Engenharia Gramatical (1º de MEI)

Trabalho Prático 2

Relatório de Desenvolvimento

Maria Beatriz Lacerda	Maria Barros	Renata Teixeira
A89535	PG47488	PG47603

9 de maio de 2022

Resumo

Foi nos proposto pela equipa docente da Unidade Curricular de Engenahria Gramatical, desenvolver um Analisador de Código para uma evolução da sua linguagem de programação imperativa simples (LPIS), que foi definida no TP2 da unidade curricular de Processamento de Linguagem. Posteriormente teremos que responder a 5 perguntas propostas e gerar um ficheiro *HTML* como forma de relatório dos resultados obtidos.

Conteúdo

Capítulo 1

Introdução

Neste 2º Trabalho Prático foi proposto o desenvolvimento de um analisador de código para uma evolução da Linguagem de Programação Imperativa Simples (LPIS) definida anteriormente no TP2 de PL. Esta nova versão deve permitir declarar variáveis atómicas e estruturadas (incluindo como no Python as estruturas: conjunto, lista, tuplo, dicionário), instruções condicionais e pelo menos 3 variantes de ciclos.

Concretamente, o projeto será escrito em *Python*, usando o *Parser* e os *Visitors* do módulo para geração de processadores de linguagens Lark.Interpreter, uma ferramenta que analise programas escritos na linguagem criada e gere em HTML um relatório com os resultados da análise.

A análise feita baseia-se em 5 tópicos:

- Lista de todas as variáveis do programa indicando os casos de: **redeclaração** ou **não-declaração**, **usadas mas não inicializadas** e **declaradas mas nunca mencionadas**.
- Total de variáveis declaradas *versus* os Tipos de dados estruturados usados.
- Total de instruções que formam o corpo do programa: o número de instruções de cada tipo.
- Total de situações em que estruturas de controlo surgem aninhadas em outras estruturas de controlo do mesmo ou de tipos diferentes.
- Presença de *ifs* aninhados indicando os casos em que *ifs* aninhados possam ser substituídos por um só *if*.

Estrutura do Relatório

O presente documento encontra-se dividido em 5 capítulos:

- No **primeiro capítulo** é feita uma introdução do trabalho desenvolvido e dos objetivos mais gerais a cumprir.
- No **segundo capítulo** é feita uma descrição informal do problema, onde é explicado em mais detalhe as questões propostas bem como a lógica por detrás da sua resolução.
- No **terceiro capítulo** procedemos à exposição e explicação mais detalhada por detrás das decisões tomadas e raciocínio para a chegada à implementação final.
- O **quarto capítulo** encontra-se dividido em duas subsecções sendo que na primeira apresentamos um dos testes utilizados para testar o nosso código. Este teste exemplifica a sintaxe da nossa linguagem e tem vários casos que testam se as questões indicadas foram resolvidas tais como variáveis redeclaradas ou condições *if* aninhadas. Na segunda secção apresentamos os resultados obtidos. Tal como se pode verificar, estes resultados são apresentados numa página HTML.
- No **quinto capítulo** é feita uma reflexão sobre o trabalho entregue com uma síntese do documento. Adicionalmente, são apontados os aspetos que não corresponderam às expectativas, visão geral do trabalho e conclusões tiradas para trabalhos futuros.

Capítulo 2

Análise e Especificação

2.1 Descrição informal do problema

```
1    f = open("teste", "r")
2    cod = f.read()
3    p = Lark(grammar)
4    parse_tree = p.parse(cod)
5    data = MyInterpreter().visit(parse_tree)
```

Listing 2.1: Excerto de código

Inicialmente é efetuada uma leitura de um ficheiro, tal como apresentamos na listagem acima, na linha 1.

Neste ficheiro *teste* encontra-se uma linguagem que nós definimos na unidade curricular de PL, que contém exemplos de ciclos, condições e atribuições. No entanto foram efetuadas algumas alterações, sendo adicionados tipos de estrturas como dicionários, tuplos, *arrays* e *sets*.

Para conseguirmos prosseguir para as respostas às questões propostas é necessário assim definir uma gramática para posteriormente começarmos a percorrer a árvore através de um *Interpreter*.

Posteriormente, é efetuada assim a leitura da gramática através do *Lark* (linha 3 da listagem), feito o *parse* da mesma com o código lido do ficheiro teste (linha 4 da listagem) e finalmente é inicializado o *Interpreter* (linha 5 da listagem).

Finalmente podemos passar à resposta de cada pergunta.

Na pergunta 1 foi nos pedido 4 conjuntos de variáveis. Sendo elas: as variáveis que são redeclaradas, ou seja são declaradas mais do que uma vez; as variáveis que não são declaradas, ou seja variáveis que foram usadas para o desenvolvimento do código mas nunca foram declaradas; as variáveis que são usadas mas não foram inicializadas à priori. E finalmente variáveis que foram declaradas mas nunca foram usadas.

Na pergunta 2 é-nos pedido para indicar o número de variáveis de cada tipo de estrutura, ou seja indicar quantos dicionários, tuplos, *arrays* e *sets* tem no código.

Passando à pergunta 3, é-nos suposto indicar o total de instruções feitas no programa e quantas de cada tipo, isto é, quantas atribuições, ciclos, condicionais, leituras e escritas.

Já na pergunta 4 temos que indicar o total de estruturas aninhadas, ou seja quantas estrturas surgem aninhadas dentro de outras estrturas iguais ou de tipos diferentes.

Finalmente, na pergunta 5 é nos pedido para indicar se existem *ifs* aninhados em *ifs* e se os mesmos dão para substituir por um único *if*.

2.2 Especificação dos requisitos

Começamos assim por perceber como iremos obter os dados pedidos. Os nossos *testes* está dividido em **declarações** e em **instruções**.
Assim haverá perguntas em que iremos trabalhar apenas com uma das partes e algumas em que iremos trabalhar com as duas em simultâneo.

2.2.1 Pergunta 1

Depois de uma análise do ficheiro teste e da sua estrutura podemos concluir que:

- As variáveis redeclaradas exigem apenas analisar a parte das declarações;
- As variáveis não declaradas exigem apenas analisar a parte das instruções;
- As variáveis não inicializadas e as não usadas exigem analisar as duas partes, declarações e instruções.

2.2.2 Pergunta 2

Nesta pergunta, para conseguirmos obter o total de estruturas usadas, como no nosso *teste* as declarações são sempre efetuadas no seu respetivo lugar, teremos que analisar apenas a parte das declarações.

2.2.3 Pergunta 3

Nesta pergunta, tal como mencionado anteriormente pretendemos obter o total de instruções de cada tipo, isto é, o total de atribuições, escritas, leituras, condicionais e ciclos.
Aqui teremos que analisar as duas partes, sendo que as atribuições encontram-se nas declarações, a leitura e a escrita encontra-se nas duas partes e as condicionais e os ciclos encontram-se nas instruções.

2.2.4 Pergunta 4

Nesta pergunta apenas nos interessa analisar a parte das instruções, sendo que apenas neste local podem existir estruturas aninhadas.

2.2.5 Pergunta 5

À semelhança da pergunta anterior, também apenas visitamos a parte das instruções do nosso programa dado que será aqui que estarão presentes as condições aninhadas.

Capítulo 3

Conceção/ desenho da Resolução

Antes de avançar para qualquer pergunta é necessário construir a nossa gramática. Assim na listagem abaixo encontra-se um excerto do início da mesma.

```
grammar='''
start: declaracoes instrucoes
declaracoes : DECLARACAO CA decl* CF
decl : VAR ID PV
     | VAR ID IGUAL exp PV
     | DICT ID IGUAL CA dicionario* CF PV
     | VAR ID RA NUM RF PV
     | VAR ID RA NUM RF IGUAL RA array* RF PV
     | VAR ID IGUAL PA tuplo PF PV
     | VAR ID IGUAL CA tuplo CF PV

instrucoes : INSTRUCAO CA inst* CF
inst : ID IGUAL exp PV
     | FOR PA inst logica PV ifor PF DO CA insts CF
     | WHILE PA logica PF CA insts CF
     | FOREACH PA ID IN ID PF CA insts CF
     | IF PA logica PF CA insts CF
     | IF PA logica PF CA insts CF ELSE CA insts CF'''
```

Listing 3.1: Excerto da gramática

Posteriormente é iniciado o *Interpreter* para de seguida conseguirmos visitar a árvore a todos os níveis desejados. Depois de uma análise da gramática chegamos à conclusão que:

- Nas declarações o nível mais profundo que iremos visitar será a *decl*, ou seja será aqui que serão feitas todas as contas e análises de informações que necessitamos.
- Nas instruções o nível mais profundo que iremos visitar será o fator. Dependendo do tipo de instrução, o programa vai visitar diferentes campos, ou seja:
  - Se estivermos perante uma expressão do tipo **x = "exp";**, sendo apenas um exemplo, iremos visitar o campo 2;
  - Se estivermos perante um **for** iremos visitar o campo 2, 3, 5 e 9;
  - Se estivermos perante um **while** iremos visitar o campo 2 e 5;

- Se estivermos perante um **foreach** iremos visitar o campo 7;
- Se estivermos perante um **if** iremos visitar o campo 2 e 5;
- Se estivermos perante um **if & else** iremos visitar o campo 2, 5 e 9.

Assim, no final nós queremos chegar sempre ao ***Token ID***.

3.1 Pergunta 1

Para esta pergunta, inicialmente criamos 4 tipos de *sets*, aqui serão guardadas todas as variáveis que pretendemos para posteriormente responder à pergunta e criamos também mais dois *sets* de forma auxiliar, *self.vars = set()* e *self.ini_vars = set()*, sendo que o primeiro indica todas as variáveis presentes e o segundo indica todas as que são inicializadas.

3.1.1 Variáveis redeclaradas

```
1 def decl(self, tree):
2     if tree.children[1] in self.vars:
3         self.red_vars.add(str(tree.children[1]))
4     else:
5         (...)
```

Listing 3.2: Excerto da função decl

Na função apresentada a cima, conseguimos garantir se as variáveis já foram declaradas ou não. Este processo é feito através do *set* auxiliar *self_vars*. Se a variável já estiver contida neste *set*, significa que esta está a ser redeclarada, assim será adicionada ao *set self.red_vars*.

3.1.2 Variáveis não declaradas

```
1 def inst(self, tree):
2     if(tree.children[0].type == "ID"):
3         for child in tree.children:
4             if not isinstance(child, Tree):
5                 if child.type == "ID":
6                     if child not in self.vars:
7                         self.notD_vars.add(str(child))
8
9         (...)
```

Listing 3.3: Excerto da funçaõ inst

No excerto apresentado acima, conseguimos perceber como é que definimos quais as variáveis que são ou não declaradas. Se uma variável encontrada não pertencer ao *set self_vars* significa que esta não foi declarada à priori. Assim é adicionado ao *set* em questão.

3.1.3 Variáveis não inicializadas

```
def decl(self, tree):
    (...)
    else:
        self.vars.add(str(tree.children[1]))
        if tree.children[2].type == "IGUAL":
            self.ini_vars.add(str(tree.children[1]))
        elif len(tree.children) > 6:
            if tree.children[5].type == "IGUAL":
                self.ini_vars.add(str(tree.children[1]))
        else:
            self.notI_vars.add(str(tree.children[1]))
```

Listing 3.4: Excerto da função decl

Começando por tratar da parte das declarações, sempre que encontramos algum igual significa que a variável é inicializada, ou seja, caso contrário adicionamos ao *set self.notI_vars*. Posteriormente é necessário verificar se a sua inicialização é efetuada na parte das instruções.

```
def inst(self, tree):
    (...)
    if tree.children[0] in self.notI_vars:
                self.notI_vars.remove(tree.children[0])
    (...)
```

Listing 3.5: Excerto da função inst

Nesta parte, verificamos se a variável que está a ser igualada a algo está no *set self.notI_vars*. Se tal se verificar retiramos desse *set*, pois esta é inicializada posteriormente, nas instruções.

3.1.4 Variáveis declaradas mas nunca usadas

Inicialmente igualamos o *set self.inuteis_var* ao *set self.vars*. Nesta fase iremos fazer o processo contrário, isto é, sempre que encontramos uma variável a ser usada iremos retirar do *set self.inuteis_vars*. No final, neste *set* apenas as variaveis que nunca serão usadas.

```
def inst(self, tree):
    (...)
    if tree.children[0] in self.notI_vars:
                self.notI_vars.remove(tree.children[0])
    if tree.children[0] in self.inuteis_vars:
                self.inuteis_vars.remove(tree.children[0])
    (...)
```

Listing 3.6: Excerto da função inst

3.2 Pergunta 2

Com vista a solucionar esta questão criamos um dicionário que inicializamos da seguinte forma:

self.estruturas = {'dict': 0, 'tuplo': 0, 'array': 0, 'set': 0}

Deste modo, a cada estrutura existente no nosso código corresponderá um valor que indica o número de vezes que são declaradas. Estes cálculos são feitos na função *decl*, que, tal como mencionado anteriormente, contem as declarações do nosso programa.

Conforme vamos percorrendo as diferentes declarações vamos fazendo algumas verificações. Se a declaração tiver o *Token* "DICT", estamos perante a declaração de um *dicionário* e, portanto, incrementamos o valor da *key* "dict"do dicionário criado anteriormente. Se, por outro lado, a declaração conter um RA, que representa um parêntese reto a abrir, sabemos que a declaração é a de um *array*. Se, por outro lado, encontrarmos um *Token* "PA"sabemos que encontramos a declaração de um *tuplo*. Por último, verificamos se a declaração contém um CA. Se isto acontecer a declaração é a de um *set*

```
def decl(self, tree):
        (...)
        if len(tree.children) > 3:
            if isinstance(tree.children[0], Token) and tree.children[0].type == "
   DICT":
                self.estruturas["dict"] += 1
            elif isinstance(tree.children[2], Token) and tree.children[2].type ==
    "RA":
                self.estruturas["array"] += 1
            elif isinstance(tree.children[3], Token) and tree.children[3].type ==
    "PA":
                self.estruturas["tuplo"] += 1
            elif isinstance(tree.children[3], Token) and tree.children[3].type ==
    "CA":
                self.estruturas["set"] += 1
```

Listing 3.7: Excerto da função decl

3.3 Pergunta 3

Para esta pergunta criamos um dicionário que inicializámos da seguinte forma:

self.ins = {'decl': 0, 'for': 0, 'foreach': 0, 'while': 0, 'if': 0, 'ifelse': 0, 'escrever': 0, 'ler': 0}

De forma a contabilizar todas as instruções realizadas tivemos que visitar quer as nossas declarações quer as instruções do nosso programa pelo que recorremos a várias funções para esse efeito.

Primeiramente, na função *decls* conseguimos obter todas as declarações existentes. Conseguimos ainda contabilizar as operações de escrita numa variável quando esta é declarada.

```
    def decl(self, tree):
        if tree.children[2].type == "IGUAL":
            self.ins['escrever'] += 1
        elif len(tree.children) > 6 and tree.children[5].type == "IGUAL":
```

```
            self.ins['escrever'] += 1
        self.ins['decl'] += 1
```

Listing 3.8: Excerto da função decl

Adicionalmente, verificamos a escrita em mais duas funções: *inst* e *ifor*. Na primeira, estamos a ter em atenção os casos em que escrevemos numa variável no corpo do nosso programa. Na segunda estamos a contabilizar a escrita da variável de controlo que incrementa/decrementa num ciclo for.

```
def inst(self, tree):
        # IF que verifica se    um ID IGUAL exp PV
        if(tree.children[0].type == "ID"):
            self.ins['escrever'] += 1
            (...)

 def ifor(self, tree):
        # Visita ifor: ID IGUAL exp
        if(tree.children[0].type == "ID"):
            self.ins['escrever'] += 1
```

Listing 3.9: Excerto da função inst e ifor

Os cálculos para guardar o número de instruções for, while, if, if else ou foreach foram efetuados na função *inst*. Nesta função contabilizamos ainda o caso em que lemos uma variável num ciclo foreach.

```
def inst(self, tree):
    (...)
        elif tree.children[0].type == "FOR":
            self.ins['for'] += 1
            self.visit(tree.children[2])
            self.visit(tree.children[3])
            self.visit(tree.children[5])
            self.visit(tree.children[9])
        # Verifica se WHILE PA logica PF CA inst* CF
        elif tree.children[0].type == "WHILE":
            self.ins['while'] += 1
            self.visit(tree.children[2])
            self.visit(tree.children[5])
        # Verifica FOREACH PA ID IN ID PF CA inst* CF
        elif tree.children[0].type == "FOREACH":
            self.ins['ler'] += 1
            self.ins['foreach'] += 1
            if tree.children[4].type == "ID":
    (...)
     # Verifica se IF PA logica PF CA inst* CF | IF PA logica PF CA inst* CF ELSE
    CA inst* CF
        else:
            if len(tree.children) < 8:
                self.ins['if'] += 1
                self.visit(tree.children[2])
```

```
25                  self.visit(tree.children[5])
26              else:
27                  self.ins['ifelse'] += 1
28                  self.visit(tree.children[2])
29                  self.visit(tree.children[5])
30                  self.visit(tree.children[9])
```

Listing 3.10: Excerto da função inst

Finalmente, através da função *fator*, adicionamos os casos de leitura de variáveis quando estas estão presentes numa "exp"(que representa uma expressão aritmética).

```
1
2 def fator(self, tree):
3         # Visita o exp de PA exp PF
4         if len(tree.children) == 3:
5             self.visit(tree.children[1])
6         # Visita o ID
7         elif tree.children[0].type == "ID":
8             self.ins['ler'] += 1
```

Listing 3.11: Excerto da função fator

3.4 Pergunta 4

Para dar resposta a esta questão utilizamos a função *insts*. É aqui que conseguimos verificar se existe um *Token* de uma instrução do tipo *for*, *if*, *foreach* ou *while* entre os filhos de uma instrução do mesmo tipo. Se tal se verificar, incrementamos a variável criada: self.aninhadas.

```
1
2 def insts(self, tree):
3         # Visita insts : inst+
4         for child in tree.children:
5             if isinstance(child.children[0], Token) and (child.children[0].type
    == "FOR" or child.children[0].type == "IF" or child.children[0].type == "
    FOREACH" or child.children[0].type == "WHILE"):
6                 self.aninhadas += 1
7             self.visit(child)
```

Listing 3.12: Excerto da função insts

3.5 Pergunta 5

De modo a responder a esta última pergunta criamos três variáveis:

- **self.flag**
  Variável inicializada como False. Marca sempre que for encontrada, na parte das instruções do código, uma condição *if*. Isto tem efeito na função *insts*, onde esta variável toma o valor de True. Quando uma outra condição, como por exemplo, um *for*, ou mesmo um *if else*, é detetada, esta variável volta a assumir o valor de False.

- **self.idIF**
  Esta variável irá ser incrementada sempre que forem detetadas condições *if* e *if else* no código. Desta forma, cada *if* detetado poderá, mais tarde, ser identificado. Este incremento é feito na função *inst*.

- **self.tmp**
  Este array irá guardar todos os ids de ifs que foram detetados como aninhados. Isto acontece na função *insts* em que verificamos que a instrução seguinte a um if é também deste tipo.

```
1
2     def insts(self, tree):
3         (...)
4         if isinstance(child.children[0], Token) and self.flag and child.
    children[0].type == "IF":
5             self.tmp.append(int(self.idIF))
6         self.visit(child)
7
```

Listing 3.13: Excerto da função insts

Assim sendo, seguindo esta lógica, sempre que um if é detetado tem o seu ID registado e a nossa flag torna-se True. De seguida, verificamos se a instrução seguinte é também um if. Se isto se verificar, o id desse mesmo if é adicionado ao array tmp.

Mais tarde, ao gerar o código HTML, onde apresentamos os resultados, encontramos todos os ifs através de uma expressão regular. Estes ifs serão retornados pela função *findall* num array no qual o índice de cada um corresponderá ao seu ID previamente recolhido. Assim, em todos os ifs encontrados, se o seu id se encontrar no array *tmp* será, no ficheiro HTML, assinalado através de um comentário.

Capítulo 4

Testes e resultados

4.1 Teste

```
declaracoes{
    var i; var i; var w;
    var soma;
    var j = 2; var j = 2;
    dict arvore = {1:2,2:3};
    var k[2] = [1,2,3];
    var tuplo = (1,2,3);
    var set = {1,2,3};
    dict arvore = {1:2,2:3};
    var set = {1,2,3};
    var i;
    var jose; var j;
} instrucoes{
    for(f = 0; i < q; t = i + 1) do{
        soma = p + 1;
        if(i < j){
            soma = soma + 1;
            while(a < i){
                soma = soma + 1;
            }
        }
        else{
            nomeTRES = soma + 1;
        }
    }
    h = 0;
    while(a < i){
        nome = soma+1;
        if(i < j){
            while(a < i){
                soma = soma + 1;
            }
        }
        else{
            foreach (x in b){
```

```
                nomeUM = soma + 1;
                if(c < i){
                    nomeDOIS = soma + 1;

                }
            }
            nomeTRES = soma + 1;
        }
    }
    foreach (x in b){
        nomeUM = soma+1;

    }
    if(c < i){
        if(a < i){
            if(w < i){
                nomeDOIS = soma+1;
                if(i < j){
                    soma = soma + 1;
                }
                else{
                    if(c < i){
                        nomeDOIS = soma + 1;
                    }
                }
                nomeTRES = soma + 1;
            }

        }
    }
}
```

Listing 4.1: Ficheiro teste

4.2 Resultados

Depois de correr o trabalho, é gerado um ficheiro *HTML* que apresenta todos os resultados:

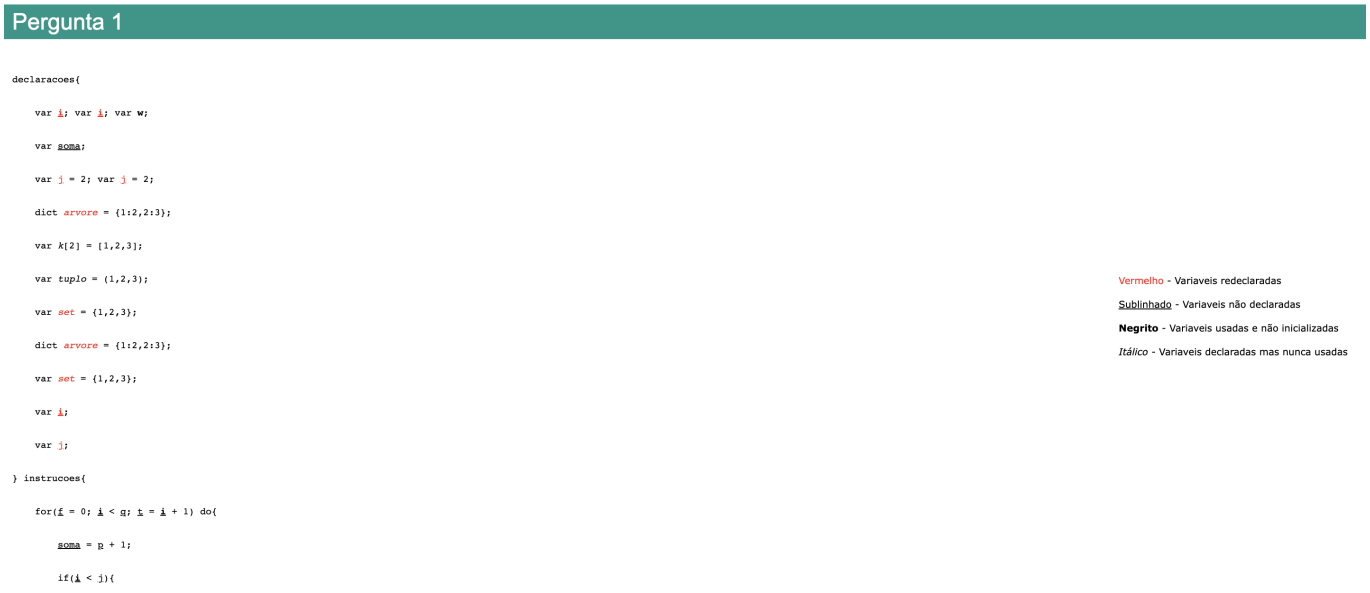

Figura 4.1: Pergunta 1

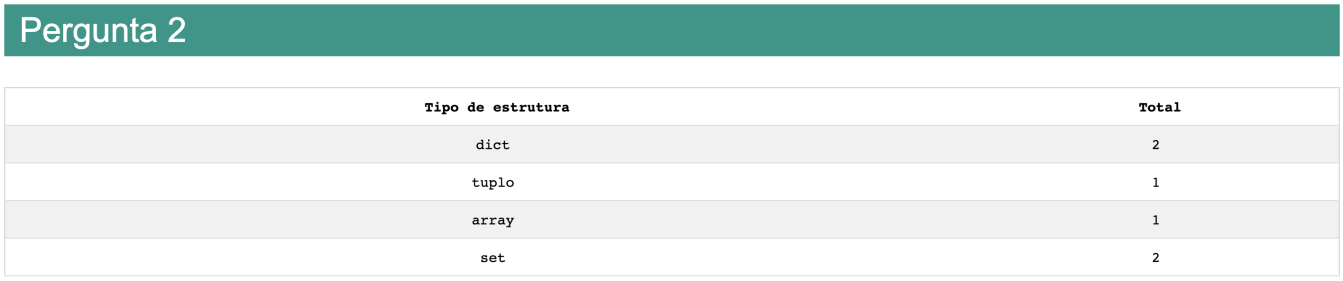

Figura 4.2: Pergunta 2

Pergunta 3

Tipo de instrução	Total
decl	14
for	1
foreach	2
while	3
if	5
ifelse	3
escrever	25
ler	41

Figura 4.3: Pergunta 3

Pergunta 4

Total de instruções - 94

Total de estruturas aninhadas - 10

Figura 4.4: Pergunta 4

```
# O seguinte par de ifs pode ser susbtituído por uma única condição
    if(c < i){
 # O seguinte par de ifs pode ser susbtituído por uma única condição
        if(a < i){
             if(w < i){
                nomeDOIS = soma+1;
                if(i < j){
                    soma = soma + 1;
                }
                else{
                    if(c < i){
                        nomeDOIS = soma + 1;
                    }
                }
                nomeTRES = soma + 1;
            }

        }
    }
}
```

Figura 4.5: Pergunta 5

Capítulo 5

Conclusão

O presente documento descreve as várias etapas deste trabalho prático da unidade curricular de Engenharia Gramatical, retratando o raciocínio por detrás da resolução dos diversos tópicos propostos.

Durante a elaboração deste projeto, foi possível consolidar conhecimentos relativos ao módulo *Lark.Interpreter*, como o *Parser* e os *Visitors* deste módulo, gerando uma ferramenta que auxilia em tarefas como a deteção de redeclarações ou variáveis inicializadas mas não usadas, situações estas que são desnecessárias e devem ser corrigidas no código, e situações de possíveis simplificações no código, como *ifs* ainhados que podem ser substituídos por apenas um.

Consideramos ter cumprido o proposto pelo enunciado, tendo sido capazes de responder corretamente aos vários tópicos utilizando os módulos especificados pelos docentes e lecionados em aula.

Apêndice A

Código do Programa

```
from ast import parse
from lark import Discard
from lark import Lark, Token, Tree
from lark.tree import pydot__tree_to_png
from lark.visitors import Interpreter
import re


class MyInterpreter(Interpreter):
    def __init__(self):
        self.output = {}
        self.vars = set()
        self.red_vars = set()
        self.ini_vars = set()
        self.notI_vars = set()
        self.notD_vars = set()
        # Variaveis que nunca sao usadas no codigo
        self.inuteis_vars = set()
        self.estruturas = {'dict': 0, 'tuplo': 0, 'array': 0, 'set': 0}
        self.ins = {'decl': 0, 'for': 0, 'foreach': 0,
                    'while': 0, 'if': 0, 'ifelse': 0, 'escrever': 0, 'ler': 0}
        self.isFilho = False
        self.aninhadas = 0
        self.flag = False
        self.tmp = []
        self.idIF = 0

    def start(self, tree):
        # Visitar Declaracoes
        self.visit(tree.children[0])
        self.output['vars'] = self.vars
        self.output['red_vars'] = self.red_vars
        self.output['ini_vars'] = self.ini_vars
        self.output['notI_vars'] = self.notI_vars
        self.inuteis_vars = self.vars
        # Visitar Instrucoes
        self.visit(tree.children[1])
        self.output['notD_vars'] = self.notD_vars
```

```
        self.output['inuteis_vars'] = self.inuteis_vars
        self.output['estruturas'] = self.estruturas
        self.output['ins'] = self.ins
        self.output['aninhadas'] = self.aninhadas
        self.output['tmp'] = self.tmp
        return self.output

    def declaracoes(self, tree):
        # Visitar lista de decl
        for children in tree.children:
            if isinstance(children, Tree):
                self.visit(children)

    def decl(self, tree):
        if tree.children[2].type == "IGUAL":
            self.ins['escrever'] += 1
        elif len(tree.children) > 6 and tree.children[5].type == "IGUAL":
            self.ins['escrever'] += 1
        self.ins['decl'] += 1
        # Verifica se ja foi declarada anteriormente
        if tree.children[1] in self.vars:
            self.red_vars.add(str(tree.children[1]))
        # Obtem o ID
        else:
            # Adiciona a lista geral de variaveis declaradas
            self.vars.add(str(tree.children[1]))
            # IF para todas as variaveis sem ser array inicializado
            if tree.children[2].type == "IGUAL":
                self.ini_vars.add(str(tree.children[1]))
            # IF para todas as variaveis de arrays inicializado
            elif len(tree.children) > 6:
                if tree.children[5].type == "IGUAL":
                    self.ini_vars.add(str(tree.children[1]))
            # ELSE para todas as variaveis que nao foram inicializadas
            else:
                self.notI_vars.add(str(tree.children[1]))

        if len(tree.children) > 3:
            if isinstance(tree.children[0], Token) and tree.children[0].type == "
DICT":
                self.estruturas["dict"] += 1
            elif isinstance(tree.children[2], Token) and tree.children[2].type ==
 "RA":
                self.estruturas["array"] += 1
            elif isinstance(tree.children[3], Token) and tree.children[3].type ==
 "PA":
                self.estruturas["tuplo"] += 1
            elif isinstance(tree.children[3], Token) and tree.children[3].type ==
 "CA":
                self.estruturas["set"] += 1

    def instrucoes(self, tree):
        # Visita lista de inst
        for children in tree.children:
```

```
            if isinstance(children, Tree):
                self.visit(children)

    def inst(self, tree):
        # IF que verifica se e um ID IGUAL exp PV
        if(tree.children[0].type == "ID"):
            self.flag = False
            self.ins['escrever'] += 1
            # Percorre todas as instrucoes
            for child in tree.children:
                if not isinstance(child, Tree):
                    if child.type == "ID":
                        # Verifica se nao esta declarada
                        if child not in self.vars:
                            self.notD_vars.add(str(child))
            # Remove variavies que sao inicializadas posteriormente
            if tree.children[0] in self.notI_vars:
                self.notI_vars.remove(tree.children[0])
            # Remove variavies que nao sao inuteis
            if tree.children[0] in self.inuteis_vars:
                self.inuteis_vars.remove(tree.children[0])
            # Visita as exp
            self.visit(tree.children[2])
        # verifica se FOR PA inst logica PV ifor PF DO CA insts CF
        elif tree.children[0].type == "FOR":
            self.flag = False
            self.ins['for'] += 1
            self.visit(tree.children[2])
            self.visit(tree.children[3])
            self.visit(tree.children[5])
            self.visit(tree.children[9])
        # Verifica se WHILE PA logica PF CA inst* CF
        elif tree.children[0].type == "WHILE":
            self.flag = False
            self.ins['while'] += 1
            self.visit(tree.children[2])
            self.visit(tree.children[5])
        # Verifica FOREACH PA ID IN ID PF CA inst* CF
        elif tree.children[0].type == "FOREACH":
            self.flag = False
            self.ins['ler'] += 1
            self.ins['foreach'] += 1
            if tree.children[4].type == "ID":
                child = tree.children[4]
                if child not in self.vars:
                    self.notD_vars.add(str(child))
            self.visit(tree.children[7])
        # Verifica se IF PA logica PF CA inst* CF | IF PA logica PF CA inst* CF
    ELSE CA inst* CF
        else:
            self.idIF += 1
            if len(tree.children) < 8:
                self.flag = True
                self.ins['if'] += 1
```

```
                self.visit(tree.children[2])
                self.visit(tree.children[5])
            else:
                self.flag = False
                self.ins['ifelse'] += 1
                self.visit(tree.children[2])
                self.visit(tree.children[5])
                self.visit(tree.children[9])

    def insts(self, tree):
        # Visita insts : inst+
        for child in tree.children:
            if isinstance(child.children[0], Token) and (child.children[0].type
    == "FOR" or child.children[0].type == "IF" or child.children[0].type == "
    FOREACH" or child.children[0].type == "WHILE"):
                self.aninhadas += 1
            if isinstance(child.children[0], Token) and self.flag and child.
    children[0].type == "IF":
                self.tmp.append(int(self.idIF))
            self.visit(child)

    def ifor(self, tree):
        # Visita ifor: ID IGUAL exp
        if(tree.children[0].type == "ID"):
            self.ins['escrever'] += 1
            if tree.children[0] not in self.vars:
                self.notD_vars.add(str(tree.children[0]))
            self.visit(tree.children[2])

    def logica(self, tree):
        if len(tree.children) == 3:
            self.visit(tree.children[0])
            self.visit(tree.children[2])
        elif len(tree.children) == 2:
            self.visit(tree.children[1])
        else:
            self.visit(tree.children[0])

    def cond(self, tree):
        if len(tree.children) == 3:
            self.visit(tree.children[0])
            self.visit(tree.children[2])
        else:
            self.visit(tree.children[0])

    def exp(self, tree):
        if len(tree.children) == 3:
            self.visit(tree.children[0])
            self.visit(tree.children[2])
        else:
            self.visit(tree.children[0])

    def termo(self, tree):
        if len(tree.children) == 3:
```

```
            self.visit(tree.children[0])
            self.visit(tree.children[2])
        else:
            self.visit(tree.children[0])

    def fator(self, tree):
        # Visita o exp de PA exp PF
        if len(tree.children) == 3:
            self.visit(tree.children[1])
        # Visita o ID
        elif tree.children[0].type == "ID":
            self.ins['ler'] += 1
            child = tree.children[0]
            if child not in self.vars:
                self.notD_vars.add(str(child))
            # Remove variavies que nao sao inuteis
            if child in self.inuteis_vars:
                self.inuteis_vars.remove(child)


grammar = '''
start: declaracoes instrucoes
declaracoes : DECLARACAO CA decl* CF
decl : VAR ID PV
     | VAR ID IGUAL exp PV
     | DICT ID IGUAL CA dicionario* CF PV
     | VAR ID RA NUM RF PV
     | VAR ID RA NUM RF IGUAL RA array* RF PV
     | VAR ID IGUAL PA tuplo PF PV
     | VAR ID IGUAL CA tuplo CF PV

dicionario: elemd (VIRG elemd)*

elemd : (WORD | NUM) DP (WORD | NUM)

tuplo: NUM (VIRG NUM)*

array : elema (VIRG elema)*

elema : WORD | NUM

instrucoes : INSTRUCAO CA inst* CF

inst : ID IGUAL exp PV
     | FOR PA inst logica PV ifor PF DO CA insts CF
     | WHILE PA logica PF CA insts CF
     | FOREACH PA ID IN ID PF CA insts CF
     | IF PA logica PF CA insts CF
     | IF PA logica PF CA insts CF ELSE CA insts CF

insts : inst+

ifor: ID IGUAL exp

```

```
logica : logica AND cond
       | logica OR cond
       | NOT cond
       | cond

cond : cond MAIOR exp
     | cond MENOR exp
     | cond MAIORI exp
     | cond MENORI exp
     | cond II exp
     | cond DIF exp
     | exp

exp : exp SOMA termo
    | exp SUB termo
    | termo

termo : termo MUL fator
      | termo DIV fator
      | termo MOD fator
      | fator

fator : PA exp PF | NUM | ID

CA: "{"
CF: "}"
VAR: "var"
DECLARACAO: "declaracoes"
INSTRUCAO: "instrucoes"
ID: WORD
IGUAL: "="
PV: ";"
FOR: "for"
PA: "("
PF: ")"
DO: "do"
IF: "if"
ELSE: "else"
AND: "&&"
OR: "||"
NOT: "!"
MAIOR: ">"
MENOR: "<"
MAIORI: ">="
MENORI: "<="
II: "=="
DIF: "!="
SOMA: "+"
SUB: "-"
MUL: "*"
DIV: "/"
MOD: "%"
NUM: ("0".."9")+
WHILE: "while"
```

```
FOREACH: "foreach"
IN: "in"
DICT: "dict"
RA: "["
RF: "]"
DP: ":"
VIRG: ","

%import common.WORD
%import common.WS
%ignore WS
'''

f = open("teste", "r")
cod = f.read()
p = Lark(grammar)
parse_tree = p.parse(cod)
data = MyInterpreter().visit(parse_tree)


print('\n')
print("---------------------- PERGUNTA 1 -----------------------------")
print("Variaveis redeclaradas: " + str(data['red_vars']))
print("Variavies nao declaradas: " + str(data['notD_vars']))
print("Variavies nao inicializadas mas usadas: " + str(data['notI_vars']))
print("Variavies nunca usadas: " + str(data['inuteis_vars']))

print('\n')
print("---------------------- PERGUNTA 2 -----------------------------")
for k, v in data['estruturas'].items():
    if k == 'dict':
        print("Total de dicionarios: " + str(v))
    elif k == 'array':
        print("Total de arrays: " + str(v))
    elif k == 'tuplo':
        print("Total de tuplos " + str(v))
    else:
        print("Total de sets: " + str(v))


print('\n')
print("---------------------- PERGUNTA 3 -----------------------------")
total = 0
for k, v in data['ins'].items():
    total += v
    if k == 'decl':
        print("Total de declaracoes: " + str(v))
    elif k == 'for':
        print("FOR: " + str(v))
    elif k == 'foreach':
        print("FOREACH: " + str(v))
    elif k == 'while':
        print("WHILE: " + str(v))
    elif k == 'if':
```

```
355         print("IF: " + str(v))
356     elif k == 'ifelse':
357         print("IF & ELSE: " + str(v))
358     elif k == 'escrever':
359         print("ESCRITA: " + str(v))
360     else:
361         print("LEITURA: " + str(v))
362 
363 print("Total de instrucoes: " + str(total))
364 
365 print('\n')
366 print("---------------------- PERGUNTA 4 -----------------------------")
367 print("Total de estruturas aninhadas: " + str(data['aninhadas']))
368 
369 print(data['tmp'])
370 fileTXT = open('teste', 'r')
371 fileHTML = open('relatorio.html', 'w')
372 
373 fileHTML.write(f'''
374 <!DOCTYPE html>
375 <html>
376 <head>
377     <meta charset="UTF-8">
378     <link rel="stylesheet" href="w3.css">
379     <title>Relatorio</title>
380 </head>
381 <style>.code''' + "{position: relative; display: inline-block }" + '''
382 </style>
383 <body>
384 <h1 class="w3-container w3-teal">Pergunta 1</h1>
385 <div class="w3-container w3-margin w3-margin-top w3-padding w3-display-right">
386     <p><span style="color:red"> Vermelho</span> - Variaveis redeclaradas</p>
387     <p><u>Sublinhado</u> - Variaveis nao declaradas</p>
388     <p><b>Negrito</b> - Variaveis usadas e  nao inicializadas</p>
389     <p><i>Italico</i> - Variaveis declaradas mas nunca usadas</p>
390 
391 </div>
392 <pre class="w3-container"><code>
393 ''')
394 alterado = False
395 for line in fileTXT:
396     mat = re.findall(r'((var|dict)\s)([a-zA-Z]+)', line)
397     variavesHTML = set()
398     for ele in mat:
399         variavesHTML.add(ele[2])
400     for red in data['red_vars']:
401         if red in variavesHTML:
402             alterado = True
403             line = re.sub(
404                 rf'\b{red}\b', '<span style="color:red">' + red + '</span>', line
    )
405 
406     for ele in data['notD_vars']:
407         if re.search(rf'\b{ele}\b', line):
```

```
            line = re.sub(rf'\b{ele}\b', '<u>' + ele + '</u>', line)

    for ini in data['notI_vars']:
        if re.search(rf'\b{ini}\b', line):
            line = re.sub(rf'\b{ini}\b', '<b>' + ini + '</b>', line)

    for inuteis in data['inuteis_vars']:
        if re.search(rf'\b{inuteis}\b', line):
            line = re.sub(rf'\b{inuteis}\b', '<i>' + inuteis + '</i>', line)

    fileHTML.write('<p class="code">' + line + "</p>\n")
fileHTML.write(f'''<h1 class="w3-container w3-teal">Pergunta 2</h1>
<table class="w3-table-all w3-centered w3-light-grey">
    <tr>
        <th> Tipo de estrutura</th>
        <th> Total</th>
    </tr>
    <tr>
    ''')
for k, v in data['estruturas'].items():
    fileHTML.write('<td>' + k + '</td>')
    fileHTML.write('<td>' + str(v) + '</td></tr>')

fileHTML.write(f'''</table><h1 class="w3-container w3-teal">Pergunta 3</h1>
<table class="w3-table-all w3-centered w3-light-grey">
    <tr>
        <th> Tipo de instrucao</th>
        <th> Total</th>
    </tr>
    <tr>
    ''')
for k, v in data['ins'].items():
    fileHTML.write('<td>' + k + '</td>')
    fileHTML.write('<td>' + str(v) + '</td></tr>')

fileHTML.write('</table><h1 class="w3-container w3-teal">Pergunta 4</h1></table><h3><p>Total de instrucoes - ' + str(
    total) + '</p><p> Total de estruturas aninhadas - ' + str(data['aninhadas']) + '</p></h3><h1 class="w3-container w3-teal">Pergunta 5</h1><pre class="w3-container"><code>')

i = 1
fileTXT = open('teste', 'r')
for line in fileTXT:
    ifs = re.findall(r'^ *if\(', line)
    if ifs:
        if i in data['tmp']:
            l = '<span style="color:green"># O seguinte par de ifs pode ser susbtituido por uma unica condi  o </span><span>' + \
                "\n" + line + ' </span>'
        else:
            l = "<span>" + line + "</span>"
        i = i+1
    else:
```

```
458         l = "<span>" + line + "</span>"
459
460     fileHTML.write(l)
461
462 fileHTML.write("</code></pre></body></html>")
```