Street Network Definition and Querying Language

## 1) Street Network Definition Language

**a)** Defining a one way street network:

The street network (SN) is treated as an object and below is the constructor. The constructor takes the token -> to indicate that the street network is one-way.

```
sn1 = SN(->);
```

**b)** Defining a two way street network:

Two way street networks are defined similarly. The constructor takes the token => to indicate that the street network is two-way.

```
sn2 = SN(=>);
```

**c)** Point definition and properties:

A point is defined as the following:

```
p1 = Point();
```

Its type is dynamically allocated and it has an empty constructor.

We can define properties generally as shown below. In this definition, the first parameter is the name and the second parameter is the value of the property.

```
property1 = Property("name", "Bilkent University");
property2 = Property("sells", map1); //m1 is a Map
property3 = Property("sells", list1); //list1 is a List
property4 = property("type", set1); //set1 is a Set
property5 = property("fee", 2); //value is an integer
property6 = property("fee", 2.5); //value is a float
```

A property can be added to a point using the following function:

```
p1.addProperty(property1); //property1 is a Property object
```

We can also remove properties using the following function:

```
p1.removeProperty(3); //deletes property at index 3.
```

**d)** Street definition and properties:

A street is defined in the following way:

```
st1 = Street (30, p1, p2); //30 is the average crossing time of
                           //the street in minutes, p1 is the
                           //starting point of the street and p2
                           //is the ending point.
```

The first parameter is the average time it takes to pass the street in. If it is a one-way street, the second parameter is the beginning, and the third parameter is the end point. If it is a two-way street, additionally another street beginning at parameter three and ending at parameter two is defined.

A street delay property can be added in the following way:

```
s1.addProperty (property1);
```

We can also add multiple properties using this function.

We can remove properties using the following function:

```
s1.removeProperty(3); //deletes property at index 3.
```

We can also add temporary properties to our street. A delay is added in the following manner:

```
s1.addDelay(20, "Road-work"); //There is a 20 minute delay on
                              //the street due to road-work.
```

We can also add multiple delays using the above function.

We can remove a delay using the following function:

```
s1.removeDelay(0); //removes the delay at index 0.
```

The street can also be temporarily closed using the following function:

```
s1.closed(t1, t2, "Traffic"); // t1 and t2 give us the time
                              //interval for which the street
                              //is closed. The following string
                              //gives the reason for which the
                              //interval is closed.
```

Here t1 and t2 are Time objects defined in the following way:

```
t1 = Time(10, 30); //time is 10:30
```

The street can be reopened using the following function:

```
s1.opened();
```

**e)**

**Primitive Types:**

Since our language supports dynamic type allocation, strings, integers, characters and floats can be assigned in the following way:

```
a = "Sena, Nazli and Mert";
b = 3;
c = 4.5;
d = 'c';
```

**Lists** are defined in the following way:

```
list1 = List();
```

We can append new values to the list using the following functions:

```
list1.append(map1); //map1 is a Map
list1.append("Visa"); //Visa is a string
list1.append(list2); //list2 is a list
```

We can remove properties from the list using the following function:

```
list1.remove(2); //remove the value at index 2.
```

**Maps** can be defined as the following:

```
map1 = Map();
```

We can add new key, value pairs to the map using the following functions:

```
map1.add("name", "Nazli");
map1.add("name", list1); //list1 is a List
```

We can remove a key and it's associated values using the following function:

```
map1.removeKey("name");
```

**Sets** can be defined as the following:

```
set1 = Set();
```

We can add new values to the set using the following functions:

```
set1.add("Sena");
set1.add(list1); //list1 is a List
set1.add(set2); //set2 is a Set
```

We can remove a key and it's associated values using the following function:

```
set1.removeValue("Sena");
```

**2) Street Network Querying Language**

We have Boolean properties that are shown arbitrarily as "predicates". For example:

```
predicate1 = ("name" == "Bilkent University");
```

**a) Concatenation** is defined as the following:

```
(predicate1)concat(predicate2) //concatenates property1 and
                               //property2
```

**Alternation** is defined as the following:

```
(predicate1)alter(predicate2) //chooses between property1 and
                              //property2
```

**Repetition** is defined as the following:

```
(predicate1)rep(2) //repeats property1 2 times.
```

The precedence order is from highest to lowest is; repetition, concatenation, alternation.

We will find all routes using the following function:

```
streetNetw1.findRoute(predicate1,(
predicate2)concat(predicate3)alter(predicate4), predicate5);
//starting point has predicate1, along the way we must pass (
//predicate2 and predicate3) or predicate4, the ending point
//has predicate5.

streetNetw1.findRoute(predicate1, predicate2); //starting point
            //has predicate1, ending point has predicate2)
```

This querying operation returns us a list of lists. This list is a list of routes, where we have a list of streets we can pass through to fulfill the requirements.

We will define the following arithmetic operations:

A point/street that does not have a property:

```
not(predicate1)
```

A point/street that has a property greater than a constant value:

```
predicate1 = ("price" == 3.5);
(predicate1)greater(3)  //predicate1's value (price) must be
                        //greater than 3.
(predicate1)greaterOrEqual(3) //predicate1's value (price) must
                              //be greater than or equal to 3.
```

A point/street that has a property less than a constant value:

```
(predicate1)less(3) //predicate1's value (price) must be less
                    //than or equal to 3.
(predicate1)lessOrEqual(3) //predicate1's value (price) must be
                           //less than or equal to 3.
```

A point/street that has a property that starts with a certain character:

```
(predicate2)startsWith('A') //property2's value must start with
                            //'A'.
```

We will sort the routes according to the shortest route, shortest distance and simplest route using the following function:

```
      list1 = streetNetw1.findRoute(predicate1,(
predicate2)concat(predicate3)alter(predicate4), predicate5);
                  //finds routes according to the route query and
                  //returns a list of routes.
      streetnetw1.sortShortestRoute(list1);
                  //sorts routes according to route times from
                  //shortest to highest.
      streetnetw1.sortShortestDistance(list1);
                  // sorts routes according to route distances from
                  //shortest to longest.
      streetnetw1.sortSimplestRoute(list1);
                  // sorts routes according to the number of streets
                  //on the route, from minimum to the maximum.
```

This function modifies the list it is given as a parameter and sorts it as desired.

We will sort the routes according to the shortest route, shortest distance and simplest route using the following function and return the specified number of routes:

```
      list1 = streetNetw1.findRoute(predicate1,(
predicate2)concat(predicate3)alter(predicate4), predicate5);
            //finds routes according to the route query and returns a
            //list of routes.
      list2 = streetnetw1.sortShortestRoute(list1, 3);
            //sorts routes according to route times from shortest to
            //highest and returns the best three routes.
      list3 = streetnetw1.sortShortestDistance(list1, 5);
```

```
        //sorts routes according to route distances from
        //shortest to longest and returns the best five routes.
list4 = streetnetw1.sortSimplestRoute(list1, 2);
        //sorts routes according to the number of streets on the
        //route, from minimum to the maximum and returns the best
        //two routes.
```

This function does not modify the list it is given as a parameter. Instead it sorts the list and returns the best n (the integer in the second parameter) number of routes in a new list.

**b)** Having variables in route queries:

To define a property with a variable as the value, we use the notation var(x). This means that x is a variable with unknown value.

```
predicate1 = ("name" == vary(x))
```

**c)** Regarding modulation, using our methods of concatenation, alternation and repetition, we can define endless predicates to satisfy our route queries. We can give names to each piece of the query:

```
predicate1 = (predicate2)concat(predicate3)rep(2)
```

We can use predicate1 in a higher level query now:

```
predicate5 = (predicate1)alter(predicate6)
```