

Perfil EL - Engenharia de Linguagens
(1º ano do MEI)
Engenharia Gramatical
Trabalho Prático 2
Relatório de Desenvolvimento

João Gonçalves
PG46535

Sara Queirós
PG47661

25 de abril de 2022

Resumo

O presente documento tem como principal objetivo explicar o desenvolvimento de um analisador de código para uma linguagem LPIS2, criada por nós, e gerar um HTML com o resultado das análises feitas ao documento em causa.

Conteúdo

1	Introdução	2
2	Análise e Especificação	3
2.1	Descrição informal do problema	3
2.2	Especificação do Requisitos	3
2.2.1	Linguagem	3
2.2.2	Analizador	3
3	Concepção/Desenho da Resolução	5
3.1	Gramática da Linguagem LPIS2	5
3.1.1	Variáveis	5
3.1.2	Atribuições	6
3.1.3	Instruções Condicionais	6
3.1.4	Input e Output	7
3.1.5	Ciclos	7
3.1.6	Declaração de Funções	7
3.2	Parser + Analizador	7
3.2.1	Start	8
3.2.2	Função Auxiliar	8
3.2.3	Declarações	8
3.2.4	Instruções	10
3.2.5	Input e Output	11
3.2.6	Ciclos	12
3.2.7	IFs	14
4	Testes	19
4.1	Testes realizados e Resultados	19
4.1.1	Exemplo 1	20
4.1.2	Exemplo 2	24
5	Conclusão	27

Capítulo 1

Introdução

Ferramentas avançadas de análise de código têm como intuito ajudar em tarefas diferentes tais como embelezar o código, detetar situações que infringem boas práticas de codificação, sugestões de simplificação de código sem alterar o seu significado inicial e até mesmo avaliar a sua performance estática ou dinâmica.

Face a isto, o objetivo principal é criar um analisador de código para a linguagem de Programação Imperativa Simples, criadas por nós, e após efetuada essa análise, os resultados devem ser expostos num HTML.

Estrutura do Relatório

Este relatório é constituído por capítulos que englobam a Análise e Especificação do problema, descrevendo-o e os seus objetivos principais, Conceção e Desenho da Resolução (explorando o raciocínio utilizado na construção da gramática e do Analisador + Parser) e ainda, um capítulo com apresentação dos testes efetuados. Finalmente, no último capítulo é exposta uma conclusão e trabalho futuro.

Capítulo 2

Análise e Especificação

2.1 Descrição informal do problema

Desenvolver um analisador de código para uma linguagem definida por nós e expôr esses resultados num HTML.

2.2 Especificação do Requisitos

Para realizar este trabalho é necessário ter em conta os requisitos presentes no enunciado:

2.2.1 Linguagem

A linguagem será designada por LPIS2 e deve permitir:

- Declarar variáveis atômicas;
- Declarar variáveis estruturadas (conjunto, lista, tuplo, dicionario);
- Instruções condicionais;
- Três Variantes de ciclo, pelo menos.

2.2.2 Analisador

O analisador deve ser escrito em Python, usando o Parser e Visitors do módulo Lark.Interpreter e deve produzir resultado em HTML capaz de:

- Listar variáveis redeclaradas, não declaradas, utilizadas e não inicializadas, declaradas e nunca mencionadas.
- Tipos de estruturas para as variáveis declaradas.
- Total de Instruções do programa e o respetivo número de instruções para cada tipo: atribuições, declarações, leitura e escrita, condicionais, cíclicas e declarações de funções.

- Total de estruturas de controlo aninhadas.
- Informações face a possíveis simplificações para ifs.

Capítulo 3

Concepção/Desenho da Resolução

3.1 Gramática da Linguagem LPIS2

Para ser possível definir a linguagem foi gerada a produção inicial:

```
start: code
code: (variaveis | cond | output | ciclos | funcao)+
```

Assim, o código pode conter variáveis, declarações de output, funções de output, estruturas condicionais e de controlo.

Nota: Grande parte dos símbolos terminais poderiam ser substituídos pelos respetivos caracteres nas produções em que são utilizados. No entanto, como queremos reutilizá-los para elaborar o HTML, consideramos conveniente defini-los desta forma.

3.1.1 Variáveis

As principais operações que se podem efetuar com variáveis são declarações e atribuições.

```
variaveis: declaracoes | atribuicoes
```

Em relação às declarações, existem 9 tipos que definimos como possíveis:

```
declaracoes: decint | decstring | declista | decdict | deconj | dectuplos | decfloat |
             decinput | decvallist
```

Para cada um dos tipos foi definida uma produção que permite identificar as respetivas declarações:

```
decvallist: (INTW | STRINGW | FLOATW) WORD IGUAL WORD (PRE INT PRD)+ PV
decint : INTW WORD (IGUAL INT (operacao))?. PV
declista : INTW WORD PRE PRD IGUAL CE (INT ( VIR INT)*)?. CD PV
decstring: STRINGW WORD (IGUAL ESCAPED_STRING)?. PV
decdict: DICTW WORD (IGUAL DICTW PE PD )?. PV
deconj: CONJW WORD (IGUAL CE (ESCAPED_STRING (VIR ESCAPED_STRING)*)?. CD )* PV
```

```
dectuplos: TUPLEW WORD (IGUAL PE var (VIR var)+ PD)* PV
decfloat: FLOATW WORD (IGUAL FLOAT)* PV
decinput: STRINGW IGUAL input
```

Cada símbolo não terminal que acabe com a letra "W" tem o intuito de representar a palavra correspondente ao tipo:

```
INTW: "int"
INPUTW: "input"
OUTPUTW: "print"
STRINGW: "string"
DICTW: "dict"
CONJW: "conj"
TUPLEW: "tuple"
FLOATW: "float"
```

3.1.2 Atribuições

Uma vez que em atribuições não é necessário fazer o controlo dos tipos das variáveis, foram apenas definidas as seguintes produções para gerir os novos valores dados às variáveis.

```
atribuicoes: WORD IGUAL ((var operacao? PV) | input | lista | dicionario)
var: INT | WORD | ESCAPED_STRING | FLOAT
operacao: ((SUM|SUB|MUL|DIV|MOD) INT)+
input: INPUTW PE PD PV
lista: WORD (PRE INT PRD)+ PV
dicionario: CE ESCAPED_STRING DP (INT | WORD)(VIR ESCAPED_STRING DP (INT | WORD))* CD PV
```

Desta forma, conseguimos generalizar para atribuir valores como sendo operações aritméticas, leituras de inputs, outras variáveis e até mesmo elementos de listas, conjuntos e dicionários, que são geridos da mesma forma.

3.1.3 Instruções Condicionais

Relativamente às instruções condicionais, a produção principal é "cond". Assim, esta é constituída principalmente pelos Tokens que representam as instruções, a condição que caracteriza o "if" e ainda um possível "else", caso se queira implementar.

```
cond: IFW PE condicao PD CE code? CD elsee? PV
condicao: var ((II|MAIOR|MENOR|DIF|E|OU) var)?
elsee: ELSEW CE code CD
IFW: "if"
ELSEW: "else"
```

No corpo destas instruções é possível escrever código, que inclui qualquer outro tipo de instrução.

3.1.4 Input e Output

Para as opções de interação com o utilizador, criamos duas produções essenciais. Para imprimir no ecrã deverá ser invocada a função "print". Por sua vez, para ler input, basta recorrer à invocação da função "input":

```
input: INPUTW PE PD PV
output: OUTPUTW PE ESCAPED_STRING PD PV
INPUTW: "input"
OUTPUTW: "print"
```

3.1.5 Ciclos

Na nossa linguagem foram definidas 3 variantes de ciclo que podem ser utilizadas: for, while e do while que são traduzidas pela gramática seguinte.

```
ciclos: whilee | forr | dowhile
whilee: WHILEW PE condicao PD CE code? CD PV
forr: FORW PE variaveis condicao PV WORD IGUAL tipo PD CE code? CD PV
dowhile: DOW CE code? CD WHILEW PE condicao PD PV
WHILEW: "while"
DOW: "do"
FORW: "for"
```

É de notar que cada ciclo é composto por respetivas condições e pode existir mais código no corpo do ciclo

3.1.6 Declaração de Funções

Finalmente, na nossa linguagem também é possível definir funções seguinte uma notação parecida a python, apenas distinguindo na troca do símbolo ":" por chavetas.

```
funcao: DEFW WORD PE args PD CE code? return? CD
args: (types WORD VIR)* types WORD
types: (STRINGW | DICTW | INTW | TUPLEW | FLOATW | CONJW)
return: RETURNW (WORD VIR)* WORD
```

Tal como é possível perceber através da gramática, a função pode receber vários argumentos e retornar os elementos que o utilizador quiser.

3.2 Parser + Analisador

Para gerar a ferramenta de análise, usando o Parser e os Visitors do módulo para geração de processadores de linguagens Lark.Interpreter, começamos por definir a inicialização do parser, e face aos requisitos, criar as variáveis para controlo das estatísticas.

É de notar que decidimos apresentar o código original com anotações em relações as variáveis não declaradas, não inicializadas e redeclaradas e, ainda, possíveis simplificações nas intruções condicionais no ficheiro resultado HTML. Por esse motivo, à medida que analisamos a árvore adicionamos o conteúdo da mesma à variável "self.html", o que fez com que o código do parser se tornasse um pouco mais complexo.

```
def __init__(self):
    self.varsDecl = dict() #variaveis declaradas
    self.varsNDecl = dict () #variaveis nao declaradas
    self.varsRDecl = dict() #variaveis redeclaradas
    self.tipoInstrucoes = {'declaracoes': 0, 'atribuicoes': 0, 'io': 0, 'ciclos': 0,
        'cond': 0, 'funcoes': 0}
    self.totalInst = 0
    self.forC = 0 #Controlo auxiliar dentro de um ciclo for
    self.inInst = {'atual': 0, 'maior': 0, 'total': 0} #Controlo Inst aninhadas
    self.aninhavel = False #Verificar se da para juntar ifs
    self.html = str(...) #Inicializacao do HTML com definicoes necessarias
```

3.2.1 Start

Na produção de start apenas é inicializada a visita à árvore, desencadeando toda a análise da mesma. Após efetuada a análise, os dados finais são escritos no HTML recorrendo à função "dadosfinais()" que será explorada ao longo deste relatório.

```
def start(self, tree):
    #start: code
    self.visit(tree.children[0])
    self.html = self.html + self.dadosfinais() + str('</body></html>')
    return self.html
```

3.2.2 Função Auxiliar

Com o intuito de calcular o maior nível de aninhamento existente entre instruções foi criada a função seguinte, que cada vez que é invocada verifica se o registo que possui face ao maior valor ainda se mantém atualizado. Caso não esteja, atualiza-o.

```
def maior(self):
    if self.inInst['atual'] > self.inInst['maior']:
        self.inInst['maior'] = self.inInst['atual']
```

3.2.3 Declarações

Assim que é reconhecida a regra de declaração de uma variável, é recalculado o valor do aninhamento, uma vez que declarações podem ocorrer dentro de instruções condicionais, e incrementados

os contadores das respetivas instruções:

```
def declaracoes(self, tree):
    self.maior()
    self.totalInst +=1
    self.tipoInstrucoes['declaracoes'] += 1
```

Depois, é iniciada a visita ao primeiro elemento da árvore, que indica o tipo de variável em questão. Por gestão de apresentação em HTML nos ciclos for, se esta declaração não acontecer no interior do corpo de um, deve adicionar-se um parágrafo.

```
var = self.visit(tree.children[0])
if (self.forC== 0):
    self.html = self.html + '<p>'
```

Posto isso, é iniciado o tratamento das variáveis. Caso esta ainda não tenha sido declarada, é adicionada ao respetivo dicionário, etiquetando o seu tipo e inicialização.

```
#Verificar que a var ainda nao foi declarada
if var[1] not in self.varsDecl:
    if(var[2] != ";"):
        if(var[2] == "["): #caso de var lista
            self.varsDecl[var[1]] = {"tipo" : var[0] + "[]", "inic" : 1, "utilizada": 0}
        else:
            self.varsDecl[var[1]] = {"tipo" : var[0], "inic" : 1, "utilizada": 0}
    else:
        self.varsDecl[var[1]] = {"tipo" : var[0], "inic" : 0, "utilizada": 0}
    self.html = self.html + "<p class='code'>"+ "\n" + var[0] + " " + var[1] + " "
```

Para o caso de a variável já ter sido declarada anteriormente, esta é anotado a azul (com uma classe própria para tal) no HTML gerado.

```
#Se ja foi declarada e anunciada com uma classe propria para tal
else:
    self.varsRDecl[var[1]] = {"tipo" : var[0]}
    self.html = self.html + "<p class='code'><div class='redeclared'>"+ var[0] + " " +
        var[1]+ "<span class='redeclaredtext'>Varivel redeclarada</span></div> "
```

Para os restantes terminais existentes, verificam-se casos particulares como operações com recursos a outras variáveis, para fazer a respetiva interpretação das mesmas, caso sejam Tokens do tipo WORD, tal como anteriormente.

```
for elem in var[2:]:
    #decl tuples e ints com operacoes
    if isinstance(elem, Token):
        #Casos como: int x = y + 1
        if elem.type == 'WORD' and elem not in self.varsDecl:
            self.html = self.html + "<p class='code'><div class='error'> "+ elem + "
                <span class='errortext'>Varivel no declarada</span></div> "
```

```

else:
    if elem.type == 'WORD':
        self.varsDecl[elem]["utilizada"] += 1
        self.html = self.html + elem + " "

```

Para símbolos que não sejam Tokens, itera-se o conjunto dos seus filhos de forma a obter todos os terminais utilizados para escrita no HTML.

```

else:
    for i in elem:
        self.html = self.html + i + " "
if (self.forC == 0):
    self.html = self.html + '</p>'
self.html = self.html + "</p>\n"

```

3.2.4 Instruções

De uma forma muito semelhante ao racínio utilizado nas declarações, iniciou-se a função "instrucoes".

```

def atribuicoes(self, tree):
    self.totalInst += 1
    self.maior()
    self.tipoInstrucoes['atribuicoes'] += 1
    var = self.visit_children(tree)
    if (self.forC == 0):
        self.html = self.html + '<p>'

```

Mais uma vez é efetuado o controlo face às variáveis que estão a ser utilizadas, utilizando e anotando cada uma na respetiva cor face ao alerta que geram.

```

if (var[0] not in self.varsDecl):
    self.varsNDecl[var[0]] = {}
    self.html = self.html + "<p class='code'><div class='error'> "+ var[0]+ "<span
        class='errortext'>Varivel no declarada</span></div>"
else:
    self.varsDecl[var[0]]["utilizada"] += 1
    self.varsDecl[var[0]]["inic"] = 1
    self.html = self.html + "<p class='code'>" + var[0] + " "

```

Os restantes filhos constituem os valores dados na atribuição e são analisados através de um ciclo, verificando se se trata de outra variável, ou casos particulares que não sejam Tokens. Assim, analisa-se a sua validade em contexto, anotando-a no HTML caso nunca tenha sido declarada.

Caso contrário, por se tratarem de símbolos terminais são diretamente adicionados à string.

Cada vez que uma variável declarada é utilizada é incrementado o contador que esta possui para o efeito.

```

for elem in var[1:]:
    if isinstance(elem, Token):
        self.html = self.html + elem + " "
    else:
        for i in elem:
            #Utilizacao de variaveis nao declaradas direita da operao
            if i.type == 'WORD':
                if i not in self.varsDecl:
                    self.varsNDecl[i] = {}
                    self.html = self.html + "<p class='code'><div class='error'> "+ i + "
                        <span class='errortext'>Varivel no declarada</span></div> "
                else:
                    self.varsDecl[i]["utilizada"] += 1
                    self.html = self.html + i + " "
            else:
                self.html = self.html + i + " "
if (self.forC == 0):
    self.html = self.html + '</p>'
self.html = self.html + "</p>\n"

```

3.2.5 Input e Output

Na função de input apenas retorna a visita que faz ao seus filhos, sendo este resultado utilizado nas declarações de variáveis e atribuições pois só faz sentido ler input para uma variável. Mais uma vez, as variáveis de controlo de número de instruções são atualizadas.

```

def input(self, tree):
    self.maior()
    self.totalInst += 1
    self.tipoInstrucoes['io'] += 1
    return self.visit_children(tree)

```

Por sua vez, através da regra para identificar a menção de instruções de output, é feita a visita aos filhos da árvore correspondente e, por apenas possuir tokens, estes são adicionados ao HTML.

```

def output(self, tree):
    self.maior()
    tokens = self.visit_children(tree)
    self.totalInst += 1
    self.tipoInstrucoes['io'] += 1
    self.html = self.html + "<p><p class='code'>"
    for t in tokens:
        self.html = self.html + t + " "
    self.html = self.html + "</p></p>\n"
    return tree

```

3.2.6 Ciclos

A principal produção representante dos ciclos, para além das instruções comuns, apenas faz a visita à árvore do filho que corresponde à instrução em causa, ou seja, uma das 3 variantes de ciclo

```
def ciclos(self, tree):
    self.maior()
    self.totalInst += 1
    self.tipoInstrucoes['ciclos'] += 1
    result = self.visit(tree.children[0])
    return result
```

While

Considerando a produção para o while, os dois primeiros símbolos são Tokens e podem ser tratados da mesma forma.

```
def whilee(self, tree):
#whilee: WHILEW PE condicao PD CE code? CD PV
    self.html = self.html + "<p><p class='code'>"
    for i in range(2):
        self.html = self.html + tree.children[i] + " "
```

A "condicao", por se tratar de outra produção com análise complexa, requer uma visita à árvore que gera.

```
self.visit(tree.children[2])
self.html = self.html + tree.children[3] + " " + tree.children[4] + " "
```

Para ser possível contar a quantidade de estruturas aninhadas, a variável "inInst['atual']" é incrementada pois estamos dentro de uma variante de ciclo. Caso existe código no interior desta estrutura, será possível efetuar a contagem corretamente para a eventual existência de outras estruturas.

```
#Significa que tem codigo no meio
self.inInst['atual'] += 1
if len(tree.children) == 8:
    self.visit(tree.children[5])
    self.html = self.html + tree.children[6] + " " + tree.children[7]
else:
    self.html = self.html + tree.children[5] + " " + tree.children[6]
```

Ao chegar a esta fase de análise significa que todo o código no corpo do ciclo já foi processado e então, está a terminar o while e por isso, o contador do nível de aninhamento pode ser decrementado. Caso se trate da estrutura mais "exterior" aos aninhamentos, incrementa-se a variável que conta o total de ocorrências deste tipo.

```
self.inInst['atual'] -= 1
```

```

if self.inInst['atual'] == 1:
    self.inInst['total'] += 1
self.html = self.html + "</p></p>\n"

```

For

Um ciclo for é composto por 3 componentes separadas por ";". Após os procedimentos iniciais de incrementar as variáveis relativas aos números de instruções, são visitados os símbolos "variaveis" e "condicao", que correspondem respetivamente à 1ª e 2ª componentes referidas.

```

def forr(self, tree):
    #forr: FORW PE variaveis condicao PV WORD IGUAL tipo PD CE code? CD PV
    self.inInst['atual'] += 1
    self.forC = 1
    self.html = self.html + "<p><p class='code'>" + tree.children[0] + " " +
        tree.children[1]
    self.totalInst += 1
    self.tipoInstrucoes['atribuicoes'] += 1
    self.visit(tree.children[2])
    self.visit(tree.children[3])
    self.html = self.html + tree.children[4]

```

Relativamente à 3ª componente, esta é composta geralmente por uma atribuição. Por envolver variáveis, são realizados os procedimentos habituais de verificação da validade da variáveis e a sua escrita no HTML em função desse resultado. A variável "forC" é utilizada para perceber necessidade, ou não, da utilização de parágrafos dentro do for.

```

if (tree.children[5] not in self.varsDecl):
    self.varsNDecl[tree.children[5]] = {}
    self.html = self.html + "<p class='code'><div class='error'> "+ tree.children[5]+
        "<span class='errortext'>Varivel no declarada</span></div>"
else:
    self.varsDecl[tree.children[5]]["utilizada"] += 1
    self.varsDecl[tree.children[5]]["inic"] = 1
    self.html = self.html + tree.children[5] + " "
    self.html = self.html + tree.children[6] + " "
    self.visit(tree.children[7])
    self.html = self.html + tree.children[8] + " " + tree.children[9] + " "
    self.forC = 1

```

Devido à possibilidade de existir código dentro da estrutura, caso o filho na posição 10 não seja um Token, sabe-se que é necessário efetuar a visita a essa árvore. Na situação de ser um Token, significa que não existe mais código no interior e os restantes símbolos podem ser tratados como Tokens. Mais uma vez, utiliza-se a lógica da variável "inInst" para a deteção da presença de aninhamentos.

```

if (isinstance(tree.children[10], Token)):

```

```

        self.html = self.html + tree.children[10] + " " + tree.children[11] + " "
    else:
        self.visit(tree.children[10])
        self.html = self.html + tree.children[11] + " " + tree.children[12] + " "
    self.inInst['atual'] -= 1
    if self.inInst['atual'] == 1:
        self.inInst['total'] += 1
    self.html = self.html + "</p></p>\n"

```

Quando visita a árvore da produção "tipo" no ciclo for, são extraídos apenas os Tokens, através de visitas às árvores dos respectivos filhos para utilização destes no HTML.

```

def tipo(self, tree):
    #var operacao?
    #operacao: ((SUM|SUB|MUL|DIV|MOD) INT)+
    var = self.visit(tree.children[0])
    self.html = self.html + var[0]
    operacao = self.visit(tree.children[1])
    for op in operacao:
        self.html = self.html + op

```

Do-While

Por se tratar de uma instrução simples, os elementos que não forem Tokens são visitados e os restantes são obtidos diretamente.

```

def dowhile (self, tree):
    #dowhile: DOW CE code? CD WHILEW PE condicao PD PV
    self.html = self.html + "<p><p class='code'>"
    for elem in tree.children:
        if isinstance(elem, Token):
            self.html = self.html + elem + " "
        else:
            self.visit(elem)
    self.html = self.html + "</p>\n"

```

3.2.7 IFs

As instruções comuns, novamente, são realizadas no início da função "cond".

```

def cond(self, tree):
    #cond: IFW PE condicao PD CE code? CD else? PV
    self.totalInst += 1
    self.inInst['atual'] += 1
    self.tipoInstrucoes['cond'] += 1

```



```
self.html = self.html + "<p><p class='code'>"
```

Os primeiros 5 elementos da regra são tratados de forma muito semelhante aos anteriormente mencionados, distinguindo os Tokens dos restantes, tendo tratamentos distintos entre si.

```
for elem in tree.children[:5]:
    if isinstance(elem, Token):
        self.html = self.html + elem + " "
    else:
        self.visit(elem)
```

A partir do 5º elemento, inicia-se a análise de possíveis simplificações de condições. Para isso é necessário que a não haja nenhuma instrução entre o primeiro if e o aninhado. Assim, se o primeiro filho da produção "code" for "cond", significa que é possível simplificar.

```
if not isinstance(tree.children[5], Token): #condies para os ifs aninhados,
    sinalizar se puder simplificar
if tree.children[5].children[0].data == 'cond':
    self.aninhavel = True
else:
    self.aninhavel = False
self.visit(tree.children[5])
self.html = self.html + tree.children[6] + " "
```

Após isto, analisamos se existe algum else associado. Se houver, será feita a respetiva visita.

```
if not isinstance(tree.children[7], Token): #Se houver um else
    self.visit(tree.children[7])
    self.html = self.html + tree.children[8] + " "
else:
    self.html = self.html + tree.children[7] + " "
```

No caso em que não existe código dentro do if mas existe "else", esta função terá um comportamento semelhante. No final, é efetuado o controlo de aninhamentos.

```
else:
    self.html = self.html + tree.children[5] + " "
    if not isinstance(tree.children[6], Token): #Se houver um else sem codigo no if
        self.visit(tree.children[6])
        self.html = self.html + tree.children[7] + " "
    else:
        self.html = self.html + tree.children[6] + " "
self.html = self.html + "</p>\n"
self.inInst['atual'] -=1
if self.inInst['atual'] == 1:
    self.inInst['total'] += 1
```

A função "else" comporta-se de forma muito simples, semelhante a funções descritas anteriormente:

```
def elsee(self,tree):
    #elsee: ELSEW CE code CD
    for elem in tree.children:
        if not isinstance(elem, Token):
            self.visit(elem)
        else:
            self.html = self.html + elem + " "
```

Funções

Após possuir o comportamento habitual, os 6 primeiros símbolos são analisados de forma comum.

```
def funcao(self, tree):
    #funcao: DEFW WORD PE args PD CE code? return? CD
    self.totalInst += 1
    self.tipoInstrucoes['funcoes'] += 1
    self.html = self.html + "<p><p class='code'>"
    for elem in tree.children[:6]:
        if isinstance(elem, Token):
            self.html = self.html + elem + " "
        else:
            self.visit(elem)
```

A análise varia a partir do 6º elemento pois existe a possibilidade de o 6º e o 7º não existirem. Assim é necessário garantir as várias possibilidades de visita e análise às várias combinações existentes com "code" e "return".

```
if not isinstance(tree.children[6], Token):
    if tree.children[6].data == 'code':
        self.visit(tree.children[6])
        if not isinstance(tree.children[7], Token): #SE nao for um token return
            self.html = self.html + "<p>"
            for i in tree.children[7].children:
                self.html = self.html + i + " "
            self.html = self.html + "</p>" + tree.children[8] + " "
        else:
            self.html = self.html + tree.children[7] + " "
    elif tree.children[6].data == 'return':
        self.html = self.html + "<p>"
        for i in tree.children[6].children:
            self.html = self.html + i + " "
        self.html = self.html + "</p>" + tree.children[7] + " "
```

Por se tratar de uma função e as variáveis dadas como input não se poderem reutilizar fora, têm que ser eliminadas:

```

d = copy.deepcopy(self.varsDecl)
for key in d:
    if (self.varsDecl[key]['tipo'] == None):
        del self.varsDecl[key]

```

A única visita que é feita com garantias a partir da produção "funcao" é a "args" que representa a lista de argumentos que podem ser dados à função na sua declaração. Esta pode ser tratada com bastante simplicidade como se vê abaixo:

```

def args(self, tree):
    for elem in tree.children:
        if isinstance(elem, Token):
            self.html = self.html + elem + " "
            if (elem.type == 'WORD'):
                self.varsDecl[elem] = {"tipo": None, "inic": 1, "utilizada": 0}
        else:
            self.html = self.html + elem.children[0] + " "

```

As variáveis dadas como argumento são provisoriamente colocadas no dicionário de variáveis declaradas.

Condição

Tal como mencionado nos tópicos anteriores, cada instrução condicional possui uma condição que deve ser analisada e processada:

```

def condicao(self, tree):
    #var ((II|MAIOR|MENOR|DIF|E|OU) var)?
    self.totalInst += 1
    child = self.visit_children(tree)
    flag = True #Para saber se h algum operador a escrever no meio
    first = child[0][0]
    if len(child) == 3:
        l = [first, child[2][0]]
    else:
        l = [first]
    flag = False

```

Para verificar possíveis simplificações de código, se a flag "aninhavel" estiver a True, a condição é escrita a verde no HTML (através de uma classe criada para tal).

```

if self.aninhavel == True: #Se for aninhavel colocamos a verde
    self.html = self.html + "<div class='aninh'>"

```

Além disto disto, verifica-se a validade das variáveis utilizadas e a sua cor varia, tal como já foi descrito anteriormente.

```

for var in l:
    if var.type == "WORD": #Se for uma variavel temos de ver se ela declarada ou
        assim para anotar o codigo
        if var not in self.varsDecl:
            self.varsNDecl[var] = {}
            self.html = self.html + "<div class='error'> "+ var + "<span
                class='errortext'>Varivel no declarada</span></div>"
        else:
            self.varsDecl[var]["utilizada"] += 1
            if self.varsDecl[var]["inic"] == 0:
                self.html = self.html + "<div class='notinic'> "+ var + "<span
                    class='notinictext'>Varivel no inicializada</span></div>"
            else:
                self.html = self.html + var
        else:
            self.html = self.html + " " + var
    if (flag):
        self.html = self.html + " " + child[1] + " "
        flag = False
if self.aninhavel == True:
    self.html = self.html + "<span class='aninhtext'>Pode simplificar com a condio
        anterior utilizando '&&'</span></div>"

```

Capítulo 4

Testes

4.1 Testes realizados e Resultados

As anotações existentes no código possuem cores diferentes para resultados diferentes. Ao passar o rato por cima é possível ver a mensagem que lhe é subjacente:

Variável redeclarada
`dict d = dict () ;`

Figura 4.1: Anotações a azul

Variável não declarada
`x1 = 2 + 4 ;`

Figura 4.2: Anotações a vermelho

Variável não inicializada
`while (h < 4) {`

Figura 4.3: Anotações a cor-de-laranja

Pode simplificar com a
condição anterior utilizando
'&&'
`if (x > 5) {`

Figura 4.4: Anotações a verde

4.1.1 Exemplo 1

Neste exemplo são apresentados todos os casos mais básicos relativos a todas as funcionalidades que a nossa linguagem permite, nomeadamente a declaração dos diferentes tipos de variáveis e atribuições, exemplos dos 3 tipos de estruturas cíclicas, assim como condicionais e, ainda, declarações de funções e operações de input/output.

Destaca-se também a existência de estruturas aninhadas para efeitos de contagem e análise de possíveis simplificações que são anotadas.

```
int x = 4;
int h;
string l;
string m = "algo";
int list [] = {};
int lista [] = {12, 12};
dict d;
string s = lista[0];
s = s[1];
s = m[1];
dict d = dict();
conj c;
conj c1 = {"algo", "mais algo"};
tuple t;
tuple t1 = (1, "cenas");
tuple t2 = (1, "cenas", 2);
float f;
float f1 = 3.3;
x = "coisas";
x1 = 2 + 4;
s = input();
print("teste");
if( x > 5){};
if("cois" == "coisa" ){
    print(" at funciona");
};
while (h < 4){
    string x1 = "w" ;
};
for ( int i = 0; i < 1 ; i = i + 1) {};
do{
    s = input();
} while( 1 == 1);
string s = "string";
if( x > 5){
if(x > 5 ){
    print(" at funciona");
    if(x == 25 ){
        if(s ){}}
```

```
        else {  
            print("fixe"); }  
        };  
    };  
};  
d = {"algo":1};
```

Código anotado:

```
int x = 4 ;
int h ;
string l ;
string m = "algo" ;
int list [ ] = { } ;
int lista [ ] = { 12 , 12 } ;
dict d ;
string s = lista [ 0 ] ;
s = s [ 1 ] ;
s = m [ 1 ] ;
dict d = dict ( ) ;
conj c ;
conj c1 = { "algo" , "mais algo" } ;
tuple t ;
tuple t1 = ( 1 , "cenas" ) ;
tuple t2 = ( 1 , "cenas" , 2 ) ;
float f ;
float f1 = 3.3 ;
x = "coisas" ;
x1 = 2 + 4 ;
s = input ( ) ;
print ( "teste" ) ;
if ( x > 5 ) { } ;
if ( "cois" == "coisa" ) {
print ( "até funciona" ) ;
} ;
while ( h < 4 ) {
string x1 = "w" ;
} ;
for (int i = 0 ; i < 1 ; i = i+1) { } ;
do {s = input ( ) ; } while ( 1 == 1 ) ;
string s = "string" ;
if ( x > 5 ) {
if ( x > 5 ) {
print ( "até funciona" ) ;
if ( x == 25 ) {
if ( s ) { } else {
print ( "fixe" ) ;
} ;
} ;
} ;
} ;
d = { "algo" : 1 } ;
```

22

Figura 4.5: Exemplo de Código

-----Análise Geral-----

Variáveis declaradas:

- **x**: int
- **h**: int
- **l**: string
- **m**: string
- **list**: int[]
- **lista**: int[]
- **d**: dict
- **s**: string
- **c**: conj
- **c1**: conj
- **t**: tuple
- **t1**: tuple
- **t2**: tuple
- **f**: float
- **f1**: float
- **x1**: string
- **i**: int

Variáveis não declaradas:

- **x1**

Variáveis redeclaradas:

- **d**
- **s**

Variáveis declaradas e nunca mencionadas:

- **l**
- **list**
- **c**
- **c1**
- **t**
- **t1**
- **t2**
- **f**
- **f1**
- **x1**

-----Análise Instruções-----

Nº Declarações	19
Nº Atribuições	8
Nº Input/Output	6
Nº Ciclos	3
Nº Inst. Condicionais	6
Nº Funções	0
Total Instruções	51

NOTA: Existem **1** situações de aninhamento e o nível máximo de instruções condicionais aninhadas é **4**. Sugestões de simplificação são mencionadas no código acima.

Figura 4.6: Resultado devolvido pelo nosso analisador

4.1.2 Exemplo 2

Para este o seguinte input, obtém-se o output HTML exposto a seguir:

```
def exemplo(string s){
  int x = 4 * 4 - 3;
  s = "algo";
  int lista [] = {12, 14};
  dict d = dict();
  dict d;
  string h;
  int num = list[0];
  if(num == x ){
    print("isto at funciona");
  };
  while (h < "letra"){
    string x1 = "w" ;
    if(x == 25 ){
      if(lista){
      } else {
        tuple t1 = (1, "cenas");
        conj c1 = {"algo", "mais algo"};
      };
    };
  };
  for ( int i = 0; i < 1 ; i = i + 1) {};
  do{
    s = input();
  } while( 1 == 1);
  d = {"algo":1};
  return x, y
}
```

Código anotado:

```
def exemplo ( string s ) {  
    int x = 4 * 4 - 3 ;  
    s = "algo" ;  
    int lista [ ] = { 12 , 14 } ;  
    dict d = dict ( ) ;  
    dict d ;  
    string h ;  
    num = lista [ 0 ] ;  
    if ( num == x ) {  
        print ( "isto até funciona" ) ;  
    } ;  
    while ( h < "letra" ) {  
        string x1 = "w" ;  
        if ( x == 25 ) {  
            if ( lista ) { } else {  
                tuple t1 = ( 1 , "cenas" ) ;  
                conj c1 = { "algo" , "mais algo" } ;  
            } ;  
        } ;  
    } ;  
    for (int i = 0 ; i < 1; i = i+1) { } ;  
    do {s = input ( ) ; } while ( 1 == 1) ;  
    d = { "algo" : 1 } ;  
    return x , y  
}
```

-----Análise Geral-----	
Variáveis declaradas:	
<ul style="list-style-type: none"> • x: int • lista: int[] • d: dict • h: string • x1: string • t1: tuple • c1: conj • i: int 	
Variáveis não declaradas:	
<ul style="list-style-type: none"> • num 	
Variáveis redeclaradas:	
<ul style="list-style-type: none"> • d 	
Variáveis declaradas e nunca mencionadas:	
<ul style="list-style-type: none"> • x1 • t1 • c1 	
-----Análise Instruções-----	
Nº Declarações	9
Nº Atribuições	5
Nº Input/Output	2
Nº Ciclos	3
Nº Inst. Condicionais	3
Nº Funções	1
Total Instruções	29
NOTA: Existem 1 situações de aninhamento e o nível máximo de instruções condicionais aninhadas é 3 . Sugestões de simplificação são mencionadas no código acima.	

Figura 4.8: Análise Geral

Capítulo 5

Conclusão

Após a elaboração deste trabalho, podemos dizer que, como balanço geral, as maiores dificuldades surgiram no âmbito da geração do HTML, ao qual gostaríamos de ter implementado indentação, pois consideramos que a visualização de erros e sugestões, tornam a análise do utilizador, muito mais fácil.

Para além disso, de forma global, ao efetuar a elaboração da gramática tivemos uma melhor perceção de todas as variações que podem existir para código de qualquer linguagem. A abertura a vários tipos de estruturas de dados requer maior atenção e cuidado na gestão das informações contidas nelas. Com isto, a nossa perspetiva face a analisadores de código mudou, considerando a dimensão e precisão dos resultados que estes nos fornecem.

Em conclusão, o módulo Lark.Interpreter, facilita o alcançar do objetivo principal, uma vez que, a utilização de árvores é intuitiva. Desta forma, pretendemos utilizador o código desenvolvido e implementar aquilo que não conseguimos e estender a gramática, com o intuito de abranger outras possíveis situações.