Perfil EL - Engenharia de Linguagens
(1º ano do MEI)
Engenharia Gramatical

Trabalho Prático 3

Relatório de Desenvolvimento

João Gonçalves
PG46535

Sara Queirós
PG47661

15 de junho de 2022

Resumo

O presente documento tem como principal objetivo explicar os procedimentos e raciocínio desenvolvidos para gerar grafos capazes de analisar a linguagem LPIS2, criada por nós, no TP2. O resultado do programa em causa deve gerar um HMTL com o resultado das análises feitas ao documento em causa, assim como os grafos CFG e SDG.

Conteúdo

Capítulo 1

Introdução

Ferramentas avançadas de análise de código têm como intuito ajudar a embelezar o código, detetar situações que infringem boas práticas de codificação, sugestões de simplificação de código sem alterar o seu significado inicial e até mesmo avaliar a sua performance estática ou dinâmica.
Para enriquecer ainda mais a análise feita, podem construir-se grafos que também permitem estudar o comportamento dos programas-fonte, nomeadamente estudar a sua complexidade, analisar instruções cíclicas, detetar grafos de ilha, etc.

Face a isto, o objetivo principal é gerar grafos de controlo de fluxo e de dependências do sistema, complementando a análise previamente elaborada e já presente no HTML.

Estrutura do Relatório

Este relatório é constituído por capítulos que englobam a Análise e Especificação do problema, descrevendo-o juntamente com os seus objetivos principais (Concepção e Desenho da Resolução) e ainda, um capítulo com apresentação dos testes efetuados. Finalmente, no último capítulo é exposta uma conclusão e trabalho futuro.

Capítulo 2

Análise e Especificação

2.1 Descrição informal do problema

Desenvolver um gerador de grafos para o código-fonte dado como input e complementar a análise previamente elaborada:

- Control Flow Graph
- System Dependency Graphs

2.2 Especificação do Requisitos

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

2.2.1 Grafos

A construção dos grafos CFG e SDG (desconsiderando o fluxo de dados) devem ter em consideração a representação de:

- Estruturas cíclicas: for, while , do-while (implementadas na linguagem LPIS2).
- Estrutura condicional if-else.
- Instruções de declaração, atribuição e input/output.

2.2.2 Análise de código utilizando grafos

Após efetuar a construção dos grafos, estes podem ser analisados para tirar conclusões:

- Zonas de código inalcansável (grafos de ilha);
- A complexidade de McCabe's para um determinado excerto de código.

Capítulo 3

Concepção/Desenho da Resolução

3.1 Melhorias à linguagem LPIS2

Após a entrega do Trabalho Prático 2, percebemos a necessidade de implementar melhorias na linguagem e na sua respetiva interpretação. Dessa forma, a gramática foi restruturada, possuindo a seguinte ideologia:

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

De forma a evitar a repetição do Token PV(;) em todas as produções, este foi colocado em evidência. Para além disso, a definição das listas foi modificada de forma a poder casos mais realistas permitindo a presença de listas no interior de listas, assim como em Python.

```
'''
variaveis: (declaracoes | atribuicoes) PV
declaracoes: decint | decstring | decdict | declist | decconj | dectuplos | decfloat |
    decinput
decint : INTW WORD (IGUAL (INT | operacao))?
  operacao : (NUMBER|WORD) ((SUM | SUB | MUL | DIV | MOD) (NUMBER|WORD))+
decstring : STRINGW WORD (IGUAL (ESCAPED_STRING|input))?
decdict : DICTW WORD (IGUAL DICTW PE PD)?
declist : INTW WORD (PRE NUMBER? PRD)+ (IGUAL (content | ultracontent))?
  content : CE (INT (VIR INT)*)* CD
  ultracontent: CE (content (VIR content)*)* CD
decconj: CONJW WORD (IGUAL CE (ESCAPED_STRING (VIR ESCAPED_STRING)*)? CD )*
dectuplos: TUPLEW WORD (IGUAL PE var (VIR var)+ PD)*
  var: INT | WORD | ESCAPED_STRING | FLOAT
decfloat: FLOATW WORD (IGUAL FLOAT)*
decinput: STRINGW IGUAL input
'''
```

As restantes produções mantêm-se relativamente semelhantes, havendo novamente, alterações para maximizar a eficiência do processamento do código, reaproveitando corretamente produções já definidas:

```
'''
atribuicoes: WORD IGUAL (var | operacao | input |lista | dicionario)
  input: INPUTW PE PD
  lista: WORD (PRE INT PRD)+
  dicionario: CE ESCAPED_STRING DP (INT | WORD)(VIR ESCAPED_STRING DP (INT | WORD))* CD

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

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

output: OUTPUTW PE (ESCAPED_STRING|WORD) PD PV

ciclos: (whilee | forr | dowhile) PV
whilee: WHILEW PE condicao PD CE code? CD
forr: FORW PE variaveis condicao PV atribuicoes PD CE code? CD
dowhile: DOW CE code? CD WHILEW PE condicao PD
'''
```

3.2 Alterações à apresentação de resultados

Com o intuito de corrigir algumas falhas na análise das estruturas condicionais e adequar as possíveis sugestões de simplificação de código, foram introduzidas as seguintes condições:

- A instrução condicional em causa não pode ter um 'else' associado;
- A primeira instrução no corpo desse 'if' tem de ser uma intrução condicional;
- Esse 'if', para ser aninhável com o mais exterior, também não pode possuir um 'else' associado.

Estas exigências traduzem-se na seguinte condição, adicionada à função da produção correspondente ao processamento de instruções condicionais:

```
    #cond: IFW PE condicao PD CE code? CD else? PV

    if not isinstance(tree.children[5], Token): #Tem codigo no corpo do if
```

```
    size = len(tree.children[5].children) #tamanho do corpo do if
    sizehere = len(tree.children) #tamanho da arvore do nivel atual
    sizeeee = len(tree.children[5].children[size-2].children)-2 #posio do 'else' do if
        aninhado
    if tree.children[5].children[0].data == 'cond' and
        isinstance(tree.children[5].children[size-2].children[sizeeee], Token) and
        isinstance(tree.children[sizehere-2], Token): #o 1 elem do codigo  um if
      self.aninhavel = True
    else:
      self.aninhavel = False
  else:
    self.aninhavel = False
```

Para além disso, na sequência das alterações efetuadas à gramática, tomamos algumas decisões de forma a tornar mais apelativo e intuitivo o relatório final do resultados das análises feitas ao código-fonte.
Deste modo, criámos secções para apresentar os resultados:

Declaração de Variáveis

As variáveis passaram a ser apresentadas numa tabela indicativa dos respetivos tipos. Para não causar confusão visual, os resultados são apresentados por ordem de tipos, sendo sempre assinalados os tipos da esquerda para a direita.

→ Variáveis Declaradas

Variável	conj	dict	float	int	list	string	tuple
c1	X						
d		X					
f			X				
x				X			
m				X			
i				X			
xw				X			
t					X		
st						X	
t1							X

Figura 3.1: Apresentação das variáveis declaradas no HTML

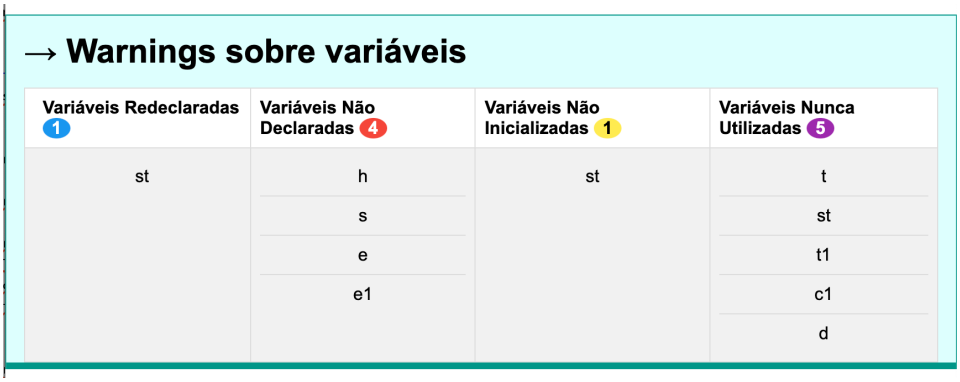
Warnings sobre variáveis

Considerando as 4 principais situações que podem gerar alertas sobre as variáveis podemos considerar:

1. Redeclaração de variáveis
2. Não declaração de uma variável mencionada
3. Utilização de uma variável não inicializada

4. Variáveis declaradas mas nunca utilizadas

Para estas categorias, a informação passou a ser representada numa tabela:



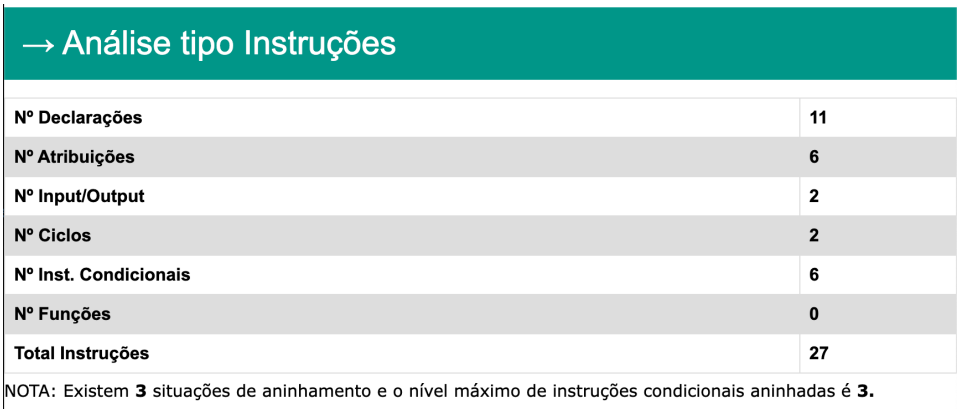
→ Warnings sobre variáveis

Variáveis Redeclaradas 1	Variáveis Não Declaradas 4	Variáveis Não Inicializadas 1	Variáveis Nunca Utilizadas 5
st	h	st	t
	s		st
	e		t1
	e1		c1
			d

Figura 3.2: Apresentação de warnings apenas relativos às variáveis

Análise dos Tipos de Instruções

Paralelamente à informação apresentada sobre os tipos e quantidades de instruções, colocamos a informação relativa aos níveis de aninhamento e a quantidades de situações em que aninhamentos acontecem.



→ Análise tipo Instruções

Nº Declarações	**11**
Nº Atribuições	**6**
Nº Input/Output	**2**
Nº Ciclos	**2**
Nº Inst. Condicionais	**6**
Nº Funções	**0**
Total Instruções	**27**

NOTA: Existem **3** situações de aninhamento e o nível máximo de instruções condicionais aninhadas é **3.**

Figura 3.3: Tipo de Instruções

Warnings Globais

Alternativamente às anotações de código que tinhamos, decidimos converter essas informações em warnings do programa apresentando sugestões de simplificação de estruturas condicionais e warnings sobre as variáveis, apresentando linha e coluna onde estes acontecem.
É de notar que os warnings são apresentados pela ordem na qual aparecem no código, de forma a facilitar a interpretação por parte do utilizador.

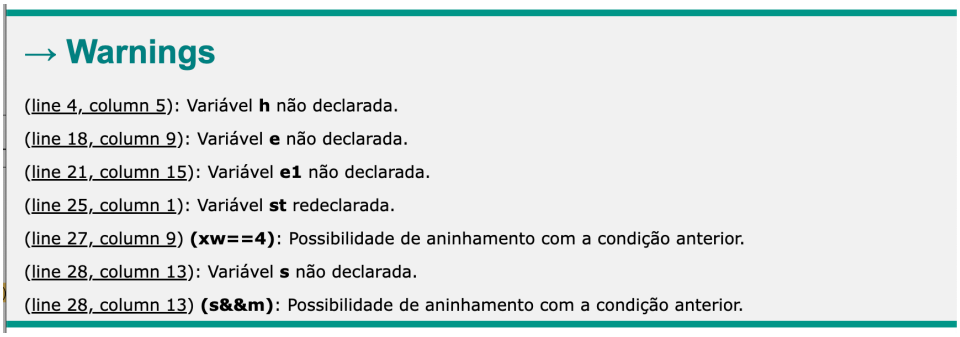


Figura 3.4: Warnings

3.3 Control Flow Graph

Control Flow Graph é uma representação gráfica da computação efetuada durante a execução do programa em causa. Trata-se de um grafo orientado que possui um nodo de entrada e um nodo de saída representando, respetivamente, o início e fim do programa.

Grande parte da gestão dos nodos e arestas foi efetuada no ficheiro *graph.py*.

De forma a poder ser feita a gestão dos nodos são utilizadas as seguintes variáveis de controlo:

```
self.graphControl = {'aninh': 0 , 'total': 0, 'inFor': False}
self.nodeAnt = "beginCode"
self.mccabe = {'nodes': 0 , 'edges': 0 }
```

O raciocínio geral prende-se com a seguinte ideia: criar uma aresta entre o nodo que está a ser visitado e o nodo anterior, guardado na variável 'nodeAnt'.

3.3.1 Variáveis, Input/Output

Para instruções que envolvam declaração e atribuição de valores a variáveis e instruções de input/output o raciocínio é muito semelhante:

- Invocação da função 'buildNodeDec()', 'buildNodeAtr()', def buildNodeIO(), em que o nodo atual é construído com base nos tokens que definem a regra em causa.
  Tal como já foi mencionado, para esse nodo construído é criada uma aresta com o anterior e o nodo em causa passa a ser considerado o 'nodeAnt'.

```
def buildNodeDec(self, dec, tokensList, g):
    edge = dec[0] + " " + dec[1] + " "
    for tok in tokensList:
        edge = edge + tok+ " "
    g.node(edge, fontcolor='blue', color='blue')
    self.mccabe['nodes'] +=1
    if self.nodeAnt != "":
        g.edge(self.nodeAnt, edge)
     if self.nodeAnt != 'beginCode':
```

```
            self.mccabe['edges'] +=1
        self.nodeAnt = edge
    return edge


    def buildNodeIO(self, tokens, g):
        edge = ''
        for tok in tokens:
        if tok != ";":
            edge = edge + tok
        g.node(edge, fontcolor='brown', color='brown')
        self.mccabe['nodes'] +=1
        if self.nodeAnt != "":
            g.edge(self.nodeAnt, edge)
            if self.nodeAnt != 'beginCode':
                self.mccabe['edges'] +=1
        self.nodeAnt = edge
    return edge
```

3.3.2 Instruções Condicionais

Em relação às instruções condicionais, existem dois nodos que demarcam o abrangência destas: 'if' e 'endif'. Por convenção, caso o 'if' possua duas arestas, a da esquerda representa o fluxo caso a condição se verifique, e a da direita, caso contrário.
Assim, o nodo, e a respetiva aresta, que dão início à representação deste fluxo, em particular, são gerados com a função 'buildNodeCond':

```
def buildNodeCond(self, nodeCond, g):
    edge = 'if ' + nodeCond
    g.node(edge, fontcolor='red', color='red', shape='diamond')
    self.mccabe['nodes'] +=1
    if self.nodeAnt != "":
        g.edge(self.nodeAnt, edge)
        if self.nodeAnt != 'beginCode':
            self.mccabe['edges'] +=1
        self.nodeAnt = edge
    return edge
```

No caso da existência de código do corpo da instrução, este é representado segundo a lógica do nodo anterior, explicada anteriormente.
Visualmente, o fim da visita a um if/else é representado pela função 'buildNodeCondEnd', que coloca como último nodo 'endif', com a agregação do contador da correspondente instrução.

```
def buildNodeCondEnd(self, g, counter):
    endif = 'endif'+ str(counter)
    counter += 1
```

```
        g.node(endif, fontcolor='red', color='red', shape='diamond')
        self.mccabe['nodes'] +=1
        if self.nodeAnt != "":
            g.edge(self.nodeAnt, endif)
            if self.nodeAnt != 'beginCode':
                self.mccabe['edges'] +=1
            self.nodeAnt = endif
        return endif
```

3.3.3 Instruções Cíclicas

For

Um ciclo for é constituído por 3 partes:

1. Inicialização da variável de controlo de ciclo;
2. Condição;
3. Alteração da variável envolvida na condição.

O fluxo de representação de um 'for' inicia-se com o nodo representativo da parte 1 identificada. Ao longo da visita à árvore, esse nodo é adicionado imediatamente através das declarações. Após se obter a string que representa a condição, é concatenada ao 'for' para representar o inicio do ciclo:

```
    def buildNodeCondFor(self, g, cond):
        edgefor = 'for '+ cond
        g.node(edgefor, fontcolor='purple', color='purple')
        self.mccabe['nodes'] +=1
        if self.nodeAnt != "":
            g.edge(self.nodeAnt, edgefor)
            if self.nodeAnt != 'beginCode':
                self.mccabe['edges'] +=1
            self.nodeAnt = edgefor
        return edgefor
```

Por sua vez, o nodo representativo da parte 3 é guardado para ser representado no fim do ciclo, criando uma aresta entre si e o 'for', nunca chegando a ser considerado o nodo anterior:

```
    g.edge(atr, edgefor)
```

Assim, no fim do ciclo, a variável 'nodeAnt' possui o valor do nodo do for, sendo possível continuar a representação do fluxo.

While

De forma muito semelhante ao ciclo for, para representar o while, recorremos à função 'buildNodeWhile':

```
def buildNodeWhile(self, cond, g):
    edgewhile = 'while '+ cond
    g.node(edgewhile, fontcolor='purple', color='purple')
    self.mccabe['nodes'] +=1
    if self.nodeAnt != "":
        g.edge(self.nodeAnt, edgewhile)
        if self.nodeAnt != 'beginCode':
            self.mccabe['edges'] +=1
        self.nodeAnt = edgewhile
    return edgewhile
```

Assim que termina a visita ao código no interior do ciclo, é adicionada a aresta entre a última instrução lida e o 'while':

```
g.edge(self.nodeAnt, nodeWhile)
```

Do-While

Reaproveitando o raciocínio utilizado para as outras instruções condicionais, utilizamos a função 'buildNodeWhileDo' para gerir os nodos principais: 'DO' e 'WHILE'.

```
def buildNodeWhileDo(self, text, g):
    g.node(text,fontcolor='purple', color='purple')
    self.mccabe['nodes'] +=1
    g.edge(self.nodeAnt, text)
    self.mccabe['edges'] +=1
    self.nodeAnt = text
    return text
```

Como forma de evidenciar a existência de um ciclo, no final é adicionada a aresta que liga o 'DO' e o 'WHILE':

```
g.edge(whiledo, node)
```

3.4 Complexidade de McCabe's

De forma a ser possível efetuar o cálculo da complexidade de McCabe's é necessário saber o número de vértices e arestas que o gráfico possui.
Assim, tal como foi possível visualizar nas funções apresentadas anteriormente, criamos a seguinte variável

```
self.mccabe = {'nodes': 0 , 'edges': 0 }
```

em que, a cada vez que um nodo ou aresta são adicionados, esta é incrementada .

No final, a complexidade do excerto é calculada com base na fórmula:

```
McCabe's complexity = E - V + 2
```

3.5 System Dependency Graph

System Dependency Graph é um grafo que modela um sistema (conjunto de funções) através da representação do fluxo e dependência de dados das funções que o constituem.
Para efetuar a gestão independente dos nodos e arestas foi criado o ficheiro *sdg.py*.
De forma geral, o nodo 'principal' é denominado de 'ENTRY', a partir do qual, as instruções não dependentes de fluxos/condições derivam.

3.5.1 Variáveis e Input/Output

Declarações, Atribuições, Input e Output são as instruções mais primordiais que podem ser encontradas no código-fonte e estas podem encontrar-se em 2 situações:

1. Inseridas em instruções cíclicas/condicionais
2. Independentes de quaisquer outras instruções

Esse tipo de controlo é efetuado considerando o nível de aninhamento atual de cada instrução. Caso o nível de aninhamento seja 0, é construído um nodo com 'ENTRY', caso contrário, com recurso a uma stack, a aresta é criada com entre o nodo e a última 'instrução mãe', da qual ele deriva:

```
def sdgDec(self, node, sdg):
    sdg.node(node, fontcolor='blue', color='blue')
    if self.inInst['atual'] == 0 or self.sdgControl['inFor']:
        sdg.edge('ENTRY', node)
    else:
        sdg.edge(self.sdgControl['instMae'][-1], node)
```

3.5.2 Instruções Condicionais

Para as instruções condicionais, a cada if procede um 'then', que evidencia o fluxo caso a condição se verifique. Assim, é invocada a função 'sdgIfs', que após perceber o nível de aninhamento em que se encontra, adiciona a aresta e adiciona à stack a instrução if (o respetivo 'then') atual, para conectar corretamente as instruções presentes no código do corpo da condição.

```
def sdgIfs(self, node, sdg, counter):
    sdg.node(node, fontcolor='red', color='red', shape='diamond')
    then = 'then' + str(counter)
    if self.inInst['atual'] == 1:
        sdg.edge('ENTRY', node)
    else:
```

```
            sdg.edge(self.sdgControl['instMae'][-1], node)
        sdg.edge(node, then)
        self.sdgControl['instMae'].append(then)
```

Caso existe um else, é adicionado um nodo 'else' a partir do qual se aplica a mesma lógica anterior:

```
    def sdgElse(self,beginIf, nodeElse, sdg, counter):
        sdg.node(nodeElse+str(counter))
        sdg.edge(beginIf, nodeElse+str(counter))
        self.sdgControl['instMae'].append(nodeElse+str(counter))
```

No fim, assim que acaba a análise desta produção, é efetuado o pop() do último elemento da stack, uma vez que as possíveis instruções derivantes já terminaram:

```
    self.sdgControl['instMae'].pop()
```

3.5.3 Intruções Cíclicas

For

Tal como já mencionado, o ciclo for é constituído por 3 partes. A instrução de inicialização da variável do ciclo é considerada como "independente"do 'for' e por isso é tratada como não derivante do ciclo mas sim da instrução mãe para o nível que se encontra. Dado o processamento desta, inserimos o nodo 'for', seguindo a lógica da stack criada.

```
    def sdgFor(self, edgefor, sdg):
        sdg.node(edgefor, fontcolor='purple', color='purple')
        if self.inInst['atual'] == 1:
            sdg.edge('ENTRY', edgefor)
        else:
            sdg.edge(self.sdgControl['instMae'][-1], edgefor)
        self.sdgControl['instMae'].append(edgefor)
```

Uma vez que o ciclo é demarcado pela aresta entre a atribuição que influencia a condição, cria-se uma aresta entre esses dois nodos em causa:

```
    def sdgForAtr(edgefor, atr, sdg):
        sdg.node(atr, fontcolor='purple', color='purple')
        sdg.edge(atr, edgefor)
```

O pop() é efetuado no final do processamento da árvore correspondente a esta produção.

While e Do-While

O processamento de ambos os whiles, neste tipo de grafo é o mesmo, e apenas difere do anterior na inserção de uma aresta para si próprio, evidenciando assim a existência de um ciclo:

```
def sdgWhile(self, node , sdg):
    sdg.node(node, fontcolor='purple', color='purple')
    sdg.edge(node, node)
    if self.inInst['atual'] == 0:
        sdg.edge('ENTRY', node)
    else:
        sdg.edge(self.sdgControl['instMae'][-1], node)
    self.sdgControl['instMae'].append(node)
```

Mais uma vez, o pop() é efetuado no final.

Capítulo 4

Testes

4.1 Testes realizados e Resultados

Como forma de o grafo se tornar de fácil interpretar cada tipo de instruções possui uma cor/forma diferente:

- Vermelho - Instruções condicionais
- Azul - Declarações
- Verde - Atribuições
- Roxo - Instruções cíclicas
- Castanho - Instruções input/output

A seguir apresentam-se alguns exemplos que abrangem todas as capacidades do programa.

Exemplo 1

Para o seguinte input, obtén-se os grafos apresentados a seguir:

```
int x =0;
for(int i = 0; i < 16; i = i+ 1){
    if(s){
        print("ola");
        int t [];
    };
};
int xw = 4;
 while(xw < 12){
     i = 0;
     x = 3 +3;
};
if(m){
    if(xw == 4){
        if(s && m ){
             dict d = dict();
             float f = 1.0;
             f = f + 2.7;
        }else{};
    };
};
```

Resultado:

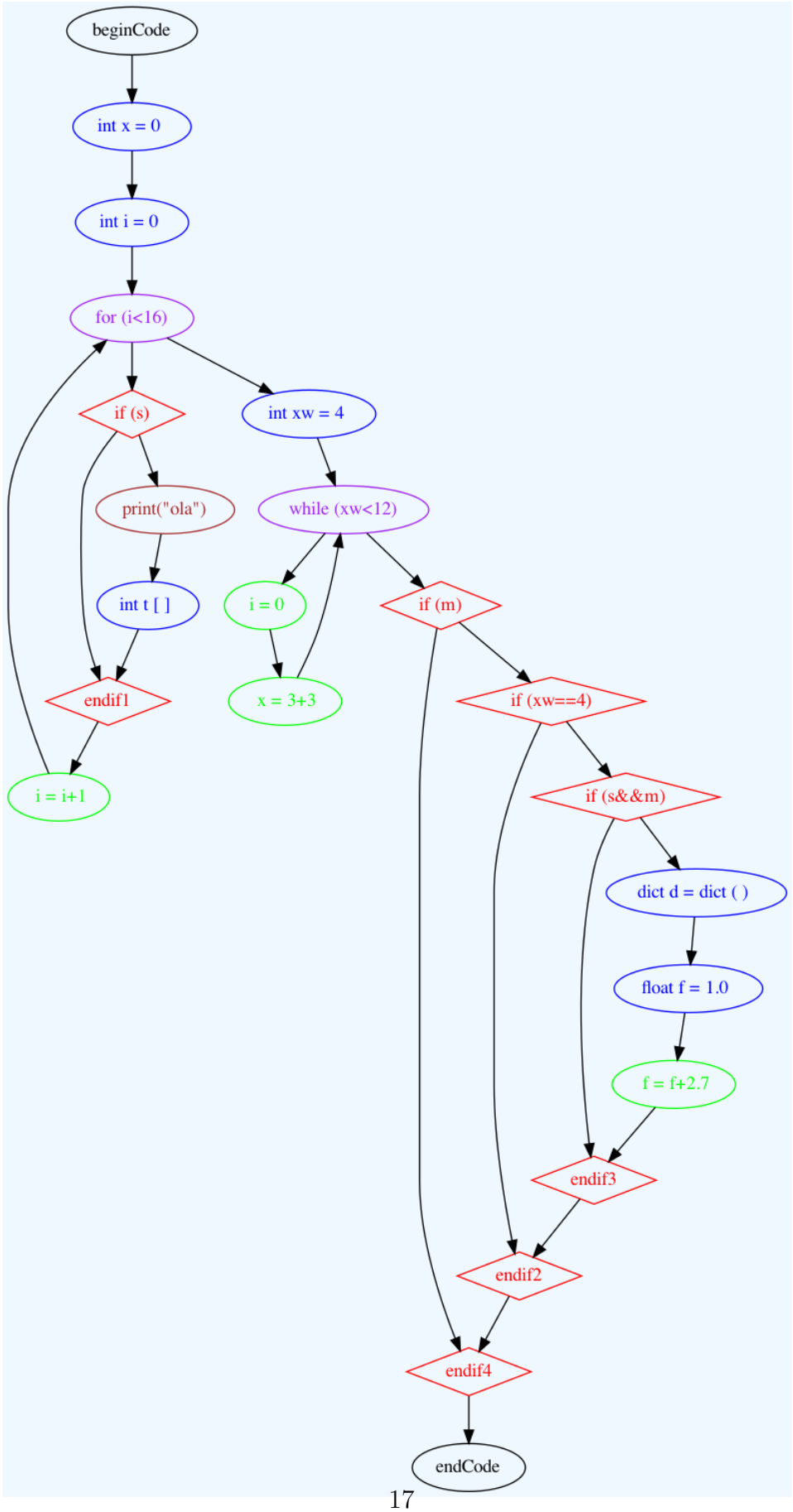


Figura 4.1: Grafo CFG obtido para o input do exemplo 1

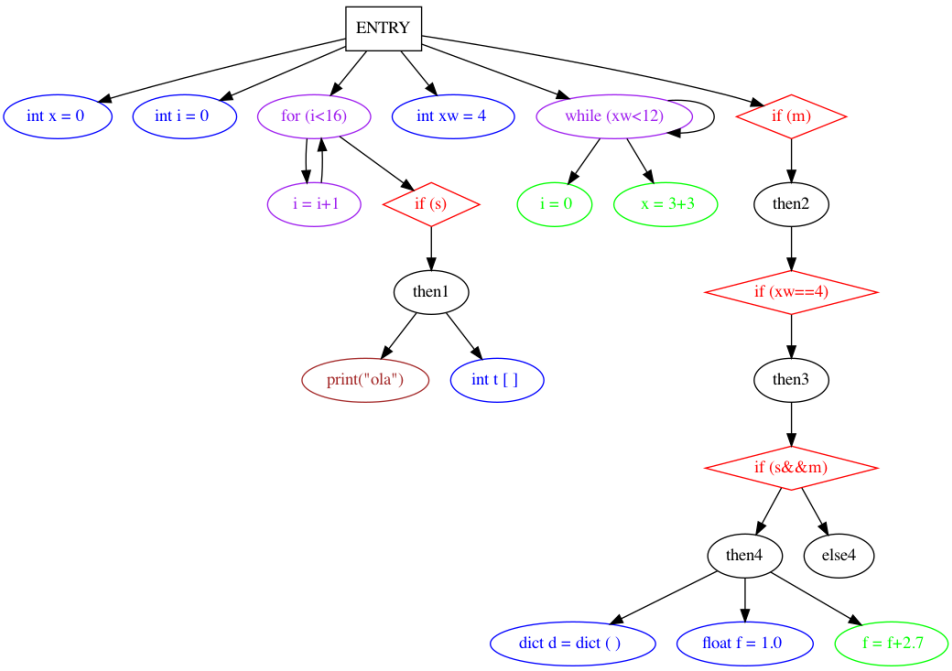


Figura 4.2: Grafo SDG obtido para o input do exemplo 1

Complexidade de McCabe´s é 6, considerando 22 vértices e 26 arestas.

Exemplo 2

Para o seguinte input, obtén-se os grafos apresentados a seguir:

```
int x =0;
do {
    string str;
    for (int y= 25; y != 16; y = y-1){
        print("Ainda no obtive o valor");
        if (str == "ola"){
            y = y -2 ;
        }else{
            y = y*2;
        };
    };

    str = input();

}while(x < 15);
```

Resultado:

Complexidade de McCabe´s é 3, considerando 14 vértices e 15 arestas.

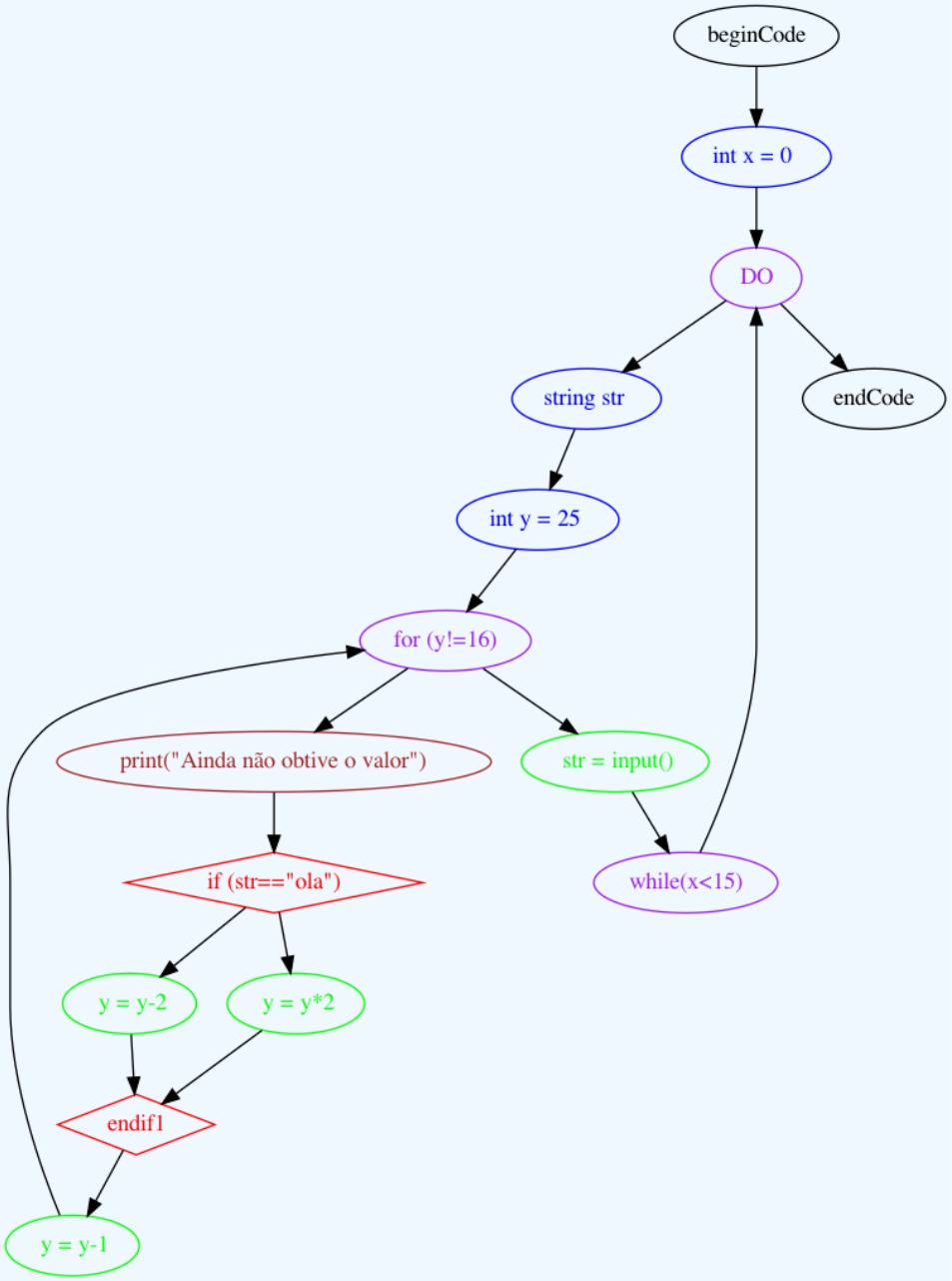


Figura 4.3: Grafo CFG obtido para o input do exemplo 2

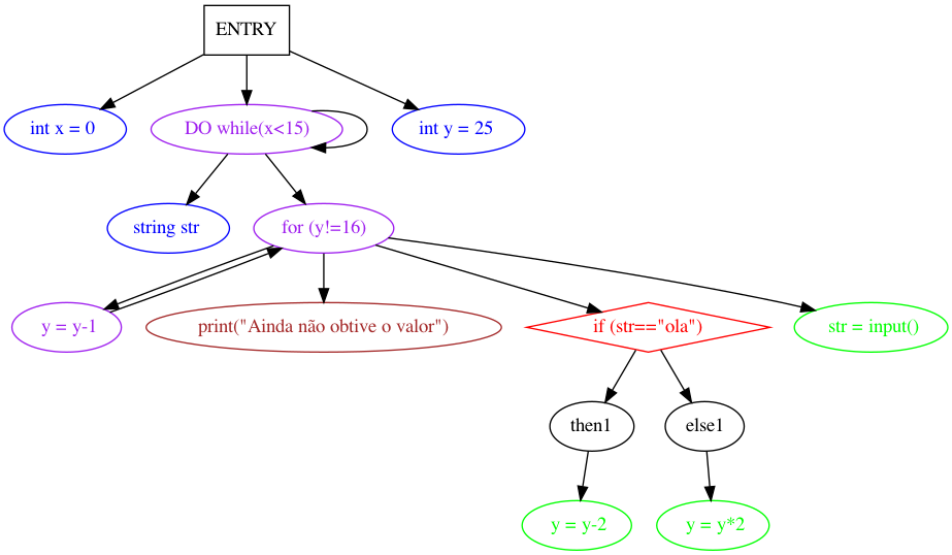


Figura 4.4: Grafo SDG obtido para o input do exemplo 2

Capítulo 5

Conclusão

Após a elaboração deste trabalho, podemos dizer que, como balanço geral, as maiores dificuldades surgiram no âmbito da gestão da ligação das arestas nas intruções condicionais devido à complexidade envolvida nos aninhamentos existentes.
Para além disso, tentamos implementar o controlo de dependências de dados no SDG, no entanto, por questão de gestão temporal, não nos foi possível completar essa tarefa, e por isso optamos por não colocar no trabalho. Na imagem seguinte segue um exemplo das tentativas de implementação, representando a tracejado as depedendências de dados.

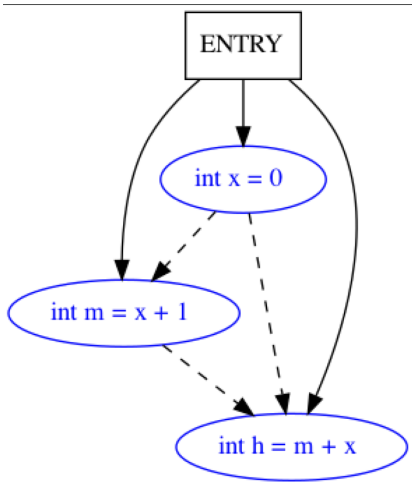


Figura 5.1: Exemplos de SDG com dependências de dados

Como trabalho futuro pretendemos terminar completa e correta implementação de um System Dependency Graph, e permitir que o programa forneça uma elaborada análise ao utilizador face ao código-fonte em questão.