BİLKENT UNIVERSITY

ENGINEERING FACULTY

DEPARTMENT OF COMPUTER ENGINEERING

CS 315 Project – Part 1

GEEZ : A Graph Description-Query Language

REPORT

Muhammed Duha Çelik - 21400794

Selin Fildiş - 21402228

Başak Melis Öcal - 21400668

Group 5

TABLE OF CONTENTS

1) Graph Definition Language

a) Supporting Directed Graph Definitions

Directed graphs can be defined with the *DirGraph* keyword in Geez. After *DirGraph* keyword, user can specify a name for his/her graph. And after the name is specified, inside the curly brackets, properties of the graph can be defined. As its name implies, *connect* keyword takes place in this process to connect the edges and vertices as:

```
DirGraph g1{
     Vertex firstVertex;
     Vertex anotherVertex;
     Edge bridge;
     connect (firstVertex, anotherVertex, bridge);
}
```

Directed graph means that there is a specified direction between the edges that are connected to each other. Geez supports this feature with the *connect* keyword, i.e., when the connect keyword is used while defining a directed graph, it also gives a direction from first written vertex to second one. In our example, it specifies a direction from firstVertex to anotherVertex.

b) Supporting Undirected Graph Definitions

Undirected graphs are supported almost in the same way with the directed graphs. Only difference is the keyword that is used when the graph is defined. Instead of *DirGraph*, *UndirGraph* keyword is used to specify an undirected graph.

```
UndirGraph g2{
     Vertex firstVertex;
     Vertex anotherVertex;
     Edge bridge;
     connect (firstVertex, anotherVertex, bridge);
     connect (anotherVertex, firstVertex, bridge);
}
```

This time however, since the undirected graph means that there is no direction between the two vertices that are connected to each other, order is not important when the *connect* keyword is used. It means that, connect (firstVertex, anotherVertex, bridge) and connect (anotherVertex, firstVertex, bridge) gives the same output.

c) Supporting Vertex Definitions

In Geez, vertices are supported with the keyword *Vertex* followed by a name that user prefers. It is possible to add more than one properties to a desired vertex. For this purpose, there is a method that is named as *includeProperty*. This method can simply be used by using the predefined keyword *property* as nameOfVertex.includeProperty(propertyName) or just nameOfVertex.includeProperty(name, value).

```
Vertex firstVertex;

Property p1 = (“Spring Fest”, {“Duman”: [1], “Sertab
Erener”: [2]});

firstVertex.includeProperty(“Hello World”, 10);

firstVertex.includeProperty(p1);
```

d) Supporting Edge Definitions

Like vertices, edges are also supported in Geez language. Edges can be defined with the keyword *Edge* followed by its name. It is possible to add more than one properties to a desired edge. For this purpose, *includeProperty* method and *property* keyword can also be used for edges as well.

```
Edge e1;

Property p1 = (“Stadiums”, [“Anfield Road”, “San Siro”,
“Santiago Bernabeu”, “St.James Park”]);

e1.includeProperty(“Signal Iduna Park”, “BEST”);

e1.includeProperty(p1);
```

<u>Supporting Properties</u>

Property keyword followed by the name of the property can be used to define properties. These properties then can be used at any time for both the vertex and edge definitions. Basically this keyword makes the users’ life easier. Instead of specifying same property again and again, *Property* keyword can be used to specify the property once and then use it at any time after. *Property* is defined as pairs. 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.

```
//second field is also a string:

Property prop1 = (“Ali”, “Cengiz”);

//second field is a float:

Property prop2 = (“Fenomen”, 98.8);

//second field is a set:
```

```
Property prop3 = (“Turkey”, [“Mediterranean”, “Black
Sea”, “Aegean Sea”]);

//nested structures are also supported:

Property prop4 = (“World Map”, {“Asia”: [0,10], “Europe”:

[10,17], “America”: [17,26]});
```

e) Supporting Dynamic Type System

Geez is designed as a dynamically typed language which supports both primitive and collection types.

Primitive types:

- int
- float
- string
- boolean

```
number = 100;                  //integer

numberf = 100.678;             //float

flag = true;                   //boolean

str = "You know nothing!"      //string
```

Collection types:

- List: Lists are defined by using square brackets, “[]”. They can be used for a collection from the same types or different types.
- Map: Curly brackets, “{}” will be used to define maps. Each entry in the map consists of a key-value pair in which key must be string.
- Set: Sets are defined by using square brackets, “[]”.

```
//List of strings and floats.

list1 = ["Sheldon","Leonard","Penny"];

//Map from a string to a list of integers.

map2 = {“Bugra”: [315,442], “Ugur”: [202,319],
“Billur”: [391,300]};

//Sets of strings and integers.

set1 = [“See the money”, “wanna stay”, “for your meal”];
```

Arbitrary nestings of collection types are allowed aiming to provide a flexible experience to the user.

➢ Since Geez supports dynamic typing which means type checking is done at run-time rather than compile time, user doesn’t need to specify the type of the variable while defining or initializing.

```
//It’s type (int) will be determined at run-time.
```

```
y = 5;
```

2) Graph Query Language

a) Supporting regular path queries

-Predicates are used to support regular path queries as they are the building blocks of queries. They check whether an edge or a vertex of a graph satisfies the specified property. They can be implemented with the *Predicate* keyword followed by a specific name. When the predicates are defined, "#" is used for vertices while "$" is used for edges in Geez.

- $P and $P can be used consecutively for edges and vertices, that the user wants to pass without searching for a specified property.
- Arithmetic expressions and built-in functions defined in the tutorial can be used for generating predicates.

```
Predicate p1 = #equals("name", "Roberto Firmino");
//searches for a vertex that has the given name
Predicate p2 = $equals("name", "Sadio Mane");
//searches for an edge that has the given name
```

-Filters are composed of predicates in Geez. They can be implemented with the *Filter* keyword followed by a name. Filters are basically Boolean expressions that match the predicates with the vertices and edges.

```
Predicate p3 = $equals("date", 9.11);
Predicate p4 = #equals("score", 1453);
Filter f1 = p3 & p4;
```

-Queries, regular expressions that composed of filters and other queries, are supported in Geez language. They can be implemented with the keyword *Query* followed by a name. After multiple queries are defined, it is possible to do alternation ("|"), repetition ("*") and concatenation ("&") witt these queries. Not only for queries, but also for predicates and filters these operations can be applied as well to generate complex expressions.

```
Predicate p1 = #equals("name", "Roberto Firmino");
Predicate p2 = $equals("name", "Sadio Mane");
Predicate p3 = $equals("date", 9.11);
Predicate p4 = #equals("score", 1453);
Filter f1 = p3 & p4;
Filter f2 = p1 | p2;
Query q1 = (f1 | f2^)^;
```

b) Support Having Variables in Path Expressions

Geez supports dynamic type system as mentioned before. Variables can be also used in any kind of function or property as follows:

```
//Matches with paths whose consecutive vertices have the
```

```
//property named as "IQ" which is less than variable x.

Predicate p1 = #lessThan("IQ", x);

Filter f1 = p1 & $P & p1;

Query q1 = filter;

exec(myGraph,q1);
```

c) Support Modularity

Geez supports modularity by providing different building blocks that can be used for defining filters and queries. In other words, expressions for filters and queries can be divided into multiple less complex expressions.

- ➢Predicates are the smallest pieces that can be used to form filters. As they match with a specified vertex or an edge depending on the syntax, a filter can be defined which searches for the desired vertices and edges consecutively. In other words, predicates can be used in the regular expressions of filters.
- ➢Filters can be used for generating more complex filters and queries in the same manner.
- ➢Geez supports alternation ("|"), concatenation ("&") and repetition ("^") which can be applied on predicates, filters and queries in order to form more complex ones.
- ➢Queries can be also used for forming more complex queries with the operations defined above.

Defining one expression in terms of smaller ones provide modularity to the language. Also, it comes up with an ease of use as it allows user to define the desired filters/queries easily.

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