EKA Graph Definition and Querying Language Tutorial

EKA: The name of the language

What are we doing?

This language is used for defining graphs with vertices, edges and queries. It will meet the followings:

Key Words and Symbols

grd: <type> directed graph

gru: <type> undirected graph

vertex: <type> vertex

edge: <type> edge

query: <type> query

==: Boolean operator. Determines if the equality of the elements.

<-: Assign operator. Assigns the data on right hand side to left hand side.

->: Indicates the direction between vertices from left hand side to right hand side, if it exists.

--: Indicates that there is no direction between vertices.

§: Symbol for querying.

£ and /£: Symbols of starting and ending of an edge

“[“ and “]”: “[“ is the starting and “]” is the ending of the vertex when defining a query.

$i: The query performs on “include” mode.

$s: The query performs on “starts with” mode.

$e: The query performs on “ends with” mode.

$d: The query performs on “substring” mode.

$p: The query performs on “predicate” mode.

For further information see the parts below.

Note that: Concatenation, alternation and repetition operations are defined in tutorial below the "**Defining Query**" title.

1) Defining Primitive Types: Integer, String, Float

1.1) Integers and Floats

Integer and float numbers supports arithmetic operations, such as addition, subtraction, division and multiplication. These operations are left associative.

```
num <- 1+3 -2 //addition and subtraction
num <- 10/5 //division
num <- 3*2+1 //first multiplication then addition (left associative)
```

1.2) Strings

String are represented between " and ". We can slice string by string[int : int] operation as shown below.

```
myString <- "eren"
myString[0:1] // result is e
myString[:2] // result is er
```

2) Defining Map, List, Set and Variables

This language will support list, set, map and they will be defined as followings:

2.1) Map

Maps are containers which are associative and edge property values support maps. They can be considered as an unordered set of key which means ("name" : value) pair, names and values are separated with ":" symbol. Maps are defined in between "{","}". The following example states the definition of a map.

```
map myMap <- {("string" : "asd"), ("key1" : 1), ("key2" : 2), ("key3" : 3) }
```

2.2) List

List are sequence container and edge property values supports lists which are ordered collection of elements. Lists are defined in between "[","]".The following example states the definition of a list.

```
list myList <- [1,2,3,4,5]
```

2.3) Set

Sets are containers that store unique elements and edge's property values support sets which are unordered collection of unique elements. Sets are defined in between "(" ,")". The following example states the definition of a set.

set mySet <- (1,2,3,4,5,"example")

3) Defining Vertices

Vertices are the information storing nodes of the graph. They can have single or multiple properties. Each property has a name and corresponding value.

property = name ":" value

Name can only be string and value can be float, int, string, map, list or set.

vertex v1
v1 <- {("Name":"Woz"), ("Profession":"cs student") }
v1<- {"University":"Bilkent", ("Language" : {"English" : "true","Turkish" : "true", "German" : "false"})}

4) Defining Edges

Edges define the relationship between vertices. Also they can have single or multiple properties. Each property has a name and corresponding value.

A property consists of name ":" value

Name can only be string and value can be float, int, string, map, list or set. To define the relation between the vertices for the directed graph "->" is used and direction is to the left hand side to right hand side, for the undirected graph "—" is used.

vertex v1,v2

edge v1-> v2 {("School": ["Bilkent","METU"]),("Departments" : {"Primary": "CS",
"Secondary" : "EE"}), ("Ages" : (20,25,30)) }

5) Defining Directed and Undirected Graphs

In this language, directed and undirected graphs will be defined based on graph definition.

grd myGraph // grd stands for graph directed
gru myGraph // gru stands for graph undirected

5.1) Defining Directed Graph

In order to define directed graph we should use “grd" keyword and it consists of vertices and edges. It is defined as followings:

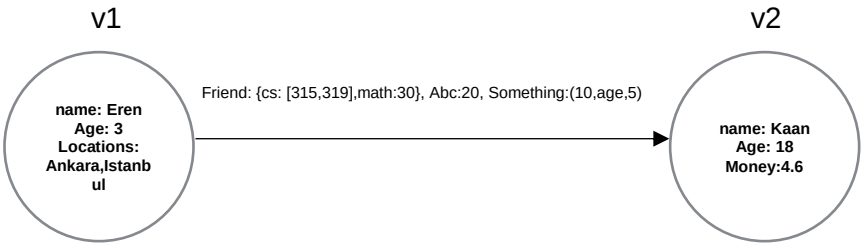
grd myGraph{

vertex v1 <- {(“name “ : “Eren”), (“Age “ : 3), (“locations “ : [“Ankara”, “Istanbul”]) }

vertex v2 <- {(“name “ : “Kaan”), (“Age “ : 18), (“Money “ : 4.6) }

edge v1->v2 {(“Friend” : {“cs”: [315,319],”math230”:[30]}), (“Abc” : 20), (“Soemthing”: (10,”age”,5)) }

}



5.2) Defining Undirected Graph

In order to define undirected graph we should use “gru" keyword and it consists of vertices and edges. It is defined as followings:

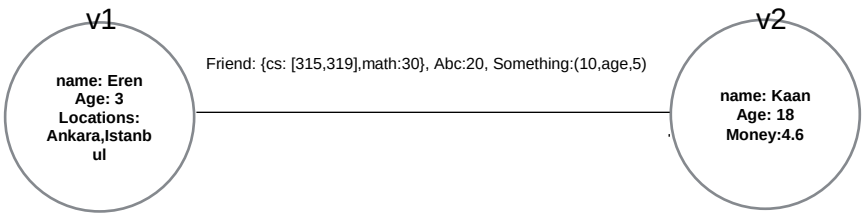
gru myGraph{

vertex v1 <- {(“name “ : “Eren”), (“Age “ : 3), (“locations “ : [“Ankara”, “Istanbul”]) }

vertex v2 <- {(“name “ : “Kaan”), (“Age “ : 18), (“Money “ : 4.6)}

edge v1--v2 {(“Friend “ : {“cs”: [315,319],”asd”:”math230”}),(“age” : 20), (“qwe” : (10,”age”,5))}

}



6) Defining Query

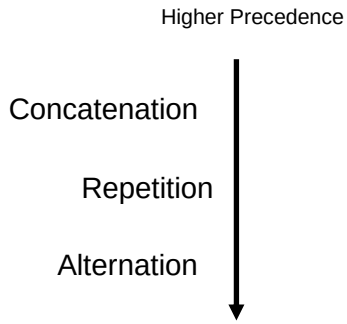
Our language supports concatenation, alternation and repetition. There is no symbol for concatenation, “|” is used for alternation and “*” or “+” used for repetition.

6.1) Using Concatenation Operator: This operator requires both sides to be vertices or edges. It concatenates the previous and following elements.

6.2) Using Alternation Operator: This operator uses “|” symbol and requires both sides to be vertices or edges. It means previous element or following element can be selected but not both.

6.3) Using Repetition Operator: There are 2 operators for this operation. “*” symbol corresponds to 0 or more times and “+” symbol corresponds to 1 or more times. Both of the repetition operations require an edge or vertex to be only on the left side.

Concatenation has the highest precedence of all of them. And repetition has lower precedence than concatenation but has higher precedence than alternation, so alternation has lowest precedence.



This language supports query on graphs. In order to query, first, you need to write the graph’s name then put the “§” symbol followed by the querying parameters. Before writing an edge’s querying parameters you must put “[“ then end of the properties you must put “]”. To write querying parameters for the edge you must put “£” at the beginning and “/£” at the end.

The check for existence can be made by using “ == “ symbol. This symbol must be placed between the name and the property that has been queried for and if the purpose of the query is checking not existence, then “!=“ must be used.

gru myGraph

myGraph § [“name” == “Eren”] £ /£ [ ] £ “age”==4 /£ [“school”==“Bilkent”]

This example queries for “name”=“Eren” then ignores the first edge and vertex that is occurred (since between £/£ is and [ ] is empty) then looks for the edge which has “age”=4 then looks for the vertex which has “school” =“ Bilkent”.

Regular Expression of Query: The behavior of the query is like that in general: V(EV)*E? , where V means that vertices, E means that edges and the other have already defined above.

Also you can create a query by passing some values for edges and values for vertices as shown in example.

query q1
q1 <- [“name”==“example”] £”Location”==“Ankara”/£

This creates a query which is looking for a vertex which has property “name”=“example” and an edge which has property “Location”=“Ankara”. Also you can use your query as in the example after creating it.

graph1 § q1

Also you can add multiple queries or look for that query 1 or more times.

graph1 § q1+ // querying for q1 1 or more times.
graph1 § q1* //querying for q1 0 or more times.
graph1 § q1 q2 // concatenation q1 with q2 then querying them.
graph1 § q1 | q2 // alternation querying for q1 or q2

6.4) Using “*” and “+” Symbols For Querying

6.4.1) Using “+”: query looks for the query q1 to occur 1 or more times.

query q1 <- [“name”==“example”]
graph1 § q1 +

6.4.2) Using "*": query looks for the query q1 to occur 0 or more times.

query q1 <- ["name"=="example"]
graph1 § q1 *

6.5) Using Concatenation For Querying

Since there is no symbol for concatenation, when you write two queries back to back separated by one space it concatenates the queries.

query q1 <- ["name"== "example1"]
query q2 <- ["name"== "example2"]
graph1 § q1 q2 // q1 q2 corresponds to the query ["name== "example1"]
// ["name"=="example2"]

6.6) Using "|" Symbols For Querying

This symbol is used for alternation operation. This symbol must be placed between queries.

query q1 <- ["name"== "example1"]
query q2 <- ["name"== "example2"]
graph1 § q1 | q2 // query for q1 or q2

Results for these operations are also queries. Thus you can perform such operations.

query q1 <- ["name"== "example1"]
query q2 <- ["name"== "example2"]
query q3 <- q1 | q2* q1

6.7) Using Predicate in Queries

Our language supports predication queries. Simply putting "$p" symbol after "[" or "£" will cause it to predicate rest of the statement. The "$p" must be placed before the name of the edge or vertex.

myGraph § [$p “name”==“value”]£”edge”==1 /£

myGraph § [“name”==“value”]£ $p “edge”==1 /£

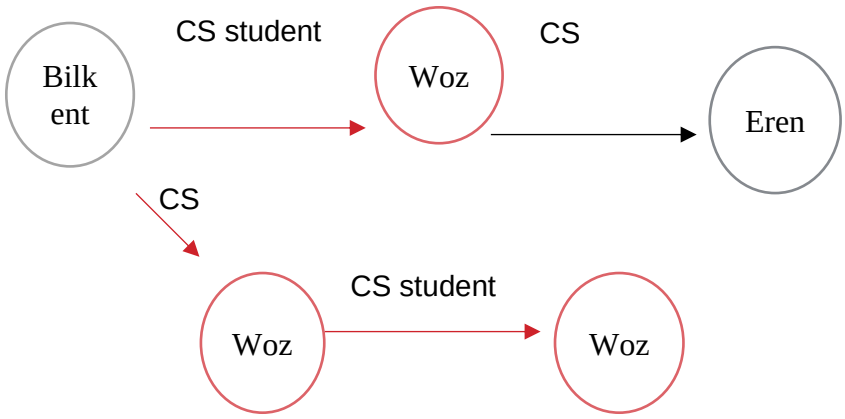
6.8) Using These Symbols For In Querying Statements

These operations have similar results for using them on queries. For example;

grd myGraph

myGraph § [] (£”profession” == “CS student”/£ [“name” == “woz”])+

In this example it ignores first vertex and looks for any edges which has a property “profession”=“CS student” if it is found, it looks for a vertices which has property “name”=“woz” and does this 1 or more times since there is “+” at the end.



In this example our query will find results for vertex “name=woz” and edge “profession=CS student”. So this query finds all CS students which has the name “woz”.

6.9) Using Variable Type Queries

Our language supports variables in queries. You can put any variable that’s type is supported by our language in place of any key or property.

```
a <- “ name”
b <- 3
graph1 § [a==b]
```

Also you can query for unknown variable type. You must use “?” symbol before the unknown variable. For example,

```
a<- “name”
graph1 § [a== ?b]
```

You can use arithmetic variable operations inside query.

```
a <- “name”
b <- 5
c <- 4
graph1 § [a== b+c]
```

6.10) Using Threshold Type Queries

Our language supports querying for less then or more than for a given number. In order to do that you must put “<“ symbol for less than and “>“ symbol for more than after the name. Also you can use “<=“ symbol for less or equal , “>=“ symbol for more or equal.

```
graph1 § [“name” <= 4] // query for the value at the vertex that corresponds to key “name” is
                       // smaller or equal to 4
graph1 § [“name” > 10] // query for the value at the vertex that corresponds to key “name” is
                       // bigger than 10
```

6.11) Using “starts with” Type Queries

Our language supports querying for “starts with “arguments. In order to do that, after the name you must put $ sign then put the string.

```
graph1 § [“name” $s “asd”] // query for the value at the vertex that corresponds to key
                           // “name” is starts with “asd”
```

Also you can query for “ends with” arguments. It is similar to “starts with” argument.

```
graph1 § [“name” $e “asd”] // query for the value at the vertex that corresponds to key
                           // “name” is ends with “asd”
```

Our language can query for “including “ arguments. It queries for the value if it includes the query parameter. $i is reserved for this operation.

graph1 § [“name” $i “asd”] // query for the value at the vertex that corresponds to key
// “name” is includes “asd”

You can query for a particular part of the value. $d(interval) is used for this operation. The “interval “ can be a “int”, “int : int” or “:int” type.

graph1 § [“name” $d(1:3) “ab”] // query for the value at the vertex that corresponds to key
// “name” has “a” and “b” char at the location 1,2.

graph1 § [“name” $d(:3) “abc”] // query for the value at the vertex that corresponds to key
// “name” has “a”,”b" and “c” char at the location 0,1,2.

graph1 § [“name” $d(1) “e”] // query for the value at the vertex that corresponds to key
// “name” has “e” char at the location 1.

Also our language supports these query types for the edges. The symbols are same for the edges.

graph1 § [“name” $e “end”] £”from” $i “Turkey”/£