# **GRACL** (.grc)


| | | |
|---|---|---|
| Defne Sonmez | dys2109 | System Architect |
| Eilam Lehrman | esl2160 | Language Guru |
| Hadley Callaway | hcc2134 | Manager |
| Maya Venkatraman | mv2731 | System Architect |
| Pelin Cetin | pc2807 | Tester |


## Table of Contents



# 1. Introduction

This manual describes GRACL (GRAph Concurrency Language), a language aiming to improve the efficiency of common graph algorithms while allowing programmers to initialize and modify graphs easily with built-in data structures specific to our language.

The syntax is inspired by some of the past projects focused on graphs (such as GRAIL from Spring 2017) with elements from Java, Python, and C. We plan to make the following features available to the programmer: graphs, nodes, edges, integer and double hash tables, edge lists, and node lists. GRACL focuses on directed graphs but the user is able to create undirected graphs by calling directed edges twice.

# 2. Constants

| | |
|---|---|
| Integer | Denoted int. Mathematical integers, eg 10 . |
| Doubles | Denoted double. Numbers that have a decimal point, e.g. 3.42 or 3.5E-10. |
| Boolean constants | Represented by the keywords True and False. |
| String literals | Series of ASCII characters delimited by double quotation marks. |

# 3. Primitive Data Types

| | |
|---|---|
| bool | Boolean value (True/False). |
| int | Integer. |
| double | A double-precision floating-point. |

# 4. Derived Types

| | |
|---|---|
| String | Array of ASCII characters. |
| NodeList | A collection of Node types. Under the hood it is implemented as a doubly linked list with O(n) lookup, they have an associated implicit mutex. |
| EdgeList | A collection of Edge types. Under the hood it is implemented as a doubly linked list with O(n) lookup, they have an associated implicit mutex. |
| IntTable | Collection of key value pairs with Node keys and int values. O(1) lookup time on keys, they have an associated implicit mutex. Has dynamic capacity; capacity doubles when the table is filled up. Uses the hash function h(item) = item % len and linear probing to resolve collisions. |

| | |
|---|---|
| DoubleTable | Collection of key value pairs with Node keys and double values. O(1) lookup time on keys, they have an associated implicit mutex. Has dynamic capacity; capacity doubles when the table is filled up. Uses the hash function h(item) = item % len and linear probing to resolve collisions. |
| Node | An object that has a String data field and an EdgeList representing connected edges, as well as an implicit mutex, and a visited boolean. |
| Edge | An object that takes in two Node objects, represents an directed Edge, has a double representing the weight. A user may create two Edge objects to represent one undirected Edge. They have an associated implicit mutex. |
| Graph | A list of Node objects connected by directed or undirected Edges representing a Graph data type. |
| void | The return type of functions that do not return anything. A variable of type void cannot be declared. |

## 5. Declarations

Declarations in a program are of the following grammar:

*program*: declarations eof
*declarations:* declarations var_dec | declarations fun_dec | **ε**

### a. Object Declarators

Each type has its own declarator, formatted in the following way:

```
int i;
bool b;
double d;
String s;
NodeList nl;
EdgeList el;
IntTable it;
DoubleTable dt;
Node n;
Edge e;
Graph g;
```

Variables can also be declared and initialized at the same time.

```
double e = 2.718;
```

Object declarations are of the following grammar:

*type*: int | bool | double | void | string | graph | node | edge | inttable | doubletable | nodelist | edgelist
*var_dec*: **type id ;** | **type id** = expression **;**

### b. Function Declarators

Every function declaration starts with first indicating the return type of the function, followed by its name and the arguments it will take in parentheses. Multiple arguments are separated by commas within the parentheses.

The scope of the function is indicated by curly brackets, starting with { and ending with }. Each function ends with a return statement right before the end of its scope.

GRACL will look for a function named main whose return type is an int to begin parsing the program.

```
returnType functionName (type arg1, type arg2, ... ){
      //This is the scope
      …
      return statement;
}
```

Function declarations are of the following grammar:

*type*: int | bool | double | void | string | graph | node | edge | inttable | doubletable | nodelist | edgelist
*formals_opt:* formal_list | **ε**
*formal_list:* **type id** | formal_list , **type id**
*func_body:* func_body var_dec | func_body statement | **ε**
*fun_dec:* **type id (** formals_opt **) {** func_body **}**

## 6. Lexical Conventions

### a. Comments

GRACL has two types of comments: multiline comment with /**/ and single-line comment with //.

The /* characters introduce a comment; the */ characters terminate a comment. They do not indicate a comment when occurring within a string literal. Comments do not nest. Once the /* introducing a comment is seen, all other characters are ignored until the ending */ is encountered.

The // characters don't need characters to terminate the comment. As soon as the user moves on to the next line, the comment will be terminated.

### b. Identifiers

An identifier, or name, is a sequence of letters, digits, and underscores (_). The first character cannot be a digit. Uppercase and lowercase letters are distinct. Name length is unlimited. The terms identifier and name are used interchangeably.

### c. Keywords

All data types, *for*, *while*, *if*, *else*, *return, True*, *False*, *hatch,* and *synch* are keywords. Functions with the same headers as functions in the standard library cannot be declared. Keywords cannot be used elsewhere.

### d. Punctuators

A punctuator is a symbol that has semantic significance but does not specify an operation to be performed. The punctuators [ ], ( ), and { } must occur in pairs, possibly separated by expressions, declarations, or statements.

## 7. Operator Conversions

| | |
|---|---|
| String doubleToString(double a) | Takes in a double and returns a corresponding String. For example, doubleToString(14.7) would return "14.7" on success. Program breaks on failure. |
| double intToDouble(int d) | Takes in an int and returns a corresponding double. For example, intToDouble(14) would return 14.0 on success. Program breaks on failure. |

## 8. Expressions and Operators

### a. Assignment Operator

| | |
|---|---|
| = | Assignment for all types. |

### b. Integer and Double Operators

| | |
|---|---|
| - + | Additive for types int and double. |
| * / % | Multiplicative for types int and double. |

### c. Logical Operators

| | |
|---|---|
| == | Equality comparison for types int, double, bool, String. |
| < <= | Relational comparisons for types int and double. |
| ! | Logical not for Boolean expressions. |

| | |
|---|---|
| && | Logical and for Boolean expressions. |
| \|\| | Logical or for Boolean expressions. |

### d. IntTable Operators

| | |
|---|---|
| table[Node key] = int value | Adds a new key-value pair to the IntTable. |
| table[Node key] | Returns the value corresponding to the key. |

### e. DoubleTable Operators

| | |
|---|---|
| table[Node key] = double value | Adds a new key-value pair to the DoubleTable. |
| table[Node key] | Returns the value corresponding to the key. |

### f. Precedence

The rows are ordered from highest to lowest precedence.

| **Operator** | **Associativity** |
|---|---|
| ( ) [ ] . | Left to Right. |
| ! - (unary) | Right to Left. |
| * / % | Left to Right. |
| + - | Left to Right. |
| *hatch synch* | Nonassociative |
| < <= | Left to Right. |
| == | Left to Right. |
| && | Left to Right. |
| \|\| | Left to Right. |
| = | Right to Left. |

## 9. Scope

### a. Block Scopes

The scope of an identifier is limited to the block in which it is defined. Each block has its own scope. No conflict occurs if the same identifier is declared in two blocks. If one block encloses the other, the declaration in the enclosed block hides that in the enclosing block until the end of the enclosed block is reached. Blocks are defined by {}.

```
if(!gracl_is_great) {
        int die = 1;
} else {
        int live = 1;
}
die = 0;          // This will not compile.
```

### b. File Scopes

Identifiers appearing outside of any block, function, or function prototype, have file scope. This scope continues to the end of the file.

```
int x;
if (plt_is_the_best_class) {
        x = x+ 1          // This is allowed.
}
```

## 10. Statements

Statements are of the following grammar:

*statement_list*: statement_list statement | **ε**

*statement*: expression **;** | **return** expression_opt **;** | **{** statement_list **}**
| **if (** expression **)** statement **else** statement | **for ( node id in id )** statement
| **for ( edge id in id )** statement | **while (** expression **)** statement
| **hatch** expression **id (** args_opt **)** statement | **synch id {** statement_list **}**

*expression_opt*: expression | **ε**

*expression*: **literal** | **func_literal** | **binary_literal** | **string_literal** | **id**
| expression **+** expression | expression **-** expression | expression ***** expression
| expression **/** expression | expression **%** expression | expression **=** expression
| expression **<** expression | expression **≤** expression | expression **&&** expression
| expression || expression | **-** expression | **!** expression | **id =** expression | **id (** args_opt **)**
| **id .** call_chain | **id [ id ]** | **id [ id ] =** expression | **(** expression **)**

*call_chain*: **id (** args_opt **)** | call_chain **. id (** args_opt **)**

*args_opt*: args_list | **ε**

*args_list*: expression | args_list **,** expression

### a. Selection Statements

| | |
|---|---|
| ; | Signifies the end of a statement. |
| if( *condition* ){ *statements* } \|<br>if( *condition* ){ *statements* }<br>else{ *statements* } | Conditional statement; curly braces only required for multiple statements. |

### b. Iteration Statements

| | |
|---|---|
| for(*Node/Edge id* in *NodeList/EdgeList*)<br>{ *statements* }<br><br>while( *condition* ){ *statements* } | Loops; curly braces only required for multiple statements. The for...in loop goes over all elements of a NodeList or an EdgeList. |

### c. Function-Call Statements

| | |
|---|---|
| functionName**(** args_list **);** | The function-call statement is used when a defined function is called. Parameters in args_list can be any of the primitive types or objects or expressions that evaluate to those types or objects. |

## 11. Concurrency

There are two keywords that deal with concurrency, *hatch* and *synch*. *hatch* takes an integer, a user-defined function and its parameters. It creates however many threads are specified by the integer and passes function(parameters) as the thread start routine. The curly braces following the *hatch* call denote code executed by the parent process before it begins to wait for the threads to terminate. The parent process may not execute code after the final curly brace until all child threads have been reaped.

Here is an example:

```
// Spawns 6 threads that execute startRoutine.
// The arguments that the function takes may all be of different types.

hatch 6 startRoutine(arg1, arg2, arg3, ...) {

        // Code executed by the parent thread before threads joined.

}

// Code executed after threads reaped.
```

*hatch* takes in a positive integer for the number of threads

*synch* takes the implicit mutex (created by the compiler) for an object and within the curly braces following synch, holds the lock associated with the object. Only after the second curly brace is the lock released. This helps to prevent user deadlocking since every *synch* call includes release of the lock.

Here is an example, the curly braces are required no matter how many statements:

```
synch visited {          // locks before access of shared data
        for ( Node n in visited ) {
                print('visited');
        }
}                        // unlocks after shared data access is complete
```

## 12. Library Functions

Note: the .func(params) notation indicates that the given function is called like object.func(params) for the given referred to object

Note: If a function errors in a non-recoverable way, the program breaks

### a. NodeList Properties and Built-in Functions

| | |
|---|---|
| int .length() | Returns the length of the list as an int on success. Returns -1 on failure. |
| bool .empty() | Returns True if the list is empty and False if it is not. |
| int .remove(Node n) | Removes the passed-in item from the list. Returns 0 on success and -1 on failure. |
| Node .removeFirst() | Removes the first item from the list. Returns the removed node. Program breaks if .removeFirst() is called on an empty NodeList. |
| Node .removeLast() | Removes the last item from the list. Returns the removed node. Program breaks if .removeLast() is called on an empty NodeList. |
| int .append(Node n) | Appends the passed-in Node to the end of the list. Returns 0 on success and -1 on failure. |
| int .prepend(Node n) | Prepends the passed-in Node to the beginning of the list. Returns 0 on success and -1 on failure. |

## b. EdgeList Properties and Built-in Functions

| | |
|---|---|
| int .length() | Returns the length of the list as an int on success. Returns -1 on failure. |
| bool .empty() | Returns True if the list is empty and False if it is not. |
| int .remove(Edge e) | Removes the passed-in item from the list. Returns 0 on success and -1 on failure. |
| Edge .removeFirst() | Removes the first item from the list. Returns the removed edge. Program breaks if .removeFirst() is called on an empty EdgeList. |
| Edge .removeLast() | Removes the last item from the list. Returns the removed edge. Program breaks if .removeLast() is called on an empty EdgeList. |
| int .append(Edge e) | Appends the passed-in Edge to the end of the list. Returns 0 on success and -1 on failure. |
| int .prepend(Edge e) | Prepends the passed-in Edge to the beginning of the list. Returns 0 on success and -1 on failure. |

## c. IntTable Properties and Built-in Functions

| | |
|---|---|
| NodeList .keys() | Returns a list containing all Node keys in the hashtable. |
| int .delete(Node n) | Deletes the key-value pair with the passed-in Node as the key from the IntTable. Returns 0 on success and -1 on failure. |
| bool .includes(Node n) | Returns a bool as to whether or not a key exists in the IntTable. |

## d. DoubleTable Properties and Built-in Functions

| | |
|---|---|
| NodeList .keys() | Returns a list containing all Node keys in the hashtable. |
| int .delete(Node n) | Deletes the key-value pair with the passed-in Node as the key from the DoubleTable. Returns 0 on success and -1 on failure. |
| bool .includes(Node n) | Returns a bool as to whether or not a key exists in the DoubleTable. |

## e. Node Properties and Built-in Functions

| | |
|---|---|
| String .data() | Returns data stored in the Node object. |
| NodeList .neighbors() | Returns a list of neighbor Nodes. |
| bool .visited() | Returns a boolean representing if the Node has already been visited. |
| int .updateData(String data) | Updates the data field on the Node to be the new data passed in and returns 0 on success. On failure, returns -1. |
| int .updateVisited(bool tf) | Updates the visited field on the Node to be the new bool passed in and returns 0 on success. On failure, returns -1. |
| bool .equals(Node n) | Compares the two Node objects. Returns True if they are the same and returns False if they're not the same. Under the hood, this is implemented by comparing the implicit id associated with the Node object. |

## f. Graph Properties and Built-in Functions

| | |
|---|---|
| NodeList .nodes() | Returns a list of Node objects in the graph. |
| EdgeList .edges() | Returns a list of directed Edges in the graph. |
| Node .createNode(String data) | Creates a new Node, setting its data. It returns a pointer to a Node object that the user can save in a Node variable. |
| int .removeNode(Node n) | Removes the passed-in Node from the graph and deletes corresponding Edges. Returns 0 on success |

| | |
|---|---|
| | and -1 on failure. |
| Edge .addEdge(Node source, Node dest, double weight) | Adds the Edge e to a Node's EdgeList, and returns that Edge. |
| int .removeEdge(Edge e) | Removes the given Edge e and returns 0, returns -1 if the passed in Edge does not exist in the graph. |

### g. Edge Properties and Built-in Functions

| | |
|---|---|
| void .updateEdge(double a) | Updates the directed Edge to have a new weight. |
| double .weight() | Returns the weight of the Edge object. |

### h. String Properties and Built-in Functions

| | |
|---|---|
| int .length() | Returns the length of the String. |
| bool .equals(String s) | Returns True if both strings represent the same array of characters, False otherwise. |

Note: String creation should be done by assigning a String literal to a String variable, similar to how primitive types are initialized. For example: String s = "GRACL";

### i. General Library Functions and Constants

| | |
|---|---|
| void print() | Print function that prints String objects. |
| infinity | Double type constant representing infinity. Useful for graph algorithms that have infinite edge weights. |
| Graph createGraph() | Creates and returns an empty Graph. |
| NodeList createNodeList() | Creates and returns an empty NodeList. |
| EdgeList createEdgeList() | Creates and returns an empty EdgeList. |
| IntTable createIntTable() | Creates and returns an empty IntTable with Node keys and int values. |
| DoubleTable createDoubleTable() | Creates and returns an empty DoubleTable with Node keys and double values. |

14. Sample Programs
    a. Graph Syntax

```
// Creates empty graph
Graph g = createGraph();

// Creates nodes and adds it to graph
Node n = g.createNode("New York");
Node c = g.createNode("Chicago");
Node s = g.createNode("San Francisco");

// Adds an edge to graph g
Edge nc = g.addEdge(n, c, 796.1);
Edge cs = g.addEdge(c, s, 2131.72);
Edge ns = g.addEdge(n, s, 2900.0);

// Returns a list of nodes in the graph
NodeList n = g.nodes();

// Updates the Node n in the graph
// Checks to see that update was successful
if (n.updateData("New York City") < 0) {
    print("Failure updating the data");
}

// Removes node c from the graph and corresponding edges
// Checks to see that removal was successful
if (g.removeNode(c) < 0) {
    print("Failure removing the node");
}
```

b. Concurrent DFS Graph Traversal

```
// Assume Graph g is already created and populated with nodes and edges

NodeList path = createNodeList(); // Creates empty NodeList

bool goalTest(Node goal, Node current){
    // Check to see if we've reached goal
    return current.equals(goal);
}

int normalDFS(Node current, Node goal, NodeList myPath){
    synch path {
        if (!path.empty()) {
            return 0;
        }
    }
    // Append current node to this thread's myPath
    if (myPath.append(current) < 0) {
        return -1;
    }
    // Check to see if this thread has reached the goal
    if (goalTest(goal, current)) {
        // Update shared memory path and return
        synch path {
            path = myPath;
        }
    } else {
        synch current {
            // Update current node to have visited = True
            if (current.updateVisited(True) < 0) {
                return -1;
            }
        }
        // Call normalDFS for each neighboring node with the same myPath
        for (neighbor in current.neighbors()) {
            // Make sure neighbor hasn't been visited before calling normalDFS on it
            bool alreadyVisited = False;
            synch neighbor {
                alreadyVisited = neighbor.visited();
            }
            if (!alreadyVisited) {
                normalDFS(neighbor, goal, myPath);
            }
        }
    }
    return 0;
}
```

```
NodeList multithreadDFS(Node start, Node goal) {
    // Check to see if start node is the goal
    if (goalTest(goal, start)) {
        return createNodeList();
    } else {
        // Update start node to have visited = True
        if (start.updateVisited(True) < 0) {
            return createNodeList();
        }
        NodeList neighbors = start.neighbors();
        NodeList myPath = createNodeList();
        // Create a thread to run normalDFS on each node neighboring start
        for (node in neighbors) {
            hatch 1 normalDFS(node, goal, myPath) {}
            return path;
        }
    }
}
```

## 15. References


- GRAIL Project (Spring 2017)
- K&R C Book
- Java Synchronized Keyword