Engenharia Gramatical

Trabalho Prático 2

Relatório de Desenvolvimento

Grupo 9

Júlio Alves (PG47390)

Rúben Cerqueira (PG47626)

25 de abril de 2022

Resumo

Neste relatório irá ser abordado o desenvolvimento de um analisador de código fonte para uma linguagem imperativa, denominada LPIS2, que permite declarar variáveis atómicas e estruturadas, intruções condicionais e cíclicas.

Conteúdo

Capítulo 1

Introdução

1.1 LPIS2

Área: Processamento de Linguagens

Para este trabalho tinhamos como objetivo a criação de um analisador de código fonte para uma linguagem imperativa desenvolvida por nós, a LPIS2. De forma a cumprir com os objetivos propostos para este trabalho, foi necessário desenvolver uma gramática e um interpretador, utilizando para tal a biblioteca Lark do Python. O interpretador teria de ser capaz de gerar um relatório em HTML com os resultados da análise, nomeadamente:

- Listar todas as variáveis do programa, indicando casos de redeclaração, não-declaração, variáveis usadas mas não inicializadas e variáveis declaradas e nunca mencionadas.
- Total de variáveis declaradas versus os tipos de dados estruturados usados.
- Total de instruções que formam o corpo do programa, indicando os seus tipos (atribuições, leitura e escrita, condicionais e cíclicas).
- Total de situações em que estruturas de controlo surgem aninhadas dentro de outras estruturas de controlo, sejam elas do mesmo tipo ou de tipos diferentes.
- Adicionar informações acerca da presença de ifs aninhados, indicando os casos em que os ifs aninhadas possam ser substituídos por apenas um if.

Estrutura do Relatório

O relatório está dividido em 5 capítulos, sendo eles, Introdução, Gramática, Interpretador, Resultados e Conclusão. Tanto no capítulo da Gramática como do Interpretador temos dois subcapítulos, sendo eles Declarações e Instruções, onde, em cada um, irão ser abordadas as características das mesmas. No capítulo dos resultados, irão ser mostrados alguns dos relatórios gerados em HTML, onde serão indicados pormenores que acreditamos serem relevantes.

Capítulo 2

Gramática

No contexto de desenvolvimento do trabalho, foi usada a sintaxe de gramática adotada pelo Lark que usa a sintaxe EBNF de forma a facilitar e simplificar a escrita de gramáticas para a sua futura utilização pelo respetivo Lark Parser.

Para tornar esta componente modular, foi realizada uma separação da gramática em dois ficheiros, um contendo lógica referente à declaração de variáveis da linguagem e o segundo contendo a gramática que dita a representação das várias instruções suportadas pela LPIS2.

O ficheiro principal será o que contém a gramática direcionada às instruções do programa, que posteriormente irá importar o ficheiro relativo à declaração de variáveis, contendo assim a totalidade da gramática necessária para ter suporte total à linguagem LPIS2.

2.1 Declaração de Variáveis

O ficheiro que contém esta gramática tem como extensão .lark de forma a que possa ser importado por outras gramáticas do contexto Lark.

A gramática escrita suporta formas de declaração semelhantes à linguagem imperativa 'C' para variáveis atómicas. São suportados os seguintes tipos atómicos, sendo 'x' uma variável arbitrária a ser declarada:

- Inteiro ("int x;")
- Float ("float x;")
- Bool ("bool x;")
- String ("str x;")

Quanto a tipos estruturados, foram suportados 4 tipos, sendo esses listas, tuplos, conjuntos e dicionários. A declaração de variáveis do tipo estruturado vem sempre acompanhada pelo tipo de dados primitivo que vão compor essas estruturas, abordados anteriormente. No caso de dicionário,

a sua declaração será acompanhada por um tuplo de tipos, que correspondem ao tipo da chave e do valor, pela ordem respetiva.
A sintaxe de declaração de cada, sendo 'x' uma variável arbitrária a ser declarada, é a seguinte:

- Lista (int list x;)
- Conjunto (int set x;)
- Tuplo (int tuple x;)
- Dicionário ((int,int) dict x;)

Aquando à declaração de uma variável, é também suportada a sua inicialização, tal como na linguagem imperativa 'C'. Porém, a gramática impede que haja inicialização de tipos de dados diferentes, isto é, uma variável declarada como lista, não pode ser inicializada com um valor atómico ou com um conjunto. Além dessa limitação, não é possível inicializar listas, conjuntos e tuplos com elementos de tipos diferentes, portanto, é necessário que todos os elementos sejam do mesmo tipo. Quanto aos dicionários, a lógica de verificação dos tipos é tratada a nível do interpretador.
Um exemplo de uma inicialização seria a seguinte:

```
int list x = [1,2,3,4];
```

Figura 2.1: C++ code

2.2 Instruções

As instruções estão declaradas num ficheiro grammar.py. Tal como as declarações, também as instruções são baseadas na linguagem "C". As instruções podem ser de três tipos, sendo elas, atribuição, condição e ciclo. A atribuição é definida da seguinte forma:

```
var "=" (expression | list | tuple | set | dict)
```

Figura 2.2: Definição de atribuição

Expression pode ser uma expressão booleana ou uma expressão matemática, cujas definições estão em anexo. Isto permite que seja atríbuido à variável um valor booleano ou, por exemplo, que a variável seja o resultado de uma operação matemática. As restantes alternativas, list, tuple, set e dict já foram abordadas na secção anterior.

A condicional é definida, como referido previamente, tal como em "C", podendo ela ser De forma a

```
if(exp){                    if(exp){
  "código"                   "código"
}                            } else{
                             "código"
                             }
```

Figura 2.3: Definição de condição

facilitar o desenvolvimento do interpreter, foi utilizado um alias para a expressão condicional if ... else. Desta forma, no interpreter, conseguimos distinguir mais facilmente o que fazer, uma vez que não temos de tratar das duas expressões condicionais na mesma definição e, como se irá ver mais tarde, irá ser benéfico separar ambas, nomeadamente para resolver a conjugação de ifs. Quanto às instruções cíclicas, estas podem ser de quatro tipos, sendo elas os ciclos while, do while, for e repeat.

```
do{                  while(exp){    for(atrib;exp;atrib){    repeat(exp){
   "código"            "código"       "código"                 "código"
  }while(exp)         }               }                       }
```

Figura 2.4: Definição de ciclo

De realçar que, assim como na linguagem "C", todos as expressões que constituem o ciclo for são opcionais, oferecendo ao utilizador mais alternativas para o desenvolvimento do seu programa. Assim como no caso das condicionais, também os ciclos estão separados em diferentes definições, oferecendo maior modularidade ao programa, facilitando o trabalho do engenheiro de desenvolver um interpretador capaz de extrair o máximo de funcionalidade possível ao trabalho e também permitindo que a gramática seja mais legível e facilmente compreendida.

Capítulo 3

Interpretador

Após ocorrido o parsing da linguagem é gerada uma árvore de parsing que irá ser percorrida por um interpretador capaz de retirar informação útil da árvore e aplicar a lógica necessária para o correto processamento da informação obtida.

Na presente aplicação, o interpretador construído irá gerar uma página HTML composta pelo código fonte processado acompanhado por um relatório contendo as informações gerais do código processado.

Este processamento do código consiste na identificação de casos de erro de semântica, ou seja, código que o parser aceita mas é inconsistente. Um exemplo adequado seria a atribuição de um valor de tipo inteiro a uma variável do tipo float. Além da identificação de casos de erro, o interpretador apresenta informações acerca do nível de aninhamento de estruturas de controlo, bem como a substituição de if's aninhados, quando esta é possível. Todo este processamento do código é anotado na página HTML gerada, nos espaços indicados, ou seja, erros de semântica na linha onde acontecem, if's juntos na estrutura pai e o nível de aninhamento a nível da linha da estrutura de controlo.

No relatório que constitui a parte final da página HTML contém informações estatísticas do código, bem como o agregado de todos os erros presentes neste e os respetivos "warnings". As informações estatísticas retratam o número de variáveis presentes no programa, distinguindo-as pelo seu tipo (atómica, lista, conjunto, tuplo, dicionário) bem como o número de instruções presentes de cada tipo (atribuição, leitura, escrita, cíclica, condicional). Todas as informações constantes no relatório apenas contabilizam as linhas de código que não contêm erros. Não é relevante apresentar informação relativa a código com erros e, por isso, esses dados não deverão ser incluídos no relatório final.

3.1 Variáveis

O interpretador terá que ter estruturas de informação que capturam os estado das variáveis e os contadores necessários até ao ponto de análise do código do programa. Isto é necessário uma vez que cada regra que o interpretador visitar irá retornar o código com as tags HTML apropriadas para posteriormente conseguir-se identificar o processamento feito na página gerada. Todas estas estruturas de informação serão variáveis de instância da classe e estas serão preenchidas à medida que os nós da árvore de parsing são visitados.

Na declaração de variáveis, grande parte das operações são inserções de informação de variáveis na estrutura. Como é permitida a inicialização ao mesmo tempo que a variável é declarada, é necessário

também atualizar essa informação na estrutura de dados. Esta estrutura responsável por guardar o estado das variáveis é composta por um dicionário que mapeia nomes de variáveis para um outro dicionário. Este segundo irá mapear o estado, o tipo dos dados e o tipo da variável (atómica ou não). O estado da variável será definido por um array de três posições, em que a primeira corresponde ao estado da declaração, a segunda ao estado de inicialização e a terceira ao uso.

Esta informação permite identificar qualquer tipo de erros referentes a variáveis, tal como atribuições de tipos não compatíveis ou a redeclaração.

Para ser possível fazer a análise dos tipos de dados em inicializações e atribuições, foi necessário criar uma variável de instância que contém o tipo de dados e o tipo de variável do ultimo nó visitado. Isto é útil porque nem todos valores vão estar guardados na estrutura de dados e por isso era necessário extrair o seu tipo de alguma forma. Daí apenas ser necessário comparar o tipo da variável na operação com o tipo de dados resultante da variável responsável por buscar a informação de tipagem do valor do lado direito da instrução.

Apenas a lógica de verificar se uma variável era mencionada no programa é feita no fim da visita da árvore, uma vez que era necessário visitar a totalidade da árvore para analisar o caso. Este processo apenas consta em verificar se a declaração foi declarada mas não inicializada ou usada, através do campo estado da estrutura de dados associada à variável em questão.

Como foi anteriormente mencionado, estes estados e contadores de variáveis, apenas tomarão em conta nós da árvore que não contenham erros de semântica, a título de exemplo, se uma variável tiver uma inicialização com um valor de tipo incompatível, está variável não terá o seu estado alterado e o respetivo contador não irá incrementar.

3.2 Instruções

Como foi dito no capítulo anterior, as instruções podem ser de três tipos, e cada um desses tipos tem processamentos diferentes.

No caso das instruções de atribuição, é feita a análise se os variáveis que estão a ser atribuídas foram declaradas. De forma a fazer esta análise, é consultado o dicionário de variáveis existentes e verifica-se se a variável em questão se encontra presente no mesmo e, em caso de resposta negativa, é lançado o erro. No entanto, ainda falta verificar se a atribuição está a ser feita corretamente, isto é, se a variável a quem está a ser feita a atribuição, se está a atribuir corretamente um valor. Para fazer esta verificação, é consultado o tipo da variável no dicionário e no caso de os seus tipos não corresponderem é lançado o erro de tipagem incorreta.

Para as instruções de condicionais, e como mencionado no capítulo anterior, possuímos duas definições diferentes para as mesmas, uma para a instrução condicional com else e outra para a condicional apenas com um if. Ambas são bastante semelhantes, tendo apenas uma diferença que será explicada brevemente. De maneira a possuir a indicação do nível de aninhamento em que nos encontramos, e também para controlar a identação do código que será apresentado no relatório, temos duas variáveis de instância, uma responsável pelo nível de identação e outra responsável pelo nível de aninhamento. Esta última é utilizada, também, para nos permitir saber quantos estruturas de controlo estão aninhadas dentro de outras, uma vez que caso o valor desta variável seja superior a zero, sabemos que nos encontramos dentro de uma estrutura de controlo e desta forma podemos incrementar a variável responsável por essa contabilização.

Relativamente à gestão do nível de controlo e de identação, funciona como se fosse uma stack, ou seja, inicializamos uma variável local que adquire o valor da variável de instância e incrementamos o valor da variável de instância. No final, a variável de instância volta a ter o valor da variável global, de forma a manter a coerência da informação.

Relativamente à diferença entre ambas as definições, esta diferença está presente na definição da condicional do if, uma vez que, quando pretendemos conjugar ifs, apenas esta instrução é válida, dado que não é logicamente correto conjugar if elses. Explicando a diferença, é consultada uma variável de instância que possui um tuplo com a expressão condicional e com o código existente na instrução condicional. Caso esta variável não esteja sem qualquer valor e o comprimento dos filhos do código seja um, condição fundamental pois o corpo da condicional não pode ter mais nenhuma instrução para além de uma condicional, visto que caso tenha, não pode haver conjugação de ifs, conjuga-se a expressão condicional com a expressão armazenada na variável de instância e o corpo da instrução atual também com a armazenada. Importante realçar que, no caso de haver possibilidade de conjugar ifs, decrementa-se o valor das variáveis de controlo aninhadas e das instruções condicionais.

Para as instruções cíclicas, é realizado um processo em tudo semelhante aos das instruções condicionais, relativamente à gestão dos níveis de controlo. A única estrutura cíclica com algo a salientar é a do for, uma vez que, como todas as suas expressões são opcionais, foi necessário um controlo maior. Para tal, inicializamos uma variável com quatro posições, posições essas relativas às três expressões e a restante ao corpo do ciclo. Conforme se vão visitando os filhos da sua subárvore, verifica-se qual o tipo desse filho e coloca-se na sua posição respetiva. Desta forma temos pleno controlo sobre o ciclo for, tendo ele todas as expressões ou nenhuma.

Capítulo 4

Testes realizados e resultados

De forma a gerar o relatório e demonstrar os resultados pretendidos, foi escrito um pequeno programa que podemos ver de seguida:

```
str a = "1";

str b = a;
str ggg;
int b;
str gh = ggg;
str set guy = {"on","tw","th","fo"};
float tuple tu = (1.2,2.,3.);
(int,str) dict di = {3:"ro",6:"he",7:"ko",9:"o"};
bool goo = 2;
str teste;
a = 1;
teste = di[1];
if(cond > 0 + 1 && a + 1 == 0 || cond < 2) {
    if(y[0] > 1) {
        if(z > 2) {
            if(z > 3) {
                z = 1; k = 3;
            }
        }
    }
    while(cond > 0) {
        if(z < 8) {
            z = 1;
        }
        else {
            x[1] = 0;
        }
    }
}
for(i = 0; i<a;a=a+1){
    a =1;
}
x=3;
```

Figura 4.1: Programa de exemplo

Na imagem apresentada de seguida, pudemos ver a parte do código onde estão as declarações. Nesta podemos ver os diferentes tipos de declarações, sendo que, no caso de haver algum erro com a mesma, a variável que está em erro é colorida com a cor vermelha e se passarmos o rato por cima das mesmas, aparece uma mensagem a indicar o erro.

```
str a = "1";

str b = a;

str ggg;

int b;


str gh = ggg;


str set guy = {"on","tw","th","fo"};

float tuple tu = (1.2,2.,3.);

(int,str) dict di = {3:"ro",6:"he",7:"ko",9:"o"};

bool goo = 2;
```

Figura 4.2: Declarações de variáveis

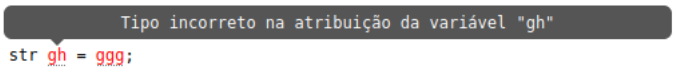

Figura 4.3: Mensagem de erro em atribuição

Relativamente às instruções, tal como nas declarações, sempre que uma variável é utilizada incorretamente, também fica colorida a vermelho, bem como apresenta a mensagem de erro se o rato estiver em cima da mesma.
No caso de haver possibilidade de juntar ifs, estes são apresentados na forma ótima e a sua expressão condicional é apresentada a verde e, assim como no caso dos erros, se o rato passar por cima da expressão, é apresentada uma mensagem a indicar que os ifs foram conjugados. Em todas as estruturas de controlo, é apresentada o nível de aninhamento em que nos encontramos, de forma a ser mais fácil perceber quão profundo estamos.
Após apresentarmos todo o código, aparece o relatório estatístico, onde indicamos o número de variáveis declaradas, o número de instruções, os erros e os warnings. É feita a distinção entre erros e warnings, uma vez que os warnings não colocam em causa o funcionamento do código, ao contrário dos erros.

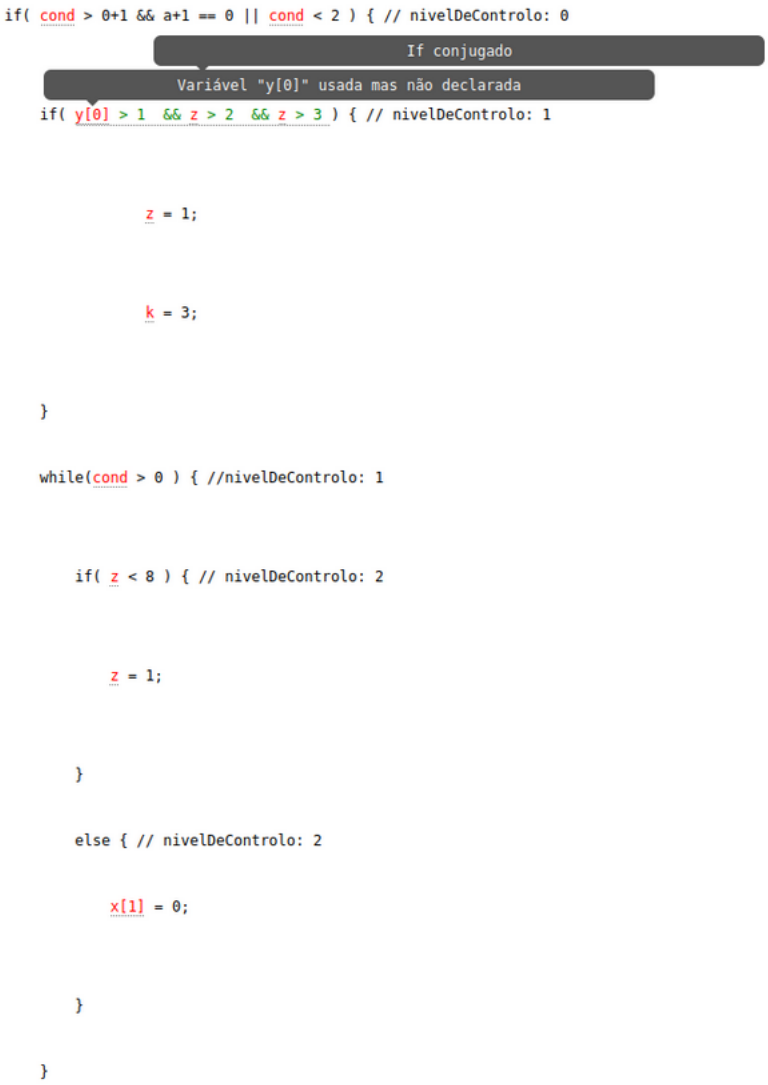

Figura 4.4: Instruções com as mensagens existentes

Número de variáveis declaradas

- Atómicas: 4
- Conjuntos: 1
- Listas: 0
- Tuplos: 1
- Dicionários: 1
- Total: 7

Figura 4.5: Número de variáveis declaradas

Número de Instruções

- Atribuições: 0
- Leituras: 3
- Escritas: 13
- Condicionais: 3
- Cíclicas: 2
- Controlo Aninhadas: 3
- Total: 21

Figura 4.6: Número de instruções no programa

Erros

- Variável "z" atribuida mas não declarada
- Variável "y[0]" usada mas não declarada
- Variável "i" usada mas não declarada
- Variável "z" usada mas não declarada
- Tipo incorreto na atribuição da variável "gh"
- Variável "k" atribuida mas não declarada
- Tipos incompatíveis na atribuição de um valor à variável "teste"
- Tipo incorreto na atribuição da variável "goo"
- Variável "cond" usada mas não declarada
- Variável "x" atribuida mas não declarada
- Variável "di[1]" usada mas não declarada
- Variável "x[1]" atribuida mas não declarada
- Variável "ggg" não atribuída
- Tipos incompatíveis na atribuição de um valor à variável "a"
- Variável "b" redeclarada
- Variável "i" atribuida mas não declarada

Figura 4.7: Erros existentes no programa

Warnings

- Variável "ggg" declarada mas nunca mencionada
- Variável "teste" declarada mas nunca mencionada

Figura 4.8: Warnings existentes no programa

Capítulo 5

Conclusão

Com a realização deste trabalho, foi possível aprofundar os nossos conhecimentos na área do processamento de linguagens, nomeadamente no desenvolvimento de gramáticas EBNF, sendo que também pudemos explorar Interpreters, que tem um funcionamento diferente dos Visitors e dos Transformers,e que também foram estudados nas aulas de UC, uma vez que ao contrário destes, dado que este funciona de forma top-down e, para visitar os ramos inferiores é necessário que se chame explicite as funções de visita.

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 vão retornar o seu código
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 código dos vários 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("Variável \"" + 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("Variável \"" + 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 atribuição da variável \"" +
                        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("Variável \"" + 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 atribuição da variável \"" +
                        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),"Dicionário 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
                    ário não são 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("Variável \"" + varName + "\" não declarada")
301             varName = utils.generateErrorTag(varName,"Variável não 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("Variável \"" + varName + "\" não atribuída")
312             varName = utils.generateErrorTag(varName,"Variável não atribuída"
                    )
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("Variável \"" + varName + "\" atribuida mas não declarada")
371             varName = utils.generateErrorTag(varName,"Variável \"" + varName + "\" atribuída mas não declarada")
372 
373         elif self.variables[varName]['type'] != self.valueType or self.variables[varName]['datatype'] != self.valueDataType:
374             self.errors.append("Tipos incompatíveis na atribuição de um valor à variável \"" + varName + "\"")
375             varName = utils.generateErrorTag(varName,"Tipos incompatíveis na atribuição")
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         # Cálculo da identação 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         # Cálculo da identação 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         # Cálculo da identação 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         # Cálculo da identação 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         # Cálculo da identação 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         # Cálculo da identação 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("Variável \"" + value + "\" usada mas não
                        declarada")
619 
620             elif self.variables[value]["state"][1] == False:
621                 errors.append("Variável \"" + value + "\" usada mas não
                        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("Índice não é do tipo int")
658                 operand = utils.generateErrorTag(operand,"Índice não é 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
```

utils.py

```
1
2 def generateErrorTag(text, errorMessage="Erro na variável"):
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>Número de variáveis declaradas</h2>
28      <ul>
29          <li>Atómicas: {numDeclaredVars['atomic']}</li>
30          <li>Conjuntos: {numDeclaredVars['set']}</li>
31          <li>Listas: {numDeclaredVars['list']}</li>
32          <li>Tuplos: {numDeclaredVars['tuple']}</li>
33          <li>Dicionários: {numDeclaredVars['dict']}</li>
34          <li>Total: {numDeclaredVars['atomic'] + numDeclaredVars['set'] +
                numDeclaredVars['list'] + numDeclaredVars['tuple'] +
                numDeclaredVars['dict']}</li>
35      </ul>
36      <br>
37      <h2>Número de Instruções</h2>
38      <ul>
39          <li>Atribuições: {numInstructions['atribution']}</li>
40          <li>Leituras: {numInstructions['read']}</li>
41          <li>Escritas: {numInstructions['write']}</li>
42          <li>Condicionais: {numInstructions['condition']}</li>
43          <li>Cíclicas: {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> Análise de código </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
```