**MEDER KUTBIDIN UULU**
**DIAS ALYMBEKOV**
**CÜNEYT EREM**

## 1. (45pts) The street network definition language shall support:

### (a) (5pts) Defining a street network with one-way streets.

One-way streets can be defined as oneWayStreet and the name of the network is followed by it. Afterwards, properties of streets and points will be defined.

```
oneWayStreet network1{
        m -> n //which means, there is only way from point m to point n.
        s  = n -> l //there is a way from point n to point l.
        Street s{
                S.name = "Bahceli"
        }
}
```

***Look to tutorial: One-way street networks***

### (b) (5pts) Defining a street network with only two-way streets.

Two-way streets can defined as twoWayStreet and the specifying name of the network comes afterwards. Inside the network, point and street properties can be defined. For instance,

```
twoWayStreet network2{
        m --- n //these three dashes mean that there are two ways: both from point m to n and from point n to m.
        Street s
}
```

***Look to tutorial: Only two-way street networks***

### (c) (10pts) Defining point properties.

- A point property is some data value associated with a point.
- It shall be possible to attach multiple properties to a point.

  In SQL, it is possible to assign multiple values to a particular point. For instance,

```
oneWayStreet network1 {
m -> n //which means, there is only way from point m to point n.
Point m{
        m."stationName" = "Petrolium"
```

m.”type” = “Gas”}

As you can see above, point m has both ‘stationName’ and ‘type’ properties.

***Look to tutorial: Defining point and street properties**

- A property should be a (name, value) pair. For instance, if a point represents a gas station, then a point property can be *(‘name’, ‘Bilkent Petrol’)*. Remember that there may be multiple such properties, such as *(‘type’, ‘gas station’), (‘name’, ‘Bilkent Petrol’), (‘sells’, ‘LPG’), (‘payment-options’, ‘cash’)*.

Our SQL program supports the (name, value) property. Users of the program can set as many properties as he/she wants, there are no limitations. Let’s say we have a Point p, and the values to the p can be set as follows:

p.’name’ = ‘Bilkent Petrol’;
p.’type’ = ‘Gas Station’;
p.’sell’ = ‘LPG’; // and the list goes on

***Look to tutorial: Defining point and street properties**

---

(d) (10pts) Defining street properties

• It shall be possible to attach multiple properties to a street.

The SQL language supports to set several properties to a street.

```
oneWayStreet villageNetwork {

point a;
point b;

street street1;
street1 = a->b;

//street properties
street street1 {
        “avgtime” = “60”,
        “delay” = “10”
}
```

As shown above, streets can get multiple properties.

***Look to tutorial: Defining point and street properties**

• The streets shall have a property to describe the average time to pass the street.

From the given example previously, the average passing time is given as a property of the street.

```
street street1 {
        “avgtime” = “60”,
        “delay” = “10” }
```

(e) (15pts) A dynamic type system for the property values (property names should be strings)

- Supports *strings*, *integers*, and *floats* as primitive types. For instance, the 'average-time' property of the streets may have an *integer* value, whereas the 'name' attribute may have a *string* value.

SQL is a dynamically typed language. You do not have to specify a data type for any variable.

$variable = 4.0 //float type
$variable = 'Mederbeck' //string
$variable = true; //boolean
***Look to tutorial: Primitive types & Variables**

- Supports lists, sets, and maps as collection types. For instance, it should be possible to have a property like: *('sells', {'LPG', 'Diesel'})*. This would be an example of a property, whose value is list of strings. Arbitrary nesting should be possible as well. For instance, *('sells', {'gas-types':{'LPG', 'Diesel'}, 'payment-options': {'Cash', 'Visa', 'MasterCard'}})*. This is an example where the value type is a map from a string to a list.

All the collection types that are required are being supported by SQL:

a."sells" = list1["LPG", "Diesel"] //point list as string
street1."avgtime" = list2[15, 30] //street list as integer
***Look to tutorial: List**

a."paymentOpition" = set1["Cash", "Visa"] //point set as string
a."paymentOpition" = a."paymentOpition" + "Mastercard";
*//a."paymentOpition" becomes = set1["Cash", "Visa", "Mastercard"]*
***Look to tutorial: Set**

a."sells" = map1["gas-types" : ["LPG", "Diesel"]] //string to list
b."sells" = map2["payment-options" : ["Cash", "Visa", "Mastercard"]] //string to list
street1."avgtime" = map3["time" : [35]] //string to int
***Look to tutorial: Map**

2. The street network querying language shall support:

(a) (30pts) Creating route queries. A route query is an expression specifying a route while visiting several streets and/or points. A route is an alternating series of points and streets. Importantly, we are not asking you to evaluate queries. We are asking you to create language to express them. A route query shall be able to specify:

- Concatenation, alternation and repetition

‘&’ stands for concatenation, ‘|’ stands for alternation, ‘*’ and ‘+’ stand for repetition.

- Provides filtering constraints as Boolean expressions, which are composed of predicates defined over street properties as well as incident point properties.
  – For instance, one may want to find all routes, from a specific building at Bilkent to a shopping mall while stopping at a gas station, formally, the start point of the first street has a property *‘name’ = ‘Bilkent Building*’ passes through a street with a starting point that has properties ‘*type’ = ‘gas station*’ and *‘sells’ = ‘Diesel’*, and finally the last street end point has a property *‘type’ = ‘Shopping Mall’*.
  – Another one may want to find the routes from a train station at Ankara to his home, formally, the start point of the first street has properties *‘type’ = ‘train station’*, *‘City’ = ‘Ankara’*, the last street has a property of *name = ‘13th St*.’ and the end point has a property of *house-number = ‘25’*

SQL language uses following syntax for querying the routes in a given network. First, we need to give properties to a starting point, give properties to a street through which a user wants to pass and finally set final point properties. Examples are given below:

Network1 >> [‘name’ = ‘ Bilkent Building’] <’type’ = ‘gas station’, ‘sells’ = ‘Diesel’> [‘type’ = ‘Shopping mall’];
Network2 >> [‘type’ = ‘train station’, ‘city’ = ‘Ankara’]<’name’ = ‘13th St.’> [‘house-number’ = ‘25’];

In this case, Network1 and Network2 are given, first square bracket is the properties of the starting point, whereas the triangular bracket is the properties of the street which is being passed through and finally, the last square bracket is final point properties.
***Look to tutorial: Query**

---

- (6pts) Support for existence predicates as well as arithmetic expressions and functions in predicate expressions. For instance: a point containing or not containing a property with a given name or value; or a street that has a certain property whose value is greater than a constant; or a point that has a certain property whose value is a string that starts with ‘A’ (this would require a string indexing function).

Network >> [contain(name) == true] < sqrt(“avg_time”) < x >[‘name’[0] == ‘A’]
In this example, first square bracket is checking if the name is contained in the starting point. The second bracket is to check whether the square root of the avg_time is less than a given specific variable and the third bracket is to check if the first letter of the name of the end point is equal to ‘A’. If there is any route which satisfy to these requirements, then those routes will be listed. And the functions used here are the built-in function and they are explained in Tutorial, Functions section.

***Look to tutorial: Query**

- (6pts) Support for sorting based on: “Shortest route in time”, “Shortest route in distance” or “Simplest Route” (minimum number of streets on route).

network >> [“name” = “Bilkent Building 81”] <min_time = true> [“type” ==”Shopping Mall”];
*//This query is to find the routes which are shortest in time.*

network >> [“name” = “Bilkent Building 81”] <min_dist = true> [“type” ==”Shopping Mall”];
*//the query is to find the routes which are shortest in distance*

network >> [“name” = “Bilkent Building 81”] <simplest = true> [“type” ==”Shopping Mall”];
*//this query is to find the simplest route among the routes.*

***Look to tutorial: Query**

- (6pts) Support to limit the number of routes on the result. For instance, one may want to limit the query to return only the best three routes.

  network >> [“name” = “Bilkent Building 81”]<limit_results = 3>[“type” ==”Cepa”];

  *//there is an option to get the best three results among many other routes. This option can be used just limiting the routes by using **‘limit_results’**.*

***Look to tutorial: Query**

---

(b) (15pts) Having variables in route queries. For instance, you may want to query all routes where both start and end streets in the route have a property called name with the same property value, but the value is not known. In this case, that value becomes a variable.

network >> [“name” = $point]<limit_results = 3>[“type” == $point];

*//here value behaves as a variable, $point.*

***Look to tutorial: Query**

(c) (10pts) Modularity, that is dividing regular route queries into multiple pieces that are specified separately. This would require giving names to each piece and being able to use those names in a higher-level query.

query1 = query2  ( query3 || query4 ) *

*//The query1 unites the query2 with query3 or query4, all possible requests are included in the query 1.*

***Look to tutorial: Query**