The Harika Programming Language is an efficient and simple graph definition/query language. The language is designed to be highly capable for representing relationships between data with graph.

The language must be simple. This means that the language should support complex operations while still having an appealing syntax. Harika manages to incorporate both of these features by being a Dynamically Typed Language. The interpreter does most of the work while letting the programmer work freely and independently.

## Why Use Harika?

## Graphs are a very abstract concept, which means that they run the danger of meaning something only to the creator of the graph. Often, simply showing the structure of the data says very little about what it actually means, even though it’s a perfectly accurate means of representing the data. Everything looks like a graph, but almost nothing should ever be drawn as one.[[1]](#footnote-1)

## Harika offers simple mechanics to relate lots of data. While maintaining its easy to use properties, it falls no short of supporting complex operations. The users can define Graph types with simple parameters in addition to those already provided. Assuming you already know about graphs concept, Harika will be very easy to learn and use.

# Requirements Criteria

## 1) Support of Entry Point

Like the main() function commonly used in C-family programming languages, Harika uses the method main() as an entry point. When the application is invoked, the main() function instantiates the variables in the background. This instantiation process and the identifiers of the variables are all isolated from the user to provide a simple and clean interface. The code segment of entry point:

main() {

//do nothing

}

Here, curly braces are used to capsulate statements into a “scope”, just like the C.

## Dynamically Typed Language

Harika supports strings, floats, integers, characters and boolean as primitive data types.

Here are the illustrations of the declaration of primitive types and some supported operations:

Type “string”:

string love = "I love "

string course = "CS315"

string love\_course = love + course // “I love CS315”

“string” type also supports concatenation by using addition operator (+).

Type “int”:

int x = 15

int y = 18

int z = x + y

Type “char”:

char c1 = 'T'

char c2 = 'C'

string TC = c1 + c2 // “TC”

It is also supported to concatenate two chars into a string with the addition operator (+).

Type “bool”:

bool t1 = true

bool t2 = 1

“bool” type supports the usage of both “true”, “false” keywords and “0”,”1”.

Type “float”:

float a = 2.08f

float b = 7.92f

float c = a + b // 10.00f

Type “double”:

double k = 25.88

double l = 5.38

double m = k - l // 20.50

Harika also supports logical and arithmetic operations. These include; subtraction (-), division (/), multiplication (\*), modulus (%), and(&&), or(||), equal to(==), less than(<), greater than(>), less than or equal to(<=) and greater than or equal to(>=).

Examples:

int y = (x \* 2) - 4

double result = (x + y) / 2

int remainder = 10 % 4

bool and\_condition = t1 && f2 // false

bool or\_condition = t2 || f1 // true

Harika supports three collection types.

Lists

List is a collection type which is similar to an array.

int[] grades\_315 = [100, 80]

int[] grades\_101 = [90, 98]

Sets

A set is an unordered collection of items. Every element is unique (no duplicates) and must be immutable (which cannot be changed). However, the set itself is mutable. We can add or remove items from it. Sets can be used to perform mathematical set operations like union, intersection, symmetric difference etc. A Set can be created as follows:

set class = {"CS", 315, "Bugra Gedik", "Boran", "Eren", "Umitcan", 2017}

Maps

Creating a map is done by placing items inside curly braces {} separated by comma. An item has a key and the corresponding value expressed as a pair, key: value.

/\* This would be an example of a property, whose value is a map from strings to integers. \*/

map first\_grades = {"CS315" : 100, "CS101" : 90}

/\*This is an example where the value type is a map from a string to a list of integers. \*/

map all\_grades = {“CS315” : [100, 80], “CS101” : [90, 98]}

1. **Defining Vertex Properties**

Vertex is holding the data of the Graph. In Harika, it is possible to attach multiple properties to a vertex. A property should be a (name, value) pair. For instance, assume a vertex represents a student then the syntax will be the following:

Vertex student = ("id" = 21402338, "name" = "Eren", "grades" = {"CS315" = 100, "CS319" = 90})

An object of Vertex class is created with identifier student whose id is 21402338, name is Eren and a list of grades.

1. **Defining Edge Properties**

Edge is the connection of the vertices and it is very easy to specify an edge between two vertices.

After the creation of two Vertex objects, the edges are specified with the syntax following:

// undirected edge

student2 -- student3

// directed edge from student2 to student3

student1 -> student2

/\* student2 directs to student3 with weight 5, student3 directs to student2 with weight 8 \*/

student2 5--8 student3

/\* student3 directs to student1 with weight 6, student1 directs to student3 with weight default(1) \*/

student3 6-- student1

// student2 directs to student1 with weight 12

student2 12-> student1

## Defining Directed and Undirected Graphs

In Harika programming language, if there exists a directed edge between any two vertices in a graph, the graph is a directed graph. If there are not any directed edges, then the graph is an undirected graph.

The code below is an example of the instantiation of a directed graph.

Graph students

// generate student1 as a vertex

Vertex student1 = ("id" = 21402338, "name" = "Eren",

"grades" = {"CS315" = 100, "CS319" = 90})

// generate student2 as a vertex

Vertex student2 = ("id" = 21401947, "name" = "Boran",

"grades" = {"CS315" = 90, "CS319" = 90})

// generate student3 as a vertex

Vertex student3 = ("id" = 21402314, "name" = "Umitcan",

"grades" = {"CS315" = 90, "CS319" = 100})

// adds vertices to the graph

students.add(student1)

students.add(student2)

students.add(student3)

// edges initialization

student2 5--8 student3

/\* (undirected) student2 directs to student3 with weight 5, student3 directs to student2 with weight 8 \*/

student3 -- student1

/\*(undirected) student3 and student1 directs each other with weight default(1) \*/

student2 12-> student1

//(directed) student2 directs to student1 with weight 12

If the edge between student2 and student1 was not directed, the graph would be an undirected graph.

1. Ben Fry’s quote from https://dhs.stanford.edu/algorithmic-literacy/everything-is-a-graph-and-drawing-it-as-such-is-always-the-best-thing-to-do/ [↑](#footnote-ref-1)