2: Programa Principal

Na produção acima mencionada podemos constatar que, tal como mencionado, a declaração de variáveis é feita antes das instruções do nosso programa. Assim o código gerado para máquina será sempre: declarações + 'start' + instruções + 'stop'.

4.3.2 Bloco de Declarações

Como forma de identificar o início e o fim do bloco de declarações fazemos uso dos tokens 'DECL' e 'ENDECL', estando dentro destes as declarações de inteiros, arrays de inteiros de 1 ou 2 dimensões, podendo estas ter ou não a atribuição de uma expressão, ou então não existir qualquer declaração.

```
1 def p_Decl(p):
2     "Decl : DECL Declaracoes ENDECL "
3     p[0] = p[2]
4
5 def p_Declaracoes(p):
6     "Declaracoes : Declaracoes Declaracao "
7     p[0] = p[1] + p[2]
8
9 def p_Declaracoes_Empty(p):
10    "Declaracoes : "
11    p[0] = ""
12
13 def p_Declaracao_Variaveis(p):
14    "Declaracao : int Variaveis ',' "
15    p[0] = p[2]
16
17 def p_Declaracao_Arrays(p):
18    "Declaracao : int Array ',' "
19    p[0] = p[2]
```

Listing 3: Bloco de Declarações

Para o armazenamento das variáveis para consulta posterior, fazemos uso de um dicionário onde guardamos como key o ID da variável e como valor o offset da máquina em que foi guardado. No caso de um inteiro depois de adicionado à stack o offset é aumentado em 1, no caso de um array de uma dimensão é aumentado pelo seu tamanho e no caso de um array de duas dimensões é aumentado pelo produto do tamanho das linhas com o tamanho das colunas.

```
1 def p_Variaveis(p):
2     "Variaveis : Variaveis ',' Variavel "
3     p[0] = p[1] + p[3]
4
5 def p_Variaveis_Simples(p):
6     "Variaveis : Variavel "
7     p[0] = p[1]
8
9 def p_Variavel_Vars(p):
10    "Variavel : Var"
11    p[0] = p[1]
12
13 def p_Variavel_VarsDeclaradas(p):
14    "Variavel : VarDeclarada"
15    p[0] = p[1]
16
17 def p_Array_Uma_Dimensao(p):
18    "Array : ArrayDimensaoSimples"
19    p[0] = p[1]
```

```

20
21 def p_Array_Duas_Dimensoes(p):
22     "Array : ArrayDimensaoDupla"
23     p[0] = p[1]
24
25 def p_ArrayDimensaoSimples(p):
26     "ArrayDimensaoSimples : '[' NUM ']' ID"
27     p[0] = 'pushn ' + p[2] + '\n'
28     p.parser.registers[p[4]] = p.parser.registerindex
29     p.parser.registerindex += int(p[2])
30
31 def p_ArrayDimensaoDupla(p):
32     "ArrayDimensaoDupla : '[' NUM ']' '[' NUM ']' ID"
33     p[0] = 'pushn ' + str(int(p[2])*int(p[5])) + '\n'
34     p.parser.registers[p[7]] = p.parser.registerindex
35     p.parser.registerindex += int(p[2])*int(p[5])
36     p.parser.matrizes[p[7]] = int(p[5])
37
38 def p_Var_ID(p):
39     "Var : ID"
40     p[0] = 'pushi 0\n'
41     p.parser.registers[p[1]] = p.parser.registerindex
42     p.parser.registerindex += 1
43
44 def p_VarDeclarada(p):
45     "VarDeclarada : ID EQUALS Exp"
46     p[0] = p[3]
47     p.parser.registers[p[1]] = p.parser.registerindex
48     p.parser.registerindex += 1

```

Listing 4: Declaração de variáveis

4.3.3 Bloco de Instruções

Já para o bloco de instruções, utilizamos os tokens 'INSTR' e 'ENDINSTR' para delimitar o bloco de instruções, sendo que dentro deste pode não existir qualquer instrução, ou existir uma ou mais das seguintes instruções:

- Atribuição do valor de expressões numéricas a variáveis.
- Ler do standart input.
- Escrever no standart output.
- Efetuar instruções condicionais.
- Efetuar instruções cíclicas.

```

1 def p_Instr(p):
2     "Instr : INSTR Instrucoes ENDINSTR "
3     p[0] = p[2]
4

```

```

5 def p_Instrucoes(p):
6     "Instrucoes : Instrucoes Instrucao"
7     p[0] = p[1] + p[2]
8
9 def p_Instrucoes_Empty(p):
10    "Instrucoes : "
11    p[0] = ""

```

Listing 5: Bloco de Instruções

A leitura do standart input, escrita no standart output e atribuição de expressões a variáveis simples são algo relativamente simples e fazem apenas uso dos tokens respetivos.

```

1 def p_Instrucao_Print(p):
2     "Instrucao : print '(' Exp ')', ';' "
3     p[0] = p[3] + 'writei' + '\n'
4
5 def p_Instrucao_Read(p):
6     "Instrucao : input '(' ID ')', ';' "
7     if p[3] in p.parser.registers:
8         p[0] = 'read\atoi\n' + 'storeg' + str(p.parser.registers.get(p[3])) +
9         '\n'
10    else:
11        raise Exception
12
13 def p_Instrucao_Atrib(p):
14     "Instrucao : ID EQUALS Exp ';'"
15     if p[1] in p.parser.registers:
16         p[0] = p[3] + 'storeg' + str(p.parser.registers.get(p[1])) + '\n'
17     else:
18         raise Exception

```

Listing 6: Gramática para Instruções de Leitura/Escreita/Atribuição

Já a atribuição de expressões a posições de arrays de uma ou duas dimensões é algo mais complexo, uma vez que para um array de uma dimensão temos de aceder ao offset do índice indicado, somando esse valor ao offset base do array; para aceder a uma posição de um array de duas dimensões devemos multiplicar o índice presente na linha pelo tamanho da coluna do array, somando o índice presente na coluna.

```

1 def p_Instrucao_Atrib_Array(p):
2     "Instrucao : ID '[' Exp ']' EQUALS Exp ';'"
3     if p[1] in p.parser.registers:
4         p[0] = 'pushgp\npushi' + str(p.parser.registers.get(p[1])) + '\npadd\n'
5         + p[3] + p[6] + 'storen\n'
6     else:
7         raise Exception
8
9 def p_Instrucao_Atrib_Matrix(p):
10    "Instrucao : ID '[' Exp ']' '[' Exp ']' EQUALS Exp ';'"
11    if p[1] in p.parser.registers:
12        p[0] = 'pushgp\npushi' + str(p.parser.registers.get(p[1])) + '\npadd\n'
13        + p[6] + p[3] + 'pushi' + str(p.parser.matrizes[p[1]]) + '\nmul\nadd\n' +
14        p[9] + 'storen\n'
15    else:
16        raise Exception

```

```
13 raise Exception
```

Listing 7: Atribuição a Arrays de 1 ou 2 dimensões

As condicionais são delimitadas pelos tokens 'IF' e 'ENDIF'. Seguido do 'IF' estão as condições delimitadas por parêntesis. Caso cumpra com as condições, então no seu interior tem o bloco de instruções que serão realizadas.

```
1 def p_Instrucao_If(p):
2     "Instrucao : IF '(' Conds ')' Instrucoes ENDIF "
3     p[0] = f'{p[3]}jz endif{p.parser.ifs}\n{p[5]}endif{p.parser.ifs}:\n'
4     p.parser.ifs += 1
```

Listing 8: Condicional IF

Existem também as condicionais do tipo *if then else*, sendo o bloco das instruções do else delimitado pelos tokens 'ELSE' e 'ENDELSE', o qual se encontra dentro do bloco 'IF' e 'ENDIF'.

```
1 def p_Instrucao_If_Else(p):
2     "Instrucao : IF '(' Conds ')' Instrucoes Else ENDIF "
3     p[0] = f'{p[3]}jz else{p.parser.elses}\n{p[5]}jump endelse{p.parser.elses}\nelse{p.parser.elses}:\n{p[6]}jump endelse{p.parser.elses}\nendelse{p.parser.elses}:\n'
4     p.parser.elses += 1
5
6 def p_Else(p):
7     "Else : ELSE Instrucoes ENDELSE"
8     p[0] = p[2]
```

Listing 9: Condicional IF then ELSE

Existem diversas operações relacionais a ter em consideração, maior, maior ou igual, menor, menor ou igual, equivalente ou diferente, bem como as operações lógicas, 'E', 'OU' e 'NÃO'. Assim, estas são as definições da gramática que permitem controlar o fluxo dos blocos condicionais, podendo haver uma ou mais condições.

```
1
2 def p_Conds_And(p):
3     "Conds : Conds AND Cond"
4     p[0] = p[1] + p[3] + 'add\npushi 2\nnequal\n'
5
6 def p_Conds_Or(p):
7     "Conds : Conds OR Cond"
8     p[0] = p[1] + p[3] + 'add\npushi 0\nsup\n'
9
10 def p_Conds_Unica(p):
11     "Conds : Cond"
12     p[0] = p[1]
13
14 def p_Cond_Not(p):
15     "Cond : NOT '(' Conds ')'"
16     p[0] = p[3] + 'pushi 1\ninf\n'
17
18 def p_Cond_Equivalent(p):
19     "Cond : Exp EQUIVALENT Exp"
20     p[0] = p[1] + p[3] + 'equal\n'
```

```

21
22 def p_Cond_Different(p):
23     "Cond : Exp DIFFERENT Exp"
24     p[0] = p[1] + p[3] + 'equal\nnot\n'
25
26 def p_Cond_Greater(p):
27     "Cond : Exp GREATER Exp"
28     p[0] = p[1] + p[3] + 'sup\n'
29
30 def p_Cond_Greater_Equal(p):
31     "Cond : Exp GREATEREQUAL Exp"
32     p[0] = p[1] + p[3] + 'supeq\n'
33
34 def p_Cond_Lesser(p):
35     "Cond : Exp LESSER Exp"
36     p[0] = p[1] + p[3] + 'inf\n'
37
38 def p_Cond_Lesser_Equal(p):
39     "Cond : Exp LESSEREQUAL Exp"
40     p[0] = p[1] + p[3] + 'ineq\n'

```

Listing 10: Condicionais

Já o ciclo *repeat-until* é delimitado pelos tokens 'REPEAT' e 'UNTIL' sendo a condição colocada a seguir a este último, também ela entre parêntesis. No interior dos tokens delimitadores, encontra-se o conjunto de instruções a executar.

```

1 def p_Instrucao_Repeat_Until(p):
2     "Instrucao : REPEAT Instrucoes UNTIL '(' Conds ')'"
3     p[0] = f'r{p.parser.ciclos}:\n{p[2]}{p[5]}jz r{p.parser.ciclos}\n'
4     p.parser.ciclos += 1

```

Listing 11: Ciclo repeat-until

Adicionalmente, elaborámos também a gramática tradutora para o ciclo *while-do*, o qual é delimitado pelos tokens 'WHILE', sendo a condição do ciclo colocada a seguir a este dentro de parêntesis, e 'ENDWHILE', sendo que a seguir ao token 'DO' se encontra o bloco de instruções a executar.

```

1 def p_Instrucao_While_Do(p):
2     "Instrucao : WHILE '(' Conds ')'" DO Instrucoes ENDWHILE"
3     p[0] = f'while{p.parser.ciclos}:\n{p[3]}jz fimwhile{p.parser.ciclos}\n{p[6]}
4     jump while{p.parser.ciclos}\nfimwhile{p.parser.ciclos}:\n'
5     p.parser.ciclos += 1

```

Listing 12: Ciclo while-do

4.4 Expressões Aritméticas e Gerais

Na presente secção serão apresentadas as produções da gramática para operações aritméticas básicas como adição, subtração, multiplicação, divisão e módulo, bem como o estabelecimento de IDs, NUMs e arrays de 1 ou 2 dimensões.

```

1

```

```

2 def p_Exp_Termo_add(p):
3     "Exp : Exp ADD Termo"
4     p[0] = p[1] + p[3] + 'add\n'
5
6 def p_Exp_Termo_sub(p):
7     "Exp : Exp SUB Termo"
8     p[0] = p[1] + p[3] + 'sub\n'
9
10 def p_Exp_Termo(p):
11     "Exp : Termo"
12     p[0] = p[1]
13
14 def p_Termo_Fator_mul(p):
15     "Termo : Termo MUL Fator"
16     p[0] = p[1] + p[3] + 'mul\n'
17
18 def p_Termo_Fator_div(p):
19     "Termo : Termo DIV Fator"
20     if (p[3] != 'pushi 0\n'):
21         p[0] = p[1] + p[3] + 'div\n'
22     else:
23         p[0] = 'pushi 0\n'
24
25 def p_Termo_Fator_mod(p):
26     "Termo : Termo MOD Fator"
27     if (p[3] != 'pushi 0\n'):
28         p[0] = p[1] + p[3] + 'mod\n'
29     else:
30         p[0] = 'pushi 0\n'
31
32 def p_Termo_Fator(p):
33     "Termo : Fator"
34     p[0] = p[1]
35
36 def p_Fator_ID(p):
37     "Fator : ID"
38     if p[1] in p.parser.registers:
39         p[0] = 'pushg ' + str(p.parser.registers.get(p[1])) + '\n'
40     else:
41         raise Exception
42
43 def p_Fator_ID_Array(p):
44     "Fator : ID '[' Exp ']' "
45     if p[1] in p.parser.registers:
46         p[0] = 'pushgp\npushi ' + str(p.parser.registers.get(p[1])) + '\npadd\n'
47         + p[3] + 'loadn\n'
48     else:
49         raise Exception
50
51 def p_Fator_ID_Matrix(p):
52     "Fator : ID '[' Exp ']' '[' Exp ']' "
53     if p[1] in p.parser.registers:
54         p[0] = 'pushgp\npushi ' + str(p.parser.registers.get(p[1])) + '\npadd\n'
55         + p[6] + p[3] + 'pushi ' + str(p.parser.matrizes[p[1]]) + '\nmul\nadd\n' + '

```

```

loadn\n'
54     else:
55         raise Exception
56
57 def p_Fator_num(p):
58     "Fator : NUM"
59     p[0] = 'pushi ' + p[1] + '\n'
60 def p_Fator_Exp(p):
61     "Fator : '(' Exp ')'"
62     p[0] = p[2]

```

Listing 13: Expressões Aritméticas Básicas e Variáveis

5 Controlo de Erros

Como forma de controlar os erros introduzidos na escrita da linguagem por nós definida, testámos escrever erradamente de forma propositada para perceber o tipo de erro lançado pelo interpretador do Python.

Assim chegámos à conclusão que a exceção *TypeError* é aquela que é lançada

Para além destes, fazemos também o controlo de uso de variáveis não declaradas nas instruções, algo que considerámos que não deva acontecer, nomeadamente porque no ficheiro de output que seria introduzido na máquina essa variável aparecerá como *None*.

Desta forma, quando um destes erros é apanhado é lançada uma mensagem de erro para o utilizador e o ficheiro onde seria escrito o código máquina é apagado, Como forma de contrariar possíveis escritas erradas do código da máquina VM, e como forma de preservar o nosso compilador

6 Programas de Teste

Como forma de testar o trabalho é necessário, estando explícito no enunciado, o desenvolvimento de um conjunto de testes que demonstrem o código Assembly gerado bem como o programa a correr na máquina virtual VM. Estes testes deverão ser escritos em programas-fontes redigidos na linguagem por nós definida.

O conjunto de testes a desenvolver é então:

- Ler 4 números e dizer se podem ser os lados de um quadrado.
- Ler um inteiro N , depois ler N números e escrever o menor deles.
- Ler N (constante do programa) números e calcular e imprimir o seu produtório.
- Contar e imprimir os números ímpares de uma sequência de números naturais.

Tendo em conta a opção acima escolhida das funcionalidades adicionais devemos também escolher um outro teste a desenvolver de entre os seguintes:

- Ler e armazenar N números num array; imprimir os valores por ordem inversa.
- Invocar e usar num programa nosso uma função "potencia()", que começar por le do input a base B e o expoente E e retorna o valor B^E .

Uma vez que a nossa escolha foi a capacidade de declarar e manusear arrays de inteiros, o teste a desenvolver adicionalmente será o primeiro acima mencionado.

Para cada teste a seguir mencionado, apresentaremos o código do programa-fonte escrito na nossa linguagem, bem como o resultado da sua compilação com o nosso trabalho.

6.1 Lados do Quadrado

Objetivo: Ler 4 números e dizer se podem ser os lados de um quadrado.

Teste na Linguagem Definida:

```
1 DECL
2     int a,b,c,d;
3 ENDDECL
4 INSTR
5     input(a);
6     input(b);
7     input(c);
8     input(d);
9     IF(a==b AND b==c AND c==d)
10        print(1);
11    ELSE
12        print(0);
13    ENDELSE
14    ENDIF
15 ENDINSTR
```

Listing 14: Lados do Quadrado na Linguagem Definida

Código Máquina Gerado:

```
1 pushi 0
2 pushi 0
3 pushi 0
4 pushi 0
5 start
6 read
7 atoi
8 storeg 0
9 read
10 atoi
11 storeg 1
12 read
13 atoi
14 storeg 2
15 read
16 atoi
17 storeg 3
18 pushg 0
19 pushg 1
20 equal
21 pushg 1
22 pushg 2
23 equal
24 add
25 pushi 2
26 equal
27 pushg 2
28 pushg 3
29 equal
```

```
30 add
31 pushi 2
32 equal
33 jz else0
34 pushi 1
35 writei
36 jump endelse0
37 else0:
38 pushi 0
39 writei
40 jump endelse0
41 endelse0:
42 stop
```

Listing 15: Código Máquina Gerado

6.2 Menor de N Inteiros

Objetivo: Ler um inteiro N, depois ler N números e escrever o menor deles.

Teste na Linguagem Definida:

```
1 DECL
2     int n,num;
3     int min=99999999;
4 ENDDECL
5 INSTR
6     input(n);
7     REPEAT
8         input(num);
9         IF(num<min)
10             min = num;
11         ENDIF
12     n = n-1;
13 UNTIL (n==0)
14     print(min);
15 ENDINSTR
```

Listing 16: Menor de N números na Linguagem Definida

Código Máquina Gerado:

```
1 pushi 0
2 pushi 0
3 pushi 99999999
4 start
5 read
6 atoi
7 storeg 0
8 r0:
9 read
10 atoi
11 storeg 1
12 pushg 1
13 pushg 2
14 inf
15 jz endif0
16 pushg 1
17 storeg 2
18 endif0:
19 pushg 0
20 pushi 1
21 sub
22 storeg 0
23 pushg 0
24 pushi 0
25 equal
26 jz r0
27 pushg 2
28 writei
29 stop
```

Listing 17: Código Máquina Gerado

6.3 Produtório de N Números

Objetivo: Ler N (constante do programa) números e calcular e imprimir o seu produtório.
Teste na Linguagem Definida:

```
1 DECL
2     int n,num;
3     int prod=1;
4 ENDDECL
5 INSTR
6     input(n);
7     REPEAT
8         input(num);
9         prod = prod * num;
10        n = n-1;
11    UNTIL (n==0)
12    print(prod);
13 ENDINSTR
```

Listing 18: Produtórios de N Números na Linguagem Definida

Código Máquina Gerado:

```
1 pushi 0
2 pushi 0
3 pushi 1
4 start
5 read
6 atoi
7 storeg 0
8 r0:
9 read
10 atoi
11 storeg 1
12 pushg 2
13 pushg 1
14 mul
15 storeg 2
16 pushg 0
17 pushi 1
18 sub
19 storeg 0
20 pushg 0
21 pushi 0
22 equal
23 jz r0
24 pushg 2
25 writei
26 stop
```

Listing 19: Código Máquina Gerado

6.4 Contabilização e Impressão dos Números Ímpares Naturais de uma Sequência

Objetivo: Contar e imprimir os números ímpares de uma sequência de números naturais.
Teste na Linguagem Definida:

```
1 DECL
2     int count;
3     int num=1;
4 ENDDECL
5 INSTR
6     REPEAT
7         input(num);
8         IF(num%2!=0)
9             count = count + 1;
10            print(num);
11        ENDIF
12    UNTIL (num==0)
13    print(count);
14 ENDINSTR
```

Listing 20: Contabilização e Impressão dos Números Ímpares na Linguagem Definida

Código Máquina Gerado:

```
1 pushi 0
2 pushi 1
3 start
4 r0:
5 read
6 atoi
7 storeg 1
8 pushg 1
9 pushi 2
10 mod
11 pushi 0
12 equal
13 not
14 jz endif0
15 pushg 0
16 pushi 1
17 add
18 storeg 0
19 pushg 1
20 writei
21 endif0:
22 pushg 1
23 pushi 0
24 equal
25 jz r0
26 pushg 0
27 writei
28 stop
```

6.5 Leitura, Armazenamento e Impressão por Ordem Inversa dos Números de um Array

Objetivo: Ler e armazenar N números num array; imprimir os valores por ordem inversa.

Teste na Linguagem Definida:

```
1 DECL
2     int[5] valores;
3     int valor,i;
4     int max=5;
5 ENDDECL
6 INSTR
7     REPEAT
8         input(valor);
9         valores[i] = valor;
10        i = i+1;
11    UNTIL (i==max)
12    WHILE(max>0)
13    DO
14        print(valores[max]);
15        max = max-1;
16    ENDWHILE
17 ENDINSTR
```

Listing 21: Leitura de Inteiros num Array e Impressão Inversa na Linguagem Definida

Código Máquina Gerado:

```
1 pushn 5
2 pushi 0
3 pushi 0
4 pushi 5
5 start
6 r0:
7 read
8 atoi
9 storeg 5
10 pushgp
11 pushi 0
12 padd
13 pushg 6
14 pushg 5
15 storen
16 pushg 6
17 pushi 1
18 add
19 storeg 6
20 pushg 6
21 pushg 7
22 equal
23 jz r0
24 r1:
25 pushgp
26 pushi 0
27 padd
```



```
28 pushg 7
29 loadn
30 writei
31 pushg 7
32 pushi 1
33 sub
34 storeg 7
35 pushg 7
36 pushi 0
37 inf
38 jz r1
39 stop
```

Listing 22: Código Máquina Gerado

7 Conclusão

A elaboração deste trabalho ajudou a aprofundar o conhecimento com as ferramentas Yacc e Lex do PLY, desenvolvendo assim bases e conhecimento que era o objetivo desta cadeira. Apesar de algumas dificuldades e erros que nos apareceram, conseguimos sempre rever, descobrir qual era o erro, o porquê isso de acontecer, o que nos fez procurar sempre ultrapassar e tentar fazer o melhor que conseguíssemos.

Após muito empenho e dedicação, pensamos que o projeto corresponde às expectativas, na medida em que conseguimos desenvolver um trabalho que satisfaz a todos os requisitos.

No entanto existe sempre espaço para melhorar e otimizar certos aspetos do trabalho, bem como acrescentar funcionalidades adicionais e expandir o mesmo para, por exemplo, ser capaz de interpretar informação como floats e strings, ou realizar um ciclo *for-do*.

Desta forma, depois de concluirmos o trabalho, conseguimos reconhecer que o nosso conhecimento no que diz respeito à construção de gramáticas e das ferramentas Lex e Yacc cresceu exponencialmente, na medida em que nos consideramos bastante aptos e com boas bases para desenvolver eventuais projetos futuros que possam aparecer.

8 Apêndicas

8.1 vim_tokens.py

Segue em anexo o código presente no ficheiro referente à ferramenta Lex do PLY.

```
1 import ply.lex as lex
2
3 reserved = {
4     'DECL' : 'DECL',
5     'ENDDECL' : 'ENDDECL',
6     'int' : 'int',
7     'INSTR' : 'INSTR',
8     'ENDINSTR' : 'ENDINSTR',
9     'print' : 'print',
10    'input' : 'input',
11    'IF' : 'IF',
12    'ENDIF' : 'ENDIF',
13    'ELSE' : 'ELSE',
14    'ENDELSE' : 'ENDELSE',
15    'AND' : 'AND',
16    'OR' : 'OR',
17    'REPEAT' : 'REPEAT',
18    'UNTIL' : 'UNTIL',
19    'WHILE' : 'WHILE',
20    'DO' : 'DO',
21    'ENDWHILE' : 'ENDWHILE',
22    'NOT' : 'NOT'
23 }
24
25 literals = ['(', ')', ';', ',', '[', ']']
26
27 tokens = ['NUM', 'ID',
28          'ADD', 'SUB', 'MUL', 'DIV', 'MOD',
29          'GREATER', 'LESSER', 'GREATEREQUAL', 'LESSEREQUAL', 'EQUIVALENT',
30          'DIFFERENT', 'EQUALS'] + list(reserved.values())
31
32 def t_ID(t):
33     r'[a-zA-Z_][a-zA-Z0-9_]*'
34     t.type = reserved.get(t.value, 'ID')
35     return t
36
37 def t_ADD(t):
38     r'\+'
39     t.type = reserved.get(t.value, 'ADD')
40     return t
41
42 def t_SUB(t):
43     r'\-'
44     t.type = reserved.get(t.value, 'SUB')
45     return t
46
47 def t_MUL(t):
48     r'\*'
49     t.type = reserved.get(t.value, 'MUL')
50     return t
```

```

48     t.type = reserved.get(t.value, 'MUL')
49     return t
50
51 def t_DIV(t):
52     r '/'
53     t.type = reserved.get(t.value, 'DIV')
54     return t
55
56 def t_MOD(t):
57     r '%'
58     t.type = reserved.get(t.value, 'MOD')
59     return t
60
61 def t_EQUIVALENT(t):
62     r '=='
63     t.type = reserved.get(t.value, 'EQUIVALENT')
64     return t
65
66 def t_DIFFERENT(t):
67     r '!='
68     t.type = reserved.get(t.value, 'DIFFERENT')
69     return t
70
71 def t_GREATEREQUAL(t):
72     r '>='
73     t.type = reserved.get(t.value, 'GREATEREQUAL')
74     return t
75
76 def t_GREATER(t):
77     r '>'
78     t.type = reserved.get(t.value, 'GREATER')
79     return t
80
81 def t_LESSEQUAL(t):
82     r '<='
83     t.type = reserved.get(t.value, 'LESSEQUAL')
84     return t
85
86 def t_LESSER(t):
87     r '<'
88     t.type = reserved.get(t.value, 'LESSER')
89     return t
90
91
92 def t_EQUALS(t):
93     r '='
94     t.type = reserved.get(t.value, 'EQUALS')
95     return t
96
97 def t_NUM(t):
98     r '\-?\d+'
99     t.type = reserved.get(t.value, 'NUM')
100    return t
101

```

```

102 t_ignore = " \t\n"
103
104 def t_error(t):
105     print(f'Carater ilegal: {t.value[0]}')
106     t.lexer.skip(1)
107     return t
108
109 #build the lexer
110 lexer = lex.lex()

```

Listing 23: Código presente no Lex

8.2 vim_yacc.py

Segue em anexo o código presente no ficheiro referente à ferramenta Yacc do PLY.

```

1 import ply.yacc as yacc
2 from vim_tokens import tokens
3 import sys
4 import os
5
6 # Programa -> Decl Instr
7 #
8 # Decl -> DECL Declaracoes ENDDECL
9 #
10 # Declaracoes -> Declaracoes Declaracao
11 #      |
12 #
13 # Declaracao -> int Variaveis ;
14 #      | int Array ;
15 #
16 # Variaveis -> Variaveis ',' Variavel
17 #      | Variavel
18 #
19 # Variavel -> Var
20 #      | VarDeclarada
21 #
22 # Array -> ArrayDimensaoSimples
23 #      | ArrayDimensaoDupla
24 #
25 # Var -> ID
26 #
27 # VarDeclarada -> ID EQUALS Exp
28 #
29 # ArrayDimensaoSimples -> [NUM] ID
30 #
31 # ArrayDimensaoDupla -> [NUM] [NUM] ID
32 #
33 # Instr -> INSTR Instrucoes ENDINSTR
34 #
35 # Instrucoes -> Instrucoes Instrucao
36 #      |
37 #
38 # Instrucao -> print ( Exp ) ;

```

```

39 #         | input ( ID ) ;
40 #         | ID EQUALS Exp ;
41 #         | ID [ Exp ] EQUALS Exp ;
42 #         | ID [ Exp ] [ Exp ] EQUALS Exp ;
43 #         | IF ( Conds ) Instrucoes ENDIF
44 #         | IF ( Conds ) Instrucoes Else ENDIF
45 #         | REPEAT Instrucoes UNTIL ( Conds )
46 #
47 # Else -> ELSE Instrucoes ENDELSE
48 #
49 # Conds -> Conds AND Cond
50 #         | Conds OR Cond
51 #         | Cond
52 #
53 # Cond -> Exp EQUIVALENT Exp
54 #         | Exp DIFFERENT Exp
55 #         | Exp GREATER Exp
56 #         | Exp GREATEREQUAL Exp
57 #         | Exp LESSER Exp
58 #         | Exp LESSEREQUAL Exp
59 #         | NOT ( Conds )
60 #
61 # Exp -> Exp ADD Termo
62 #         | Exp SUB Termo
63 #         | Termo
64 #
65 # Termo -> Termo MUL Fator
66 #         | Termo DIV Fator
67 #         | Termo MOD Fator
68 #         | Fator
69 #
70 # Fator -> ( Exp )
71 #         | NUM
72 #         | ID
73 #         | ID [ Exp ]
74 #         | ID [ Exp ] [ Exp ]
75
76 #####
77 #                               Main Program                               #
78 #####
79 def p_Programa(p):
80     "Programa : Decl Instr"
81     p[0] = f'{{p[1]}}start\n{{p[2]}}stop'
82
83 #####
84 #                               Bloco Declaracoes                               #
85 #####
86 def p_Decl(p):
87     "Decl : DECL Declaracoes ENDDECL "
88     p[0] = p[2]
89
90 def p_Declaracoes(p):
91     "Declaracoes : Declaracoes Declaracao "
92     p[0] = p[1] + p[2]

```

```

93
94 def p_Declaracoes_Empty(p):
95     "Declaracoes : "
96     p[0] = ""
97
98 def p_Declaracao_Variaveis(p):
99     "Declaracao : int Variaveis ';' "
100     p[0] = p[2]
101
102 def p_Declaracao_Arrays(p):
103     "Declaracao : int Array ';' "
104     p[0] = p[2]
105
106 def p_Variaveis(p):
107     "Variaveis : Variaveis ',' Variavel "
108     p[0] = p[1] + p[3]
109
110 def p_Variaveis_Simples(p):
111     "Variaveis : Variavel "
112     p[0] = p[1]
113
114 def p_Variavel_Vars(p):
115     "Variavel : Var"
116     p[0] = p[1]
117
118 def p_Variavel_VarsDeclaradas(p):
119     "Variavel : VarDeclarada"
120     p[0] = p[1]
121
122 def p_Array_Uma_Dimensao(p):
123     "Array : ArrayDimensaoSimples"
124     p[0] = p[1]
125
126 def p_Array_Duas_Dimensoes(p):
127     "Array : ArrayDimensaoDupla"
128     p[0] = p[1]
129
130 def p_ArrayDimensaoSimples(p):
131     "ArrayDimensaoSimples : '[' NUM ']' ID"
132     p[0] = 'pushn ' + p[2] + '\n'
133     p.parser.registers[p[4]] = p.parser.registerindex
134     p.parser.registerindex += int(p[2])
135
136 def p_ArrayDimensaoDupla(p):
137     "ArrayDimensaoDupla : '[' NUM ']' '[' NUM ']' ID"
138     p[0] = 'pushn ' + str(int(p[2])*int(p[5])) + '\n'
139     p.parser.registers[p[7]] = p.parser.registerindex
140     p.parser.registerindex += int(p[2])*int(p[5])
141     p.parser.matrizes[p[7]] = int(p[5])
142
143 def p_Var_ID(p):
144     "Var : ID"
145     p[0] = 'pushi 0\n'
146     p.parser.registers[p[1]] = p.parser.registerindex

```

```

147     p.parser.registerindex += 1
148
149 def p_VarDeclarada(p):
150     "VarDeclarada : ID EQUALS Exp"
151     p[0] = p[3]
152     p.parser.registers[p[1]] = p.parser.registerindex
153     p.parser.registerindex += 1
154
155 #####
156 #                               Bloco Instrucoes                               #
157 #####
158
159 def p_Instr(p):
160     "Instr : INSTR Instrucoes ENDINSTR "
161     p[0] = p[2]
162
163 def p_Instrucoes(p):
164     "Instrucoes : Instrucoes Instrucao"
165     p[0] = p[1] + p[2]
166
167 def p_Instrucoes_Empty(p):
168     "Instrucoes : "
169     p[0] = ""
170
171 def p_Instrucao_Print(p):
172     "Instrucao : print '(' Exp ')', ',';"
173     p[0] = p[3] + 'writei' + '\n'
174
175 def p_Instrucao_Read(p):
176     "Instrucao : input '(' ID ')', ',';"
177     if p[3] in p.parser.registers:
178         p[0] = 'read\natoi\n' + 'storeg ' + str(p.parser.registers.get(p[3])) +
179         '\n'
180     else:
181         raise Exception
182
183 def p_Instrucao_Atrib(p):
184     "Instrucao : ID EQUALS Exp ',';"
185     if p[1] in p.parser.registers:
186         p[0] = p[3] + 'storeg ' + str(p.parser.registers.get(p[1])) + '\n'
187     else:
188         raise Exception
189
190 def p_Instrucao_Atrib_Array(p):
191     "Instrucao : ID '[' Exp ']' EQUALS Exp ',';"
192     if p[1] in p.parser.registers:
193         p[0] = 'pushgp\npushi ' + str(p.parser.registers.get(p[1])) + '\npadd\n'
194         + p[3] + p[6] + 'storen\n'
195     else:
196         raise Exception
197
198 def p_Instrucao_Atrib_Matrix(p):
199     "Instrucao : ID '[' Exp ']' '[' Exp ']' EQUALS Exp ',';"
200     if p[1] in p.parser.registers:

```



```

199     p[0] = 'pushgp\npushi ' + str(p.parser.registers.get(p[1])) + '\npadd\n'
200     + p[6] + p[3] + 'pushi ' + str(p.parser.matrizes[p[1]]) + '\nmul\nadd\n' +
201     p[9] + 'store\n'
202     else:
203         raise Exception
204
205 def p_Instrucao_If(p):
206     "Instrucao : IF '(' Conds ')' Instrucoes ENDIF "
207     p[0] = f'{p[3]}jz endif{p.parser.ifs}\n{p[5]}endif{p.parser.ifs}:\n'
208     p.parser.ifs += 1
209
210 def p_Instrucao_If_Else(p):
211     "Instrucao : IF '(' Conds ')' Instrucoes Else ENDIF "
212     p[0] = f'{p[3]}jz else{p.parser.elses}\n{p[5]}jump endelse{p.parser.elses}\nelse{p.parser.elses}:\n{p[6]}jump endelse{p.parser.elses}\nendelse{p.parser.elses}:\n'
213     p.parser.elses += 1
214
215 def p_Else(p):
216     "Else : ELSE Instrucoes ENDELSE"
217     p[0] = p[2]
218
219 def p_Instrucao_Repeat_Until(p):
220     "Instrucao : REPEAT Instrucoes UNTIL '(' Conds ')' "
221     p[0] = f'r{p.parser.ciclos}:\n{p[2]}{p[5]}jz r{p.parser.ciclos}\n'
222     p.parser.ciclos += 1
223
224 def p_Instrucao_While_Do(p):
225     "Instrucao : WHILE '(' Conds ')' DO Instrucoes ENDWHILE"
226     p[0] = f'while{p.parser.ciclos}:\n{p[3]}jz fimwhile{p.parser.ciclos}\n{p[6]}jump while{p.parser.ciclos}\nfimwhile{p.parser.ciclos}:\n'
227     p.parser.ciclos += 1
228
229 #####
230 # Bloco Condicoes #
231 #####
232
233 def p_Conds_And(p):
234     "Conds : Conds AND Cond"
235     p[0] = p[1] + p[3] + 'add\npushi 2\nnequal\n'
236
237 def p_Conds_Or(p):
238     "Conds : Conds OR Cond"
239     p[0] = p[1] + p[3] + 'add\npushi 0\nsup\n'
240
241 def p_Conds_Unica(p):
242     "Conds : Cond"
243     p[0] = p[1]
244
245 def p_Cond_Not(p):
246     "Cond : NOT '(' Conds ')' "
247     p[0] = p[3] + 'pushi 1\ninf\n'
248
249 def p_Cond_Equivalent(p):

```

```

248     "Cond : Exp EQUIVALENT Exp"
249     p[0] = p[1] + p[3] + 'equal\n'
250
251 def p_Cond_Different(p):
252     "Cond : Exp DIFFERENT Exp"
253     p[0] = p[1] + p[3] + 'equal\nnot\n'
254
255 def p_Cond_Greater(p):
256     "Cond : Exp GREATER Exp"
257     p[0] = p[1] + p[3] + 'sup\n'
258
259 def p_Cond_Greater_Equal(p):
260     "Cond : Exp GREATEREQUAL Exp"
261     p[0] = p[1] + p[3] + 'supeq\n'
262
263 def p_Cond_Lesser(p):
264     "Cond : Exp LESSER Exp"
265     p[0] = p[1] + p[3] + 'inf\n'
266
267 def p_Cond_Lesser_Equal(p):
268     "Cond : Exp LESSEREQUAL Exp"
269     p[0] = p[1] + p[3] + 'infeq\n'
270
271
272 #####
273 #                               Bloco Express es                               #
274 #####
275
276 def p_Exp_Termo_add(p):
277     "Exp : Exp ADD Termo"
278     p[0] = p[1] + p[3] + 'add\n'
279
280 def p_Exp_Termo_sub(p):
281     "Exp : Exp SUB Termo"
282     p[0] = p[1] + p[3] + 'sub\n'
283
284 def p_Exp_Termo(p):
285     "Exp : Termo"
286     p[0] = p[1]
287
288 def p_Termo_Fator_mul(p):
289     "Termo : Termo MUL Fator"
290     p[0] = p[1] + p[3] + 'mul\n'
291
292 def p_Termo_Fator_div(p):
293     "Termo : Termo DIV Fator"
294     if (p[3] != 'pushi 0\n'):
295         p[0] = p[1] + p[3] + 'div\n'
296     else:
297         p[0] = 'pushi 0\n'
298
299 def p_Termo_Fator_mod(p):
300     "Termo : Termo MOD Fator"
301     if (p[3] != 'pushi 0\n'):

```

```

302     p[0] = p[1] + p[3] + 'mod\n'
303 else:
304     p[0] = 'pushi 0\n'
305
306 def p_Termo_Fator(p):
307     "Termo : Fator"
308     p[0] = p[1]
309
310 def p_Fator_ID(p):
311     "Fator : ID"
312     if p[1] in p.parser.registers:
313         p[0] = 'pushg ' + str(p.parser.registers.get(p[1])) + '\n'
314     else:
315         raise Exception
316
317 def p_Fator_ID_Array(p):
318     "Fator : ID '[' Exp ']' "
319     if p[1] in p.parser.registers:
320         p[0] = 'pushgp\npushi ' + str(p.parser.registers.get(p[1])) + '\npadd\n'
321         + p[3] + 'loadn\n'
322     else:
323         raise Exception
324
325 def p_Fator_ID_Matrix(p):
326     "Fator : ID '[' Exp ']' '[' Exp ']' "
327     if p[1] in p.parser.registers:
328         p[0] = 'pushgp\npushi ' + str(p.parser.registers.get(p[1])) + '\npadd\n'
329         + p[6] + p[3] + 'pushi ' + str(p.parser.matrizes[p[1]]) + '\nmul\nadd\n' + '
330         loadn\n'
331     else:
332         raise Exception
333
334 def p_Fator_num(p):
335     "Fator : NUM"
336     p[0] = 'pushi ' + p[1] + '\n'
337
338 def p_Fator_Exp(p):
339     "Fator : '(' Exp ')'"
340     p[0] = p[2]
341
342 #####
343 #                               Bloco Yacc Geral                               #
344 #####
345
346 def p_error(p):
347     print(f'Syntax Error: {p}')
348
349 while True:
350     try:
351         fileIn = open(f"{input('Introduce path to program to be compiled: ')}", 'r')
352         break
353     except OSError:

```

```

352         print('Invalid path to file')
353
354 while True:
355     try:
356         fileOut = input('Introduce path to outputfile: ')
357         break
358     except OSError:
359         print('Invalid path')
360
361
362 # Build parser
363 parser = yacc.yacc()
364 parser.registers = {} # id : offset
365 parser.matrizes = {} # id : tamanho das colunas
366 parser.registerindex = 0 # valor do offset
367 parser.ifs = 0 # total de ifs
368 parser.elses = 0 # total de elses
369 parser.ciclos = 0 # total de ciclos
370 parser.fileOut = open(fileOut, 'w+')
371
372 content = ''
373 for linha in fileIn:
374     content += linha
375
376
377 try:
378     result = parser.parse(content)
379     parser.fileOut.write(result)
380 except (TypeError, Exception):
381     print("An error occurred during the compiling, no output could be given!")
382     os.remove(fileOut)
383
384 fileIn.close()
385 parser.fileOut.close()

```

Listing 24: Código do Parser