Engenharia Gramatical
1º ano de Mestrado em Engenharia Informática

Trabalho Prático 3

Relatório de Desenvolvimento
Grupo 9

Júlio Alves (PG47390)
Rúben Cerqueira (PG47626)

30 de maio de 2022

Resumo

O presente relatório abordará a adição de conteúdo do trabalho prático anterior que consistiu num analisador de código da linguagem LPIS2. O interpretador será enriquecido com uma análise comportamental, com a geração de vários grafos que contêm informação relevante de forma a incentivar a escrita de código mais eficiente e correto.

Conteúdo

1 Introdução **2**

2 Descrição da Resolução **3**
2.1 Control Flow Graph 3
2.1.1 Especificação 3
2.1.2 Implementação 4
2.2 System Dependency Graph 5
2.2.1 Especificação 5
2.2.2 Implementação 6
2.2.3 Análise de Código utilizando SDG 7

3 Codificação e Testes **8**
3.1 Testes realizados e Resultados 8

4 Conclusão **14**

A Código do Programa **15**

Capítulo 1

Introdução

Num contexto profissional, grandes empresas têm vários ficheiros com enormes quantidades de linhas de código de forma a que as suas aplicações sejam o mais completas e apelativas possível. Porém, à medida que a complexidade de uma aplicação aumenta, a dificuldade de análise de código aumenta, daí ser necessário haver uma base automática que permita realizar essa tarefa de forma rápida e sem falhas.

A partir desta necessidade foram idealizadas aplicação de análise de código fonte, semelhante à aplicação realizada no trabalho prático anterior. No entanto, há a necessidade de realizar uma análise a outros detalhes do código, tal como uma análise comportamental.

Daí, no âmbito da Unidade Curricular de Engenharia Gramatical ser proposto um enriquecimento do analisador de código. Esta adição constitui na capacidade de geração de grafos que permitem ao utilizador visualizar o comportamento da aplicação de uma forma mais simplista. O programa é capaz de gerar dois tipos de grafos, CFG (Control Flow Graph) e SDG (System Dependency Graph).

Estas novas adições serão adicionadas no output em HTML juntamente com a informação gerada do TP anterior.

Capítulo 2

Descrição da Resolução

Para introduzir esta análise comportamental de excertos de código em LPIS2 foram criados dois ficheiros separados de modo a conseguir-se modularizar a lógica do analisador.
Logo, foram criados 2 ficheiros, um para a geração do Control Flow Graph e outro para a geração do System Dependency Graph. Estes dois ficheiros irão conter um interpretador responsável para a construção do respetivo grafo.

2.1 Control Flow Graph

2.1.1 Especificação

O Control Flow Graph consiste num grafo que analisa os vários caminhos que um fluxo de execução pode tomar. É um grafo que é comummente usada para análise estática de código e aplicação de compiladores. Esta ferramenta é útil na medida em que tem várias aplicações:

- Entendimento do código através do fluxo da aplicação.
- Localização de comportamento indesejável.
- Localização de código inalcançável.
- Cálculo da cobertura de testes.

A título de exemplo, para o código apresentado:

```
int a = 0;
if( a == 1 ) {
    a = 6;
}
else {
    a = 1;
}
```

O grafo correspondente é o seguinte:

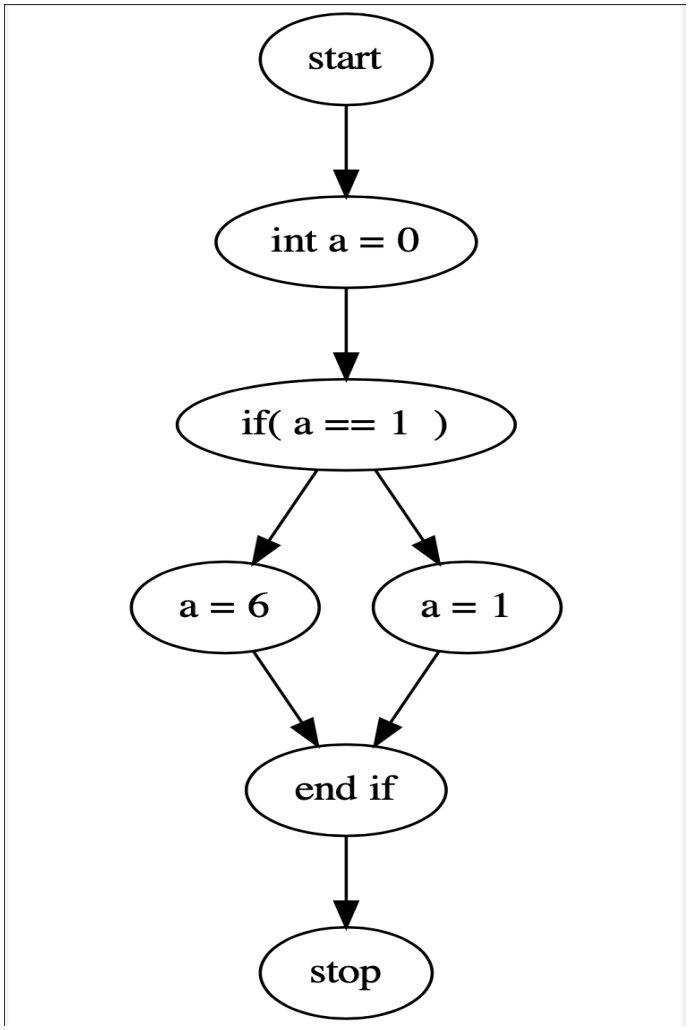


Figura 2.1: CFG para o código exemplo apresentado

2.1.2 Implementação

Tal como foi dito anteriormente, a geração do CFG foi isolado da lógica do trabalho prático anterior. É constituído por um interpretador que visita as regras da gramática estabelecida e vai construindo o grafo "on the fly".

Para realizar a construção do grafo, são necessárias 4 variáveis importantes:

- **Graph** - O grafo em construção que consistirá numa lista de adjacência.
- **GraphMap** - Mapeia cada nodo do grafo à label correspondente. A label será o pedaço de código correspondente ao nodo em causa.
- **NodeBefore** - Corresponde ao nodo anterior. Esta variável é util para a estabelecer as ligações entre nodos.
- **NodeCount** - Dá id's aos nodos para que seja possível fazer a ligação entre eles usando as suas identificações.

Em regra geral, à medida que se visita cada instrução de código, é adicionada uma nova entrada a ambos os grafos com a informação correspondente à instrução em causa. Após a adição nos grafos é feita a ligação do nodo anterior com o atual e atualizada a variável que contém o nodo anterior, de modo a que quando se visite a próxima instrução se consiga ligar com a atual. Por fim, incrementa-se a variável responsável por dar id's aos nodos do grafo.

Porém, para estruturas de controlo, a lógica é diferente, uma vez que o caminho não é linear. Para a geração do grafo de if's, teriam que originar dois caminhos, que seriam o caminho caso a condição se verifique e o caminho caso a condição não se verifique. Para conseguir esta lógica, o código que irá ser executado caso a condição se verifique é visitado de forma linear como nas instruções que não são de controlo, guardando o nodo do if. De seguida altera-se a variável que guarda o nodo anterior para o nodo do if para que se consiga ter os dois caminhos esperados, e assim, poder-se continuar a visitar as regras que faltariam visitar pelo interpretador.

No caso dos ciclos, já não se faz dois caminhos lineares a partir de um nodo, mas liga-se o último nodo do código à condição para ser novamente verificada, logo, ao invés de atualizar a variável que guarda o nodo anterior, liga-se diretamente o nodo que foi visitado por último ao nodo responsável pela condição do ciclo. Após feita essa ligação, por fim, é conectado o nodo da condição para o código posterior ao ciclo.
Relativamente à complexidade de McCabe, cuja fórmula é número de arestas menos número de nodos mais 2, é necessário saber quantos nodos e quantas arestas o grafo possui. Para tal, utilizamos as estruturas usadas para construir o grafo, sendo que uma indica os nodos existentes no grafo e a outra é uma estrutura que indica os nodos para onde cada nodo terá de ter ligação. Depois de termos os dados necessários, é só aplicar a fórmula para descobrir o complexidade de McCabe.

2.2 System Dependency Graph

2.2.1 Especificação

O System Dependency Graph é um grafo capaz de modelar um sistema, apresentando informações de controlo de fluxo e dependência de dados entre as funções constituintes. Este consegue juntar utilidades do Control Flow Graph e do Data Dependency Graph, consistindo num grafo com grande quantidade de informação. O grafo é capaz de fazer uma análise completa ao sistema, procurando padrões para assegurar boas práticas de programação, descobrir onde fazer mudanças pois consegue apresentar uma boa compreensão do código e até permite analisar a segurança do sistema.
Com o mesmo código apresentado em cima:

```
int a = 0;
if( a == 1 ) {
    a = 6;
}
else {
    a = 1;
}
```

O SDG correspondente (sem ter o fluxo de dados em conta) é o seguinte:

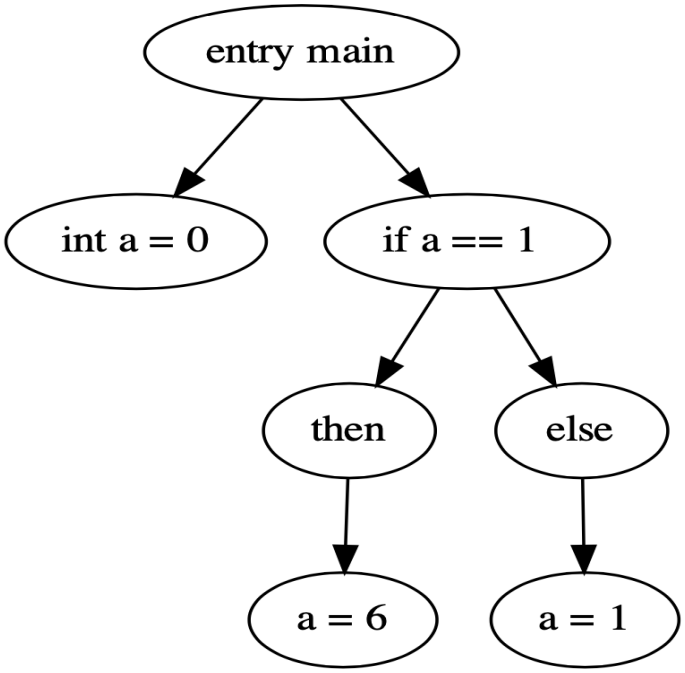


Figura 2.2: SDG para o código exemplo apresentado

2.2.2 Implementação

Ao contrário do Control Flow Graph, o System Dependency Graph não faz ligações com o nodo anterior mas sim a um nodo que funciona como nodo "pai". Este nodo pai irá se ligar com todas as instruções que o caraterizam e com a ordem de execução da esquerda para a direita, de cima para baixo. Para o efeito, foram usadas as mesmas variáveis que no interpretador do CFG, exceto a **nodeBefore** que foi substituída por **connectParent**. Esta nova variável será uma lista de nodos que funcionará como uma stack. Esta mudança foi efetuada pois já não é necessário manter a informação do nodo anterior mas sim do nodo pai a que as instruções devem-se ligar. O armazenamento do grafo em memória e das labels associadas é semelhante ao adotado no interpretador de geração de CFG.

Tal como na CFG, as estruturas de controlo vão ter uma lógica diferente associada na construção do grafo, uma vez que este cresce verticalmente. Na regra de if's é criado um novo nodo **then** que é ligado ao if e um **else** caso esteja presente. Todo o código pertencente a essa estrutura de controlo estará ligado ao **then** ou ao **else**, sendo esses os novos pais.

Tendo este caso em conta, justifica-se o uso de uma stack para o armazenamento do pai atual da instrução em visita. Cada vez que é encontrada uma estrutura de controlo, é feito um push desse nodo para a stack, para que o código interior se possa ligar ao novo pai. No fim de visitar o código interior à estrutura de controlo, é feito um pop da estrutura de dados para recuperar o nodo pai anterior. A diferença da representação de ciclos para if's é a ausência dos nodos **then** e **else**, a restante lógica é semelhante.

2.2.3 Análise de Código utilizando SDG

De forma a descobrir zonas de código inalcansável, foi necessário realizar uma análise estática ao código. Existem 2 cenários em que podemos calcular se uma zona é inalcansável ou não:

- Não existem variáveis na condição
- Existem variáveis mas não mudarão o valor da condição

Quando não existem variáveis na condição, limitamo-nos a fazer a avaliação da condição, utilizando a função *eval()* nativa do Python. Se a condição tiver valor negativo e for um if, então sabemos que iremos sempre percorrer o fluxo do else (se este existir), e simbolizamos o caminho que não irá ser seguido através de arestas vermelhas. Se a condição tiver valor positivo e for um if, então seguiremos sempre pelo fluxo do then, fazendo a mesma sinalização que foi especificada anteriormente. A mesma avaliação é feita para os ciclos, sendo que apenas se verifica se a condição dos ciclos é verdadeira ou não, e caso a resposta seja negativa, fazemos as arestas vermelhas.
Se existirem variáveis, temos de verificar se estamos a fazer disjunções ou conjunções. Por exemplo, no caso das disjunções, basta que uma das condições presentes seja verdadeira para que a condição seja verdadeira e o oposto para as disjunções. Depois de fazer esta avaliação, o processo é o mesmo que no caso onde não há variáveis.

Capítulo 3

Codificação e Testes

3.1 Testes realizados e Resultados

De forma a criar os grafos, utilizou-se a biblioteca GraphViz, e o código para os gerar é o seguinte: No caso mostrado em cima, apresenta-se o excerto de código que gera o grafo SDG, e as diferenças

```
def createGraph(graph, graphMap,unreach):
    dot = graphviz.Digraph('System Dependency Graph',strict=True)

    for (beg,currentVertex) in unreach:
            depthFirst(graph,currentVertex,[])
    auxVisited = visitedList
    for i,node in enumerate(graph):
        dot.node(str(i),graphMap[i])
        for arrs in auxVisited:
            if len(arrs)>0:
                arrs = arrs[1:]
                for i in range(len(arrs)-1):
                    frm = arrs[i]
                    to = arrs[i+1]
                    dot.edge(str(frm),str(to),color="red")
        auxVisited=[]
        for row in node:
            for idx,(beg,end) in enumerate(unreach):
                if(i==beg and row == end):
                    dot.edge(str(i),str(row),color="red")
                elif(i==end):
                    dot.edge(str(i),str(row),color="red")


            dot.edge(str(i),str(row))

    dot.render("System Dependency Graph.gv",view=True)

    return dot
```

Figura 3.1: Código para gerar SDG

relativas à geração do CFG é que, como se poderá ver em seguida, é que para o SDG temos de colocar as arestas vermelhas para o código inalcansável, fazendo com que não seja necessário verificar quais os nodos inalcansáveis.

Assim como no TP2, é gerado um relatório com o código analisado, os erros e os warnings e um relatório estatístico.

Como havia sido pedido no enunciado, os grafos resultantes foram adicionados ao HTML gerado pelo TP2, tendo sido criada uma secção para cada um dos grafos, como podemos ver de seguida

No fim da imagem também é colocada a complexidade de McCabe, bem como a conta que é feita

```
def createGraph(graph, graphMap):
    dot = graphviz.Digraph('Control Flow Graph')

    for i,node in enumerate(graph):
        dot.node(str(i),graphMap[i])

        for row in node:
            dot.edge(str(i),str(row))

    dot.render("Control Flow Graph.gv",view=True)

    return dot
```

Figura 3.2: Código para gerar CFG

para chegar a esse resultado.
No System Dependency Graph, como pudemos ver, estão sinalizados os nodos pelos quais não iremos passar através das arestas vermelhas, como foi mencionado no capítulo anterior.

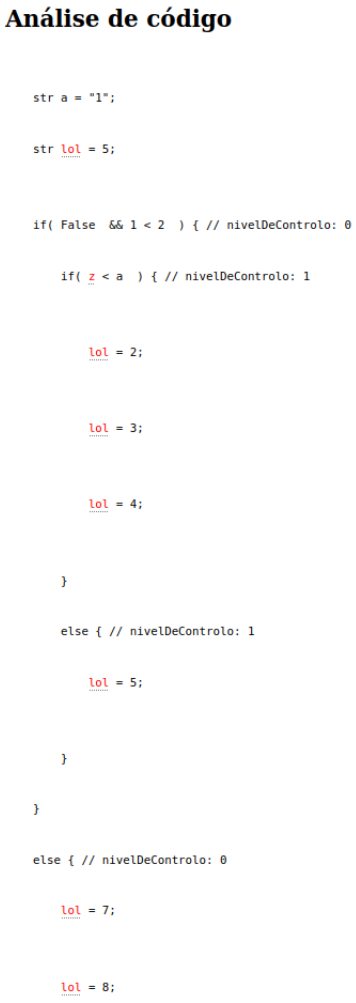
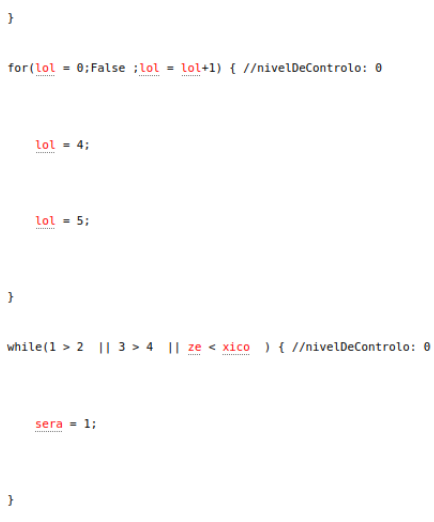
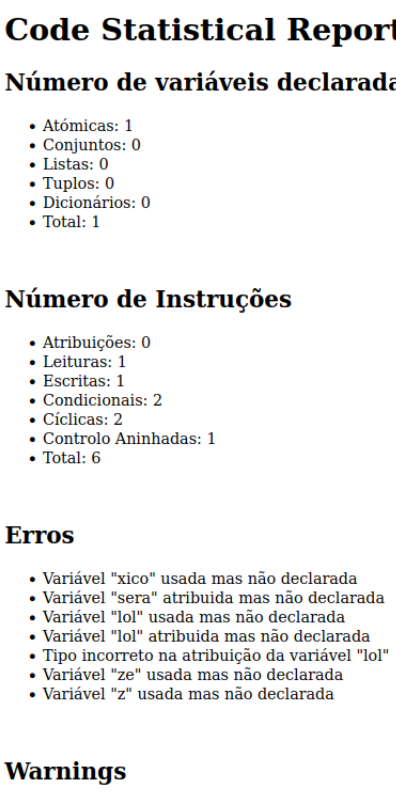


Figura 3.3: HTML com excerto de código analisado



```
}

for(lol = 0;False ;lol = lol+1) { //nivelDeControlo: 0


    lol = 4;


    lol = 5;


}

while(1 > 2  || 3 > 4  || ze < xico  ) { //nivelDeControlo: 0


    sera = 1;


}
```

Figura 3.4: HTML com excerto de código analisado



Code Statistical Report

Número de variáveis declaradas

- Atómicas: 1
- Conjuntos: 0
- Listas: 0
- Tuplos: 0
- Dicionários: 0
- Total: 1

Número de Instruções

- Atribuições: 0
- Leituras: 1
- Escritas: 1
- Condicionais: 2
- Cíclicas: 2
- Controlo Aninhadas: 1
- Total: 6

Erros

- Variável "xico" usada mas não declarada
- Variável "sera" atribuida mas não declarada
- Variável "lol" usada mas não declarada
- Variável "lol" atribuida mas não declarada
- Tipo incorreto na atribuição da variável "lol"
- Variável "ze" usada mas não declarada
- Variável "z" usada mas não declarada

Warnings

Figura 3.5: Relatório estatístico

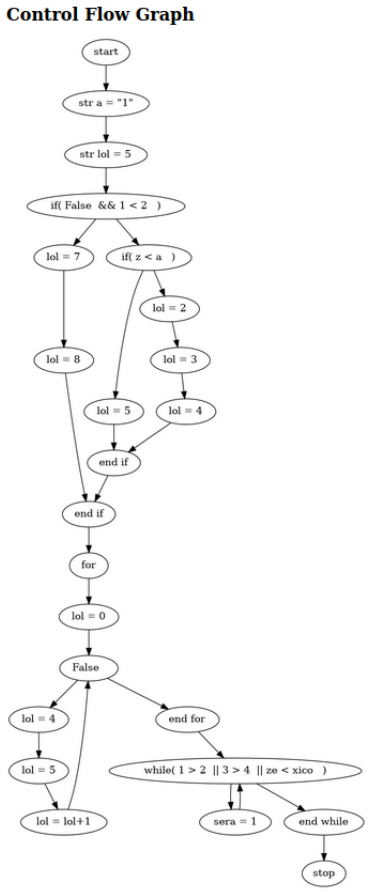


Figura 3.6: CFG no relatório HTML

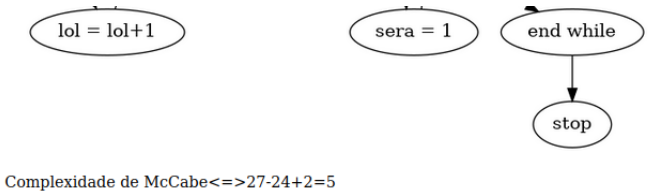


Figura 3.7: Complexidade de McCabe no relatório HTML

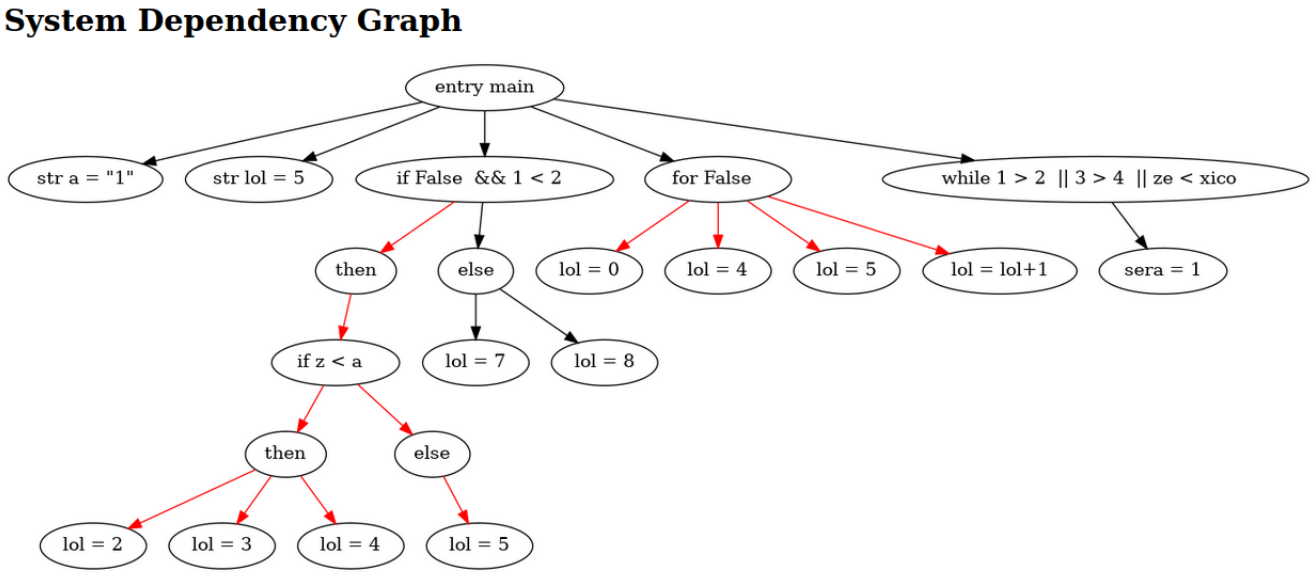


Figura 3.8: SDG no relatório HTML

Capítulo 4

Conclusão

Com a realização deste trabalho foi possível aprofundar o nosso conhecimento em System Dependency Graphs e Control Flow Graphs, bem como, neste caso, foi possível aprender a biblioteca GraphViz de forma a construir os grafos necessários.

Também serviu para complementar o trabalho anterior com a análise estática do código, que é bastante útil e nos dá uma muito melhor noção sobre possíveis problemas que o nosso código fonte possa ter

Apêndice A

Código do Programa

declarations.lark

```
// Declarations

declaration: atomic_declaration ";"
           | set_declaration ";"
           | list_declaration ";"
           | tuple_declaration ";"
           | dict_declaration ";"

atomic_declaration: TYPE var
                  | TYPE var "=" operand

set_declaration: TYPE "set" var
               | TYPE "set" var "=" set

list_declaration: TYPE "list" var
                | TYPE "list" var "=" list

tuple_declaration: TYPE "tuple" var
                 | TYPE "tuple" var "=" tuple

dict_declaration: "(" TYPE "," TYPE ")" "dict" var
                | "(" TYPE "," TYPE ")" "dict" var "=" dict

var: WORD

TYPE: "int"
    | "float"
    | "str"
    | "bool"

set: "{" list_contents "}"
list: "[" list_contents "]"
```

```
33 tuple: "(" list_contents ")"
34 dict: "{" dict_contents "}"
35
36 list_contents: int_contents
37              | float_contents
38              | string_contents
39              | bool_contents
40
41 int_contents: INT ("," INT)*
42 float_contents: FLOAT ("," FLOAT)*
43 string_contents: ESCAPED_STRING ("," ESCAPED_STRING)*
44 bool_contents: BOOL ("," BOOL)*
45
46 dict_contents: value ":" operand ("," value ":" operand)*
47
48 operand: value  -> operand_value
49        | var    -> operand_var
50
51 value: ESCAPED_STRING   -> value_string
52      | FLOAT            -> value_float
53      | INT              -> value_int
54      | BOOL             -> value_bool
55
56 BOOL: "True"
57     | "False"
58
59 %import common.WS
60 %import common.NEWLINE
61 %ignore WS
62 %ignore NEWLINE
63 %import common.INT
64 %import common.WORD
65 %import common.FLOAT
66 %import common.ESCAPED_STRING
```

grammar.py

```
1 ## Primeiro precisamos da GIC
2 grammar = '''
3 start: code*
4 code: (declaration | instruction)+
5
6 instruction: atribution ";"
7              | condition
8              | cycle
9
10 atribution: var "=" (expression | list | tuple | set | dict)
11
12
```

```
13 var: WORD
14     | WORD "[" operand "]"
15
16 condition: "if" "(" boolexpr ")" "{" code "}"
17          | "if" "(" boolexpr ")" "{" code "}" "else" "{" code "}"    ->
                condition_else
18
19 cycle: while_cycle
20       | do_while_cycle
21       | repeat_cycle
22       | for_cycle
23
24 while_cycle: "while" "(" boolexpr ")" "{" code "}"
25 do_while_cycle: "do" "{" code "}" "while" "(" boolexpr ")"
26 repeat_cycle: "repeat" "(" matexpr ")" "{" code "}"
27 for_cycle: "for" "(" atribution? ";" boolexpr? ";" atribution? ")" "{" code
       "}"
28
29 expression: boolexpr
30           | matexpr
31
32 matexpr: operand (MAT_OPERATOR operand)*
33
34 simple_bool_expr: matexpr BOOL_OPERATOR matexpr
35                  | BOOL
36
37 boolexpr: simple_bool_expr (LOGIC simple_bool_expr)*
38
39 BOOL_OPERATOR: ">"|"<"|">="|"<="|"=="|"!="
40 MAT_OPERATOR: "+"|"-"|"*"|"/"
41
42 LOGIC : "&&"
43       | "||"
44
45 operand: value | var
46
47 value: ESCAPED_STRING    -> value_string
48      | FLOAT             -> value_float
49      | INT               -> value_int
50      | BOOL              -> value_bool
51
52 %import common.WS
53 %import common.NEWLINE
54 %ignore WS
55 %ignore NEWLINE
56 %import common.INT
57 %import common.WORD
58 %import common.FLOAT
```

```
59 %import common.ESCAPED_STRING
60 %import .grammar.declarations (declaration,BOOL,TYPE,set,tuple,dict,list)
61 '''
```

interpreter.py

```
1  from distutils.log import error
2  from lark.visitors import Interpreter
3  from lark import Tree,Token
4  import interpreter.utils as utils
5  import re
6
7  global identNumber
8
9  identNumber = 4
10
11 class MainInterpreter (Interpreter):
12
13     def __init__(self):
14         self.variables = dict() # var -> {state -> (declared, assigned, used)
               , size -> int, datatype -> str, type -> str, keys -> list}
15         self.warnings = []
16         self.errors = []
17         self.valueDataType = None
18         self.valueType = None
19         self.valueSize = 0
20         self.numDeclaredVars = {'atomic':0,'set':0,'list':0,'tuple':0,'dict
               ':0}
21         self.numInstructions = {'atribution':0,'read':0,'write':0,'condition
               ':0,'cycle':0,'nestedControl':0}
22         self.identLevel = 0
23         self.controlDepth = 0
24         self.maxcontrolDepth = 0
25         self.ifData = None
26         self.codeData = None
27
28
29     def start(self,tree):
30         # Visita todos os filhos em que cada um v o retornar o seu c digo
31         res = self.visit_children(tree)
32
33         self.analyzeVariablesDeclaredAndNotMentioned()
34
35         self.errors = list(set(self.errors))
36         self.warnings = list(set(self.warnings))
37         statReport = utils.generateHTMLStatReport(self.numDeclaredVars,self.
               errors,self.warnings,self.numInstructions)
38
39         utils.generateHTML(''.join(res[0]),statReport)
```

```
40
41          output = dict()
42          # Juntar o c digo dos v rios blocos
43          output["html"] = res[0]
44          output["vars"] = self.variables
45
46          return output
47
48      def analyzeVariablesDeclaredAndNotMentioned(self):
49          for (var,value) in self.variables.items():
50              if value["state"][0] == True and value["state"][1] == False and
                    value["state"][2] == False:
51                  self.warnings.append("Vari vel \"" + var + "\" declarada mas
                        nunca mencionada")
52
53      def declaration(self,tree):
54          r = self.visit(tree.children[0])
55          return r
56
57      def __generalDeclarationVisitor(self,tree,type):
58          errors = []
59          dataType = str(tree.children[0])
60          varName = self.visit(tree.children[1])
61          childNum = len(tree.children)
62
63          # See if variable was mentioned in the code
64          if varName not in self.variables:
65              value = dict()
66              value["state"] = [True,False,False]
67              value["size"] = 0
68              value["datatype"] = dataType
69              value["type"] = type
70              self.variables[varName] = value
71
72          else:
73
74              value = self.variables[varName]
75
76              # Case if variable declared
77              if value["state"][0] == True:
78                  errors.append("Vari vel \"" + varName + "\" redeclarada")
79
80              # Update variable status
81              value["state"] = [True] + value['state'][1:]
82
83          # if variable is assigned
84          if childNum > 2:
85              # Get value assigned to
```

```
86              operand = self.visit(tree.children[2])
87
88              if self.valueDataType != value['datatype']:
89                  errors.append("Tipo incorreto na atribui  o da vari vel
                        \"" + varName + "\"")
90
91              else:
92                  value["size"] = self.valueSize
93                  value["state"][1] = True
94
95              self.valueDataType = None # Useless but for bug-free programming
96              self.valueType = None
97              self.valueSize = 0 # Useless but for bug-free programming
98
99              if errors:
100                 self.errors.extend(errors)
101                 self.variables.pop(varName)
102                 varName = utils.generateErrorTag(varName,";".join(errors))
103             else:
104                 self.numInstructions['write']+=value["size"]
105
106             code = f"{dataType}{'' if type == 'atomic' else ' ' + type} {
                    varName} = {operand};"
107
108         else:
109
110             if errors:
111                 self.errors.extend(errors)
112                 self.variables.pop(varName)
113                 varName = utils.generateErrorTag(varName,";".join(errors))
114
115             code = f"{dataType}{'' if type == 'atomic' else ' ' + type} {
                    varName};"
116
117         if not errors:
118             self.numDeclaredVars[type] += 1
119
120         return utils.generatePClassCodeTag(code)
121
122     def grammar__declarations__atomic_declaration(self,tree):
123         return self.__generalDeclarationVisitor(tree,"atomic")
124
125     def grammar__declarations__set_declaration(self,tree):
126         return self.__generalDeclarationVisitor(tree,"set")
127
128     def grammar__declarations__list_declaration(self,tree):
129         return self.__generalDeclarationVisitor(tree,"list")
130
```

```
131     def grammar__declarations__tuple_declaration(self,tree):
132         return self.__generalDeclarationVisitor(tree,"tuple")
133 
134     def grammar__declarations__dict_declaration(self,tree):
135         errors = []
136         keyDataType = str(tree.children[0])
137         valueDataType = str(tree.children[1])
138         varName = self.visit(tree.children[2])
139         childNum = len(tree.children)
140 
141         # See if variable was mentioned in the code
142         if varName not in self.variables:
143             value = dict()
144             value["state"] = [True,False,False]
145             value["size"] = 0
146             value["datatype"] = (keyDataType,valueDataType)
147             value["type"] = 'dict'
148             self.variables[varName] = value
149 
150         else:
151 
152             value = self.variables[varName]
153 
154             # Case if variable declared
155             if value["state"][0] == True:
156                 errors.append("Vari vel \"" + varName + "\" redeclarada")
157 
158             # Update variable status
159             value["state"] = [True] + value['state'][1:]
160 
161         # if variable is assigned
162         if childNum > 3:
163             # Get value assigned to
164             operand = self.visit(tree.children[3])
165 
166             if self.valueDataType != value['datatype'] and self.valueSize !=
                    0:
167                 errors.append("Tipo incorreto na atribui  o da vari vel
                        \"" + varName + "\"")
168 
169             else:
170                 value["size"] = self.valueSize
171                 value["state"][1] = True
172 
173             self.valueDataType = None # Useless but for bug-free programming
174             self.valueSize = 0 # Useless but for bug-free programming
175 
176             if errors:
```

```
177                 self.errors.extend(errors)
178                 varName = utils.generateErrorTag(varName,";".join(errors))
179             else:
180                 self.numInstructions["write"] += value["size"]
181 
182             code = f"({keyDataType},{valueDataType}) dict {varName} = {
                    operand};"
183 
184         else:
185 
186             if errors:
187                 self.errors.extend(errors)
188                 varName = utils.generateErrorTag(varName,";".join(errors))
189 
190             code = f"({keyDataType},{valueDataType}) dict {varName};"
191 
192         if not errors:
193             self.numDeclaredVars['dict'] += 1
194 
195         return utils.generatePClassCodeTag(code)
196 
197     def grammar__declarations__var(self,tree):
198         return str(tree.children[0])
199 
200     def set(self,tree):
201         self.valueType = 'set'
202         return f"{{{self.visit(tree.children[0])}}}"
203 
204     def list(self,tree):
205         self.valueType = 'list'
206         return f"[{self.visit(tree.children[0])}]"
207 
208     def tuple(self,tree):
209         self.valueType = 'tuple'
210         return f"({self.visit(tree.children[0])})"
211 
212     def dict(self,tree):
213         self.valueType = 'dict'
214         return f"{{{self.visit(tree.children[0])}}}"
215 
216     def grammar__declarations__list_contents(self,tree):
217         return self.visit(tree.children[0])
218 
219     def grammar__declarations__int_contents(self,tree):
220         self.valueDataType = 'int'
221         self.valueSize = len(tree.children)
222         elemList = []
223         for child in tree.children:
```

```
224             elemList.append(str(child))
225         return ",".join(elemList)
226 
227     def grammar__declarations__float_contents(self,tree):
228         self.valueDataType = 'float'
229         self.valueSize = len(tree.children)
230         elemList = []
231         for child in tree.children:
232             elemList.append(str(child))
233         return ",".join(elemList)
234 
235     def grammar__declarations__string_contents(self,tree):
236         self.valueDataType = 'str'
237         self.valueSize = len(tree.children)
238         elemList = []
239         for child in tree.children:
240             elemList.append(str(child))
241         return ",".join(elemList)
242 
243     def grammar__declarations__bool_contents(self,tree):
244         self.valueDataType = 'bool'
245         self.valueSize = len(tree.children)
246         elemList = []
247         for child in tree.children:
248             elemList.append(str(child))
249         return ",".join(elemList)
250 
251     def grammar__declarations__dict_contents(self,tree):
252         keyDataType = set()
253         valueDataType = set()
254         repetitiveKeys = False
255         valueKeys = []
256 
257         elemList = []
258         for key,value in zip(tree.children[0::2], tree.children[1::2]):
259             v_key = self.visit(key)
260             keyDataType.add(self.valueDataType)
261             v_value = self.visit(value)
262             valueDataType.add(self.valueDataType)
263 
264             if v_key in valueKeys:
265                 repetitiveKeys = True
266                 break
267 
268             valueKeys.append(v_key)
269 
270             elemList.append(f"{v_key}:{v_value}")
271 
```

```
272         if repetitiveKeys:
273             self.valueDataType = None
274             return utils.generateErrorTag(",".join(elemList),"Dicion rio tem
                    chave repetida")
275 
276         self.valueSize = len(elemList)
277 
278         if len(valueDataType) > 1 or len(keyDataType) > 1:
279             self.valueDataType = None
280             return utils.generateErrorTag(",".join(elemList),"Tipos do
                    dicion rio n o s o uniformes")
281 
282         elif len(valueDataType) == 0 or len(keyDataType) == 0:
283             self.valueDataType = None
284 
285         else:
286             self.valueDataType = (keyDataType.pop(),valueDataType.pop())
287 
288         return ",".join(elemList)
289 
290     def grammar__declarations__dict_value(self,tree):
291         return self.visit(tree.children[0])
292 
293     def grammar__declarations__operand_value(self,tree):
294         return self.visit(tree.children[0])
295 
296     def grammar__declarations__operand_var(self,tree):
297         varName = self.visit(tree.children[0])
298 
299         if varName not in self.variables:
300             self.errors.append("Vari vel \"" + varName + "\" n o declarada
                    ")
301             varName = utils.generateErrorTag(varName,"Vari vel n o
                    declarada")
302 
303             self.valueDataType = ''
304             self.valueSize = 0
305 
306             return varName
307 
308         value = self.variables[varName]
309 
310         if value["state"][1] == False:
311             self.errors.append("Vari vel \"" + varName + "\" n o atribu da
                    ")
312             varName = utils.generateErrorTag(varName,"Vari vel n o
                    atribu da")
313 
```

```
314             self.valueDataType = ''
315             self.valueSize = 0
316 
317             return varName
318 
319         value = self.variables[varName]
320         value["state"][2] = True
321 
322         self.valueDataType = value["datatype"]
323         self.valueSize = value["size"]
324 
325         return varName
326 
327     def grammar__declarations__value_string(self,tree):
328         self.valueDataType = "str"
329         self.valueSize = 1
330         return str(tree.children[0])
331 
332     def grammar__declarations__value_float(self,tree):
333         self.valueDataType = "float"
334         self.valueSize = 1
335         return str(tree.children[0])
336 
337     def grammar__declarations__value_int(self,tree):
338         self.valueDataType = "int"
339         self.valueSize = 1
340         return str(tree.children[0])
341 
342     def grammar__declarations__value_bool(self,tree):
343         self.valueDataType = 'bool'
344         self.valueSize = 1
345         return str(tree.children[0])
346 
347     def code(self,tree):
348         self.ifData = None
349         self.valueType = None
350         r=list()
351         for child in tree.children:
352             r.append(self.visit(child))
353         return r
354 
355     def instruction(self,tree):
356         return self.visit(tree.children[0])
357 
358     def atribution(self,tree):
359 
360         ident = (self.identLevel * identNumber * " ")
361 
```

```
362         varName = self.visit(tree.children[0])
363 
364         exp = self.visit(tree.children[1])
365 
366         if self.valueType is None:
367             self.valueType = 'atomic'
368 
369         if varName not in self.variables:
370             self.errors.append("Vari vel \"" + varName + "\" atribuida mas
                n o declarada")
371             varName = utils.generateErrorTag(varName,"Vari vel \"" + varName
                + "\" atribu da mas n o declarada")
372 
373         elif self.variables[varName]['type'] != self.valueType or self.
            variables[varName]['datatype'] != self.valueDataType:
374             self.errors.append("Tipos incompat veis na atribui   o de um
                valor    vari vel \"" + varName + "\"")
375             varName = utils.generateErrorTag(varName,"Tipos incompat veis na
                 atribui   o")
376 
377         elif not re.search(r'error',exp):
378             self.variables[varName]['state'][1] = True
379             self.numInstructions['write'] += 1
380             self.numInstructions['atribution'] += 1
381 
382         atrStr = f"{varName} = {exp};"
383 
384         self.codeData = atrStr
385 
386         return utils.generatePClassCodeTag(ident + atrStr)
387 
388 
389     def condition(self,tree):
390         identDepth = self.identLevel
391         controlDepth = self.controlDepth
392         self.controlDepth += 1
393         if(controlDepth>0):
394             self.numInstructions['nestedControl'] +=1
395 
396         self.identLevel += 1
397         self.numInstructions['condition'] += 1
398 
399         # C lculo da identa   o para pretty printing
400         ident = (identDepth * identNumber * " ")
401 
402         cond = self.visit(tree.children[0])
403 
404         code = self.visit(tree.children[1])
```

```
405
406         printCond = cond
407
408         if self.ifData is not None and len(tree.children[1].children) == 1:
409             cond = f'{cond} && {self.ifData[0]}'
410             code = self.ifData[1]
411             printCond = utils.generateSubTag(cond,"If conjugado")
412             self.numInstructions['nestedControl'] -= 1
413             self.numInstructions['condition'] -= 1
414
415
416         self.ifData = (cond,code)
417
418         taggedCode = utils.generatePClassCodeTag(ident + "if( "+printCond+")
                { // nivelDeControlo: "+str(controlDepth))
419         taggedCode += ''.join(code)
420         taggedCode +=utils.generatePClassCodeTag(ident + "}")
421
422         self.identLevel = identDepth
423         self.controlDepth = controlDepth
424
425         return taggedCode
426
427     def condition_else(self,tree):
428         identDepth = self.identLevel
429         controlDepth = self.controlDepth
430         self.controlDepth += 1
431         if(controlDepth>0):
432             self.numInstructions['nestedControl'] +=1
433
434         self.identLevel += 1
435         self.numInstructions['condition'] += 1
436
437         cond = self.visit(tree.children[0])
438         code = self.visit(tree.children[1])
439         elseCode = self.visit(tree.children[2])
440
441         # C lculo da identa  o para pretty #printing
442         ident = (identDepth * identNumber * " ")
443
444
445         taggedCode = utils.generatePClassCodeTag(ident + "if( "+cond+") { //
                nivelDeControlo: "+str(controlDepth))
446         taggedCode += ''.join(code)
447         taggedCode +=utils.generatePClassCodeTag(ident + "}")
448         taggedCode +=utils.generatePClassCodeTag(ident + "else { //
                nivelDeControlo: " + str(controlDepth))
449         taggedCode += ''.join(elseCode)
```

```
450         taggedCode +=utils.generatePClassCodeTag(ident + "}")
451 
452         self.identLevel = identDepth
453         self.controlDepth = controlDepth
454 
455         return taggedCode
456 
457     def cycle(self,tree):
458         self.numInstructions['cycle'] += 1
459         return self.visit(tree.children[0])
460 
461     def while_cycle(self,tree):
462         identDepth = self.identLevel
463         controlDepth = self.controlDepth
464         self.controlDepth += 1
465         if(controlDepth>0):
466             self.numInstructions['nestedControl'] +=1
467         self.maxcontrolDepth = self.maxcontrolDepth if self.maxcontrolDepth >
                controlDepth else controlDepth
468 
469         self.identLevel +=1
470 
471         # C lculo da identa  o para pretty printing
472         ident = (identDepth* identNumber * " ")
473 
474         bool=self.visit(tree.children[0])
475         code=self.visit(tree.children[1])
476 
477 
478         taggedCode = utils.generatePClassCodeTag(ident + "while(" + bool + ")
                { //nivelDeControlo: "+ str(controlDepth))
479         taggedCode += ''.join(code)
480         taggedCode += utils.generatePClassCodeTag(ident +"}")
481 
482         self.identLevel = identDepth
483         self.controlDepth = controlDepth
484         return taggedCode
485 
486     def do_while_cycle(self,tree):
487         identDepth = self.identLevel
488         controlDepth = self.controlDepth
489         self.controlDepth += 1
490         if(controlDepth>0):
491             self.numInstructions['nestedControl'] +=1
492         self.maxcontrolDepth = self.maxcontrolDepth if self.maxcontrolDepth >
                controlDepth else controlDepth
493 
494         self.identLevel +=1
```

```
495
496         # C lculo da identa  o para pretty printing
497         ident = (identDepth* identNumber * " ")
498
499         code=self.visit(tree.children[0])
500         bool=self.visit(tree.children[1])
501
502         taggedCode = utils.generatePClassCodeTag(ident + "do { //
                nivelDeControlo: "+str(controlDepth))
503         taggedCode += ''.join(code)
504         taggedCode += utils.generatePClassCodeTag(ident + "} while("+bool+")
                ")
505
506         self.identLevel = identDepth
507         self.controlDepth = controlDepth
508
509         return taggedCode
510
511     def repeat_cycle(self,tree):
512         identDepth = self.identLevel
513         controlDepth = self.controlDepth
514         self.controlDepth += 1
515         if(controlDepth>0):
516             self.numInstructions['nestedControl'] +=1
517         self.maxcontrolDepth = self.maxcontrolDepth if self.maxcontrolDepth >
                controlDepth else controlDepth
518
519         self.identLevel +=1
520
521         # C lculo da identa  o para pretty printing
522         ident = (identDepth* identNumber * " ")
523
524         mat=self.visit(tree.children[0])
525         code=self.visit(tree.children[1])
526
527         taggedCode = utils.generatePClassCodeTag(ident + "repeat(" + mat + ")
                { //nivelDeControlo: " + str(controlDepth))
528         taggedCode += ''.join(code)
529         taggedCode += utils.generatePClassCodeTag(ident + "}")
530
531         self.identLevel = identDepth
532         self.controlDepth = controlDepth
533
534
535         return taggedCode
536
537     def for_cycle(self,tree):
538
```

```
539         identDepth = self.identLevel
540         controlDepth = self.controlDepth
541         self.controlDepth += 1
542         if(controlDepth>0):
543             self.numInstructions['nestedControl'] +=1
544         self.maxcontrolDepth = self.maxcontrolDepth if self.maxcontrolDepth >
                controlDepth else controlDepth
545 
546         self.identLevel +=1
547 
548         # C lculo da identa  o para pretty printing
549         ident = (identDepth* identNumber * " ")
550 
551         childInfo = [None,None,None,None]
552 
553         for child in tree.children:
554             if child.data == "atribution" and childInfo[0] is None:
555                 self.visit(child)
556                 childInfo[0] = self.codeData[:-1]
557             elif child.data == "atribution" and childInfo[0] is not None:
558                 self.visit(child)
559                 childInfo[2] = self.codeData[:-1]
560             elif child.data == 'boolexpr':
561                 self.visit(child)
562                 childInfo[1] = self.codeData[:-1]
563             elif child.data == 'code':
564                 childInfo[3] = self.visit(child)
565 
566         insidePar = f'{"" if childInfo[0] is None else childInfo[0]};{"" if
              childInfo[1] is None else childInfo[1]};{"" if childInfo[2] is
              None else childInfo[2]}'
567 
568         taggedCode = utils.generatePClassCodeTag(ident + "for(" + insidePar +
                ") { //nivelDeControlo: " + str(controlDepth))
569         taggedCode += ''.join(childInfo[3])
570         taggedCode += utils.generatePClassCodeTag(ident + "}")
571 
572         self.identLevel = identDepth
573         self.controlDepth = controlDepth
574 
575 
576         return taggedCode
577 
578 
579     def expression(self,tree):
580         return self.visit(tree.children[0])
581 
582     def matexpr(self,tree):
```

```
583         r=""
584         for child in tree.children:
585             if(isinstance(child,Tree)):
586                 r+=self.visit(child)
587             else:
588                 r+=child
589         return r
590 
591     def simple_bool_expr(self,tree):
592         left = self.visit(tree.children[0])
593         center = tree.children[1]
594         right = self.visit(tree.children[2])
595 
596 
597         return f"{left} {center} {right}"
598 
599 
600     def boolexpr(self,tree):
601         r=""
602         for child in tree.children:
603             if(isinstance(child,Tree)):
604                 r+=self.visit(child)+" "
605             else:
606                 r+=child+" "
607 
608         self.codeData = r
609 
610         return r
611 
612     def operand(self,tree):
613         errors=[]
614         value=self.visit(tree.children[0])
615 
616         if(tree.children[0].data=="var"):
617             if value not in self.variables:
618                 errors.append("Vari vel \"" + value + "\" usada mas n o
                        declarada")
619 
620             elif self.variables[value]["state"][1] == False:
621                 errors.append("Vari vel \"" + value + "\" usada mas n o
                        inicializada")
622 
623             else:
624                 self.variables[value]["state"][2] = True
625                 self.numInstructions['read'] += 1
626 
627 
628         if errors:
```

```
629             self.errors.extend(errors)
630             value = utils.generateErrorTag(value,";".join(errors))
631 
632         return value
633 
634     def value_string(self,tree):
635         self.valueDataType = "str"
636         return str(tree.children[0])
637 
638     def value_float(self,tree):
639         self.valueDataType = "float"
640         return str(tree.children[0])
641 
642     def value_int(self,tree):
643         self.valueDataType = "int"
644         return str(tree.children[0])
645 
646     def value_bool(self,tree):
647         self.valueDataType = "bool"
648         return str(tree.children[0])
649 
650     def var(self,tree):
651         varName = str(tree.children[0])
652         retStr = varName
653 
654         if(len(tree.children) > 1):
655             operand = self.visit(tree.children[1])
656             if self.valueDataType != "int":
657                 self.errors.append("ndice no do tipo int")
658                 operand = utils.generateErrorTag(operand,"ndice no do
                        tipo int")
659             retStr += '[' + operand + ']'
660 
661         if varName not in self.variables:
662             self.valueDataType = None
663         else:
664             self.valueDataType = self.variables[varName]["datatype"]
665             self.valueType = self.variables[varName]["type"]
666 
667 
668         return retStr
```

InterpreterCFG.py

```
1 from distutils.log import error
2 from lark.visitors import Interpreter
3 from lark import Tree,Token
4 import interpreter.utils as utils
5 import re, graphviz
```

```
6
7 global identNumber
8
9 identNumber = 4
10
11 def createGraph(graph, graphMap):
12     dot = graphviz.Digraph('Control Flow Graph')
13
14     for i,node in enumerate(graph):
15         dot.node(str(i),graphMap[i])
16
17         for row in node:
18             dot.edge(str(i),str(row))
19
20     dot.render("Control Flow Graph.gv",view=True)
21
22     return dot
23
24
25 class MainInterpreterCFG (Interpreter):
26
27     def __init__(self):
28         self.graph = []
29         self.graphMap = {}
30         self.nodeBefore = None
31         self.nodeCount = 0
32         self.graphNext = False
33
34
35
36     def start(self,tree):
37         # Visita todos os filhos em que cada um v o retornar o seu c digo
38         self.graph.append(set())
39         self.nodeBefore = 0
40         self.graphMap[0] = 'start'
41         self.nodeCount = 1
42
43         res = self.visit_children(tree)
44
45         self.graph.append(set())
46         self.graphMap[self.nodeCount] = 'stop'
47         self.graph[self.nodeBefore].add(self.nodeCount)
48
49         graph = createGraph(self.graph, self.graphMap)
50         graph.save()
51         graphviz.render('dot','png','Control Flow Graph.gv')
52
53         print(self.graph)
```

```
54          print(self.graphMap)
55          print(graph)
56
57          numberNodes = len(self.graphMap)
58          numberEdges = 0
59          for conj in self.graph:
60              numberEdges+=len(conj)
61
62          output = dict()
63          # Juntar o c digo dos v rios blocos
64          output["html"] = res[0]
65          output["vars"] = self.variables
66          output["graph"] = graph
67          output["nodes"]=numberNodes
68          output["edges"]=numberEdges
69
70          return output
71
72      def declaration(self,tree):
73          r = self.visit(tree.children[0])
74          return r
75
76      def __generalDeclarationVisitor(self,tree,type):
77          dataType = str(tree.children[0])
78          varName = self.visit(tree.children[1])
79          childNum = len(tree.children)
80
81          self.graph.append(set())
82          self.graph[self.nodeBefore].add(self.nodeCount)
83
84          # if variable is assigned
85          if childNum > 2:
86              operand = self.visit(tree.children[2])
87
88              self.graphMap[self.nodeCount] = f"{dataType}{'' if type == '
                    atomic' else ' ' + type} {varName} = {operand}"
89
90              code = f"{dataType}{'' if type == 'atomic' else ' ' + type} {
                    varName} = {operand};"
91
92          else:
93
94              self.graphMap[self.nodeCount] = f"{dataType}{'' if type == '
                    atomic' else ' ' + type} {varName}"
95
96              code = f"{dataType}{'' if type == 'atomic' else ' ' + type} {
                    varName};"
97
```

```
98          self.nodeBefore = self.nodeCount
99          self.nodeCount += 1
100
101         return code
102
103     def grammar__declarations__atomic_declaration(self,tree):
104         return self.__generalDeclarationVisitor(tree,"atomic")
105
106     def grammar__declarations__set_declaration(self,tree):
107         return self.__generalDeclarationVisitor(tree,"set")
108
109     def grammar__declarations__list_declaration(self,tree):
110         return self.__generalDeclarationVisitor(tree,"list")
111
112     def grammar__declarations__tuple_declaration(self,tree):
113         return self.__generalDeclarationVisitor(tree,"tuple")
114
115     def grammar__declarations__dict_declaration(self,tree):
116         keyDataType = str(tree.children[0])
117         valueDataType = str(tree.children[1])
118         varName = self.visit(tree.children[2])
119         childNum = len(tree.children)
120
121         self.graph.append(set())
122         self.graph[self.nodeBefore].add(self.nodeCount)
123
124         # if variable is assigned
125         if childNum > 3:
126             # Get value assigned to
127             operand = self.visit(tree.children[3])
128
129             self.graphMap[self.nodeCount] = f"({keyDataType},{valueDataType})
                    dict {varName} = {operand}"
130
131             code = f"({keyDataType},{valueDataType}) dict {varName} = {
                operand};"
132
133         else:
134
135             self.graphMap[self.nodeCount] = f"({keyDataType},{valueDataType})
                    dict {varName}"
136
137             code = f"({keyDataType},{valueDataType}) dict {varName};"
138
139         self.nodeBefore = self.nodeCount
140         self.nodeCount += 1
141
142         return code
```

```
143
144     def grammar__declarations__var(self,tree):
145         return str(tree.children[0])
146
147     def set(self,tree):
148         return f"{{{self.visit(tree.children[0])}}}"
149
150     def list(self,tree):
151         return f"[{self.visit(tree.children[0])}]"
152
153     def tuple(self,tree):
154         return f"({self.visit(tree.children[0])})"
155
156     def dict(self,tree):
157         return f"{{{self.visit(tree.children[0])}}}"
158
159     def grammar__declarations__list_contents(self,tree):
160         return self.visit(tree.children[0])
161
162     def grammar__declarations__int_contents(self,tree):
163         elemList = []
164         for child in tree.children:
165             elemList.append(str(child))
166         return ",".join(elemList)
167
168     def grammar__declarations__float_contents(self,tree):
169         elemList = []
170         for child in tree.children:
171             elemList.append(str(child))
172         return ",".join(elemList)
173
174     def grammar__declarations__string_contents(self,tree):
175         elemList = []
176         for child in tree.children:
177             elemList.append(str(child))
178         return ",".join(elemList)
179
180     def grammar__declarations__bool_contents(self,tree):
181         elemList = []
182         for child in tree.children:
183             elemList.append(str(child))
184         return ",".join(elemList)
185
186     def grammar__declarations__dict_contents(self,tree):
187         elemList = []
188
189         for key,value in zip(tree.children[0::2], tree.children[1::2]):
190             v_key = self.visit(key)
```

```
191             v_value = self.visit(value)
192
193             elemList.append(f"{v_key}:{v_value}")
194
195         return ",".join(elemList)
196
197     def grammar__declarations__dict_value(self,tree):
198         return self.visit(tree.children[0])
199
200     def grammar__declarations__operand_value(self,tree):
201         return self.visit(tree.children[0])
202
203     def grammar__declarations__operand_var(self,tree):
204         varName = self.visit(tree.children[0])
205         return varName
206
207     def grammar__declarations__value_string(self,tree):
208         return str(tree.children[0])
209
210     def grammar__declarations__value_float(self,tree):
211         return str(tree.children[0])
212
213     def grammar__declarations__value_int(self,tree):
214         return str(tree.children[0])
215
216     def grammar__declarations__value_bool(self,tree):
217         return str(tree.children[0])
218
219     def code(self,tree):
220         r=list()
221         for child in tree.children:
222             r.append(self.visit(child))
223         return r
224
225     def instruction(self,tree):
226         return self.visit(tree.children[0])
227
228     def atribution(self,tree):
229         varName = self.visit(tree.children[0])
230
231         exp = self.visit(tree.children[1])
232
233         self.graph.append(set())
234         self.graph[self.nodeBefore].add(self.nodeCount)
235
236         self.graphMap[self.nodeCount] = f"{varName} = {exp}"
237
238         self.nodeBefore = self.nodeCount
```

```
239         self.nodeCount += 1
240 
241         atrStr = f"{varName} = {exp};"
242 
243         return atrStr
244 
245 
246     def condition(self,tree):
247         cond = self.visit(tree.children[0])
248 
249         self.graph.append(set())
250         self.graph[self.nodeBefore].add(self.nodeCount)
251 
252         self.graphMap[self.nodeCount] = "if( "+cond+" )"
253 
254         currentNode = self.nodeCount
255 
256         self.nodeBefore = self.nodeCount
257         self.nodeCount += 1
258 
259         code = self.visit(tree.children[1])
260 
261         self.graph[currentNode].add(self.nodeCount)
262 
263         self.graph.append(set())
264         self.graph[self.nodeBefore].add(self.nodeCount)
265 
266         self.graphMap[self.nodeCount] = "end if"
267 
268         self.nodeBefore = self.nodeCount
269         self.nodeCount += 1
270 
271         taggedCode = "if("+cond+") {"
272         taggedCode += ''.join(code)
273         taggedCode += "}"
274 
275         return taggedCode
276 
277     def condition_else(self,tree):
278         cond = self.visit(tree.children[0])
279 
280         self.graph.append(set())
281         self.graph[self.nodeBefore].add(self.nodeCount)
282 
283         self.graphMap[self.nodeCount] = "if( "+cond+" )"
284 
285         currentNode = self.nodeCount
286 
```

```
287         self.nodeBefore = self.nodeCount
288         self.nodeCount += 1
289 
290         code = self.visit(tree.children[1])
291 
292         trueNode = self.nodeBefore
293 
294         self.graph[currentNode].add(self.nodeCount)
295 
296         self.nodeBefore = currentNode
297 
298         elseCode = self.visit(tree.children[2])
299 
300         self.graph.append(set())
301         self.graph[self.nodeBefore].add(self.nodeCount)
302         self.graph[trueNode].add(self.nodeCount)
303 
304         self.graphMap[self.nodeCount] = "end if"
305 
306         self.nodeBefore = self.nodeCount
307         self.nodeCount += 1
308 
309         returnCode = ("if( "+cond+") {")
310         returnCode += ''.join(code)
311         returnCode +=("}")
312         returnCode +=("else {")
313         returnCode += ''.join(elseCode)
314         returnCode +=("}")
315 
316         return returnCode
317 
318     def cycle(self,tree):
319         return self.visit(tree.children[0])
320 
321     def while_cycle(self,tree):
322         bool=self.visit(tree.children[0])
323 
324         self.graph.append(set())
325         self.graph[self.nodeBefore].add(self.nodeCount)
326 
327         self.graphMap[self.nodeCount] = "while( "+bool+" )"
328 
329         currentNode = self.nodeCount
330 
331         self.nodeBefore = self.nodeCount
332         self.nodeCount += 1
333 
334         code=self.visit(tree.children[1])
```

```
335
336         self.graph[self.nodeBefore].add(currentNode)
337         self.graph[currentNode].add(self.nodeCount)
338
339         self.graph.append(set())
340         self.graphMap[self.nodeCount] = "end while"
341
342         self.nodeBefore = self.nodeCount
343         self.nodeCount += 1
344
345         returnCode = "while(" + bool + ") {"
346         returnCode += ''.join(code)
347         returnCode += "}"
348
349         return returnCode
350
351     def do_while_cycle(self,tree):
352
353         self.graph.append(set())
354         self.graph[self.nodeBefore].add(self.nodeCount)
355
356         self.graphMap[self.nodeCount] = "do"
357
358         currentNode = self.nodeCount
359
360         self.nodeBefore = self.nodeCount
361         self.nodeCount += 1
362
363         code=self.visit(tree.children[0])
364
365         bool=self.visit(tree.children[1])
366
367         self.graph.append(set())
368         self.graph[self.nodeBefore].add(self.nodeCount)
369
370         self.graphMap[self.nodeCount] = "while( "+bool+" )"
371
372         self.graph[self.nodeCount].add(currentNode)
373
374         self.nodeBefore = self.nodeCount
375         self.nodeCount += 1
376
377         self.graph.append(set())
378         self.graph[self.nodeBefore].add(self.nodeCount)
379
380         self.graphMap[self.nodeCount] = "end do while"
381
382         self.nodeBefore = self.nodeCount
```

```
383         self.nodeCount += 1
384 
385         returnCode = "do {"
386         returnCode += ''.join(code)
387         returnCode += "} while("+bool+")"
388 
389         return returnCode
390 
391     def repeat_cycle(self,tree):
392         mat=self.visit(tree.children[0])
393 
394         self.graph.append(set())
395         self.graph[self.nodeBefore].add(self.nodeCount)
396 
397         self.graphMap[self.nodeCount] = "repeat " + mat
398 
399         currentNode = self.nodeCount
400 
401         self.nodeBefore = self.nodeCount
402         self.nodeCount += 1
403 
404         code=self.visit(tree.children[1])
405 
406         self.graph[self.nodeBefore].add(currentNode)
407 
408         self.graph.append(set())
409         self.graph[self.nodeBefore].add(self.nodeCount)
410 
411         self.graphMap[self.nodeCount] = "end repeat"
412 
413         self.nodeBefore = self.nodeCount
414         self.nodeCount += 1
415 
416         returnCode = "repeat(" + mat + ") {"
417         returnCode += ''.join(code)
418         returnCode += "}"
419 
420         return returnCode
421 
422     def for_cycle(self,tree):
423         self.graph.append(set())
424         self.graph[self.nodeBefore].add(self.nodeCount)
425 
426         self.graphMap[self.nodeCount] = "for"
427 
428         self.nodeBefore = self.nodeCount
429         self.nodeCount += 1
430 
```

```
431         childInfo = [None,None,None,None]
432 
433         childInfo[0] = self.visit(tree.children[0]) # Atribution 1
434         childInfo[1] = self.visit(tree.children[1]) # Condition
435 
436         self.graph.append(set())
437         self.graph[self.nodeBefore].add(self.nodeCount)
438 
439         self.graphMap[self.nodeCount] = childInfo[1]
440 
441         conditionNode = self.nodeCount
442 
443         self.nodeBefore = self.nodeCount
444         self.nodeCount += 1
445 
446         childInfo[3] = self.visit(tree.children[3]) # Code
447 
448         childInfo[2] = self.visit(tree.children[2]) # Atribution 2
449 
450         self.graph[self.nodeBefore].add(conditionNode)
451         self.graph[conditionNode].add(self.nodeCount)
452 
453         self.graph.append(set())
454 
455         self.graphMap[self.nodeCount] = "end for"
456 
457         self.nodeBefore = self.nodeCount
458         self.nodeCount += 1
459 
460         insidePar = f'{"" if childInfo[0] is None else childInfo[0]};{"" if
                childInfo[1] is None else childInfo[1]};{"" if childInfo[2] is
                None else childInfo[2]}'
461 
462         returnCode = "for(" + insidePar + ") {"
463         returnCode += ''.join(childInfo[3])
464         returnCode += "}"
465 
466         print(returnCode)
467 
468         return returnCode
469 
470 
471     def expression(self,tree):
472         return self.visit(tree.children[0])
473 
474     def matexpr(self,tree):
475         r=""
476         for child in tree.children:
```

```
477             if(isinstance(child,Tree)):
478                 r+=self.visit(child)
479             else:
480                 r+=child
481         return r
482 
483     def simple_bool_expr(self,tree):
484         r=""
485         for child in tree.children:
486             if(isinstance(child,Tree)):
487                 r+=self.visit(child)+" "
488             else:
489                 r+=child+" "
490 
491         return r
492 
493 
494     def boolexpr(self,tree):
495         r=""
496         for child in tree.children:
497             if(isinstance(child,Tree)):
498                 r+=self.visit(child)+" "
499             else:
500                 r+=child+" "
501 
502         return r
503 
504     def operand(self,tree):
505         value=self.visit(tree.children[0])
506         return value
507 
508     def value_string(self,tree):
509         return str(tree.children[0])
510 
511     def value_float(self,tree):
512         return str(tree.children[0])
513 
514     def value_int(self,tree):
515         return str(tree.children[0])
516 
517     def value_bool(self,tree):
518         return str(tree.children[0])
519 
520     def var(self,tree):
521         varName = str(tree.children[0])
522         retStr = varName
523 
524         if(len(tree.children) > 1):
```

```
525             operand = self.visit(tree.children[1])
526             retStr += '[' + operand + ']'
527 
528         return retStr
```

interpreterSDG.py

```
1 from distutils.log import error
2 from lark.visitors import Interpreter
3 from lark import Tree,Token
4 import interpreter.utils as utils
5 import re, graphviz
6 
7 global identNumber
8 
9 identNumber = 4
10 
11 
12 visitedList = [[]]
13 
14 def depthFirst(graph, currentVertex, visited):
15     visited.append(currentVertex)
16     for vertex in graph[currentVertex]:
17         if vertex not in visited:
18             depthFirst(graph, vertex, visited.copy())
19     if(len(visited)>1):
20         visitedList.append(visited)
21 
22 def checkVars(elems):
23     hasVars=False
24     for elem in elems:
25         if not hasVars and elem.isalpha() and not(elem.lower()=="False".lower
               ()) and not(elem.lower()=="True".lower()):
26             hasVars=True
27     return hasVars
28 
29 def createGraph(graph, graphMap,unreach):
30     dot = graphviz.Digraph('System Dependency Graph',strict=True)
31 
32     for (beg,currentVertex) in unreach:
33             depthFirst(graph,currentVertex,[])
34     auxVisited = visitedList
35     for i,node in enumerate(graph):
36         dot.node(str(i),graphMap[i])
37         for arrs in auxVisited:
38             if len(arrs)>0:
39                 arrs = arrs[1:]
40                 for i in range(len(arrs)-1):
41                     frm = arrs[i]
```

```
42                      to = arrs[i+1]
43                      dot.edge(str(frm),str(to),color="red")
44          auxVisited=[]
45          for row in node:
46              for idx,(beg,end) in enumerate(unreach):
47                  if(i==beg and row == end):
48                      dot.edge(str(i),str(row),color="red")
49                  elif(i==end):
50                      dot.edge(str(i),str(row),color="red")
51
52
53              dot.edge(str(i),str(row))
54
55      dot.render("System Dependency Graph.gv",view=True)
56
57      return dot
58
59
60  def checkUnreachable(graph,graphMap):
61      aux = []
62      opDel = ">|<|>=|<=|!=|=="
63      for k in graphMap:
64
65          if("for " in graphMap[k] or "if " in graphMap[k] or "do_while " in
                graphMap[k] or
66              "while " in graphMap[k]):
67              hasVars = False
68              conditions =[]
69              exp = graphMap[k]
70              exp = exp.strip()
71              fullExp = exp.split(' ', 1)[1]
72              cond = exp.split(" ")
73              func = cond[0]
74              for elems in cond[1:]:
75                  elems = elems.strip()
76                  elems= elems.split(" ")
77
78                  hasVars = checkVars(cond[1:])
79
80              conditions = [ele for ele in conditions if ele.strip()]
81              if hasVars:
82                  sepAnd = False
83                  sepOr = False
84                  if "&&" in fullExp:
85                      sepAnd = True
86                  elif "||" in fullExp:
87                      sepOr=True
88                  if sepAnd:
```

```
89                      expToEval = fullExp.split("&&")
90                      count=0
91                      for exp in expToEval:
92                          insideVars = checkVars(exp)
93                          if(not insideVars and eval(exp)==False):
94                              count+=1
95                      if func=="if" and count>0:
96                          if(len(graph[k])>1):
97                              aux.append((k,graph[k][-1]))
98                      elif count>0:
99                          if func=="if":
100                             aux.append((k,graph[k][0]))
101                         else:
102                             for x in graph[k]:
103                                 aux.append((k,x))
104                 elif sepOr:
105                     expToEval = fullExp.split("||")
106                     count=0
107                     for exp in expToEval:
108                         insideVars = checkVars(exp)
109                         if(not insideVars and eval(exp)==True):
110                             count+=1
111                     if func=="if" and count>0:
112                         if(len(graph[k])>1):
113                             aux.append((k,graph[k][-1]))
114                     elif count>0:
115                         if func=="if":
116                             aux.append((k,graph[k][0]))
117                         else:
118                             for x in graph[k]:
119                                 aux.append((k,x))
120
121             if not hasVars:
122                 expToEval = fullExp.replace("&&","and")
123                 expToEval = expToEval.replace("||","or")
124                 value= eval(expToEval)
125                 if func=="if" and value:
126                     if(len(graph[k])>1):
127                         aux.append((k,graph[k][-1]))
128                 elif not value:
129                     if func=="if":
130                         aux.append((k,graph[k][0]))
131                     else:
132                         for x in graph[k]:
133                             aux.append((k,x))
134     return aux
135
136 def mccabeComplexity(graph,graphMap):
```

```
137     numberNodes = len(graphMap)
138     numberEdges = 0
139     for conj in graph:
140         numberEdges+=len(conj)
141     return numberEdges-numberNodes+2
142 
143 
144 class MainInterpreterSDG (Interpreter):
145 
146     def __init__(self):
147         self.graph = []
148         self.graphMap = {}
149         self.nodeBefore = None
150         self.nodeCount = 0
151         self.graphNext = False
152         self.connectParent = list()
153 
154     def start(self,tree):
155         # Visita todos os filhos em que cada um v o retornar o seu c digo
156         self.graph.append(list())
157         self.graphMap[0] = 'entry main'
158         self.nodeCount = 1
159         self.connectParent.append(0)
160 
161         res = self.visit_children(tree)
162 
163         print(self.graph)
164         print(self.graphMap)
165         unreach = checkUnreachable(self.graph,self.graphMap)
166         #complexity = mccabeComplexity(self.graph,self.graphMap)
167 
168         graph = createGraph(self.graph, self.graphMap,unreach)
169         graph.save()
170         graphviz.render('dot','png','System Dependency Graph.gv')
171         #print(graph)
172 
173         output = dict()
174         # Juntar o c digo dos v rios blocos
175         output["html"] = res[0]
176         output["vars"] = self.variables
177         output["graph"] = graph
178 
179 
180 
181         return output
182 
183     def declaration(self,tree):
184         r = self.visit(tree.children[0])
```

```
185         return r
186 
187     def __generalDeclarationVisitor(self,tree,type):
188         dataType = str(tree.children[0])
189         varName = self.visit(tree.children[1])
190         childNum = len(tree.children)
191 
192         self.graph.append(list())
193 
194         self.graph[self.connectParent[-1]].append(self.nodeCount)
195 
196         # if variable is assigned
197         if childNum > 2:
198             operand = self.visit(tree.children[2])
199 
200             self.graphMap[self.nodeCount] = f"{dataType}{'' if type == '
                atomic' else ' ' + type} {varName} = {operand}"
201 
202             code = f"{dataType}{'' if type == 'atomic' else ' ' + type} {
                varName} = {operand};"
203 
204         else:
205 
206             self.graphMap[self.nodeCount] = f"{dataType}{'' if type == '
                atomic' else ' ' + type} {varName}"
207 
208             code = f"{dataType}{'' if type == 'atomic' else ' ' + type} {
                varName};"
209 
210         self.nodeCount += 1
211 
212         return code
213 
214     def grammar__declarations__atomic_declaration(self,tree):
215         return self.__generalDeclarationVisitor(tree,"atomic")
216 
217     def grammar__declarations__set_declaration(self,tree):
218         return self.__generalDeclarationVisitor(tree,"set")
219 
220     def grammar__declarations__list_declaration(self,tree):
221         return self.__generalDeclarationVisitor(tree,"list")
222 
223     def grammar__declarations__tuple_declaration(self,tree):
224         return self.__generalDeclarationVisitor(tree,"tuple")
225 
226     def grammar__declarations__dict_declaration(self,tree):
227         keyDataType = str(tree.children[0])
228         valueDataType = str(tree.children[1])
```

```
229         varName = self.visit(tree.children[2])
230         childNum = len(tree.children)
231 
232         self.graph.append(list())
233 
234         self.graph[self.connectParent[-1]].append(self.nodeCount)
235 
236         # if variable is assigned
237         if childNum > 3:
238             # Get value assigned to
239             operand = self.visit(tree.children[3])
240 
241             self.graphMap[self.nodeCount] = f"({keyDataType},{valueDataType})
                    dict {varName} = {operand}"
242 
243             code = f"({keyDataType},{valueDataType}) dict {varName} = {
                operand};"
244 
245         else:
246 
247             self.graphMap[self.nodeCount] = f"({keyDataType},{valueDataType})
                    dict {varName}"
248 
249             code = f"({keyDataType},{valueDataType}) dict {varName};"
250 
251         self.nodeCount += 1
252 
253         return code
254 
255     def grammar__declarations__var(self,tree):
256         return str(tree.children[0])
257 
258     def set(self,tree):
259         return f"{{{self.visit(tree.children[0])}}}"
260 
261     def list(self,tree):
262         return f"[{self.visit(tree.children[0])}]"
263 
264     def tuple(self,tree):
265         return f"({self.visit(tree.children[0])})"
266 
267     def dict(self,tree):
268         return f"{{{self.visit(tree.children[0])}}}"
269 
270     def grammar__declarations__list_contents(self,tree):
271         return self.visit(tree.children[0])
272 
273     def grammar__declarations__int_contents(self,tree):
```

```
274         elemList = []
275         for child in tree.children:
276             elemList.append(str(child))
277         return ",".join(elemList)
278 
279     def grammar__declarations__float_contents(self,tree):
280         elemList = []
281         for child in tree.children:
282             elemList.append(str(child))
283         return ",".join(elemList)
284 
285     def grammar__declarations__string_contents(self,tree):
286         elemList = []
287         for child in tree.children:
288             elemList.append(str(child))
289         return ",".join(elemList)
290 
291     def grammar__declarations__bool_contents(self,tree):
292         elemList = []
293         for child in tree.children:
294             elemList.append(str(child))
295         return ",".join(elemList)
296 
297     def grammar__declarations__dict_contents(self,tree):
298         elemList = []
299 
300         for key,value in zip(tree.children[0::2], tree.children[1::2]):
301             v_key = self.visit(key)
302             v_value = self.visit(value)
303 
304             elemList.append(f"{v_key}:{v_value}")
305 
306         return ",".join(elemList)
307 
308     def grammar__declarations__dict_value(self,tree):
309         return self.visit(tree.children[0])
310 
311     def grammar__declarations__operand_value(self,tree):
312         return self.visit(tree.children[0])
313 
314     def grammar__declarations__operand_var(self,tree):
315         varName = self.visit(tree.children[0])
316         return varName
317 
318     def grammar__declarations__value_string(self,tree):
319         return str(tree.children[0])
320 
321     def grammar__declarations__value_float(self,tree):
```

```
322         return str(tree.children[0])
323
324     def grammar__declarations__value_int(self,tree):
325         return str(tree.children[0])
326
327     def grammar__declarations__value_bool(self,tree):
328         return str(tree.children[0])
329
330     def code(self,tree):
331         r=list()
332         for child in tree.children:
333             r.append(self.visit(child))
334         return r
335
336     def instruction(self,tree):
337         return self.visit(tree.children[0])
338
339     def atribution(self,tree):
340         varName = self.visit(tree.children[0])
341
342         exp = self.visit(tree.children[1])
343
344         self.graph.append(list())
345
346         self.graph[self.connectParent[-1]].append(self.nodeCount)
347
348         self.graphMap[self.nodeCount] = f"{varName} = {exp}"
349
350         self.nodeCount += 1
351
352         atrStr = f"{varName} = {exp};"
353
354         return atrStr
355
356
357     def condition(self,tree):
358         cond = self.visit(tree.children[0])
359
360         self.graph.append(list())
361
362         self.graph[self.connectParent[-1]].append(self.nodeCount)
363
364         self.graphMap[self.nodeCount] = "if "+cond
365
366         ifNodeCount = self.nodeCount
367
368         self.nodeCount += 1
369
```

```
370         self.graph.append(list())
371         self.graph[ifNodeCount].append(self.nodeCount)
372         self.graphMap[self.nodeCount] = "then"
373         self.connectParent.append(self.nodeCount)
374 
375         self.nodeCount += 1
376 
377         code = self.visit(tree.children[1])
378 
379         self.connectParent.pop()
380 
381         taggedCode = "if("+cond+") {"
382         taggedCode += ''.join(code)
383         taggedCode += "}"
384 
385         return taggedCode
386 
387     def condition_else(self,tree):
388         cond = self.visit(tree.children[0])
389 
390         self.graph.append(list())
391 
392         self.graph[self.connectParent[-1]].append(self.nodeCount)
393         self.graphMap[self.nodeCount] = "if "+cond
394 
395         ifNodeCount = self.nodeCount
396 
397         self.nodeCount += 1
398 
399         self.graph.append(list())
400         self.graph[ifNodeCount].append(self.nodeCount)
401         self.graphMap[self.nodeCount] = "then"
402         self.connectParent.append(self.nodeCount)
403 
404         self.nodeCount += 1
405 
406         code = self.visit(tree.children[1])
407 
408         self.connectParent.pop()
409 
410         self.graph.append(list())
411         self.graph[ifNodeCount].append(self.nodeCount)
412         self.graphMap[self.nodeCount] = "else"
413         self.connectParent.append(self.nodeCount)
414 
415         self.nodeCount += 1
416 
417         elseCode = self.visit(tree.children[2])
```

```
418
419         self.connectParent.pop()
420
421         returnCode = ("if( "+cond+") {")
422         returnCode += ''.join(code)
423         returnCode +=("}")
424         returnCode +=("else {")
425         returnCode += ''.join(elseCode)
426         returnCode +=("}")
427
428         return returnCode
429
430     def cycle(self,tree):
431         return self.visit(tree.children[0])
432
433     def while_cycle(self,tree):
434         bool=self.visit(tree.children[0])
435
436         self.graph.append(list())
437         self.graph[self.connectParent[-1]].append(self.nodeCount)
438
439         self.graphMap[self.nodeCount] = "while "+ bool
440         self.connectParent.append(self.nodeCount)
441
442         self.nodeCount += 1
443
444         code=self.visit(tree.children[1])
445
446         self.connectParent.pop()
447
448
449         returnCode = "while(" + bool + ") {"
450         returnCode += ''.join(code)
451         returnCode += "}"
452
453         return returnCode
454
455     def do_while_cycle(self,tree):
456
457         self.graph.append(list())
458         self.graph[self.connectParent[-1]].append(self.nodeCount)
459
460         bool=self.visit(tree.children[1])
461
462         self.graphMap[self.nodeCount] = "do_while " + bool
463
464         self.connectParent.append(self.nodeCount)
465         self.nodeCount += 1
```

```
466
467         code=self.visit(tree.children[0])
468
469         self.connectParent.pop()
470
471         returnCode = "do {"
472         returnCode += ''.join(code)
473         returnCode += "} while("+bool+")"
474
475         return returnCode
476
477     def repeat_cycle(self,tree):
478         mat=self.visit(tree.children[0])
479
480         self.graph.append(list())
481         self.graph[self.connectParent[-1]].append(self.nodeCount)
482
483         self.graphMap[self.nodeCount] = "repeat " + mat
484
485         self.connectParent.append(self.nodeCount)
486         self.nodeCount += 1
487
488         code=self.visit(tree.children[1])
489
490         self.connectParent.pop()
491
492         self.nodeCount += 1
493
494         returnCode = "repeat(" + mat + ") {"
495         returnCode += ''.join(code)
496         returnCode += "}"
497
498         return returnCode
499
500     def for_cycle(self,tree):
501         self.graph.append(list())
502         self.graph[self.connectParent[-1]].append(self.nodeCount)
503
504         childInfo = [None,None,None,None]
505         childInfo[1] = self.visit(tree.children[1]) # Condition
506
507         self.graphMap[self.nodeCount] = "for " + childInfo[1]
508
509         self.connectParent.append(self.nodeCount)
510         self.nodeCount += 1
511
512         childInfo[0] = self.visit(tree.children[0]) # Atribution 1
513         childInfo[3] = self.visit(tree.children[3]) # Code
```

```
514         childInfo[2] = self.visit(tree.children[2]) # Atribution 2
515 
516         self.connectParent.pop()
517 
518         insidePar = f'{"" if childInfo[0] is None else childInfo[0]};{"" if
                childInfo[1] is None else childInfo[1]};{"" if childInfo[2] is
                None else childInfo[2]}'
519 
520         returnCode = "for(" + insidePar + ") {"
521         returnCode += ''.join(childInfo[3])
522         returnCode += "}"
523 
524         print(returnCode)
525 
526         return returnCode
527 
528 
529     def expression(self,tree):
530         return self.visit(tree.children[0])
531 
532     def matexpr(self,tree):
533         r=""
534         for child in tree.children:
535             if(isinstance(child,Tree)):
536                 r+=self.visit(child)
537             else:
538                 r+=child
539         return r
540 
541     def simple_bool_expr(self,tree):
542         r=""
543         for child in tree.children:
544             if(isinstance(child,Tree)):
545                 r+=self.visit(child)+" "
546             else:
547                 r+=child+" "
548 
549         return r
550 
551 
552     def boolexpr(self,tree):
553         r=""
554         for child in tree.children:
555             if(isinstance(child,Tree)):
556                 r+=self.visit(child)+" "
557             else:
558                 r+=child+" "
559 
```

```
560         return r
561
562     def operand(self,tree):
563         value=self.visit(tree.children[0])
564         return value
565
566     def value_string(self,tree):
567         return str(tree.children[0])
568
569     def value_float(self,tree):
570         return str(tree.children[0])
571
572     def value_int(self,tree):
573         return str(tree.children[0])
574
575     def value_bool(self,tree):
576         return str(tree.children[0])
577
578     def var(self,tree):
579         varName = str(tree.children[0])
580         retStr = varName
581
582         if(len(tree.children) > 1):
583             operand = self.visit(tree.children[1])
584             retStr += '[' + operand + ']'
585
586         return retStr
```

utils.py

```
1
2 def generateErrorTag(text, errorMessage="Erro na vari vel"):
3     retStr = '<div class="error">'
4     retStr += text
5     retStr += f'<span class="errortext">{errorMessage}</span></div>'
6
7     return retStr
8
9 def generateSubTag(text,subMessage="If conjugado"):
10     retStr = '<div class="sub">'
11     retStr += text
12     retStr += f'<span class="subtext">{subMessage}</span></div>'
13     return retStr
14
15
16
17 def generatePClassCodeTag(text):
18     retStr = f'''
19 <p class="code">
```

```
20      {text}
21  </p>'''
22
23      return retStr
24
25  def generateHTMLStatReport(numDeclaredVars,errors,warnings,numInstructions):
26      html = f'''<h1>Code Statistical Report</h1>
27      <h2>N mero de vari veis declaradas</h2>
28      <ul>
29          <li>At micas: {numDeclaredVars['atomic']}</li>
30          <li>Conjuntos: {numDeclaredVars['set']}</li>
31          <li>Listas: {numDeclaredVars['list']}</li>
32          <li>Tuplos: {numDeclaredVars['tuple']}</li>
33          <li>Dicion rios: {numDeclaredVars['dict']}</li>
34          <li>Total: {numDeclaredVars['atomic'] + numDeclaredVars['set'] +
                numDeclaredVars['list'] + numDeclaredVars['tuple'] +
                numDeclaredVars['dict']}</li>
35      </ul>
36      <br>
37      <h2>N mero de Instru  es</h2>
38      <ul>
39          <li>Atribui  es: {numInstructions['atribution']}</li>
40          <li>Leituras: {numInstructions['read']}</li>
41          <li>Escritas: {numInstructions['write']}</li>
42          <li>Condicionais: {numInstructions['condition']}</li>
43          <li>C clicas: {numInstructions['cycle']}</li>
44          <li>Controlo Aninhadas: {numInstructions['nestedControl']}</li>
45          <li>Total: {numInstructions['atribution'] + numInstructions['read'] +
                 numInstructions['write'] + numInstructions['condition'] +
                numInstructions['cycle']}</li>
46      </ul>
47      <br>
48      <h2>Erros</h2>
49      <ul>'''
50
51      for error in errors:
52          html += f'''
53          <li>{error}</li>'''
54
55      html += '''
56      </ul>
57      <br>
58      <h2>Warnings</h2>
59      <ul>'''
60
61      for warning in warnings:
62          html += f'''
63          <li>{warning}</li>'''
```

```
64
65      html += '''
66      </ul>'''
67
68      return html
69
70
71
72
73
74  def generateCSS():
75      retStr = '''
76  <style>
77      .error {
78          position: relative;
79          display: inline-block;
80          border-bottom: 1px dotted black;
81          color: red;
82      }
83
84      .code {
85          position: relative;
86          display: inline-block;
87      }
88
89      .error .errortext {
90          visibility: hidden;
91          width: 500px;
92          background-color: #555;
93          color: #fff;
94          text-align: center;
95          border-radius: 6px;
96          padding: 5px 0;
97          position: absolute;
98          z-index: 1;
99          bottom: 125%;
100         left: 50%;
101         margin-left: -40px;
102         opacity: 0;
103         transition: opacity 0.3s;
104     }
105
106     .error .errortext:after {
107         content: "";
108         position: absolute;
109         top: 100%;
110         left: 8%;
111         margin-left: -5px;
```

```
112         border-width: 5px;
113         border-style: solid;
114         border-color: #555 transparent transparent transparent;
115     }
116 
117     .error:hover .errortext {
118         visibility: visible;
119         opacity: 1;
120     }
121 
122 
123     .sub {
124         position: relative;
125         display: inline-block;
126         border-bottom: 1px dotted black;
127         color: green;
128     }
129 
130     .sub .subtext {
131         visibility: hidden;
132         width: 500px;
133         background-color: #555;
134         color: #fff;
135         text-align: center;
136         border-radius: 6px;
137         padding: 5px 0;
138         position: absolute;
139         z-index: 1;
140         bottom: 300%;
141         left: 50%;
142         margin-left: -40px;
143         opacity: 0;
144         transition: opacity 0.3s;
145     }
146 
147     .sub .subtext:after {
148         content: "";
149         position: absolute;
150         top: 100%;
151         left: 8%;
152         margin-left: -5px;
153         border-width: 5px;
154         border-style: solid;
155         border-color: #555 transparent transparent transparent;
156     }
157 
158     .sub:hover .subtext {
159         visibility: visible;
```

```
160             opacity: 1;
161         }
162 </style>'''
163 
164         return retStr
165 
166 def generateHTML(body,report):
167 
168         html = '''<!DOCTYPE html>
169 <html>'''
170 
171         html += generateCSS()
172 
173         html += '''
174 
175 <body>
176 
177         <h2> An lise de c digo </h2>
178 
179         <pre><code>'''
180 
181         html += body
182 
183         html += '''
184 
185         </code></pre>'''
186 
187         html += report
188 
189         html += '''
190 </body>
191 
192 </html>'''
193 
194         with open("index.html","w",encoding="utf-8") as f:
195             f.write(html)
196 
197         return None
198 
199         def insertGraphsHTML(html,nodes,edges):
200             soup = BeautifulSoup(open('index.html'), 'html.parser')
201             tagCFG =soup.new_tag("h1")
202             tagCFG.string = "Control Flow Graph"
203             soup.body.append(tagCFG)
204             imgCFG = soup.new_tag("img",src="Control Flow Graph.gv.png")
205             soup.body.append(imgCFG)
206             tagComp =soup.new_tag("p")
207             complexidade = edges-nodes+2
```

```
208         tagComp.string = "Complexidade de McCabe<=>"+str(edges)+"-"+str(nodes
                )+"+2="+str(complexidade)
209         soup.body.append(tagComp)
210         tagSDG =soup.new_tag("h1")
211         tagSDG.string = "System Dependency Graph"
212         soup.body.append(tagSDG)
213         imgSDG = soup.new_tag("img",src="System Dependency Graph.gv.png")
214         soup.body.append(imgSDG)
215 
216         with open("index.html", "w") as file:
217             file.write(str(soup))
```