BİLKENT UNIVERSITY

ENGINEERING FACULTY

DEPARTMENT OF COMPUTER ENGINEERING

CS 315 Project – Part 1

GEEZ : A Graph Description-Query Language

TUTORIAL

Muhammed Duha Çelik - 21400794

Selin Fildiş - 21402228

Başak Melis Öcal - 21400668

Group 5

Table of Contents

1. OVERVIEW

Geez is a simple graph defining and querying language which is designed to deal with various graphs. The syntax of Geez is kept quite simple aiming to provide a user-friendly experience to the user. Geez supports extensibility feature by allowing users to define functions, create complex queries and benefit from many of the primitive/collection types. Furthermore, modularity is supported by complex queries created from the simple ones.

Throughout this tutorial, the overall structure of Geez will be explained via example code segments intending to provide a starting point to the user. The core features such as types it supports, syntax, graph defining, querying and filtering are some of the topics to be discussed. At the end of the tutorial, hopefully, users will be able to define and query graphs easily.

2. A SIMPLE PROGRAM

```
UndirGraph color{
   Vertex red;
   Vertex green;
   Vertex blue;
   Edge yellow;
   Edge purple;
   Edge cyan;
   connect(red,green,yellow);
   connect(green,blue,cyan);
   connect(blue,red,purple);
   blue.includeProperty("status", "my favoruite");
   red.includeProperty("lovers",3);
   cyan.includeProperty("status", "don't know");
   green.includeProperty("status", "spring");
}

Predicate p1 = #equals("status", "myFavourite");
Predicate p2 = $equals("status", "don't know");
Predicate p3 = #equals("status", "spring");
Predicate p4 = #moreThan("lovers",2);
```

```
Filter f1 = p1 & $P;
Filter f2 = p4|p3;
Query q1 = f1&f2;
exec(color, q1);
//Returns blue purple red.
```

3. COMMENTS

Since it's required for users to write easily easy-to-read code, Geez supports single line commenting feature. The characters written after "//" will be ignored by the compiler for only that line. As only single line commenting is supported, users should be careful about adding slashes to each line when they want to comment out multiple lines consecutively.

```
//Anything written to that line can be ignored.
//Don't forget to use double slashes while commenting multiple
//lines!
```

4. TYPES AND VARIABLES

- ➢ Geez is designed as a dynamically typed language. In other words, type checking is performed at run-time rather than compile-time.
- ➢ Two kind of types are supported which are primitive and collection types.

4.1 Primitive Types

Geez supports four primitive types as follows:

- **int:** Represents integers which are composed of one or more digits. 10, 2226839 are integers, whereas 89.32 isn't a one.
- **float:** Represents floating point numbers which are integers and decimal numbers.
- **boolean:** Represents boolean values which can be either true or false.
- **string:** Represents string which are defined as any combination of characters in a double quotation mark. "Hodor!" is a string whereas 'Hodor!' isn't a one.

```
number = 100;                  //integer
numberf = 100.678;             //float
flag = true;                   //boolean
str = "You know nothing!"      //string
```

4.2 Collection Types

Geez supports three collection types as follows:

- **List:** Lists can be defined by using square brackets, “[]”. List can be used for make a collection from the same types or different types.

```
//List of strings and floats.
list1 = ["Duha","Selin","Melis"];
list2 = [48.2,56.5,21.8];
```

- **Map:** Maps are defined in a similar way in Python. Curly brackets, “{}” can be used to define maps. Each entry in the map consists of a key-value pair. The key must be a string and the value can be any type.

```
//Maps of strings and ints.
map1 = {“Harvey”: “Senior partner”, “Donna”: “Legal
secretary”, “Mike”: “Freud”};
map2 = {“Bugra”: 315, “Ugur”: 319, “Billur”: 391};
```

- **Set:** Sets are defined by using square brackets, “[]”.

```
//Sets of strings and integers.
set1 = [“See the money”, “wanna stay”, “for your meal”];
set2 = [1,2,3,4,5];
```

✓ Arbitrary nestings of them are possible.

```
//Map from a string to a list of integers.
map2 = {“Bugra”: [315,442], “Ugur”: [202,319], “Billur”:
[391,300]};
```

4.3 Variables

➢ Since Geez is a dynamically typed language, user doesn’t need to specify the type of the variable. Type of the variable will be determined at run-time.
➢ There is no limitation for the length of the variable names.
➢ Variable names are case sensitive.

```
//Variables are different from each other.
Pancake = 5;
pancake = 5;
```

➢ A variable name can’t start with a number. “_” (underscore char.) and alphanumeric

numbers are allowed to use. However, any other kind of special characters aren't accepted in variable names.

5. BASIC OPERATIONS

- **Arithmetic Operations**

  Basic arithmetic operations such as addition, subtraction, multiplication and division on integers and floats are supported. Arithmetic operations can be used on floats and integers by using the specified symbols.

```
int1 = 25;
int2 = 5;
f1 = 89.5;
f2 = 59.2;
result1 = int1 + int2;          //result1 = 25 + 5 = 30
result2 = third - forth;        //result2 = 89.5 - 59.2 = 30.3
result3 = int1 * int2;          //result3 = 25 * 5 = 125;
result4 = int1 :: int2;         //result4 = 25 / 5 = 5;
```

- **Concatenation**

  Geez supports the string concatenation by the symbol "~", similar to any of the languages. Strings or string with a float or integer can be also concatenated. Concatenation with a different primitive type rather than string requires the type casting of other primitive type to string.

```
str1 = "The Beatles: ";
str2 = "When I'm ";
int1 = 6;
int2 = 4;
str3 = str1 ~ str2 ~ int1 ~ int2;
//str3 = "The Beatles: When I'm 64"
//int1 and int2 are type casted to string.
```

- **Logical Operations**

  Logical NOT, logical AND, logical OR operations applied on boolean values using the specified operations are supported by Geez. The following code segment illustrates the use of logical operations:

```
myFlag = true;
yourFlag = false;
flag1 = !myFlag;                    //Logical NOT
flag2 = myFlag && yourFlag;         //Logical AND
```

```
flag3 = flag1 || flag2;              //Logical OR
```

Operation	Name	Operator	Types Supported
Arithmetic	Multiplication	*	integer, float
	Division	/	integer, float
	Addition	+	integer, float
	Subtraction	-	integer, float
String	Concatenation	~	string
Logical	AND	&&	boolean
	OR	||	boolean
	NOT	!	boolean

Table 1. Basic operations supported by Geez

6. DEFINING VERTICES AND EDGES

6.1 Property

A property is a pair such as (“name”, “Geez”) or (“groupNo”, 5) which can be assigned to both vertices and edges. The first field in the pair is the name of the property whose type must be string. The second field can be any kind of primitive or collection type such as float, string, map, set etc. Properties are defined by the keyword “Property” followed by its name. Vertices and edges can have multiple properties.

```
Property p1 = (“Status”, “Unavailable”);
Property p2 = (“Density”, 53,6);
Property p3 = (“Dessert”, {“iceCream”: [5,10], “cheesecake”:
[10,12], “cookie”: [4,8]});
// Map from a string to a list of integers as a property.
```

6.2 Vertex

Vertices are building blocks of the graphs which are defined with the keyword “Vertex”. The definition is completed with the name of the vertex after the keyword. In order to include a property to a vertex, nameOfVertex.includeProperty(name, value) or nameOfVertex.includeProperty(property) methods will be called.

6.3 Edge

Edges connect the vertices of the directed and undirected graphs. Each edge is defined with the keyword “Edge” followed by its specific name. Multiple properties can be

assigned to an edge with the use of nameOfEdge.includeProperty(name, value) or nameOfVertex.includeProperty(property) methods.

```
//The following code segment illustrates the use of properties.
//Graph definition isn't added for the simplicity.
Vertex v1;
Vertex v2;
Vertex v3;
Edge e1;
Edge e2;
Property p1 = ("Status", "Unavailable")
Property p2 = ("Distance", 484);
v1.includeProperty("city", "Ankara");
v2.includeProperty("city", "Eskisehir");
v3.includeProperty("city", "Istanbul");
v1.includeProperty("places", ["lake eymir", "Anitkabir",
"Seymenler"]);
e1.includeProperty(p1);
e2.includeProperty(p2);
connect(v1,v2,e1);
connect(v1,v3,e2);
```

7. DEFINING GRAPHS

Graphs that can be defined by Geez are classified as directed and undirected graphs. Each graph consists of any number of vertices and edges. Multiple properties can be assigned to the edges and vertices.

7.1 Directed Graphs

A directed graph is defined by the keyword "DirGraph" followed by its name. Curly brackets are used to embrace the definition of the graph which consists of vertex, edge definitions and connect method calls. In order connect method to be invoked, vertices and edges of the graph are defined with the keywords and their unique names. connect(Vertex v1, Vertex v2, Edge e1) method connects the vertices and edges generating the desired graph.

```
//Directed graph is generated
//Properties aren't added for simplicity.
DirGraph dg1{
      Vertex v1;
```

```
        Vertex v2;
        Vertex v3;
        Edge e1;
        Edge e2;
        connect (v1, v2, e1);
        connect (v2, v3, e2);
}
```

The code segment above illustrates a simple DirGraph implementation. Three vertices and two edges are defined. connect(…) method is used for connecting v1 - v2 with Edge e1 and v2 – v3 with Edge e2.

✓ While defining the directed graphs, users should be careful about the connection direction. For instance, connect (v1, v2, e1) method specifies the direction of e1 as from v1 to v2.

7.2 Undirected Graphs

Undirected graphs are defined as the same way with the directed graphs except that the keyword “UndirGraph” is used instead of “DirGraph”. All of the other implementation specifics are the same with the implementation of directed graphs.

```
//Undirected graph is generated
//Properties aren’t added for simplicity.
UndirGraph dg1{
        Vertex v1;
        Vertex v2;
        Vertex v3;
        Edge e1;
        Edge e2;
        connect (v1, v2, e1);
        connect (v2, v3, e2);
}
```

The code segment above demonstrates the implementation of a basic undirected graph. Vertices v1-v2 are connected with the Edge e1 and v2-v3 is connected with the Edge e2.

✓ connect() method called in undirected graph definition doesn’t specify a direction for the edge. In other words, connect(v1, v2, e1) and connect(v2, v1, e1) are the same.

8. PREDICATES

Predicates are used to check whether the edge or a vertex satisfies the desired property. They’re defined with the keyword “Predicate” followed by its name. If the

desired property is searched for an edge, “$” symbol should be used on the right hand side after the “=” symbol. On the other hand, if the desired property is searched for a vertex, “#” symbol should be used on the right hand side after the “=” symbol. If a match occurs for an edge or a vertex, function returns true. “$P” and “#P” symbols will be used for the edges and vertices consecutively whenever the user wants to pass edge/vertex without searching for any property.

```
//#equals(…)searches for a vertex which has a property name
//“name” and property value “Sheldon”.
Predicate p1 = #equals(“name”, “Sheldon”);

//#less(…)searches for a vertex which has a property name “IQ”
//and property value 145.
Predicate p2 = #lessThan(“IQ”, 145);

//#equals(…)searches for a vertex which has a property name
//“name” and property value “The Big Bang Theory”.
Predicate p3 = $equals(“name”,”The Big Bang Theory”);
//If a match occurs, functions will return true.
```

9.BUILT-IN FUNCTIONS

Geez supports various built-in functions aiming to provide ease of use to its users. With its basic features, built-in functions are helpful for defining graphs, predicates etc., working with properties and executing queries.

- connect(v1,v2,e1) : Vertices v1 and v2 are connected with the edge e1. In a directed graph, direction of edge is from v1 to v2. In an undirected graph, edge doesn’t have a direction.
- vertexName.includeProperty(name, value) : Includes the specified property which is composed of a name-value pair to the specified edge or vertex. Name of the property (name) is a string and value can be any of the primitive/collection type.
- equals(name, value) : Compares the given name-value pair with the edges or vertices of a graph depending on the “$” and “#” symbols before the function. If a match occurs, function returns true.
- lessThan(name, value) : Returns true if the value of edge’s/vertex’s property is less than the specified value.
- moreThan(name,value): Returns true if the value of edge’s/vertex’s property is more than the specified value.
- charAt(name, desiredIndex, desiredChar) : Returns true if the value of a property has

the specified char at the specified location.

- length(name, desiredLength) : Returns true if the value of the property which is a string has the length equals to the desiredLength.
- exec(graph, query): Executes the query for the given graph.

```
//Examples
Vertex v1;
v1.includeProperty("name", "Leonard");
Predicate p1 = #charAt("name", 2, "o");//Returns true, matches v1
Predicate p2 = #length("name", 5);    //Returns false
//Usage of other built-in functions are provided throughout the
//tutorial.
```

10. FILTERS

Filters are composed of predicates which are defined over edge and vertex properties. They're boolean expressions basically which match the predicates with the vertices and edges. They can be defined by using the keyword "filter" followed by the filter name. Geez supports concatenation, alternation and repetition of predicates, filters and queries aiming to provide modularity to the language.

- **Concatenation:** Two or more building block such as predicates, filters and queries can be concatenated by using the symbol "&".

```
//Filter f1 searches for a vertex which has a property
//("name","Jon Snow")followed directly by an edge which has a
//property ("name","Ygritte").
Predicate p1 = #equals("name","Snow");
Predicate p2 = $equals("name","Ygritte");
Filter f1 = p1 & p2;
```

- **Alternation:** Alternation is provided by using the symbol "|".

```
Predicate p3 = #equals("name", "Lannister");
Filter f2 = (p1&p2)| p3;
```

- **Repetition:** Repetition is provided with Kleene star, which has a symbol "^".

```
//Given expression can be repeated more than one or zero
//times.
Filter f4 = ((p1&p2) | p3)^
```

➢ As mentioned before, "$P" and "#P" are used for passing the edges and vertices without searching for a specific property.

➢ More complex filters can be generated from both predicates and filters.

```
Predicate p1 = #equals("name","Jon Snow");
Predicate p2 = $equals("name","Tyron Lannister");
```

```
Predicate p3 = #charAt("name", 3, "b");
Predicate p4 = $lessThan("closenessToThrone", 2);
//Searches for a vertex-edge pair satisfying the above
//properties.
Filter f1 = p1 & p2;
//Search for a vertex whose char at 3rd position is b and
//which is repeating through the query. Edges don't need to
//satisfy any property.
Filter f2 = (p3 & $P)^;
//Complex filter.
Filter f3 = (f1|(f2&f4))^
//Search for arbitrary vertex-edge pairs, which don't
//satisfy any property. Also the query can be empty.
Filter f4 = (#P & $P)^;
```

11. QUERIES

A query which is also supported by Geez is a regular expression composed of filters and queries. Alternation, concatenation, repetition are also allowed for their definition. They are defined by using the keyword "query" followed by its name. After the definition is completed, queries will be executed with the use of function exec(graph,query).

```
Query q1 = (f1|(f2&f4))^;
Query q2 = f5&f6;
Query q3 = (f1&f2&f3&(#P))^;
Query q4 = q1 |((q2^)&q3);
exec(myGraph,q4);
```

12. CONDITIONALS AND LOOPS

Geez supports conditionals such as if or if/else, while and for loops. The syntax used for them is kept similar to the programming language Java intending to provide an ease of use to the user.

- **If/Else statements**

```
flag = true;
If(flag)
{
   …
}
Else
{
```

```
   …
}
```

➢ Elseif statement is also supported with the same syntax.

- **while loop:**

```
//x's current value is added to v1 as a property until x
//becomes 7.
x = 0;
While(x <= 6)          //boolean expression
{
   x = x + 1;
   v1.includeProperty("current x", x);
}
```

- **for loop:**

```
y = 6;
For(x = 0, x < 5; x++) //initialization, boolean expr, update
{
   y = y * x;
}
```

13.BUILT-IN CLASSES

Geez supports built-in classes which provides an easiness to the user in terms of defining more complex terms such as property, query, edge etc. rather than primitive or collection types. Supported built-in classes are as follows:

- Property
- Edge
- Vertex
- UndirGraph
- DirGraph
- Filter
- Predicate
- Query

Their usage and parameters are explained in detail throughout the tutorial.