Universidade Federal de São Carlos

Construção de Compiladores 2

Linguagem para manipulação de grafos

Manual da Linguagem

Discentes:

André Levi Zanon	RA: 619922
Rodrigo Ferrari de Souza	RA: 619795
Victor Hugo Domingos de Abreu	RA: 619841
Victor Santos Ferrari	RA: 552437

Docentes:

Helena Caseli
Daniel Lucrédio

Bacharelado em Ciência da Computação

São Carlos, 09 de dezembro 2017.

Sumário

1. Apresentação Geral

A teoria dos grafos faz parte de um ramo da matemática que estuda as relações entre os objetos de um determinado conjunto. Para isso, são utilizadas estruturas denominadas grafos, que contém um conjunto de vértices e arestas que podem representar várias aplicações. Em nosso dia a dia, existem diversas situações que podem ser representadas por grafos e muitos problemas de interesse prático podem ser formulados como questões sobre certos grafos

A linguagem para manipulação de grafos é um projeto de linguagem de programação desenvolvida com o objetivo de permitir a construção de grafos, bem como a aplicação de alguns algoritmos conhecidos nesses grafos construídos, podendo, por exemplo, se obter uma árvore de caminhos mínimos dado um grafo inicial. Este trabalho consiste na implementação de um compilador para a linguagem de manipulação de Grafos, capaz de realizar análise léxica, sintática, semântica da linguagem, e quando estes códigos escritos em linguagem estiverem corretos, realizar geração de código em linguagem *Python* e gerar um arquivo *.png* ilustrando o grafo final, ou exibir mensagens de erro, no caso de estar incorreto.

Esse projeto tem como objetivo servir de auxílio ao aprendizado de grafos, através desta linguagem é possível aplicar vários conceitos importantes da Teoria de Grafos, e alguns dos mais importantes algoritmos desta área da computação servindo de auxílio durante os estudos de Teoria dos Grafos, bem como, ajudar a rever alguns conceitos sobre a área.

2. Análise léxica/sintática

A análise léxica e sintática implementada para o compilador, seguindo as especificações da gramática proposta, trata os seguintes erros:

- Declaração de variáveis;
- Digitação de comandos e de seus respectivos parâmetros;
- Instanciação de comandos com retorno *void* em atribuição;
- Impossibilidade de utilização de números reais;
- Em for (para) somente é possível iterar sobre vetores
- Comentário não fechado;
- *Tokens* não permitidos;

3. Análise Semântica

A análise semântica implementada para o compilador, seguindo as especificações da gramática proposta, trata os seguintes erros:

- Variável e identificador não declarado;
- Variável e identificador declarado mais de uma vez;
- Compatibilidade de parâmetros com funções pré-definidas (ex: Busca em Largura, Busca em Profundidade e Dijkstra);
- Custo de aresta e vértice não pode ser negativo;
- Compatibilidade de tipos (grafo, lista, vértice, *int*) em atribuições;
- Variável *int* não inicializada;
- No tipo de dados vetor somente é possível adicionar o tipo de dados relativo ao primeiro, mesmo que depois tenha sido limpada a lista;
- Um grafo não pode ser desconexo;

4. Gramática

A figura abaixo ilustra a gramática final do compilador da Linguagem para manipulação de grafos.

```
grammar Grafos;

@members{...}

algoritmo : 'declaracoes' declaracao 'inicio' codigo 'fim' ;

declaracao : variavel declaracao | ;

variavel : tipo IDENT mais_variavel ;

mais_variavel : ',' IDENT mais_variavel | ;

tipo :  'grafo'     |
        'vertice'   |
        'int'       |
        'vetor'     ;

codigo : (instrucoes)* ;
                //cria aresta em um grafo, delimitado por dois veritices e um peso
instrucoes :    'aresta' '(' pa_grafo=IDENT ',' pa_vertice1=IDENT ',' pa_vertice2=IDENT ',' a_int=INTEIRO')'
                // remove de um grafo o vertice
                | 'remove_vert' '(' pr_grafo=IDENT ',' pr_vertice= IDENT ')'
                //define-se em um grafo, o custo para chegar no vertice
                | 'set_custo_para_vertice' '(' ps_grafo=IDENT ',' ps_vertice=IDENT ',' int_ou_ident ')'
                //imprime uma variavel ou cadeia
                | 'imprime' '(' print_grafo=IDENT ')'
                | 'listar' '(' var_ou_cadeia ')'
                | 'empilha' '(' pem_vetor=IDENT ',' int_ou_ident ')'
                | 'enfileira' '(' pen_vetor=IDENT ',' int_ou_ident ')'
                //lacos e comandos condicionais
                | 'se' expressao 'entao' codigo senao_opcional 'fim_se'
                | 'para' para=IDENT 'em' vetores_para 'faca' codigo 'fim_para'
                | 'enquanto' expressao 'faca' codigo 'fim_enquanto'
                //chamada de atribuicao
                | atribuicao=IDENT '<-' expressao
                //funcao dijktra pre-implementada, retorna um grafo
                | 'dijkstra' '(' pd_grafo=IDENT ',' pd_vertice=IDENT ')'
                //funcao prim pre-implementada, retorna um grafo
                | 'prim' '(' pp_grafo=IDENT ',' pp_vertice=IDENT ')'
                //funcao dfs pre-implementada, retorna um grafo
                | 'dfs' '(' pdfs_grafo=IDENT ',' pdfs_vertice=IDENT ')'
                //funcao bfs pre-implementada, retorna um grafo
                | 'bfs' '(' pbfs_grafo=IDENT ',' pbfs_vertice=IDENT ')'
                | instrucoes_com_retorno
                | instrucoes_de_vetores
                ;
int_ou_ident : INTEIRO | IDENT;

instrucoes_com_retorno :
                //retorna em um grafo o peso de uma aresta
                'get_peso' '(' pgp_grafo=IDENT ',' pgp_vertice1=IDENT ',' pgp_vertice2=IDENT ')'
                //retorna o custo para chegar em um vertice
                | 'get_custo_para_vertice' '(' pgc_grafo=IDENT ',' pgc_vertice=IDENT')'
                //retorna a quantidade de vertices em um grafo
                | 'qtde_vert' '(' pqv_grafo=IDENT ')' //grafo
                //retorna o vertice desempilhado
                | 'desempilha''(' pdem_vetor=IDENT ')' //vetor
                //retorna o vertice desenfileirado
                | 'desenfila' '(' pden_vetor=IDENT ')' //vetor
                //funcao que retorna o tamanhao de um vetor
                | 'tamanho_vetor' '('ptam_vetor=IDENT')' //vetor
                ;

instrucoes_de_vetores :
                //retorna um vetor de vizinhos a partir de um grafo e um vertice
                'vizinhos' '(' pv_grafo=IDENT ',' pv_vertice=IDENT ')'  //grafo,vertice
                //retorna um vetor de todos os vertices do grafo
                | 'vertices' '(' pver_grafo=IDENT ')' //grafo (retorna lista com vertices)
                ;
```

```
vetores_para : instrucoes_de_vetores | IDENT ;

var_ou_cadeia : CADEIA | IDENT ;

senao_opcional : 'senao' codigo | ;

expressao : exp_aritmetica op_opcional;

exp_aritmetica : termo outros_termos;

outros_termos : (op_adicao termo)*;

termo : fator outros_fatores;

outros_fatores: (op_multiplicacao fator)* ;

fator : parcela outras_parcelas;

outras_parcelas : ('%' parcela)*;

parcela : IDENT | INTEIRO | instrucoes_com_retorno | instrucoes_de_vetores |'(' expressao ')';

op_opcional : op_relacional exp_aritmetica | ;

op_relacional : '=' | '<>' | '>=' | '<=' | '>' | '<';

op_multiplicacao : '*' | '/';

op_adicao : '+' | '-';


/*Tokens: */
IDENT      : ('a'..'z' | 'A'..'Z' | '_') ('a'..'z' | 'A'..'Z' | '0'..'9' | '_')*;

INTEIRO    : ('+' | '-')? ('0'..'9')('0'..'9')* | 'INFINITO' | 'infinito';

CADEIA     : '"' ~('\n' | '\r' | '\'' | '\t')* '\'' | '"' ~('\n' | '\r' | '"' | '\t')* '"';

COMENTARIO : '{' ~('\n'|'}')* '}' {skip();};

WS         :    (' ' | '\t' | '\r' | '\n') {skip();};

/* Tokens errados: para tratarmos os mesmos foram setados tipos especificos(setType(int)) para cada um
 * assim para os tokens nao declarados foi setado o inteiro ERRO_TOKEN que possui por valor 10 e para os
 * os comentarios incompletos foi declarado ERRO_COMENT com o valor 11. Ambos os valores foram escolhidos
 * de forma parcial
 */
NAO_DECL   : ('@'| '|' | '!' | '"') {setType(ERRO_TOKEN);};

COMENTARIO_INCOMPLETO : '{' ~('\n'|'}')*'\n' {setType(ERRO_COMENT);};
```

Figura 4.1- Gramática Final.

4.1. Instruções

Neste tópico será dada uma breve descrição de cada instrução presente no compilador.

A instrução ***'aresta'*** (Figura 4.1.1) cria uma aresta com um peso definido ligando dois vértices do grafo. Parâmetros (grafo, vértice, vértice, int).

```
'aresta' '(' pa_grafo=IDENT ',' pa_vertice1=IDENT ',' pa_vertice2=IDENT ',' a_int=INTEIRO')'
```

Figura 4.1.1 – Instrução aresta.

A instrução ***'remove_vert'*** (Figura 4.1.2) remove um vértice do grafo. Parâmetros (grafo, vértice).

```
| 'remove_vert' '(' pr_grafo=IDENT ',' pr_vertice= IDENT ')'
```

Figura 4.1.2 – Instrução remove_vert.

A instrução ***'set_custo_para_vertice'*** (Figura 4.1.3) define o custo total para chegar em um determinado vértice do grafo. Parâmetros (grafo, vértice, vértice, int).

```
| 'set_custo_para_vertice' '(' ps_grafo=IDENT ',' ps_vertice=IDENT ',' int_ou_ident ')'
```

Figura 4.1.3 – Instrução set_custo_para_vertice.

A instrução ***'imprime'*** (Figura 4.1.4) imprime uma variável ou uma cadeia. Parâmetros (grafo).

```
| 'imprime' '(' print_grafo=IDENT ')'
```

Figura 4.1.4 – Instrução imprime.

A instrução ***'listar'*** (Figura 4.1.5) imprime os elementos de um vetor. Parâmetros (cadeia).

```
'listar' '(' var_ou_cadeia ')'
```

Figura 4.1.5 – Instrução listar.

A instrução ***'empilha'*** (Figura 4.1.6) adiciona um elemento que pode ser um vértice, grafo ou inteiro no topo de um vetor. Parâmetros (vetor, elemento).

```
'empilha' '(' pem_vetor=IDENT ',' int_ou_ident ')'
```

Figura 4.1.6 – Instrução empilha.

A instrução ***'enfileira'*** (Figura 4.1.7) adiciona um elemento que pode ser um vértice, grafo ou inteiro em um vetor. Parâmetros (vetor, elemento).

```
'enfileira' '(' pen_vetor=IDENT ',' int_ou_ident ')'
```

Figura 4.1.7 – Instrução enfileira.

Instrução condicional ***'se' 'então'*** (Figura 4.1.8).

```
'se' expressao 'entao' codigo senao_opcional 'fim_se'
```

Figura 4.1.8 – Instrução se entao.

Instrução de repetição ***'para'*** (Figura 4.1.9).

```
'para' para=IDENT 'em' vetores_para 'faca' codigo 'fim_para'
```

Figura 4.1.9 – Instrução para.

Instrução de repetição ***'enquanto'*** (Figura 4.1.10).

```
'enquanto' expressao 'faca' codigo 'fim_enquanto'
```

Figura 4.1.10 – Instrução enquanto.

A instrução ***'dijkstra'*** (Figura 4.1.11) executa o algoritmo Dijkstra pré-implementado, retornando o grafo resultante da execução. Parâmetros (grafo, vértice inicial).

```
| 'dijkstra' '(' pd_grafo=IDENT ',' pd_vertice=IDENT ')'
```

Figura 4.1.11 – Instrução Dijkstra.

A instrução ***'prim'*** (Figura 4.1.12) executa o algoritmo Prim pré-implementado, retornando o grafo resultante da execução. Parâmetros (grafo, vértice inicial).

```
'prim' '(' pp_grafo=IDENT ',' pp_vertice=IDENT ')'
```

Figura 4.1.12 – Instrução Prim.

A instrução ***'dfs'*** (Figura 4.1.13) executa o algoritmo Busca em Largura pré-implementado, retornando o grafo resultante da execução. Parâmetros (grafo, vértice inicial).

```
'dfs' '(' pdfs_grafo=IDENT ',' pdfs_vertice=IDENT ')'
```

Figura 4.1.13 – Instrução DFS.

A instrução ***'bfs'*** (Figura 4.1.14) executa o algoritmo Busca em Profundidade pré-implementado, retornando o grafo resultante da execução. Parâmetros (grafo, vértice inicial).

```
'bfs' '(' pbfs_grafo=IDENT ',' pbfs_vertice=IDENT ')'
```

Figura 4.1.14 – Instrução BFS.

A instrução ***'get_peso'*** (Figura 4.1.15) retorna o peso de uma aresta que liga dois vértices do grafo. Parâmetros (grafo, vértice, vértice).

```
'get_peso' '(' pgp_grafo=IDENT ',' pgp_vertice1=IDENT ',' pgp_vertice2=IDENT ')'
```

Figura 4.1.15 – Instrução get_peso.

A instrução ***'get_custo_para_vertice'*** (Figura 4.1.16) retorna o custo total para chegar em um determinado vértice do grafo. Parâmetros (grafo, vértice).

```
'get_custo_para_vertice' '(' pgc_grafo=IDENT ',' pgc_vertice=IDENT')'
```

Figura 4.1.16 – Instrução get_custo_para_vertice.

A instrução ***'qtde_vert'*** (Figura 4.1.17) retorna a quantidade de vértices presentes no grafo. Parâmetros (grafo).

```
| 'qtde_vert' '(' pqv_grafo=IDENT ')'
```

Figura 4.1.17 – Instrução qtde_vert.

A instrução ***'desempilha'*** (Figura 4.1.18) retorna o vértice no topo do vetor. Parâmetros (vetor).

```
'desempilha''(' pdem_vetor=IDENT ')'
```

Figura 4.1.18 – Instrução desempilha.

A instrução ***'desenfila'*** (Figura 4.1.19) retorna o primeiro vértice do vetor. Parâmetros (vetor).

```
'desenfila' '(' pden_vetor=IDENT ')'
```

Figura 4.1.19 – Instrução desenfila.

A instrução ***'tamanho_vetor'*** (Figura 4.1.20) retorna o número de elementos presentes no vetor. Parâmetros (vetor).

```
'tamanho_vetor' '('ptam_vetor=IDENT')'
```

Figura 4.1.20 – Instrução tamanho_vetor.

A instrução ***'vizinhos'*** (Figura 4.1.21) retorna um vetor contendo todos os vértices que possuem uma aresta ligando-os a um outro vértice passado como parâmetro. Parâmetros (grafo, vértice).

```
'vizinhos' '(' pv_grafo=IDENT ',' pv_vertice=IDENT ')'
```

Figura 4.1.21 – Instrução vizinhos.

A instrução ***'vertices'*** (Figura 4.1.22) retorna um vetor contendo todos os vértices de um grafo. Parâmetros (grafo).

```
'vertices' '(' pver_grafo=IDENT ')'
```

Figura 4.1.22 – Instrução vertices.

5. Instruções para compilação

- Para compilar, é necessário clicar em Build -> Build Artifacts -> Rebuild no IntelliJ;
- Caso o contribuidor tenha feito modificações no Grafos.g4 usar o seguinte comando dentro da pasta src do projeto: java -jar antlr-4.7-complete.jar -package trabalho3 -visitor Grafos.g4;
- Para rodar o compilador de grafos no Windows, basta acessar a pasta principal do projeto e executar o seguinte comando: java -jar compilador.jar <Caminho do arquivo em linguagem de manipulação de grafos.txt> <Caminho destino da linguagem gerada em python>;
- Então, executar o arquivo .py em uma IDE para python, como a IDE Spyder.
- Em caso de utilização do compilador no Linux é preciso importar as bibliotecas necessárias ao projeto. Para importar as bibliotecas os seguintes comandos são necessários:
    - sudo apt-get install python-pip
    - sudo pip install numpy
    - sudo pip install networkx
    - sudo apt-get install python-matplotlib.
- Após ter as bibliotecas instaladas, executar o seguinte comando: java -jar compilador.jar <Caminho do arquivo em linguagem de manipulação de grafos.txt> <Caminho destino da linguagem gerada em python>
- Então, acessar pelo terminal a pasta em que foi criada o arquivo .py e executar o seguinte comando: python <Nome do arquivo criado.py>

6. Testes

Neste tópico serão apresentados os testes realizados para avaliar a análise sintática, análise semântica e geração de código.

6.1. Sintático

- Tipo não identificado.

Entrada:

```
declaracoes:
    grafo g
    vertice v1, v2, v3
    int valor

inicio

    empilha(1, v1)

    aresta(g, v1, v2, 1)
    aresta(g, v2, v3, 1)
    aresta(g, v1, v3, 1)
fim
```

Saída:

```
Linha 8: erro sintatico proximo a 1
Fim da compilacao
```

6.2. Semântico

- Repetição de declaração de variável;
- Variável e identificador não declarado;
- Custo de aresta e vértice não pode ser negativo;

Entrada:

```
declaracoes:
    grafo g
    vertice v1, v2, v3,
        g

inicio

    valor <- valor + 1
    empilha(v4, 1)
    aresta(g, v1, v2, -1)
    aresta(g, v2, v3, 1)
    aresta(g, v1, v3, 1)

    set_custo_para_vertice(g, v1, -1)
fim
```

Saída:

```
Linha 4: identificador g ja declarado anteriormente
Linha 8: identificador valor nao declarado
Linha 8: identificador valor nao declarado
Linha 9: identificador v4 nao declarado
Linha 10: peso nao pode ser negativo do identificador -1
Linha 14: peso nao pode ser negativo do identificador -1
Fim da compilacao
```

6.3. Geração de Código

- Dijkstra

Entrada.

```
declaracoes:
grafo g1

vertice v0
vertice v1
vertice v2
vertice v3
vertice v4
vertice v5

inicio
    aresta(g1, v0, v1, 1)
    aresta(g1, v0, v2, 1)
    aresta(g1, v0, v5, 6)
    aresta(g1, v1, v2, 1)
    aresta(g1, v2, v3, 4)
    aresta(g1, v2, v4, 1)
    aresta(g1, v3, v4, 2)
    aresta(g1, v4, v5, 1)
    aresta(g1, v3, v5, 1)

    dijkstra(g1,v5)

fim
```

Saída:

```
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
gl = nx.Graph()
gl.add_edge('v0', 'v1', weight = 1)
gl.add_edge('v0', 'v2', weight = 1)
gl.add_edge('v0', 'v5', weight = 6)
gl.add_edge('v1', 'v2', weight = 1)
gl.add_edge('v2', 'v3', weight = 4)
gl.add_edge('v2', 'v4', weight = 1)
gl.add_edge('v3', 'v4', weight = 2)
gl.add_edge('v4', 'v5', weight = 1)
gl.add_edge('v3', 'v5', weight = 1)

CopiaGrcopy0 = nx.Graph()
CopiaGrcopy0 = gl

def Dijkstra(G, CopiaG, Rl):
    for i in G.nodes():
        G.node[i]['peso'] = np.inf
    G.node[Rl]['peso'] = 0
    Predecessor = {}

    for i in G.nodes():
        Predecessor[i] = None

    while G.number_of_nodes() > 0:
        pesoMinimo = np.inf

        for i in G.nodes():
            if G.node[i]['peso'] <= pesoMinimo:
                verticePesoMinimo = i
                pesoMinimo = G.node[i]['peso']

        for i in G.neighbors(verticePesoMinimo):
            if G.node[i]['peso']>G.get_edge_data(verticePesoMinimo, i)['weight']+G.node[verticePesoMinimo]['peso']:
                G.node[i]['peso'] =G.get_edge_data(verticePesoMinimo, i)['weight']+G.node[verticePesoMinimo]['peso']
                Predecessor[i] = verticePesoMinimo
        G.remove_node(verticePesoMinimo)

    for key in Predecessor:
        print("predecessor de ", key, "e = ", Predecessor[key])

    vertice = []
    arestas = []

    for key in Predecessor:
        vertice.append(key)

    for key in Predecessor:
        arestas.append((key, Predecessor[key]))

    vertice = []
    arestas = []

    for key in Predecessor:
        vertice.append(key)

    for key in Predecessor:
        arestas.append((key, Predecessor[key]))

    newGdij = nx.Graph()
    newGdij.add_nodes_from(vertice)
    newGdij.add_edges_from(arestas)
    pos = nx.spring_layout(newGdij, k = 1, iterations=30)
    nx.draw_networkx(newGdij, pos)
    plt.show()

Dijkstra(gl, CopiaGrcopy0, 'v5')

#Fim da compilacao
```

```
predecessor de  v0 e =  v2
predecessor de  v1 e =  v2
predecessor de  v2 e =  v4
predecessor de  v5 e =  None
predecessor de  v3 e =  v5
predecessor de  v4 e =  v5
```