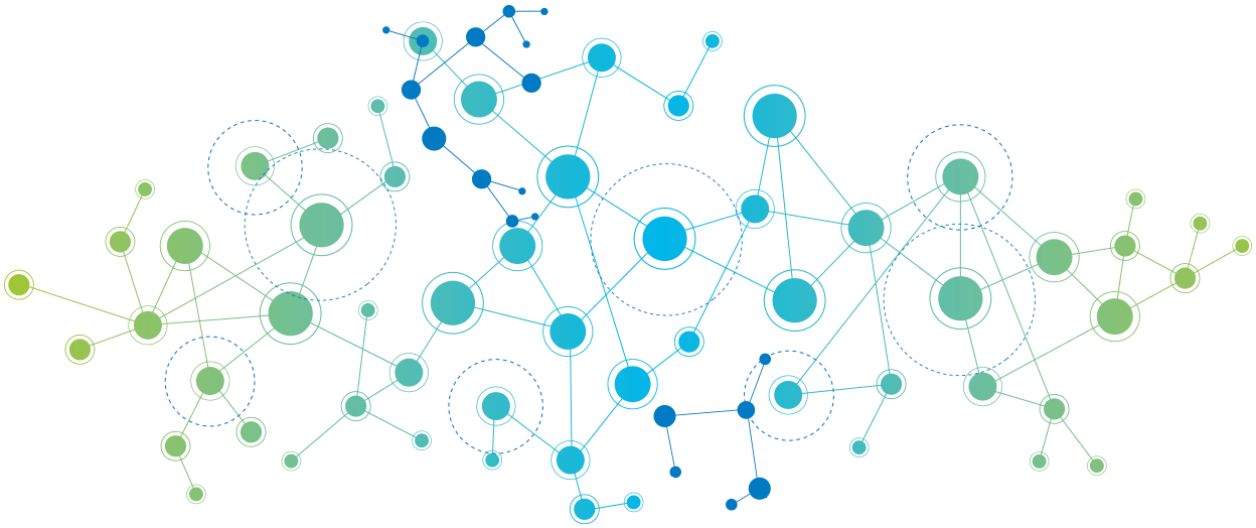

YAGL

Yet Another Graph Language

Language Reference Manual

Adam Carpentieri | AC4409
Jack Hurley | JTH2165
James Mastran | JAM2454
Shvetank Prakash | SP3816

1 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 C's syntax & features that we appreciate most.

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 have also built a Standard Library that has easily implemented many of the widely used graph algorithms.

2 Lexical Conventions

In YAGL, identifiers, keywords, constants, operators, and separators are all considered tokens. The lexeme associated with each token is composed of characters from the ASCII character set. Tokens must be separated using whitespace, comments, or any other kind of separator token.

2.1 Identifiers

Identifiers are used for naming and must begin with an alphabetical letter. This first character can then be followed by any sequence of digits, letters, and underscores. Names can be of any length but must be unique. Uppercase and lowercase letters are distinct. The regular expression for identifiers is:

```
['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']*
```

2.2 Keywords

The following words are reserved in YAGL for programming constructs, defining types, and special constants and may not be used in any other context such as naming:

```
while       int

if          char

else        bool
```

```
bfs         float
return      Graph
true        String
false       Edge
void        Node
            NULL
```

Each lexeme associated with each of these tokens above is the keyword's respective spelling.

2.3 Constants

There are 5 different types constants used to represent literals:

2.3.0.1 Integer Constants

Integer constants are composed of any sequence of digits (0-9) and an optional hyphen (-) for the sign of the integer to represent a literal between -2,147,483,648 to 2,147,483,647. All integer constants are interpreted as base 10 numbers. The regular expression for integer constants is:

```
['0' - '9']+
```

2.3.0.2 Character Constants

Character constants are represented using single quotes around a single character from the ASCII character set encoding. Sometimes two characters are needed in special cases to represent the character. In these cases the first character is a backslash which is also called the escape character in this context:

'\'' : single quote

'\n' : new line

'\r' : carriage return

'\t' : tab

'\b' : backspace

'\\' : backslash

The regular expression for these escape characters is:

```
"'" ('\\' ['b' 't' 'r' 'n']) "'"
```

The regular expression for all other ASCII characters is:

```
"'" [' ' - '~'] "'"
```

2.3.0.3 Floating Point Constants

Floating point constants are composed of an integer sequence followed by a decimal point and then a fractional sequence. The integer sequence can contain an optional hyphen to show the sign followed by any sequence of digits to represent an integer literal between 1.2E-38 to 3.4E+38. The decimal point is a simple period (.) . The fractional portion can be any sequence of digits to represent six decimal points of precision. The first two components (integer portion, decimal point) are required for the constant to be interpreted as a floating point. The regular expression for floating point constants is:

```
['0' - '9']+ '.'  ['0' - '9']* ( ['e' 'E'] ['+' '-']? ['0' - '9'] )?
```

2.3.0.4 String Constants

String constants are represented using double quotes “” around a string literal, which is simply a sequence of ASCII characters (including escape chars). The regular expression for string constants is:

```
'"' (ascii* escapeChars*)+ '"'
```

where the regular expressions for `ascii` and `escapeChars` are defined in 2.3.0.2 above.

2.3.0.5 Boolean Constants

There are only two kinds of boolean constants and these literals are reserved as keywords in the language: `true and false`. The lexeme associated with each of these two tokens is the keyword’s respective spelling.

2.4 Operators

The following characters are reserved as operators:

```
+       -       /

*       =       !

<       >       &&

||      ==
```

Each one of these is considered a separate token with the exception of – followed by an integer or floating point constant.

2.5 Separators

There are 9 separator tokens in YAGL:

```
(           )

[           ]

{           }

.

;

,
```

These are used in the language to separate code in various ways and each have a different use case outlined later in the LRM. A left parenthesis, square bracket, or curly brace needs to be followed eventually by a respective right closing character.

2.6 Whitespace

All white space (spaces, tabs, new lines) are ignored and not considered tokens. They are simply used to separate other tokens.

2.7 Comments

All comments begin with a `/*` and end with `*/`. Comments are ignored and not considered tokens **but can be used to separate tokens**.

3 Data Types

Data types are split into two different categories: primitive and derived. Primitives are the base data types and consist of `int`, `char`, `bool`, `float` and `void`. Derived data types are built from primitive data types. These types include `Node`, `Edge`, `Graph`, `Array` and `String`. **Note that bitwise representations of data types are not exposed to the user.**

3.1 Primitive

3.1.0.1 int

Used to store whole number values with a storage size of 32 bits. Integer types are stored in 2's complement and so the range is from -2,147,483,648 to 2,147,483,647. Characters (declared, and hereinafter called, char) are chosen from the ASCII set; they occupy the rightmost seven bits of an 8-bit byte.

3.1.0.2 char

Stores character values with a size of 8 bits. Character types are 1 byte unsigned integers and so their value range is from 0 to 255.

3.1.0.3 bool

Either stores the value `true` or `false`.

3.1.0.4 float

Stores floating point numbers (i.e. fractions/decimals). Floating types have a storage size of 32 bits and are formatted in IEEE 754 (1-bit for the sign, 8-bits for the exponent, 23-bits for the value). The value range is from 1.2E-38 to 3.4E+38 and has 6 decimal places of precision.

3.1.0.5 void

Specifies no value is available. It is used for functions to return nothing, functions to not take arguments and used to point to an address of an object but not its type.

3.1.0.6 Array

A contiguous chunk of memory storing multiple instances of the same type. All arrays are of fixed length and one-dimensional.

3.2 Derived

3.2.0.1 Node

Contains one or more attributes of any type (string, int, bool, etc.). Similar to a dictionary in Python.

Adding a Node to the Graph is shown in the example section.

3.2.0.2 Edge

Connects two nodes and contains a reference to its source and destination nodes. It will hold an int which can correspond to the edge's weight. All edges are directed, but you can "create" a bidirectional edge by having two edges connecting the same nodes but in opposite directions.

The information stored in an Edge is the source node, destination node, and some associated descriptor which is of type int. All these are accessible. For example, to access the destination node, source node, or attribute (weight) of an Edge E, it is simply E.dest, E.src, or E.attr, respectively;

Adding an Edge to the Graph is shown in the example section.

3.2.0.3 Graph

A Graph stores two sets of arrays: an array for the nodes contained within the Graph and an array of Edges for relationships between the nodes. To keep track of the size of the arrays, two ints are stored as well. These arrays are pointers.

Graphs are immutable thus when a new Graph is created it does not contain any nodes nor any edges and therefore the edges and nodes arrays point to null. When a node is added into a Graph, the graph mallocs enough space for the number of previous nodes plus 1. The old set of nodes are copied into the new allocated memory in addition to the new node. When an edge is added into a Graph, the graph mallocs enough space for the number of previous edges plus room for another one. The old array of edges are copied into the new allocated memory in addition to the new node. This new chunk of memory is stored in a new graph.

One can access the array of `edges` or `nodes` of a graph. For example, to access Graph G's array of nodes, it is simply `G.nodes;`

Adding nodes and edges to a Graph is shown in the example section.

3.2.0.4 String

An array of characters terminated with a null character. String types contain a pointer to the array with a size of 8 bytes.

4 Expressions and Operators

4.1 Unary

The unary operators are !, -

4.1.0.1 Accessor: variable.variable

The . operator is used on a variable of a specific type and accesses its internal data/variables.

4.1.0.2 Negation: !bool

The ! operator placed immediately before an expression is the negation operator. It works on `bool` types. If the `bool` is true, then the negation of the `bool` becomes false. If the `bool` is false, then the negation of the `bool` becomes true.

4.1.0.3 Negation: -expression

The - operator placed immediately before an expression is the negative operator. The result is the negative of the expression and works on `int` and `float` types.

4.1.0.4 Array accessor: []

The [index] operator is used after an array variable to access and dereference the contents of the memory location held in the index number (0 based) of the array.

4.1.0.5 expressions not supported

The ++ (increment) and -- (decrement) unary operators are not supported in our language since these are equivalent to `expression = expression + 1` or `expression = expression - 1`, respectively.

Bitwise operators are not supported at this time.

4.2 Binary

The binary operators are +, -, *, /, ==, <, >, =, &&, and ||.

4.2.0.1 Multiplication: expression * expression

The * operator with two expressions indicates multiplication. Both expressions must be of the same type. Therefore, this operator works on `int * int`, `float * float`, or `char * char` expressions. No other combinations are allowed. The result is another expression of the same type as the expressions used with this operator. This operator is left evaluated (grouped left-to-right).

4.2.0.2 Division: expression / expression

The / operator with two expressions indicates division. Both expressions must be of the same type. Therefore, this operator works on `int / int`, `float / float`, or `char / char` expressions. No other combinations are allowed. The result is another expression of the same type as the expressions used with this operator. This operator is left evaluated (grouped left-to-right).

4.2.0.3 Addition: expression + expression

The + operator with two expressions indicates addition. Both expressions must be of the same type. Therefore, this operator works on `int + int`, `float + float`, or `char + char` expressions. No other combinations are allowed. The result is another expression of the same type as the expressions used with this operator. This operator is left evaluated (grouped left-to-right).

4.2.0.4 Subtraction: expression - expression

The - operator with two expressions indicates subtraction. Both expressions must be of the same type. Therefore, this operator works on `int - int`, `float - float`, or `char - char`

expressions. No other combinations are allowed. The result is another expression of the same type as the expressions used with this operator. This operator is left evaluated (grouped left-to-right).

4.2.0.5 Equality: expression == expression

The == operator is the equal-to operation. It determines whether two expressions are equivalent of the form expression == expression and returns a bool type. One aspect to note is the lower precedence; for example, `a>b==c>d` is 1 if a > b and c > d or a < b and c < d (thus both have the same truth value). For comparing Nodes, Edges, or Graphs, it compares memory location. This operator is left-to-right evaluated.

4.2.0.6 Graph special operator: expression : expression

This operator is to define an expression to affect the graph on the left-hand side. The first expression must be a Graph and the second must be a valid operation on the Graph (such as the arrow operator or question mark operator). This operator is left-to-right evaluated.

4.2.0.7 Less-than: expression < expression

The < operator is the less-than operation. It determines whether the left side of the operator is less-than the right side and if so it yields true; otherwise, false. Both expressions must be of the same type. Does not apply to the bool type. This operator is left evaluated (grouped left-to-right).

4.2.0.8 Greater-than: expression > expression

The > operator is the greater-than operation. It determines whether the left side of the operator is greater-than the right side and if so it yields true; otherwise, false. Both expressions must be of the same type. Does not apply to the bool type. This operator is left evaluated (grouped left-to-right).

4.2.0.9 Arrow-operator: expression ->expression expression

The -> operator is the arrow operation. It is used to create an edge from the left-hand side directed towards the right-hand side expression. The expression in the middle is a literal (integer) number and is optional (defaults to 1). The middle expression must be an integer and the other two expressions must be a Node. This operator is left evaluated (grouped left-to-right).

4.2.0.10 Question-Mark-operator: expression ? expression

The ? operator is the question mark operation. Both expressions must be of the Node type. This operator follows the Graph-operator (:) and retrieves the attribute of the edge (an integer) between the first expression and the second expression (which are both nodes) in the graph. This operator is left evaluated (grouped left-to-right).

4.2.0.11 Assignment: expression = expression

The assignment operator groups right-to left, that is the right side of the binary operator is assigned to the left side. The assignment operator returns a value that is equal to the right side of the assignment and is of the same type. Both expressions must be of the same type, no other combinations are allowed.

4.2.0.12 OR: bool || bool

The || operator yields true if either (or both) of the booleans are true; otherwise, it yields false. This operator guarantees left-to-right evaluation.

4.2.0.13 AND: bool && bool

The && operator yields true if (and only if) both of the booleans are true; otherwise, it yields false. This operator guarantees left-to-right evaluation.

4.2.0.14 expressions not supported

Greater than or equal and less than or equal comparators.

4.2.0.15 boolean short circuiting

Boolean expressions are **not** short circuited.

4.3 Operators Precedence

The precedence order is unary operators, multiplication and division, addition and subtraction, comparators (<, >, ==), the and/or logical operators, and then lastly the assignment operator.

Parenthesis in logical evaluation or arithmetic can be used to override inherent precedence. For example 2 + 3 * 4 should evaluate to 14, but (2+3) * 4 should evaluate to 20.

The overarching precedence is in order of the major sections above (i.e. unary operations receive higher precedence over binary operations).

To summarize precedence (in order of increasing precedence):

Operator Name	**Symbol**	**Associativity**
Assignment	=	Right-to-Left Associativity
Or	||	Left-to-Right Associativity
And	&&	Left-to-Right Associativity

Equal Graph-operator	== :	Left-to-Right Associativity
Less-than Greater-than	< >	Left-to-Right Associativity
Arrow-operator	->	Left-to-Right Associativity
Plus Minus	+ -	Left-to-Right Associativity
Times Divide	* /	Left-to-Right Associativity
Not	!	Right-to-Left Associativity
QMark-operator	?	Left-to-Right Associativity
Array-access-operator	array[expr]	Left-to-Right Associativity
Accessor-operator	.	Left-to-Right Associativity

5 Functions

5.1 Functions

Functions in our language will follow the syntax:

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

The `return` statement is a mechanism for returning a value to the caller. Any expression can follow return:

```
Return expression;
```

Our language does not support functions as pointers.

5.2 Calling a function

Once a function is defined, it may be called in any place inside another function. The syntax is

```
function_name(arg1, arg2, …, arg n);
```

6 Statements

6.1 Declarations

6.1.0.1 Primitive Data Types

To declare a primitive data type, the syntax is `type var_name = expression of same type;`

6.1.0.2 Arrays

To declare an array, the syntax is `type[num_of_elements] name;`

6.1.0.3 Nodes

To declare a Node, the syntax is `Node name = {type name1 = value1, ..., type nameN = valueN};`

6.1.0.4 Edges

An Edge is not declared directly. It is declared by adding an edge to a graph. Assume a Graph G exists contain Nodes A, B and C, two edges from A to B (with weight 5) and B to C (with weight 7) can be declared as:

G: A ->5 B ->7 C

6.1.0.5 Graph

To declare a Graph, the syntax is `Graph name;` which will create a graph type and store its address in the variable `name`.

6.1.0.6 String

To declare a string, the syntax is `String name = "contents of the string";` Since Strings are fixed arrays, their length cannot be changed.

6.2 Statements

Statements are executed in their order written in the code.

6.2.0.1 Expression Statement

Any expression of the form `expression;` is a statement.

6.2.0.2 The BFS Control Flow Statement

The BFS statement is of the form:

```
bfs(Graph G; Node n; int x) {
      statement
}
```

Where n is the Node currently being iterated (originally, the start node) and x is the depth. Therefore, if it was desired to get all the neighboring nodes of the starting node, one could code:

```
node = n; //initialize start_node to node n so node n is not touched
int x;
Node[10] nodes;
bfs(Graph G; node; x) {
      if (x < 1) {
            nodes.add(node)
      }
}
```

6.2.0.3 Add Node to Graph

```
Node A = {int example : 5}
Graph G;
G = G + A;
```

6.2.0.4 Add Nodes to Graph

```
Node A = {int example : 5}
Node B = {int example : 5}
int nodes[2];
nodes[0] = A;
nodes[1] = B;
Graph G;
G = G + nodes;
```

6.2.0.5 Add Edge to Graph

```
Graph G;
G: h ->{5} e;
```

Note: Graphs are immutable, so this makes a copy of G and adds the edge and sets G to point to the updated version.

6.2.0.6 Get an Edge from a Graph

Given a Graph G with an Edge that connects Node A and Node B:

```
Edge x = G: A ? B
```

6.2.0.7 Conditional Statements

The forms of conditional statements supported are:

```
if (expression) { statement }

if (expression) { statement } else { statement }
```

6.3 Return Statement

A function returns to its caller through the return statement which has the following forms:

```
return;

return (expression);
```

6.4 null Statement

The null statement is simply:

```
;
```

7 Control Flow and Scope

Programs are executed from top to bottom, but you are able to loop (while, BFS) or skip (if/else) blocks of code based on a condition. Also, programs include variables that are accessible depending on where they are declared.

7.1 if/else statement

An if/else statement takes a condition. If the condition is true, then the block of code inside the curly braces that follows the “if” is executed. If it is false, the block of code is skipped and the block of inside the curly braces following the “else” is executed. Users are given the option to omit the “else”.

```
if { }

else { }
```

7.2 while loop

A while loop checks a condition and if it is true, then the block of code inside the curly braces is executed. The program goes back to the top of the loop and checks the condition again. If it is still true, the execution is repeated. This goes on until the condition is false and you break out of the loop.

```
while( boolean expression) { }
```

7.3 bfs loop

The BFS loop is used to traverse the nodes in a graph. The loop is implemented using the Breadth First Search algorithm and the user is able to control the how many neighbor levels that are visited. For example, the first neighbor level of Node n would be the nodes that are connected by only one edge to n.

```
bfs(Graph G; Node n; int x) {

    statement

}
```

7.4 scope

Variables that are declared outside any curly braces are available to be used anywhere in the program. Otherwise, variables are only accessible inside the curly braces. For example, if the declaration takes place inside a function, then the variable isn’t available to other functions. This is true for if/else statements, while loops and bfs loops.

8 Library Functions

Below are some of the kinds of 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.

8.1 Dijkstra: Shortest Path

`Edge[] shortest_path(Graph G, Node A, Node 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).

8.2 Reverse Edges

`Graph reverse_edges(Graph A)`

Reverses all the edges in Graph A. Returns a graph.

8.3 Depth First Search

`Graph depth_first_search(Graph G, Node A, Node B)`

Returns a Graph that depicts the DFS traversal from A to B.

8.4 Find All

`Node[] find_all(Graph G, Node src, int 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. Example implementation:

```
Nodes[] find_all (Graph G, Node src, int filter_attr) {
   Node[1000] neighbors;
   Node[on] valids;
   Node n = src;
   int x;
   int on;
   int pos = 0;

   bfs(G; n; x) {
     /*get edge between src and n*/
```

```
    Edge e = G.src ? n;
     if x < 1 && e.attr != filter_attr {
           neighbors[on] = n;
          on = on + 1;
     }
   }

   while (pos < on) {
      valids[pos] = neighbors[pos];
      pos = pos + 1;
   }

   return valids;
}
```

9 References

[1] Dennis M. Ritchie. C Reference Manual. https://www.bell-labs.com/usr/dmr/www/cman.pdf.

[2] Edwards, Stephen. ”Programming Language and Translators.” MicroC.

[3] Tutorials Point. https://www.tutorialspoint.com/cprogramming/c_data_types.htm.

10 Example

10.1 Hello World

```
Node h = {char name = 'h'};
Node e = {char name = 'e'};
Node l = {char name = 'l'};
Node l2 = {char name = 'l'};
Node o = {char name = 'o'};

Node space = {char name = ' '};

Node w = {char name = 'w'};
Node o2 = {char name = 'o'};
Node r = {char name = 'r'};
Node l3 = {char name = 'l'};
Node d = {char name = 'd'};

Graph HW;
```

```
HW = HW + h + e + l + l2 + o + space + w + o2 + r + l3 + d;

HW: h -> e -> l -> l2 -> o -> space -> w -> o2 -> r -> l3 -> d;

Node cur_node = h;
int depth;

bfs(HW; cur_node; depth) {
      printf("%C", cur_node.name);
}

printf("\n");
```

10.2 Cities and Shortest Paths

```
Node Pittsburgh = {int pop:  302205};
Node Philly     = {int pop: 1579000};
Node New_York   = {int pop: 8336000};

Graph cities;

Cities = Cities + Pittsburgh + Philly + New_York;

/* Add bidirectional edges between cities */

Cities: Pittsburgh ->304 Philly ->304 Pittsburgh ->371 New_York ->371
Pittsburgh;

Cities: Philly ->95 New_York ->95 Philly;

/* Get shortest path from Pittsburgh to New York */

Node[] path = shortest_path(cities, Pittsburgh, New_York);
```