Bilkent University
Department of Computer Engineering

QICIRLANG

CS315 Project: A Street Network Definition and Querying Language

Mert Aytöre
Nazlı Özge Uçan
Sena Kıcıroğlu

Course Instructor: Ercüment Çiçek
TA: Arda Ünal

Contents

GETTING STARTED

Qicirlang is a simplistic street network definition language which provides users to define custom street networks which can be one-way or two-way. These street networks are collections of streets and the points they connect together. The language also allows for querying within a street network. We can specify the properties we would like to visit along a route and use predefined functions to create lists of routes. We can also sort these routes according to certain specifications, such as the minimum number of streets needed to pass, etc.

COMMENTS

As it is commonly known, commenting is a very useful tool while writing & debugging programs and understanding them at first look. Qicirlang also supports this essential functionality. Qicirlang provides commenting with a single option; characters after double '/' are accepted as commentted and will be ignored by the compiler. However, the user should be careful since they are only single line comments.
An example is shown below:

```
buildingList = List(); //creates a list
//The list will contain building names
```

TYPES AND VARIABLES

Qicirlang supports dynamic type allocation and strings, integers, floats and characters and can be used in the program in the following way:

```
str = "qicirlang"; // str is given the value "qicirlang"
num = 1; // num is given the value 1
numfloat = 12.5; // the variable numfloat is 12.5
character = 't'; // character has the value 't'
```

- Variables in qicirlang can only start with a letter and rest can be followed by letters, digits or underscores.
- Variable names are case sensitive, meaning that variables with same names but different cases are different.
  e.g. example = 1; and eXaMpLe = 1; are different variables.

LISTS

Elements of lists in qicirlang can be composed of many different data types. These lists will be used to define values of properties. Lists are defined with the syntax as the following:

```
list1 = List(); // An empty List named list1 is created.
```

Inserting new values to the a list can be done in the following way:

```
//a Map named map1 is inserted into list1 from the end
list1.append(map1);
//"Mert" is added to the end of list1
list1.append("Mert");
// another list, named list2, is added to the end of //list1 below:
list1.append(list2);
// The value with the index 2 in the list, named list1, //is removed
below:
list1.remove(2);
```

MAPS

Maps in qicirlang are used to match values to keys for easy access. They can be inserted either to maps, lists or properties (as values). Maps are defined with the syntax in the following:

```
map1 = Map(); //A Map named map1 is created
```

New values can be added to a map using the following functions:

```
//string with the value "Mert" is inserted to the Map, //named map1,
under the key of "name" below:
map1.add("name", "Mert");
//Predefined List, named list1 is added to map1 under the //key of
name in the following:
map1.add("name", list1);
```

Keys and the values that are associated with them can be removed from the map with the following function:

```
// The key "name" and the values that are associated with //it are
removed below:
map1.removeKey("name");
```

SETS

Sets in qicirlang are unordered collection of various elements. They cannot have duplicate values. Sets store data and they can also be inserted into maps, properties (as values), or lists. Sets are defined with the syntax in the following:

```
set1 = Set(); // Set named set1 is created.
```

New values can be added to a set using the following functions:

```
//string with the value "Mert" is added to the Set, named //set1
below:
set1.add("Mert");
//Predefined Lists, named list1, can be added to set1 //below:
set1.add(list1);
//Sets independent from the existing set can be added //with the
function shown below: (inserting into set1, //new Set: set1)
set1.add(set2);
//string with the value "Mert" is removed from the Set, //named set1
below:
set1.removeValue("Mert");
```

DEFINING A STREET NETWORK

Qicirlang's main purpose is to define and query street networks. There can be two different types of street networks; one-way street networks and two-way street networks.

One-Way Street Network:

A one-way street network is composed of one way streets and their beginning and ending points.

A one-way street network can be created in the following way:

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

It is important to note that the token -> is used to characterise one-way street networks. The code segment above creates an instance of a one-way street network object, sn1, which can store many street objects.

Two-Way Street Network:

A two-way street network is composed of two way streets and the points they are between.

A two-way street network can be created in the following way:

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

As you can see it is very similar to the one-way street network definition. It is important to note that the token => is used to characterise two-way street networks. The code segment above creates an instance of a two-way street network object, sn2, which can store many street objects.

We can add Street objects to street networks using the following predefined function. (As a bonus we can also see that to each street we can define beginning and ending points):

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

```
point1 = Point(); //a point is defined
point2 = Point();
//a street is defined
street1 = Street (20, point1, point2);
//a street is added to sn1 (our one-way street network)
sn1.addStreet( street1 );
```

We can add as many streets as we want to our street networks using this function.

DEFINING A POINT

Points are specific locations which are connected by streets. A point can be thought of as a place such as a supermarket, a university, a park and so on.

A Point object is instantiated in the following way:

```
point1 = Point();
```

This creates a default point object.

To add properties to this point, we must first define a Property object. Property objects are very essential to qicirlang. They are needed both for defining properties of a Point and a Street. A property is instantiated in the following way:

```
property1 = Property("Type", "University");
```

Properties are composed of a name and a value. In our example, the name is “Type”, and the value is “University”.
The value can be a string, an integer, a float, a list, a map or a set. Let us see a couple of examples. Below is an example of adding a list as the value of a property.

```
property1 = Property();
//Our list will be: {'Engineering', 'Fine Arts', 'Law'}
departmentList = List();
departmentList.append("Engineering");
//add new values to the list
departmentList.append("Fine Arts");
departmentList.append("Law");
//Now we will add this list as the value of our property
property1 = Property("Departments", departmentList);
```

Let us add a set as the value of a property:

```
property2 = Property();
//Our set will be: {'A Building',  'F Building', 'B Building'}
buildingSet = Set();
buildingSet.add("A Building");
```

```
buildingSet.add("F Building");
buildingSet.add("B Building");
//Now we will add this set as the value of our property
property2 = Property("Departments", buildingSet);
```

Let us add a map as the value of a property.

```
property3 = Property();
//Our map will be: {{'Gas Types': 'LPG'}, {'Payment Options': 'Cash'}}
saleOptionsMap = Map();
saleOptionsMap.add("Gas types", "LPG");
saleOptionsMap.add("Payment Options", "Cash");
//Now we will add this map as the value of our property
property3 = Property("Sells", saleOptionsMap);
```

Now that we have learned how to create Property objects, we can add properties to our point.

```
point1 = Point();
point1.addProperty(property1); //We have formed property1 above.
```

We can add as many properties as we like using this function. This function adds properties to the point like a list, each new property is appended to the end.
To remove properties we use the following function:

```
point1.removeProperty(0); //removes property at index 0.
```

Once we remove a property, each property after the given index will be shifted up by one element.

Points are also necessary for the instantiation of Street objects, which we will look into next.

DEFINING A STREET

Streets are used to connect points together and form a street network. They can be added later on to the street-network objects as we saw previously.

A street object is instantiated in the following way:

```
point1 = Point(); //a point is defined
point2 = Point();
//a street is defined
street1 = Street (20, point1, point2);
```

To create a street we need to know the average time it takes to pass through the street and which points it is connecting together. The first parameter is the average time in minutes (in our example, 20 minutes). 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.

Properties can be added to streets in a similar way to points. We use the following function.

```
street1.addProperty(property1);
//property1 and street1 were created above
```

We can add as many properties as we like using this function. This function adds properties to the point like a list, each new property is appended to the end.
To remove properties we use the following function:

```
street1.removeProperty(0); //removes property at index 0.
```

Once we remove a property, each property after the given index will be shifted up by one element.

We can also add temporary properties to streets. We can add delays using the following function.

```
street1.addDelay(20, "Traffic"); //20 minute delay because of traffic
```

This function has two parameters, the first one specifies the average time of the delay and the second one specifies the reason. We can add as many delays as we like using this function. We can also remove delays using the following function:

```
street1.removeDelay(2); //removes delay at index 2
```

We can also indicate that a street has been closed for a certain period. We use the following function:

```
time1 = Time(10:30); //creates a Time object, 10:30
time2 = Time(11:30); //11:30
//street is closed from 10:30 to 11:30 due to road work.
street1.closed(time1, time2, "Road-work");
```

This function takes three parameters, the first and the second parameter are the period of time for which the road will be closed (which in our case is 10:30-11:30), and the third parameter is the reason.
We can reopen a street using the following function:

```
street1.opened();
```

STREET NETWORK QUERYING

LANGUAGE

In our language we define boolean expressions as follows:

```
predicate1 = ("name" == "Bilkent University");
predicate2 = ("name" == "Shell Gas Station")
//predicate1 and predicate2 are boolean objects.
```

We can use concatenation, alternation and repetition operations on boolean expressions. The precedence order of those are from highest to lowest is; repetition, concatenation and alternation. Also as general, boolean expressions in parenthesis have the highest precedence.

Concatenation

Concatenation is defined as the following:

```
(predicate1)concat(predicate2)
//concatinates predicate1 and predicate2
```

A concatenation expression specifies that predicate1 and predicate2 should both be included in the result.

Alternation

Alternation is defined as the following:

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

An alternation expression specifies that in the result there should be predicate1 or predicate2.

Repetition

Repetiton is defined as the following:

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

A repetition expression specifies that in the result there should be n times, as n is the integer in the brackets after rep expression.

Arithmetic expressions are also defined as below:

A point or street that does not have a property can be expressed as in the following:

```
not(predicate1)
//This is a boolean expression that implies the negative of
//predicate1
```

A point or 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.
```

A point or street that has a property greater than or equal to a constant value:

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

A point or 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.
```

A point or street that has a property less than or equal to a constant value:

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

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

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

In qicirlang, there can also be variables that we can use for queries. A variable with an unknown value can be shown as follows:

```
vary(x)
```

To define an expression with an unknown variable we can use the following expression:

```
predicate1 = ("name" == vary(x))
//predicate1 has the boolean value where name equals an unknown
//variable x
```

The language also has modularity. It can have various steps of independent units that can be combined together. For an example:

```
predicate1 = (predicate2)concat(predicate3)rep(2)
//We define predicate1 as predicate2 and two times predicate3.
predicate4 = (predicate1)alter(predicate5)
//predicate4 is predicate1 or predicate5.
```

In the above expression we use predicate1, which we previously constructed as an independent boolean expression with predicate5.

FINDING ROUTES

As we create our street network, add streets and points to it and specify the properties for each street and point object, we want to find a route. We can find a route by specifying the beginning and ending predicates or beginning and ending predicates as well as the conditions in between.

We can find all routes in the street network streetNetw1 from all points that satisfies predicate1 to all points that satisfy predicate2 as follows:

```
List1 = streetNetw1.findRoute(predicate1, predicate2);
```

We can find all routes in the street network streetNetw1 from all points that satisfies predicate1 to all points that satisfy predicate2 and passes from all points that satisfy predicate 4 as follows:

```
List2 = streetNetw1.findRoute(predicate5, predicate4, predicate6);
```

FindRoute function returns a list of lists where each list object is a route that satisfies the conditions.

SORTING ROUTES

After we create our routes, we can sort them in three ways:

Shortest Routes

We can sort the routes from the list that is returned from findRoute method from shortest route time to the highest as follows:

```
streetNetw1.sortShortestRoute(list1);
//sorts routes according to route times from shortest to highest by
//modifying the given list.
```

Also we can sort the list and return the best n result, as n is an integer in the following way:

```
list3 = streetNetw1.sortShortestRoute(list1, 3);
//sorts routes according to route times from shortest to highest and
//returns the best three routes in list3.
```

It is important to note that the second method returns a new list of routes while the first one modifies the given list of routes.

Shortest Distance

We can sort the routes from the list that is returned from findRoute method from shortest route distance to the highest as follows:

```
streetNetw1.sortShortestDistance(list1);
//sorts routes according to route distances from shortest to highest
//by modifying the given list.
```

Also we can sort the list and return the best n result, as n is an integer in the following way:

```
list3 = streetNetw1.sortShortestDistance(list1, 3);
//sorts routes according to route distances from shortest to highest
//and returns the best three routes in list3.
```

It is important to note that the second method returns a new list of routes while the first one modifies the given list of routes.

Simplest Route

We can sort the routes from the list that is returned from findRoute method according to the number of routes as follows:

```
streetNetw1.sortSimplestRoute(list1);
//sorts routes according to route distances from minimum to maximum
//by modifying the given list.
```

Also we can sort the list and return the best n result, as n is an integer in the following way:

```
list3 = streetNetw1.sortSimplestRoute(list1, 3);
//sorts routes according to the number of routes from minimum to the
//maximum and returns the best three routes in list3.
```

It is important to note that the second method returns a new list of routes while the first one modifies the given list of routes.