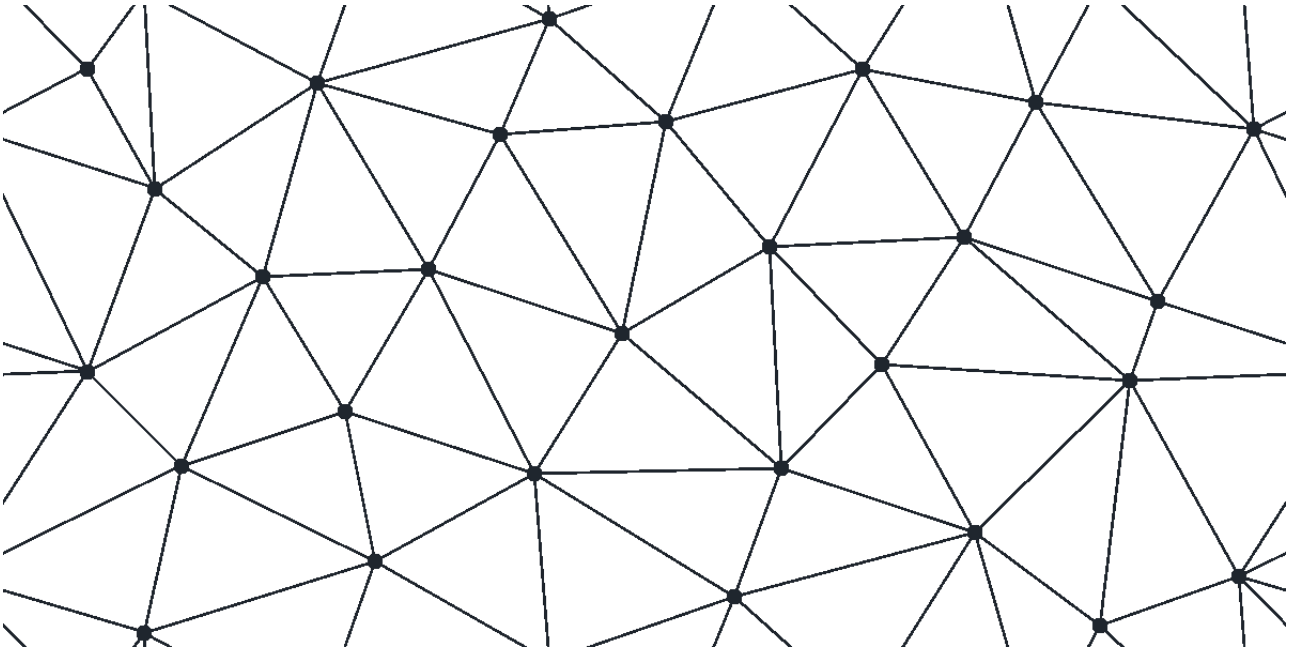
Harika Programming Lang.

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

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



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

Harila 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, Division and Multiplication on numbers
int x = 10
int y = (x * 2) - 5
double result = (x + y) / 2
//average will be 15
```

```
//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:

  ```
  // list ----------------
  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 -----------------
  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" = 1996, "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 a Bounding Box

In Ellie Programming Lang, the concept of a bounding box implies that; if a shape does not have a size, than it should be drawn inside the box which bounds it. In our case, this is the application window. If the shape does not have a specified Size, then it will be drawn to fit the bounding box (See section 4.A for concept of optional parameters).

B. Drawing an Oval

In the following example, we will be drawing 10 ovals next to each other, fitting all in the bounding box. First, we create an Oval, than we call its drawElli() method with the specified parameters.

```
     Ellie oval = Oval();
     /* draw 10 ovals next to each other horizontally, without
specifying absolute values for width and height */
     oval.drawEllie(count: 10);
```

```
     The next example shows drawing a single Oval inside the
bounding box using no parameters at all.
```

```
     Ellie oval = Oval();
     oval.drawEllie();
```

This is as simple it could get when drawing shapes with the Ellie Programming Lang. The power of optional parameters really show here. Drawing shapes with more parameters will be shown in the part of this section called Drawing a Rectangle.

C. Drawing a Rectangle

In the following example, we will be drawing a rectangle using all available parameters of the drawEllie() method.

```
Colour rectColour = Colour(r: 70, g: 90, b: 155);
Loc rectLoc = Loc(x: 10, y: 25);
Size rectSize = Size(height: 30, width: 10);
Ellie rect = Rect();
rect.drawEllie(filledState: True, fillColour: rectColour,
loc: rectLoc, size: rectSize, count: 2);
```

This example draws two rectangles horizontally with colour **rectColour**, location **rectLoc** and size **rectSize**. And the rectangles are filled with the **rectColour** as well since **filledState** is True.

D. Drawing a Line

Being different than other shapes, line takes one arbitrary parameter called direction. The accepted forms of this parameter are limited to “S”, “N”, “E”, “W”, “SE”, “SW”, “NE”, “NW”. An example for drawing a line is shown below.

```
     Loc lineLoc = Loc(x: 22, y: 90);
     Size lineSize = Size(height: 100); //width is an optional
parameter for the Size class
     Ellie line = Line(direction: "SW");
     line.drawEllie(loc: lineLoc, size: lineSize);
```

This example draws a line starting from location lineLoc and continues for a height of lineSize. The direction is specified as “SW” as seen in the Line constructor.

E. Drawing a String

Ellie language has different types called String and EllieString. String is for primitive operations on characters and EllieString is for drawing Strings. An example is as follows:

```
     Loc stringLoc = Loc(x:23, y: 88);
     Size stringSize = Size(height: 20); //the height of the text
is 20 pixels. Width depends on the height of the string
     Ellie stringDrawing = EllieString(“Hello, Ellie.”);
     stringDrawing.drawEllie(loc: stringLoc, size: stringSize); //
colour parameter is not provided here since it is an optional.
Defaults to black.
```

This example draws a String saying “**Hello, Ellie.**” with size **stringSize** and location **stringLoc.**

F. Drawing Custom Shapes

While supporting all the built-in shapes, Ellie Programming Lang. also supports defining custom shapes. However, definition of custom shapes requires an understanding of how the Core Features of the language work. Therefore, everything about drawing custom shapes is left for the next section.

In the next section, you will find the example of drawing a custom shape called a Snowman. Moreover, the next section covers some powerful language capabilities.

4. Core Features

A. Optional Parameters

Ellie Programming Lang. supports optional parameters like the Python Programming Language. The user could call a function without supplying all necessary parameters.

```
drawBugra(filledState = False, fillColour = Colour.Black, loc =
Ellie.DefaultLoc) {
     //do something
}
```

B. Intelligent Class Constructor

In many programming languages, developers need to write code to copy constructor arguments into the private variables in the language. In Ellie however, the interpreter automatically copies the constructor arguments into automatically created class variables. Here is a comparison with the Java Programming Language:

```
//assume Size and Loc classes are defined
public class Snowman extends Ellie {
     String message;
     Size snowmanSize;
     Size headSize;

     public Snowman(String message, Size snowmanSize) {
          this.message = message;
          this.snowmanSize = snowmanSize;
          this.headSize = new Size(snowmanSize.getHeight() *
          0.4);
     }
}
```

In Ellie, however,

```
Snowman -> Ellie {
     Snowman(message, snowmanSize); //Ellie class constructor
     Size headSize = Size(height: snowmanSize.Height * 0.4);
}
```

is all that is needed. The constructor creates the private variables in the background, without requiring the user to type the code. If needed, additional private variables can still be defined after the constructor. In this case, headSize is a private variable in the Snowman class.

C. Defining Classes

In Ellie, classes could be defined as follows. All classes related to drawing shapes must inherit from the Ellie class, which contains the drawEllie() method. After overriding the drawEllie() method, the user could call the method with optional parameters. The following code example may look messy because it does not fit on an A4 sized paper. However, it showcases the power of intelligent Ellie constructors and simple custom shape drawing syntax. Please take time to read it and understand fully before proceeding.

```
     Snowman -> Ellie { //We will be creating a class called
Snowman, which extends Ellie. We say “->” instead of "extends"

          Snowman(message);

          Ellie head = Oval();
          Ellie body = Oval();
          Ellie message = EllieString(message); //message will
have the default optional parameters

          //this is the perfect spot to show the optional
parameters
          drawEllie(filledState = False, fillColour =
Colour.Black, loc = Ellie.DefaultLoc, size = Ellie.DefaultSize)
{ //Ellie.DefaultLoc is provided as Loc(x: 0, y:0)
               Size headSize = Size(height: size.Height * 0.4)//
this is the ratio of the head over the body in case the user draws
this inside a bounding box

               drawOval(fs: filledState, fc: fillColour, l: loc,
s: headSize);
               drawOval(fs: filledState, fc: fillColour, l: loc,
s: snowmanSize);
               drawString(message);
          }

          //Oval's width is the same as the height unless
specified. It is an optional parameter as well.
     }
```

```
     //this will draw a snowman with head size 4, body size 10 and
the following message
     Size mrSnowmansSize = Size(height: 10);
     Ellie mrSnowman = Snowman(message: "Frozen 2 is coming!");
     mrSnowman.drawEllie(filledState: False, fillColour =
Colour.Black, size: mrSnowmansSize);
```

Here, we defined a class called Snowman which inherits from Ellie. The constructor takes two arguments. The types of the constructors are not know until the constructor runs. These arguments are automatically copied into hidden private variables after the contractor runs. Then the program continues with creating two Oval objects called head and body. Then, the an EllieString object gets created. If the user has supplied anything other than a String for the message argument in the constructor, Ellie will throw an exception and terminate. This way, we do type checking and at the same time, utilise the intelligent class constructors while still keeping the code clean.

D. Shapes Have No Attributes

Ellie Programming Lang.'s mother class (see section 2.A) defines a class called Ellie which contains an abstract drawEllie method. Since all shapes inherit from Ellie, they must implement their own drawEllie method. The drawEllie() method takes a size: Size, a location: Loc, a colour: Colour and a fillState: Bool.

Therefore, Ellie allows a very flexible way of drawing any shape. An example with an Oval is shown below:

```
     //Example for drawing an oval with specified fillState,
     location, colour and dimensions
     Colour ovalColour = Colour(r: 70, g: 90, b: 155);
     Loc ovalLoc = Loc(x: 5, y: 3);
     Size ovalSize = Size(height: 10, width: 8);
     Ellie oval = Oval();
     oval.drawEllie(filledState: False, fillColour: ovalColour,
     loc: ovalLoc, size: ovalSize);

     //the simplest way to draw an Oval inside the bounding box
     Ellie oval = Oval();
     oval.drawEllie();
```

As the example shows, the Oval object's constructor takes no parameters. Everything is controlled by the drawEllie() method. Of course, custom shapes like the Snowman from section 4.C supports arbitrary parameters (like the message parameter).

E. Elegant Getters and Setters

The Ellie programming language takes a new approach for getter and setter methods present in other programming languages. Private variables can be accessed as shown in the following example:

```
//Create a location and size for an Oval
Loc ovalLoc = Loc(x: 5, y: 3);
Size ovalSize = Size(height: 10, width: 8);

//get accessors
Int ovalLocX = ovalLoc.X;
Int ovalSize = ovalSize.Height;

//set accessor
ovalSize.Height = 7;
```

Notice these accessors do not have get or set prefixes like Java Programming Lang.

5. Conclusion

In conclusion, Ellie Programming Lang. Incorporates a Dynamically Typed language's advantages with a very focused syntax design. Since the language is centred solely on drawing shapes, we were able to create a highly specialised design pattern. With the Core Features provided, we believe this is a competitive candidate over complex programming languages. Combining simplicity, regularity, ease of use and high functionality, Ellie is an amazing tool for any person willing to work with drawing shapes. Also with the possibility of creating custom shapes and using them like the predefined ones, Ellie is sure to administer the needs of all users.