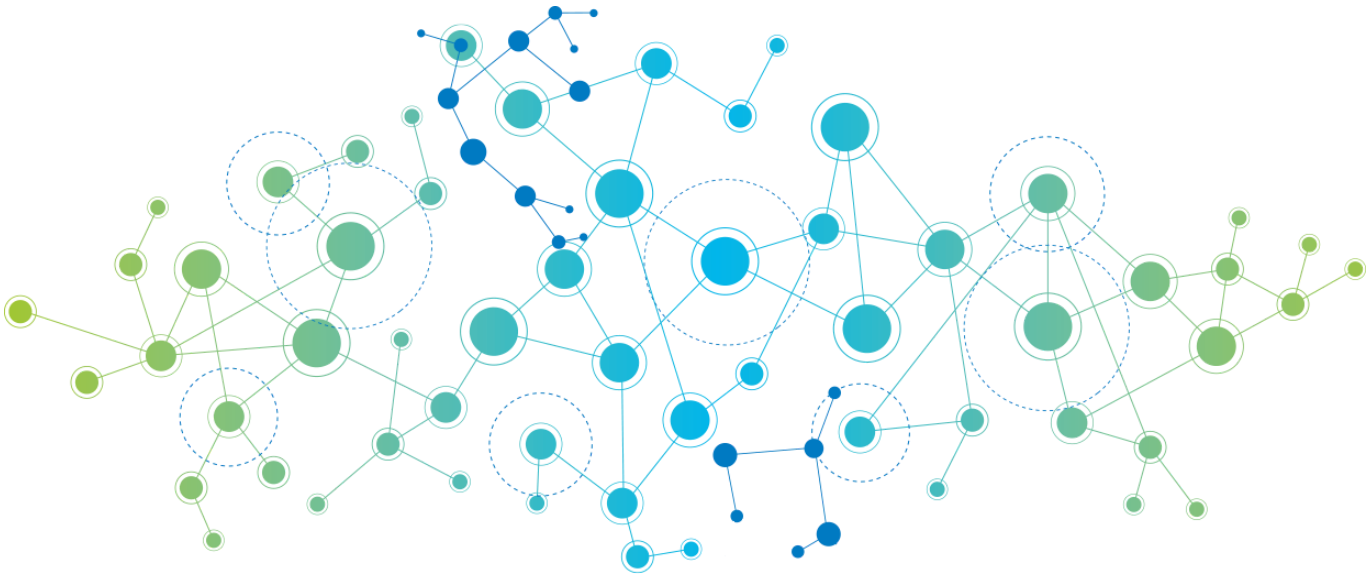

YAGL Yet Another Graph Language

Project Proposal

Adam Carpentieri ● AC4409 ● **Manager**
Jack Hurley ● JTH2165 ● **Tester**
James Mastran ● JAM2454 ● **Language Guru**
Shvetank Prakash ● SP3816 ● **System Architect**

Introduction **2**

Motivation **3**

Language Design and Syntax **3**

- Basic Features and Paradigms 3
- Primitive Data Types 3
- Derived Data Types 3
- Further Explanation: Primitive Data Types and their Supported Operators 3
- Further Explanation: Derived Data Types with their Primitive Operators 5
- Standard Library Functions to build using Graph Primitives 8
- Graph Library Algorithms 9
- Keywords 10
- Conditional and Logical Operators 10
- Functions 10
- Comments 11
- Control Structures and Flow 11

Example Code **11**

- Example 1: Creating Graphs, Find_All, Dijkstra, and BFS Operation 12
- Example 2: Implementing One of our Standard Library Graph Functions: find_all 13

Introduction

YAGL may be just that, Yet Another Graph Language, but it is unlike any other— hopefully. The pervasiveness of graphs in computer science makes them a great candidate to be added to the list of classical types that are widely used in other languages. This language aims to make implementing graphs and their algorithms much simpler and easier! While we are creating our own language syntax and design, we do plan on adopting some of Java's and C's syntax & features that we appreciate most.

Motivation

Graphs are fundamental in data structures and algorithms. They are ubiquitous and can be used to represent almost anything: social media connections, roads that connect cities, flights between cities, relationships or friendships, and many other mathematical & logical problems. Our language aims to simplify the use of graphs in computation by nicely wrapping many of the operations used in well known algorithms into a neat & compact syntax. Using these commonly used graph operations & operators as our building blocks, we hope to build a "Standard Library" that can easily implement many of the widely used graph algorithms.

Language Design and Syntax

Basic Features and Paradigms

- Statically typed
- Strongly typed
- Manual memory management (i.e. no garbage collection, must use "*free()*" and "*malloc()*")
- Strict evaluation
- Mutability

Primitive Data Types

`int, char, bool, float`

Derived Data Types

`Node, Edge, Graph, Array, String`

Further Explanation: Primitive Data Types and their Supported Operators

1. **int**: An int in our language is the same as pretty much any other language in which it can hold the value of an integer that is 4 bytes long.

   Assume we have an int a:

Instantiation	`int a = 6;` `int b = 5;`

Integer Arithmetic Operations
 	
Since our language is strongly typed, these operations require two ints and a float can not be re-interpreted as an int.
	`a + b;  /* evaluates to 11 */`
	`a - b;  /* evaluates to -1 */`
	`a / b;  /* evaluates to 1 */`
	`a * b;  /* evaluates to 30 */`

2. **char**: A char in our language is the same as a char in the C language (i.e. a 1 byte integer)

   Assume we have a char c1 and char c2:

Instantiation	`char c1  = 'a';` `char c2 = 'b';`

Character Operations (same as integer arithmetic operations)
 	
Since our language is strongly typed, these operations require two chars and a float or int can not be re-interpreted as a char.
	`c2 + c1; /* evaluates to 195 */`
	`c2 - c1; /* evaluates to 1 */`
	`c2 / c1; /* evaluates to 1 */`
	`c2 * c1; /* overflow for 1 byte*/`

3. **bool**: A bool in our language can be one of the boolean values (i.e. true or false). Classical logical operators similar to C can operate on booleans.

   Assume we have a bool a and bool b:

Instantiation	`bool a  = false;` `bool b = true;`
Boolean Operations	`a || b; /* evaluates to true`

We will support AND, OR, and NOT.	`*/`
	`a && b; /* evaluates to false*/`
	`!a ;  /* evaluates to true */`

4. **float**: A float in our language is the same as pretty much any other language in which it can hold the value of a number containing a fractional portion (i.e. a decimal or "floating" point number).

   Assume we have a float a and float b:

Instantiation	`float a  = 10.0;` `float b = 4.0;`

Float Arithmetic Operations
 	
Since our language is strongly typed, these operations require two floats and an int or char can not be re-interpreted as a float.
	`a + b;  /* evaluates to 14.0 */`
	`a - b;  /* evaluates to 6.0 */`
	`a / b;  /* evaluates to 2.5 */`
	`a * b;  /* evaluates to 40.0 */`

Further Explanation: Derived Data Types with their Primitive Operators

5. **Node**: A Node in our language is a collection of one or more attributes of any type. It is similar to a dictionary in other languages like Python. Nodes are essentially a reference to the first attribute in the dictionary.

   Assume we have a Node A:

Instantiation	`Node A(String attr1: "Name", int attr2: 22, bool attr3: True, type attr4: ...);`

Accessing and modifying attributes	`A.attr1; /* returns "Name" string */`
	`A.attr3; /* returns True bool */`
	`A;  /* returns all attributes */`
	
	`/* Will return whatever is stored in A's data variable. Returns the type of that data point. */`

6. **Edge:** An edge in our language connects two Nodes. All edges are directed and contain a source and destination Node. In addition, edges can hold an attribute of any type. Thus, our Edge type can be defined as a container holding a reference to a source Node, a reference to a destination node, and an attribute of any type. We will provide pseudo undirected edges by simply creating two directed edges with one in each direction.

   Assume we have an Edge E, Node A, and Node B:

Instantiation	Edges can only be instantiated within a Graph. See Graph below for how to create an edge. The edge's attr type is the same type as the Graph's type.
Access and Modify Source Node	`E.src; /* returns a source Node */` `E.src = Node C(String: "Pittsburgh);`
Access and Modify Destination Node	`E.dest; /* returns destination Node */` `E.dest = Node D(String: "Chicago");`

Access and Modify Edge's attribute
 	
The type of the attribute of an edge can be any type. The type of the edge is the same as the type defined by the graph instantiation.
	`E.attr = 50.0;`
	`E.attr; /* returns float 50.0 */`
	
	`E.attr = "abc";`
	`E.attr; /* returns String "abc" */`

7. **Graph**: A Graph in our language is datatype that is a container holding references to two arrays, namely a Node array and an Edge array.

   Assume we have a graph G and Node A, B:

Instantiation
 	
The <*type*> defines what type the edges' attribute is. Nodes can have many attributes, but the edge of a graph must be of a single type. If the type is not int, then the weight value for calculations is 1.
	`Graph G<int>;`
	`/* creates a graph where edges hold ints */`
	
	`Graph G<String>;`
	`/* creates a graph where edges hold Strings */`

Add a node to the graph
 	
Modifies graph G by adding Node A to it if Node A is not already in graph G. Duplicate nodes will be ignored.
 	
* You can delete Nodes too but this will be done with a library function.
	`G_new = G + A;`
	`G_new = G + A + B;`
	
	`/* This is how to add multiple Nodes at once */`
	`G_new = G + Array<Node>;`

Add bidirectional edge to the graph with attribute value x Only works on a single graph G (cannot add edges between two different graphs).	`G.A <->(x) G.B;`
Add directional edge to the graph with attribute value x Only works on a single graph G (cannot add edges between two different graphs). Cannot add two edges from A to B, if edge A to B already exists, return error.	`G.A ->(x) G.B;`

Retrieve array of all nodes in graph G	`G.nodes;`
	
	`/* returns an array of nodes */`

Retrieve array of all edges in graph G	`G.edges;`
	
	`/* returns an array of edges */`

Retrieve Edge between A and B in Graph G
 	
If the edge doesn't exist or if you are asking about two different graphs, return null.
	`G.A ? G.B;`
	`G: A ? B`
	`/* returns an edge */`

Retrieve array of neighboring Nodes to node A in graph G	`?G.A;`
	
	`/* returns an array of edges */`

8. **Array**: An array in our language is a contiguous chunk of memory storing multiple instances of the same type similar to C. However, our arrays will be dynamic in which the user can add and remove from the array without worrying about memory management. For this reason all arrays will be stored on the heap. This is similar to Java's ArrayList<>.

Assume we have an int array A:

Instantiation	`Array<int> A; // OLD`
	
	`int[x] A; where int is any type and x is a positive integer`
	
	`Sets aside contiguous amount of memory`

Updating an element
 	
Whenever you add an element and the array is full, our compiler will reallocate double the memory, copy over the existing array, and free the other memory.
 	
** The index argument is optional, if not provided, the element will be appended to the end.*
	`A.add(element, index); // OLD`
	
	`A[x] = y;`

Remove an element by index Removing an element of an array involves copying over all the elements to a new array and deallocating the original array.	`A.del(index);`
Return length of array (number of elements)	`A.length;`

9. **String**: A String is simply an array of characters in our language. This is similar to C, but String will be a built in datatype unlike in C where a header file needs to be included (i.e. <string.h>).

   Assume we have a String name:

Instantiation	`String name = "Bob";`
Add to a string Since our arrays are dynamic we can add to an existing string.	`name = name + " Meyers";`

Modify a character from a string	`/* name[index] = char */`
	`name[0] = 'R';`
	
	`name;/* returns "Rob Meyers" */`

Remove a character from a string (by index)
 	
Since our arrays are dynamic we can also remove from an existing string.
	`/* name.del(index); */`
	`name.del(0);`
	`name; /* returns "ob Meyers" */`

Standard Library Functions to build using Graph Primitives

Below are the functions we plan on implementing using our Graph primitive operators to build the "Standard Graph Library" of our language. These functions are commonly used in many graph problems.

`Graph add(Graph A, Graph B)` *Syntactic sugar:* `A + B;`	Creates one graph of two strongly connected components. Returns a new graph.
`Array<Graph> explode(Graph A)` *Syntactic sugar:* `A#;`	The explosion function. Split all SCCs into separate graphs. Returns an array of graphs.
`Array<Edge> shortest_path(Node A, Node B)` *Syntactic sugar:* `A ->? B;`	Returns the shortest path from A to B in G. If G's attribute type is int, this uses Dijkstra's algorithm. If G's attribute is not int or unweighted, all edges are of weight 1 and

	Dijkstra is equivalent to a BFS search. Returns a Path (i.e. an Array of Edges).
`Graph reverse_edges(Graph A)` *Syntactic Sugar:* `A^;`	Reverses all the edges in Graph A. Returns a graph.
`Graph breadth_first_search(Graph G, Node A, Node B)`	Returns a Graph that depicts the BFS traversal from A to B.
`Graph depth_first_search(Graph G, Node A, Node B)`	Returns a Graph that depicts the DFS traversal from A to B.
`Array<Nodes> find_all(Graph<type> G, Node src, <type> attribute)`	Returns all neighboring nodes of the source node (Node src) in Graph G such that the edge from src to any other neighbor has an attribute equal to attribute.

Graph Library Algorithms

These are some of the algorithms that will potentially be used to implement our Standard Graph Library functions above:

- Dijkstra's shortest path algorithm
- Bellman–Ford Algorithm
- Floyd Warshall Algorithm
- Johnson's algorithm for All-pairs shortest paths
- Shortest Path in Directed Acyclic Graph
- Shortest path with exactly k edges in a directed and weighted graph.

Keywords

Below are the reserved keywords in our language:

`for, while, in, if, else, BFS, DFS, int, char, bool, float, Graph, Array, String, Edge, Node, return`

Conditional and Logical Operators

We plan on implementing the following conditional and logical operators in our language:

`>, <, >=, <=`	These operators will operate on two ints, floats, or chars only. When comparing

	using these operators, the operands must be of the same type since our language is strongly typed.
`&&, ||, !`	These operators only operate on booleans and since our language is strongly typed no other types can be interpreted as bools. This is different from languages in which ints equal to 0 can be interpreted as false and anything else is interpreted as true for example.
`==, !=`	These operators will operate on all types in our language. For our primitive types, these operators compare the values. For our derived types, they will compare if they refer to the same underlying memory.

Functions

Functions in our language will follow the same syntax as in C:

```
return_type function_name (type arg1, type arg2, …, type argN){
      /* function body */
}
```

Comments

As seen in some of the code examples above, all comments in our language will follow the C multi-line comment style:

```
/* This is a comment
*/
```

Control Structures and Flow

We will be using many of the control structures similar to other languages (`if, else, else if` with `{ }`. We do not plan to have a switch/case as of now. In particular we plan on modeling the control structures of the C language such as the classic `for` & `while` loops and `if-else` controls. One addition we will be adding to our language is the "for each" control structure designed to make working with graphs easier.

```
for Edge e in p {
  /* Do something on e */
}
```

Moreover, much like how a for loop iterates through an array, Breadth First Search (BFS) is used to traverse nodes in a graph. We have also added the following control structure below to our language to make traversing graphs easier. It's required to give a graph, a starting node and the number of neighbor levels you want to visit. Node N will be updated at each step to the current node. Similarly, there will be a DFS control flow.

```
BFS(Graph G; Node N; int x) {
   /* execute statements */
}
```

Example Code

Below we have included some example code in our language with keywords bolded to delineate.

Example 1: Creating Graphs, Find_All, Dijkstra, and BFS Operation

```
Graph G<String>, cities<int>;

Node Pittsburgh(int pop: 500), Philly(int pop: 100),
     New_York(int pop: 8500), Boston(int pop:1000);
Node A(int age: 50, Node home: Pittsburgh, String name: "Jess"),
     B(int age: 19, Node home: New_York, String name: "John"),
     C(int age: 21, Node home: New_York, String name: "Jake");
```

```
G = G + A + B + C; /* Add Nodes A, B, C as isolated nodes to G */
cities = cities + Pittsburgh + Philly + New_York + Boston;

/* Make graph's edges. Since cities is type int, use int as attr */
cities.Pittsburgh <->(100) cities.Philly;
cities.Pittsburgh <->(120) cities.New_York;
cities.Philly <->(25) cities.New_York;
cities.New_York <->(110) cities.Boston;

/* Make G's edges. Since G is type String, use Strings as attr */
G.B <->("friends") G.C;
G.B <->("Never Met") G.A;

/* Get all of B's friends using find_all() */
Array<Node> friends = find_all(G, B, "friends");

/* Since Jake is John's friend and they both are in New_York the
/* following should print: "Jake lives in New_York which is 0
cost."
for Node x in friends {
   /* Use Dijkstra to get path cost between the friends */
   int cost = (cities.(B.home) ->? cities.(x.home)).cost;
   print(x + " lives in " + x.home + " which is " + cost + "
cost.");
}
/* BFS: returns a graph that depicts the BFS traversal*/
Graph G_path = breadth_first_search(cities, Pittsburgh, Boston);
Array<Edges> p = G_path.Pittsburgh ->? Boston;

int cost = 0;
/* The following should print: (Pittsburgh, New_York, 120)
                               (New_York, Boston, 110)
                               (Boston, null, 0) */
for Edge e in p {
   cost += e.attr;
   print(e);
}
print(cost); /* should print 230 */
```

Example 2: Implementing One of our Standard Library Graph Functions: find_all

```
Array<Nodes> find_all (Graph<String> G, Node src, String
filter_attr) {
   Array<Nodes> neighbors = ?G.src; /* ? operator returns
neighbors*/
```

```
   int pos = 0;
   for Node n in neighbors {
      Edge<String> e = G.src ? G.n; /* get edge between src and n
*/
      if (e.attr != filter_attr) {
         /* Do not increment pos because del will move all elements
            down in the array */
         neighbors.del(pos);
      } else {
         pos++;
      }
   }
   return neighbors;
}
```