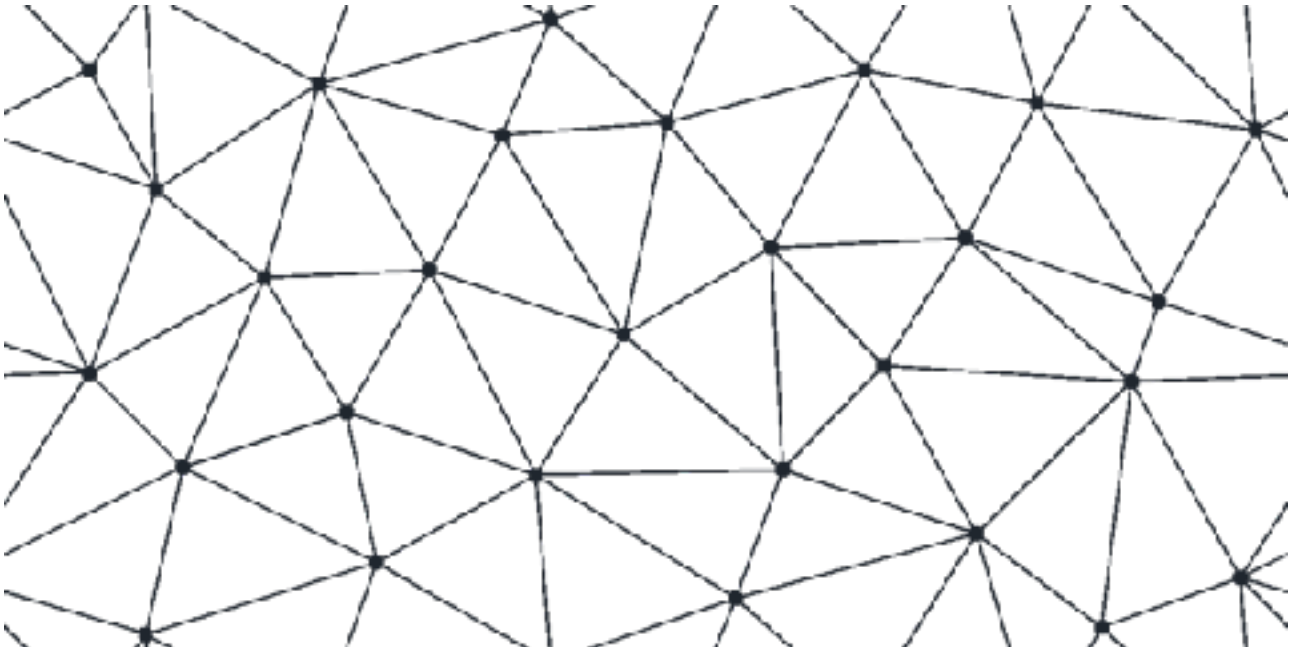
Harika Programming Lang.

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.



Report

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 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]

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.

[1] Ben Fry's quote from https://dhs.stanford.edu/algorithmic-literacy/everything-is-a-graph-and-drawing-itas-such-is-always-the-best-thing-to-do/

1) 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]}
```

2) 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, the value can also be a map. 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.

3) 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
```

4) 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.

5) Loops

Harika has two important form of loops which are for and while. For these two loops, the examples are below. Their functionalities are same, however the only difference is that they are taking different parameters to perform the same functionality.

```
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:

```
int i = 10;
while ( i < 10 )
{
     …
     i  =  i-1;
     //10 iteration
}
```

6) **Conditional Statements**

In Harika, there are two decision statements. For "if" statement, it executes first and check whether statement is true or not. If the statement is true, Harika executes partA, than go to end of the decision statements which is PartD. If it is not true, check for the else if statements with respect to their order (top to bottom) . Same rules of "if" statement are applied to "else if" statement. Lastly, if none of the statements("if" or "else if") are true, else statement executes according to "if" statement's rules.

```
if ( examGrade == 100 )
{
     //the perfect score
}
else if ( examgrade <= 99 && examGrade > 85 )
{
     //have the passing grade
}
else
{
     /not enough grade to pass the course
}
```

7) **Graph Querying**

Harika supports graph querying, including regular path queries. For instance the following code segment finds all paths of length three (because there are three identifiers) where the start vertex contains name = Umitcan, the second contains id = 21401947 and the last one contains name = Eren.

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

The parameter of queryPath(value) can include any identifier and there could be as many identifiers as it is needed.

Also, there is the check vertex query, which checks for the 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.

Harika also contains a query which finds 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.

There is another query which finds all the paths with the given length.

```
Vertex[][] paths = students.findPath(5)
```

It finds the paths of length 5 and returns a list of vertex list which is the list of paths.

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 centered solely on graphs, developer is able to create highly specialized 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.