Bilkent University

Department of Computer Engineering

CS 315 Programming Languages

Designing a Graph Definition and Querying Language

Project Part I Report

Language Name: SAPL
Section: 01 Group Number: 13

Uğurcan AYTAR, 21200487
Umut Cem SOYULMAZ, 21201744
Yusuf AVCİ, 21202512

Introduction

SAPL is a graph definition and querying language. The language is used for defining graphs and specifiying vertices of the graph and also defining regular path queries on the graph. The syntax of the language is inspired from C++ programming language's syntax. The language supports defining directed and undirected graphs. Also programmer is able to define vertexes and specify properties of these vertexes. A property consists of a name and a value. In addition to this, programmer can attach multiple properties to a vertex. Edges are also defined by the programmer. In addition to definition, programmer can also attach properties to edges.

SAPL language also possesses support for string, integer and float primitive types. In addition to these primitive types, it has a support of list, set and map collection types.

In addition to those above, SAPL also supports regular path queries. With these queries, programmer is able to write regular expressions in order to specify a path in the graph. Queries possesses support for concatenation, alternation and repetetion. In addition to this, programmers are able to define filters in their queries. These filters will specify the rules while running the queries. Also, programmer is able define queries and assign them to variables to use them later.

We have addressed the each requirement as following ways:

1. a Defining directed graphs:

We have designed a java like syntax for defining variables in SAPL language. Graphs are defined as variables and there are some functions provided on these variables. The definition of graph variables is simply made by typing the type of the graph and stating a name for the graph.

Example:

```
beginmain
        Directed myGraph
endmain
```

1.b. Defining undirected graphs:

Defining undirected graphs are similar to defining directed graphs.

Example:

```
beginmain
        Undirected myGraph
endmain
```

In SAPL, each command is seperated by lines. When we consider defining graphs as simple commands we can see why they are on seperate lines.

1.c Defining vertex properties

Defining vertex properties are explained detailed in the 6th part of the tutorial. In this case, we have simply defined a Property type and assigned values to the type. In this part syntax is also similar to java. Also, the syntax in the examples given in the requirements are strictly followed. Properties are assigned as name value pairs between paranthesis. In order to match the property with an edge programmer has to call functions related to it. In this case we have also defined vertices as types in our language. These parts are explained in 7th part of the tutorial.

Example:

```
Vertex myVertex("ali, veli")
myGraph.AddVertex(myVertex)
Property myProperty = ("id", 1234)
Vertex myVertex2(myProperty)
myGraph.AddVertex(myProperty)
```

1.d Defining edge properties

We have considered the same syntax. In this case, we have defined edges as types in our language and the matching between property and an edge is similar to the procedure in the vertex case. The procedure is explained in the 8th part of the tutorial in detail. The main difference is an edge does not have to have a property. This is why, while defining an edge programmer doesn't have to use paranthesis.

Example:

```
Edge myEdge
Property myProperty = ("id", 1234)
myEdge.AttachProperty(myProperty)
myEdge.AttachProperty("ali", "veli")
```

1.e Dynamic type system for the property values

We have defined string, integer and float types in our language. The syntax is also similar to Phyton. Programmer can create variables and use them later. This is explained in 9th, 10th and 11th parts of the tutorial.

String example 1:

```
myString2 = "mesela"
```

String example 2:

```
myString = "example string"
```

Integer example 1:

```
myInteger2 = 1234
```

Integer example 2:

```
myInteger = 1234
```

Float example 1:

myFloat2 = 1234.1234

Float example 2:

myFloat = 1234.1234

Lists, maps and sets are also defined as types in our language. Lists are specified with curly bracketts and they are used in the same way as arrays used in java with few differences. Lists can hold any type of value that is defined in our language. This is explained in 12th part of the tutorial. The difference between a set and a map is, in sets programmer can not reach the value with specifying the index of the value. In this case, a set appears as an unordered version of sets.

Example list:

```
List alist = {123, 1234, “example list string value”}
alist[0]=12
alist.Add(13)
alist.Delete(13)
```

Example map:

```
Map amap = {(2, "two"), ("five", 5)}
amap["five"]=12
amap.Add(13, "thirteen")
amap.Delete(13)
```

Example set:

```
Set aset = {1234, "ali", "mehmet, ayşe"}
aset.Add(16)
aset.Delete(13)
```

2.a Creating regular path queries

We have designed regular path queries' syntax very similar to syntax of regular expressions. In this case, the travelling between paths are provided by "->" characters. Only one way travel is permitted in our language. If one wants to travel in boths ways, he/she can use alternation property of regular path queries. The explanation of the details of regular path queries are given in the 15th part of the tutorial. In this case, concatenation is done by using "|" similar to context free grammar. Programmer can use paranthesis to create seperate regions in the query and make alternations, repetetions and concatenations on these regions. Also, query is defined with respect to properties of edges. Programmer might not define any property or might not give any condition over name or values. In this case, the query travels over that rula as it is an epsilon closure. Also, user is able to check a part of a name in properties by simply giving the index of the character after "Contains" function inside "[" and "]" characters.

2.b. In addition to these above, we have used simple boolean expressions with combination of ">", "=" and "!" just like most of the languages. In this case, conditions are also written as properties(instead of properties).

2.c. Lastly, user can divide each region into seperate queries by simply defining new variables for each of them. By default our language provides modularity in everywhere.

Example queries:

Path apath = ()->("edge1", 1245)>()->()->()->("edge2", "value of edge2")

Path bpath = apath->()->("edge3", 3)

Path cpath = (apath|bpath)->()->("edge4", 123)

Path dpath = ((apath*|bpath*)->()->("edge5",1234))*

Path epath = (Contains: "edge1")->(!Contains: "edge1")

Path fpath = (name=="edge1")->(value>13)*->()->("edge3", 123)

Path gpath = (Contains[0]: "e")*