PLT Project Proposal: GRAPL (Graph Processing Language)

September 29, 2010

1. Introduction & Motivation

GRAPL (**GRA**ph **P**rocessing **L**anguage) is a user-friendly language designed to simplify the creation and navigation of directed graphs. Creating graphs in existing languages generally requires building node objects and maintaining arrays or lists of pointers to other nodes (along with weights); keeping track of this information can be cumbersome. In addition, the structure of the graph (in the general case where any node may be connected to any other node) takes a good deal of code to set up. GRAPL is intended to hide the complexity of graph navigation and to make creating graphs and adding nodes as simple as possible.

2. Language Features

GRAPL provides the following features:

Graph construction primitives. Many difficulties arise from trying to specify in a language a simple drawing. Having this in mind, GRAPL aims to make the construction of a graph as simple as drawing one. With GRAPL you can build a complex graph with as few as a pair of lines of code, using convenient operators such as >w> for directed edges and <w> for bidirectional edges with weights. The nodes and edges are automatically induced by the GRAPL compiler.

Easy graph manipulation. Built in control flow routines make graph algorithm programming as simple as writing a typical hello world program.

Four different data types. Graph manipulation shouldn't deal with various different data types. The only data types a GRAPL programmer should worry about are the basics: Node, Edge, Graph, and Number.

User defined functions. The user can build his own function to later reuse in his programs as many times as he likes.

3. Language Overview

3.1 Identifiers, Operators, and Data Types

3.1.1 **Identifiers.** Identifiers are a sequence of English characters (both upper and lower case), digits and underlines. The first token of an identifier should be a character. Other special symbols such as &, *, and ! are not allowed in identifiers.

3.1.2 Operators

3.1.2.1 Arithmetic Operators

"+", "-", "*", and "/" are used in numeric calculations. "<", ">", "<=", ">=", "==" are used arithmetically in number comparisons.

3.1.2.2 Special Operators

"Node1" + "<" + "Weight" + ">" + "Node2" are used in graph generations, meaning creation of two new nodes and a new bidirectional edge connecting them with weight shown in the middle.

"Node1" + ">" + "Weight" + ">" + "Node2" are used in graph generations, meaning creation of two new nodes and a new unidirectional edge from "Node1" to "Node2" with weight shown in the middle.

"List" + "+" + "Node" are used to mean adding a node to the end of a list.

3.1.3 Data Types / Objects

3.1.3.1 A number data type is a primitive declared using the keyword `Number`. This data type contains integer part and decimal fraction. *Example:* `Number num = 5.3.`

3.1.3.2 A string data type is a primitive declared using the keyword `String`.
Example: `String str = "Big Panda!"`

3.1.3.3 **Node** objects are created automatically in the creation of a new graph, without explicit declaration. They can also be declared individually. Nodes have a few built-in methods and/or public attributes, including:

- · `node.visit()` // marks node as visited
- · `node.isVisited()` // returns Boolean
- · `node.unvisit()` // removes visitor marking
- · `node.print()` // prints the name of the node
- · `node.id()` // returns a unique identifier for the node

3.1.3.4 Like nodes, **Edges** are also created automatically (implicitly) in the creation of a new graph, but may be declared explicitly. Edges have an edge weight, which defaults to zero if not specified.

3.1.3.5 A **List** data type is declared using the keyword `List`**.** This data type represents lists of nodes. *Example:* `List list = {Node node1, Node node2}.` Lists may have a few public attributes and/or methods, e.g:

- · `list.length()` // get the length of a list
- · `list.print()` // print out the contents of a list

3.1.3.6 A **Graph** data type is declared using the keyword `Graph`. This data type represents graphs composed of nodes and edges.

Example: *`Graph g = [A <2> B >3> C <4> D <5> B, C <1> E];  // A, B, C, D, E are nodes`*

This example a graph with nodes A, B, C, D, E. A is connected to B with one (weight = 2) bidirectional edge. B is connected to C with one (weight = 3) unidirectional edge from B to C. The other nodes and edges are created in the same mechanism. Graphs have a few built-in methods and/or public attributes, including:

- · `graph.clearVisited()` // clears Visited flag on all nodes in graph
- · `graph.print()` // prints some reasonably useful representation of the graph
- · `graph1.insert(graph2)` // inserts the argument into graph1. graph2 could be a single node or a graph structure. Nodes with the same names as those in the graph1 are considered to be the same objects.

3.2 Control Structures

3.2.1 **If-then-else.** This works exactly as the if-then-else structure in languages like C and JAVA.

3.2.2 **forEach iterator.** This special iterator at the heart of GRAPL vastly simplifies graph traversal. The syntax is as follows:

forEach { <optionally> visited | unvisited } node1 { from | to } node0 { <optionally> withEdge (operator Number, operator Number...) }

forEach loops through all nodes "node1" that are either parents or children of node0 (depending on the “from” or “to” keyword) and satisfy the conditions (visited, unvisited, or with a constraint on edge weight). "Operator" is one of { < , > , = , <= , >= }.

Example:
```
forEach unvisited child from startNode withEdge (>10)
{
      /* do something */
}
```

3.3 **User Functions.** Users are allowed to create their own functions and implement their algorithms using this language. Function declarations are in the C style.

3.4 **Recursion.** Obviously, GRAPL supports recursive function calls; most algorithms are expected to be recursive. No “rec” keyword is needed.

4. Sample Code

4.1. Initialize and Modify Graph

```
void main(){      // Add some edges to the graph
      graph g = [a >3> b >4> c <5> d];
      // Modify the graph later if we want add more things.
      g.insert(b>4>a);
      // We can store node information in list
      List ll = null;         ll + a;     // append node a        ll +
b;    // append node b        ll.print;   // output : { a, b}.

      // Build a empty list to store the results for dfs          List
l = null;         l = dfs(l,a); // output: a,b,c,d
      l.print();        List myPath = null;     myPath = path(path,a
,c); // output a, b, c        myPath.print();    }
```

4.2 Depth First Search

```
list dfs (list l, node n) {
      n.visit();
      l + n; // appends current node n to the list
      forEach unvisited m from n { dfs (l, m); }      return l; }
```

4.3 Finding A Path To A Node

```
list path(list l, node n, node m){
      n.visit();
      l + n;
      if (n.id() == m.id())
            return l;         else
            forEach unvisited temp from n
            {
                  path(l, temp, m);
            }
}
```

4.4. Finding the node with most related nodes. Starting with node n_0, look for the node n such that 1) the distance between n_0 and n is less than 1800; and 2) n is the node in the graph (except n_0) that has the most related nodes (related nodes mean that n has at least one path to arrive at) within the distance of 100. This might be used, for example, to model the density of surrounding living areas in order to choose the best site for a new business.

```
void FindOptimalNode(Node startNode)
{
      Node nOpt;
      Number num = 0;
      forEach n from n0 withWeight (<1800)
      {
            Number num'=0;
            forEach n' from n withWeight (<100)
            {
                  num'= num' + 1;
            }
      }

      if (num' > num)
      {
            num = num';
            nOpt = n';
      }
      nOpt.print();
}
```

5.0 Future Directions and Add-ons

If time permits, we have a number of ideas for extending the language. These include:

1. 1. Including standard library functions like depth-first-search, BFS, Dijkstra, etc. (Some of these may be built in to the language).
2. 2. Extending the language to permit creation of text-based-adventure games. This would require storing additional information at each node (probably in a dictionary of some kind), and an additional notion of a User object that would navigate through the graph in an interpreted session.
3. 3. Building a simple GUI to display a created graph in a readable fashion.
4. 4. Building a GUI that allows the creation of nodes and edges graphically. (Yeah, right.)

6.0 Team Information

Team GRAPL (aka Mandarin Express) consists of:

- · Di Wen (dw2464@columbia.edu)
- · Yang Yi (yy2339@columbia.edu)
- · Lili Chen (lc2737@columbia.edu)
- · Andres Uribe (au2158@columbia.edu)
- · Ryan Turner (rct2115@columbia.edu)