1 Introduction

Hippograph will aim to be an improved version of giraph, a project from PLT Fall 2017, with improvements in graph creation and graph query capabilities. The language will support a range of graph types, like its predecessor; however, Hippograph’s graph types will be inferred from arguments passed in during instantiation. This language will be ideally suited for a wide range of graph problems that demand flexible representation of data and graph structures. Graphs can also be searched and traversed according to user-defined search strategy functions.

2 Reserved Keywords

The following Reserved Keywords cannot be used for identifiers, which are alphanumeric sequences, beginning with an alphabetic character. bool, int, float, char, fun, NULL, true, false, graph, node, edge, string, if, else, while, for, in, return, void

3 Data Types

hippograph is a statically typed language supporting the following data types:

3.1 Primitives

- bool - one byte TRUE/FALSE value
- int - signed 32-bit integers
- float - double precision floating point numbers
- char - ASCII characters
- fun - a function, with input types, return type, and name
- NULL - null
3.2 Reference Types

- **graph** - consists of nodes and edges connecting nodes. Node data, and edge weights/relationships must each have one type, but nodes do not have to be of the same type as edge.
  
  Graph declaration syntax: `graph<name type:node value type, edge rel type> new_graph;`
  
  - `graph.add_node(node)` - add node to graph.
  - `graph.add_edge(from_node, to_node, edge weight)` - add an edge in graph from from_node to to_node.
  - `graph.remove_node(node)` - removes node from graph.
  - `graph.remove_edge(edge)` - removes edge from graph.
  - `graph.has_node(node)` - returns true if node is in graph; false otherwise.
  - `graph.has_edge(edge)` - returns true if edge is in graph; false otherwise.
  - `graph.are_neighbors(from, to)` - returns true if there exists an edge from from to; false otherwise.
  - `graph.neighbors(node)` - returns a graph containing all of the neighbors of node.
  - `graph.find(data)` - returns an arbitrary node containing data from graph if one exists; otherwise, returns NULL.
  - `graph.peek()` - returns an arbitrary node in the graph without changing the state of graph.
  - `graph.print()` - pretty prints graph nodes, edges, and the data they contain.

- **edge** - consists of two nodes which the edge connects, and a weight/relationship value which can be any type. For unweighted edges, weight relationship will be set to NULL.
  
  - `edge.from()` - returns node edge originates from. In an undirected graph, there’s no guarantee which node will return
  - `edge.to()` - returns node edge goes to. In an undirected graph there’s no guarantee which node will return
  - `edge.weight()` - returns weight of edge. Returns NULL if unweighted
  - `edge.set_weight(weight/relationship)` - sets weight/relationship of specified edge.

- **node** - contains a key-value pair consisting of an ‘enumerable’ node name (int, string, or char), and node data.
  
  - `node.name()` - returns name of node.
  - `node.data()` - returns data stored in node.
  - `node.set_data(data)` - sets data in specified node. Overwrites any existing data.

- **string** - A sequence of chars enclosed in double quotes.

4 Operators and Expressions

4.1 Variable Assignment

Variables are assigned using the = operator. The left hand side argument must be an identifier, and the right hand side can be a value or identifier of the same type as the left. Type conversions are not supported.
4.2 Function Expressions

In Hippograph, functions are first-class. These are defined using the following syntax:

\[
\text{fun } \textvar \text{ = return-type (type arg1, type arg2, ...)} \{\text{expression}\};
\]

where \textvar is a new variable of type \text{fun} that refers to the newly defined function.

4.3 Node Data Assignment

Data for a node is assigned using the : operator, with the : operator having higher precedence than other graph operators.

In general, a node can be either a name-data pair where both result to a value, or a single name that evaluates to a value.

Node declarations in graph instantiations are read from left to right. When a node that has a name is created, the next time the name occurs in the declaration, it references the same node.

4.4 Graph Construction

Graphs are declared with the type signature

\[
\text{graph \langle node name type : node data type, edge data type \rangle}
\]

An edge can have the following formats:

- \(-(\text{weight})>\): a right-singly-directed edge
- \<(\text{weight})>: a left-singly-directed edge
- \-(\text{weight})- or \<(\text{weight})>: an undirected edge

Example:

\[
\text{graph<\langle char:int, int \rangle} \text{ g = ['A':2 -(3)> 'B':4 <(2)> 'C':8 <(2)- 'A']}
\]

4.5 Arithmetic Operators

The precedence follows the standard mathematical “order of operations”, with, in decreasing order of precedence:

1. Parentheses, non-associative
2. Multiplicative operators \(*, /\), left-associative
3. Additive operators \(+, -\), left-associative

4.6 Boolean Operators

In descending order of precedence:

1. \(==, \neq, \geq, \leq, >, <\), non-associative
2. \not, right-associative
3. \text{and and or}, left-associative
4.7 Comments

Comments will be formatted as (* ... *) and will not allow nested comments.

5 Control Flow

5.1 Conditionals

Conditional expressions follow C-style syntax.

```c
if (condition) { statements }
if (condition) { statements } else { statements }
if (condition1) { statements } else if (condition2) { statements } else { statements }
```

A if block may optionally be followed by any number of else if blocks. It may also be followed by a else block.

5.2 Loops

Loops follow C-style syntax.

```c
1 while ( condition ) { statement }
2 for ( initialization ; condition ; update ) { statement }
```

Hippograph also supports iteration over nodes and edges.

- `for_node(node : graph) { statements }` - Iterates through the nodes in a graph with an arbitrary ordering.
- `for_edge(edge : graph) { statements }` - Iterates through the edges in a graph with an arbitrary ordering.

6 Program Structure

Programs will consist of sequences of functions including a `main` function, which must be placed last in the program sequence. `main` will be the entry point for the program’s executable. Functions will have the following syntax, with names being mandatory (no anonymous functions are permitted):

```c
return_type name(type arg1, type arg2, ...) { body }
```

Scoping is determined through the use of curly braces {}, like in C and Java. In particular, these will define the scope of function and iteration bodies in addition to the graph instantiations.

7 Standard Library

7.1 Lists

Lists are implemented in Hippograph as trees. The following library functions are provided:

- `List.new()` - returns a new, empty list
- `List.empty(list)` - returns true if list is empty and false otherwise
• List.length(list) - returns the length of list
• List.add(list, index) - adds element at specified index in list
• List.print(list) - prints the elements of list in ascending index order
• List.merge(list1, list2) - concatenates list1 with list2
• List.get(list, index) - gets the element at index in list
• List.remove(list, index) - removes the element at index in list and shifts the elements over from the right
• List.search(list, value) - looks for an element with value in list from the left and returns its index

7.2 Strings

Strings are implemented as trees. The following functions are provided:

• String.length(string) - returns the length of string
• String.char_at(string, index) - returns the character at index in string
• String.concat(string1, string2) - concatenates the two strings
• String.index_of(string, char) - returns the index of the first occurrence of char in string
• String.replace(string, char) - replaces the first occurrence of char in string
• String.search(string, substring) - looks for substring within string and returns the index of the first match
• String.substring(string, index1, index2) - returns the substring of string between the specified indices, left inclusive and right exclusive.

7.3 Maps

Maps are implemented in Hippograph as trees with values stored within the node. The following functions are provided:

• Map.new() - returns a new empty map
• Map.add(map, key, value) - adds a key:value pair within map and returns map
• Map.remove(map, key) - removes a key:value pair from map and returns true if key was in map and false otherwise
• Map.get(map, key) - returns the value of key
• Map.contains(map, key) - returns true if key is in map and false otherwise
• Map.empty(map) - returns true if map is empty and false otherwise