CS 315

Programming Languages

GRAFI315

Tutorial

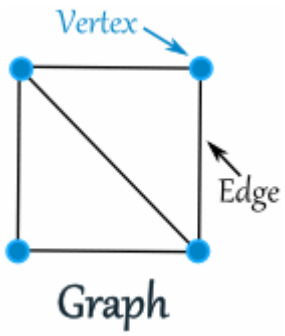
Group
Doğukan Altay
Ahmet Emre Nas
Berk Erzin

Contents

İçindekiler Tablosu

Introduction:

GRAFI315 is a graph language for constructing directed or undirected graphes. Also queries can be made with these constructed graphs. There are three types of primitive type (string, int and float) In the graphs vertex can have multiple properties.



Example Code:

```
directedGraph graph1{

Vertex v0;

V0->name = "Emre";

Vertex v1;

Edge e = v0 :>v1;

}
```

/*This code creates a graph with two vertex and one edge to connect them with direction of that goes from v0 to v1 where v0 has property of name.*/

Comments

GRAFI315 supports one-line or multiple-line comments whether it is nested or not. One-line comments starts with // and make all the line comment where in multiple-line comments the comments starts with /* and end with */. It's not important if the multiple-line comments include another comments inside of it.

Example Usage:

```
//This is one line comment

/*In GRAFI315 users can use

Multiple line of comments /*even if the comments include another

Comment */ inside of it*/
```

Types For Variables

Since GRAFI315 is staticly typed language users should declare the types of variables. There are three types of variable in GRAFI315 which are

+ Integer (int)
+ Float (float)
+ String (String)
+ Boolean (boolean)

Rules:

1. All the primitives type are case-sensitive so int and Int are not same.
2. Variable names should start with alphebetic characters or underscore(_).
3. Variable names can have numerical values if it's not at the begining of the name
4. String variables should be declerad in quatation marks (" ").
5. Float variables should use "." insted of ", "
6. Boolean types can be true or false.

Examples:

```
int number = 0;

float point = 3.5;

String name = "Emre";

boolean checked = true;
```

Operations and Usage of Primitive Types

In GRAFI315 all 4 aritmatic calculations can be used for int and float variables. For String however, only addition operator can be used. In one line there can be more than one operation.

- Addition: +
- Subtraction: -
- Multiplication: *
- Division: /

On the above the symbols of each opereation have been showed.

Examples:

```
int a = 10;

int b = 5;

int result;

String str1 = "The string is now ";

String str2 = "completed";

String completed;



result = a+b; //This will set result 15

result = a-b; //This will set result 5

result = a*b; //This will set result 50

result = a/b; //This will set result 2
```

```
result = a*b/a; //This will set result 5

Completed = str1+ str2; //This will set completed "The string is now completed"

/* Floats also used by same as integers*/
```

Graphs

In GRAFI315 there are two types of graphs. Directed and Undirected. These graph are created by the keywords “undirectedGraph” and “directedGraph”. After these keywords a graph name should come after it and a curly bracket should be open. Inside the curly brackets edges and vertex can be initliazed and be placed as desired to graph. Vertices and edges can be can created by using the keyword "vertex" and "edge".

Example:

```
directedGraph a{
        Vertex v0, v1, v2, v3;
        Edge e = v0:>;v1; // v0 and v1 are vertexes and the edge e goes from v0 to v1
        Edge b = v2:>v3;
}
```

```
undirectedGraph a{
        vertex v0, v1, v2, v3;
        edge e = v0::v1; // v0 and v1 are vertexes and the edge e goes double way
        edge b = v2::v3;
}
```

As can be seen from above in directedGraphes edges should be connected each other with symbols of ":>" where the left of the symbol is the start point of the edge and right side is the end point of the edge.

However in undirectedGraphes edges are connected each other with "::" since the start and end points are unimportant.

Properties

Propery is a value pair that can be assigned to a certain edge or vertex in GRAFI315. In GRAFI315 vertexes and edges can have multiple properties. These properties can be any primitive types (Integer, Float or String). However, using strings as property names is mandatory.

Vertex Properties

As mentioned above vertexes can have multiple properties and it does not matter what type of these properties. Vertex properties can be defined in graphes.

Example Code:

```
directedGraph graph1{

Vertex v0;

v0->”name” = "Emre"; //Name property of v0 is a string

v0->”number” = 123; //Number property of v0 is int

Vertex v1; //v1 does not have any properties

Edge e = v0 :>v1;

}
```

Edge Properties

Edges can have multiple properties just like vertexes with any primitive types that GRAFI315 supports. Syntex for edge properties is same with the vertex properties syntex.

Example Code:

```
directedGraph graph1{
Vertex v0;
v0->”name” = "Emre"; //Name property of v0 is a string
v0->”number” = 123; //Number property of v0 is int
Vertex v1; //v1 does not have any properties
Edge e = v0 :>v1;
e-> “order” = 1; //e has property of order which is integer
e->”name” = "CS315"; //e has property of name which is string
 }
undirectedGraph graph2{
Vertex v0, v1, v2, v3;
Edge e = v0::v1; // v0 and v1 are vertexes and the edge e goes double way
   // vertex initiliazing will be shown in next chapter
e-> “type” = “sports”;
}
```

Defining Properties as Map

In GRAFI315 vertex or edges properites can be set by maps. There is no keyword for a map. Defining a property with a maps is simple just follow the same rules for initiliazing properties and instead of writing the primitive types write the map code. Here is a example for this.

Example Code:

```
directedGraph mapProperty{

vertex v0, v1, v2;

edge e = v0:&gt;v1;

edge b = v1:&gt;v2;

v0 -> “scores” = { “ali”: 12, “veli”: 10, “haydar”: 11, “abdullah”: 3 } /* defining maps as a property*/

v1-> “courses” ={“ali”: <315, 319>, “veli”: <223, 202, 301> }; /*defining nested collections. Map keys assigned to sets of integers.*/

}
```

Defining Properties as List

In GRAFI315 using lists and its functions as a property is supported and vertex or edges properties can be initiliaze by putting the data inside of the brackets “[ ]”.

“property”.add(int index, value) used for adding a certain data to a certain location of the list.

“property”.remove(int index) removes the choosen index value from list and shifts the list to left.

“property”.get(int index) returns the choosen index value.

```
directedGraph listProperty{
vertex v0, v1, v2;
edge e = v0:>;v1;
edge b = v1:>v2;
v0 ->“scores” = [12, 10, 11,3 ]; // initializing a list property
v0 -> “scores”.add(0, 8); // adds “8” to the 0th index of the list, [8,12,10,11,3].
v0 -> “scores”.remove(4); // removes 4th index element from the list, [8,12,10,11].
int temp = v0 -> “scores”.get(4); /* assigns the 4th index element of the list to the temp variable. (Both should be the same primitive type in order to assign)/*
}
```

Defining Properties as Set

In GRAFI315 vertex or edges properties can be initiliaze similiar way with list or map. In set every element has unique values. The initiliazing is similar to list just insted of "[]" use "<>" . The same functions that mentioned in lists also apply to the sets.

```
vertex v0, v1, v2;

edge e = v0:>v1;

edge b = v1:>v2;

v0 ->"scores" = <12, 10, 11,3 >; // initializing a setproperty

v0 -> "scores".add(0, 8); // adds "8" to the 0th index of the set, <8,12,10,11,3>.

v0 -> "scores".remove(4); // removes 4th index element from the set, <8,12,10,11>.

int temp = v0 -> "scores".get(4); /* assigns the 4th index element of the set to the temp variable. (Both should be the same primitive type in order to assign)/*

}
```

Functions

In the Grafi315 there are some built-in functions in order to make coding easier for the programmer.

abs(); // returns the corresponding interger/float value’s absolute value.

roundUp(float); // returns the rounded up value of the corresponding float.

roundDown(float); // returns the rounded down value of the corresponding float.

hasProp(String); // returns boolean whether the edge/vertex has the corresponding property.

Quaries

The GRAFI315 supports querying. Users can query on GRAFI315 by using the “Query” keyword and it is for creating a query expression. After creating a query user can assign it by writing “ graphName.queryName;”.

In query expressions, user need to follow a syntax for both edges and vertices. For example:

```
Query path1 = { vertex-> ”name” == ”dogukan”};

Query path2 = { edge-> “job” == ”programmer”};
```

In GRAFI315, quaries support “^” as concatination, “|” as alternation and “*” as repetition. The precedence of the operators are “^”, “|”, “*” respectively from high to low. Also users can use paratheses the specift the precedence order of the expressions. As an example, the following code segment is defining a query expression that queries all the paths that have a starting vertex property (“name”, “emre”) and an edge which has a property (“relation”, “family”) or (“relation”, “friend”).

```
Query query1 = {vertex->”name” == “emre” ^ (edge->”relation”==”family” | edge->
“relation”==”friend”};
```

In the part of quary initiliazing users can use built-in functions, any modular aritmatic as boolean expressions. For example, the corresponding code segments are query expressions for querying all the paths which that the following statements are true.

```
Query query5 = {vertex->”name” == “S1” ^ edge->”name”==”emre” ^ !vertex-
>”number”==21}; /* query search for a path that has vertex name “s1”
edge name “emre” and any vertex with number different than 21.*/
Query query6 = {vertex->hasProp(“name”) ^ edge->”name”==”emre” ^
!(vertex- >”number”==21)}; /* query search for a path that has vertex property
“names” edge name “emre” and any vertex with number different than 21.*/
```

The GRAFI315 supports using variables in the query expressions. To use the variables the user do not need to follow a certain syntax. For example, in the following code segment queries all the paths that has the same amount of "apples" and "oranges" in the vertex properties. The "VALUE" has not defined in the query expression and yet it becomes a variable for this expression.

```
Query query1 = {vertex->"apples" == VALUE ^ "vertex-> "oranges"==VALUE};
```

The language supports modularity to define query expressions. As an example,

```
Query query1 = {vertex->"name" == "S1"};

Query query2 = { edge-> "path" = 1 };

Query query3 = { edge->"type"=="highway"};

Query query4 = {vertex->"id"==5+7};

Query query_final = query1^(query2| query3)^ query4;

graph1.query_final;
```