The Mauris Language

1) THE GRAPH LANGUAGE

a) Directed Graph:

The structure of a directed graph in our grammar is defined as following :

G = d: { DConnectionList } ;

where "d:" defines directed graph, "{" shows the beginning of the connectionList and "}" shows the end of the connectionList. To hold the graph's name as a variable assignment "=" and ";" can be used. G indicates the name of the directed graph variable.

connectionList: It is a token that defines the relationship between vertices and edges.

" *Vertex -Edge-> Vertex* " is a simple directed connection between two vertices. "Vertex" and "Edge" are tokens and can have properties.

Using "#", we can define more connections between vertices. For example,

Vertex -Edge-> Vertex # *Vertex -Edge-> Vertex* # . . .

b) Undirected Graph:

The structure of a undirected graph in our grammar is defined as following :

G = ud: { UDConnectionList } ;

where "ud:" defines directed graph, "{" shows the beginning of the connectionList and "}" shows the end of the connectionList. G indicates the name of the undirected graph variable.

connectionList: It is a token that defines the relationship between vertices and edges.

" *Vertex --Edge-- Vertex* " is a simple directed connection between two vertices. "Vertex" and "Edge" are tokens and can have properties.

Using “#”, we can define more connections between vertices. For example,

Vertex --Edge-- Vertex # Vertex --Edge-- Vertex # . . .

c) Vertex Properties

Vertex is a token that starts with “v” followed by “[” and ends with “]”. Vertex tokens can have properties and they must be written in between “(” and “)”. Since there can be more than one properties, they must be seperated by “,”. For example,

v[(“id”, 113), (“name”, “armağan”)]

There can be Vertex tokens that have no properties. For example,

v[ ]

d) Edge Properties

Edge is a token that starts with with “e” followed by “[” and ends with “]”. Edge tokens can have properties and they must be written in between “(” and “)”. Since there can be more than one properties, they must be seperated by “,”. For example,

e[(“year”, 2005), (“mutualFriends”, 3)]

There can be Edge tokens that have no properties. For example,

e[ ]

e) Property Value Types

Property values can be either primitive types or collection types.

Primitive Types:

String**:** It is a sequence of characters in between two quotation marks. For example:

(“name”, “armağan”) here, “armağan” is a string type.

Integer: It is a concatenation of digits where digit is defined as 0, 1, 2, 3 . . . 9. For example:

(“year”, 2005) here, 2005 is an integer type.

Float: It consists of two parts; one of them is main number and the other part is fractional part. They are seperated by "."and both of them are concatination of digits. For example:

("average", 5.8) here, 5.8 is a float type.

Collection Types:

Lists: It is an abstract data type that represents more than one primitive types. It starts with "<" and ends with ">". Each primitive type is seperated by ",". For example:

("score" , < 19, 20, 52 >)
("name", < "mert", "taner", "mertcan" >)

Sets: It is also an abstract data type that represents more than one primitive types but without holding their indexes. It starts with "{" and ends with "}". Each primitive type is seperated by "," again. For example:

("score", { 19, 20, 52 })
("name", { "mert", "taner", "mertcan" })

Maps: It is type of a container that maps values of one type to values of another type. It starts with "[" and ends with "]". Every instance is seperated by "," and mappings are defined by ":". A property whose type is map is represented as follows:

("customerInfo", ["name": "Ali", "surname": "Veli"])

2) THE QUERY LANGUAGE

A query language can be used to retrieve specific information about the graphs. Our query language has some different attributes which have different functions. All query expressions start with "Q" followed by ":". To specify different attributes after "Q:" an attribute has to be defined and "?" symbol closes the attribute definition. To hold the query's name as a variable assignment "=" and ";" can be used. Graph token comes after the "@" symbol is indicating the graph which the query will be applied on. For example:

query = Q: Attribute ? Type @ Graph ;

Attributes are tokens that have special keywords in the language. User may use query for specific features after attribute definitions and they are described below.

a) Regular Path Queries:

A regular path query can be expressed by regular expressions and finds the relevant paths according to that expression. Regular expressions are searched among the edge properties for this type of query. The “REGEX” keyword is the attribute for this purpose. For example: (“graph1” is an example graph here)

```
query  = Q: REGEX ? RegularExpression @ graph1 ;
```

<u>RegularExpression</u>: It is the regular expression to be find among the edge property values in a graph. This regular expression supports concatenation, alternation and repetition.

1- **Concatenation:** It is defined by joining edge property values with underscore (“_”) between them. For example, the concatenation of “a” and “b” is “a”_”b”.

2- **Alternation:** It is defined with “|” symbol and it is the union of two sets of patterns. For example, the alternation of “a” and “b” is “a|b” and this means the path requires either “a” value or “b” value.

3- **Repetition:** It is defined with “+” and “*” symbols. It returns a path with the edges which have the given property value one or more time if the “+” symbol is used, zero or more time if the “*” symbol is used. For example, the repetition for “a” is “a+” or “a*”.

Another path query can be defined with Boolean expressions. The query finds the relevant path according to the given boolean algebra among the property values. The “BOOL” keyword is the attribute for this purpose. For example:

```
query  = Q: BOOL ? BooleanExpression @ Graph ;
```

<u>BooleanExpression</u>: It is the boolean expression to be find among the edge properties in a graph. This boolean expression supports and (&&), or (||) and tilde (~) symbols where the tilde is the “not” expression here.

<u>pathLength</u>: This keyword is used for defining the length of the path which desired to be found. The length of a path is defined as

the number of edges between the connected vertices in a path. It can be used in BooleanExpression.

In BooleanExpression, user may use brackets, "(" and ")", to ask a specific vertex or edge's particular property value. It starts with the "v" for vertex or "e" for edge after the first opening bracket and continues with the numeric order of that edge or vertex and then the desired property with its value in another brackets. They are all seperated with ",". For example: ("graph1" is an example graph here)

```
query  = Q: BOOL? (pathLength == 3) && ( v, 1, ("name", "CS") )
                                    && ( e, 2, ("code", "315") )
                                    && ( v, 4, ("kind", "rulez!") )
                                    @ graph1 ;
```

Another query can be defined with arithmetic expressions and functions to find whether the path exists. The "EXIST" keyword is the attribute for this purpose. For example:

```
query  = Q: EXIST ? ArithmeticExpression | Function |
PropertyValue |  PropertyName | Property @ Graph ;
```

<u>ArithmeticExpression</u>: It is an expression which is defined for integers and floats. The arithmetic expressions are searched among the edges' property values. "*" indicates multiplication, "/" indicates division, "+" indicates addition, "-" indicates subtraction. For example: ("graph1" is an example graph here)

```
query  = Q: EXIST ? (3*4 + 12) / 5 @ graph1 ;
```

<u>PropertyName and PropertyValue</u>: Language also supports searching property names or other value types such as strings, but only for existence not for arithmetic operations. This example of query1 finds a path if there are edges with the string value "Ali" and query2 finds the path of edges with the property "age": ("graph1" is an example graph here)

```
query1  = Q: EXIST ? "Ali" @ graph1 ;
query2  = Q: EXIST ? "age" @ graph1 ;
```

<u>Property</u>: It is also available to find the existence of a specific property's value among edges. The given property is defined between "(" and ")" and the value is seperated from property name by ",". For example: ("graph1" is an example graph here)

```
query  = Q: EXIST ? ( "age", 30 ) @ graph1 ;
```

Function: It is a function that is already defined in the language such as “startsWith” which searches among the edges for string values that starts with the given character. Other functions that are properly defined, can be used like this to find different edges. For example: (“graph1” is an example graph here)

```
query  = Q: EXIST ? startsWith( “surname”, “B” ) @ graph1 ;
```

b) Variable Path Queries:

Variables can be used for finding paths with edges. If all the edges in a path contains that specific variable as their properties and if only if their values are same, the query will return that path. “SAME” keyword is used for this purpose. After the declaration of query attribute, user can enter the property that is desired. For example: (“graph1” is an example graph here)

```
query  = Q: SAME ? “age” @ graph1 ;
query  = Q: SAME ? “name” @ graph1 ;
```

c) Modularity:

Modularity is also supported by this language for regular path queries. Different regular path queries can be combined in another query as desired. For example:

```
query1  = Q: REGEX ? (A | B)A+ @ graph1 ;
query2  = Q: REGEX ?  B* @ graph1 ;
query3  = Q: REGEX ? (query1) _ (query2)+ @ graph1 ;
```

here, query3 defines the concatenation of query1 and query2 which is applied on a graph called “graph1”.