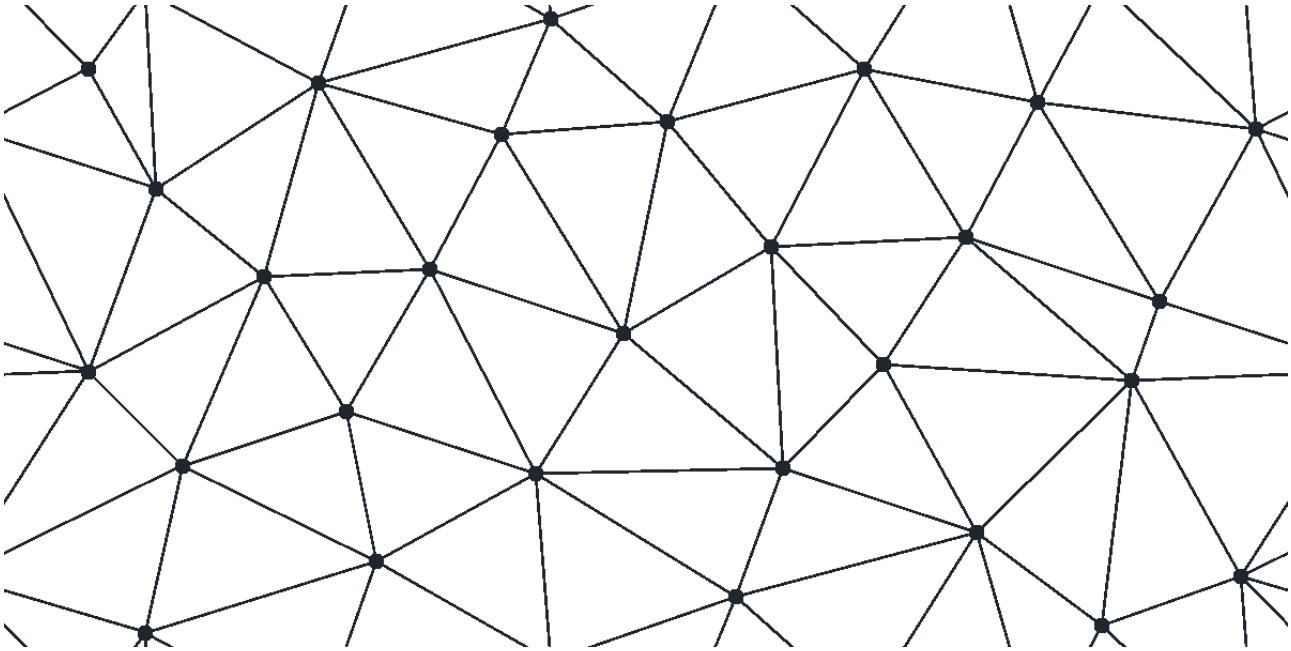
Harika Programming Lang.

Everything is a Graph and Drawing It as Such is Always the Best Thing to Do

Boran Yıldırım, H. Eren Çalık, Ümitcan Hasbioğlu CS 315 Group 1-2

In memory of Dennis Ritchie.



Documentation

1. Overview

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.

A. 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]

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.

B. Where Does the Name Come From?

In daily life, we thought it is a name that we use a lot and could easily be said by all the languages in the world. English provision of the Harika is "fantastic".

[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/

2. Introduction

A. Graph

In Harika programming language, if you want to create a graph you just type;

```
Graph names
```

and the magic starts.

After the Graph object is generated you can generate vertices and edges between them and add them to Graph object. Graph object is like a container of vertices and edges. In the following chapters, we will be discuss more about this concept.

B. Comments

Harika incorporates C style comments, because world is built on C. This way, developers coming from other languages will grasp the comment syntax easily. Examples are shown below:

```
//this is a single line comment

/* this is a multi
   line comment   */
```

C. Entry Point

Like most programming languages, Harika provides an entry point for the application. 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 shortest Harika program is written as follows:

```
main() {
     //do nothing
}
```

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

D. Concept of Functions

Harika supports modularity at the level of functions. The example below shows defining a function called sendLove() with return type void. This function gets called from the main() function.

```
main() {
     sendLove()
}

void sendLove() {
     // do something
}
```

E. Primitive Built-in Types and Operations

Harika integrates the common primitive built-in types such as integer, float, string, char, boolean and double. Here are some examples of the primitive types and some supported operations:

```
// string ---------------
string love = "I love "
string course = "CS315"

string love_course = love + course      // “I love CS315”

// character ---------------
char c1 = 'T'
char c2 = 'C'

string TC = c1 + c2      // “TC”

// boolean ---------------
bool t1 = true
bool t2 = 1

bool f1 = false
bool f2 = 0

bool and_condition = t1 && f2      // false
bool or_condition = t2 || f1       // true
```

```
// integer -------------
int x = 15
int y = 18

int z = x + y       // 33

// float ---------------
float a = 2.08f
float b = 7.92f

float c = a + b     // 10.00f

// double ---------------
double k = 25.88
double l = 5.38

double m = k - l    // 20.50
```

In string, addition operator (+) is used for concatenation purposes. However, if you want to add a numeric type to your string for example, then you have to cast it into a String before concatenation. Examples are as follows:

```
//string Concatenation
string love = "I love "
string course = "CS315"

string love_course = love + course
//reads “I love CS315”

//Casting
string class = “This class is CS“
int courseNumber = 315
string course = class + toString(courseNumber)
//reads “This class is CS315”
```

The rest of the operations are subtraction (-), division (/), multiplication (*) and modulus (%). Examples for these operations are as follows:

```
//Addition, Subtraction, Division and Multiplication on
numbers
int x = 10
int y = (x * 2) - 4
double result = (x + y) / 2
//result will be 13
```

```
//Modulus
int remainder = 10 % 4 //remainder is 2
```

F. Logical Operations

Logical operations in the Harika are the similar to other programming languages. Examples are shown below:

```
bool t1 = true
bool t2 = 1

bool f1 = false
bool f2 = 0

bool and_condition = t1 && f2       // false
bool or_condition = t2 || f1        // true
```

G. Casting

There are built in methods for casting one type to another type. This is useful when the developer wants to concatenate numbers with strings or wants to have higher precision on division operations. All casting methods and their usages are shown below.

```
string str_pi = toString(3.14) // holds “3.14”
double dbl_pi = toDouble(piString) // holds 3.14
int int_pi = toInteger(pi) // holds 3
```

H. Decision Statements

This section will show that how to use if/else if/else.

```
if ( examGrade == 100 ) {
     //get A+
}
else if ( examgrade <= 99 && examGrade > 85 ) {
     //get A
}
else {
     //get F
}
```

I. Loops

Harika has a same for loop with C. In Harika, for loops are as follows:

```
for (int i = 0; i < 10; i++) {
     //10 iteration
}
```

This simple loop iterates from 0 to 10. The index is accessible by the **i** variable specified in the loop body.

While loop is also same with C. One example is as follows:

```
while ( attendance < 70 ) {
     //attend
}
```

J. Built-in Callection Types

- Lists

  You can think lists as an array in C which stores same type data inside of the list. You don't need to specify the size of the array, Harika does it for you. A List can be created as follows:

```
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 as simple as 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]}
```

K. Built-in Object Types

Harika comes with three predefined 1 Graph class, 1 Vertex class and 3 edge types. These classes are shown in the code segment below. In the Advanced chapter all these classes and their functionalities will be covered in detail.

```
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
```

```
// representation
/*
       w:5   w:8
    (2) ------- (3)
      \          / w:1
w:(12) \       /
        \     /
         v   / w:1
          (1)
*/
```

3. Advanced

A. The Concept of Graph, Vertex and Edge

- Graph

  In this context a graph is made up of vertices which are connected by edges. A graph may be undirected, directed or contains both of them.
  Simple add operation to a graph is the following:

  ```
  students.add(student1)
  ```

  When this add function is called, the memory location of the vertex is copied into Graph so that after the insertion, every operation on the vertex will also affect the vertex in the Graph.

- Vertex

  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.

- Edge

  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
```

B. Graph Querying

The graph querying language supports creating regular path queries. A regular path query is a regular expression specifying a path. A path is a series of edges.

For instance the following code segment finds all paths of length three (because there are three identifier) where the start vertex contains name = Umitcan, the second contains id = 21401947 and the last one contains name = Eren.

```
Path allPaths = students.queryPath("name" = "Umitcan",
                    "id" = "21401947", "name" = “Eren")
```

Check for existence of a vertex of specified variable:

```
bool isExist = students.isVertexExist(with: "name" = “Boran")
```

If there is a vertex which contains name = Boran then function returns true, otherwise false.

Finding the Vertices which have a certain property whose value is a string that starts with some character.

```
Vertex[] verticesA = students.getVerticesStarts("name", ‘A’)
```

This one line code finds the all vertices which has a property “name” and starts with ‘A’ and returns a list of vertices.

4. Conclusion

In conclusion, Harika Programming Language incorporates a Dynamically Typed Language's advantages with a very clean and simple syntax design. Since the language is centred solely on graphs, developer is able to create highly specialised graphs. Combining simplicity, regularity, ease of use and high functionality is the main purpose of developing Harika. Harika is an amazing tool for any developer willing to work with graphs. Everything about graphs is implemented for you, you just start to run Harika and see the magic.