COMS W4115
Project Proposal

Group Members:
Montana Robert St.Pierre
Jason Delancey

Roles:
Manager - Montana
Language Guru - Montana
System Architect - Jason
Tester - Jason

Responsibilities:
Montana - Timely completion of deliverables
Montana - Language design
Jason - Compiler architecture, development environment
Jason - Test plan, test suites

Weekly Meeting Time:
Saturday/Sunday - 12 noon

Our language, with the working name GraphC, will be a lightweight language designed to handle problems with graphs specifically. While the language will be general purpose, it targets the domain of problems utilizing graphs and graph-like data structures. The language will be compiled ahead of time, strongly typed and statically typed. Syntax will be designed with user friendly representation of graphs as a priority, while also drawing elements from C-like languages. The aim is to develop a modern and clear syntax that is not overly verbose. To accomplish this, the language will feature both declarative functional styles and imperative styles in addition to stripping extraneous characters from code when possible.

Programs involving graphs is an example of programs meant to to be written in our language. This can be anything from mathematical graph theory problems to visualizing data in lists to functioning as a scripting language. The flexibility of the graph structure will enable a simple interface for the programmer to quickly develop algorithms on complex sets of data that other languages with similar universal data structures make difficult.

Graphs are the main element of the language, and the center of the design. They will be an associative data structure storing the representation of vertices and edges. Implementation will

be extensible, but the inherent capabilities will allow direct and undirected edges between nodes. Nodes and edges will be associative with their name serving as a key and their value being data. This representation will allow other data structures to be easily represented as most are types of graphs. For example, singly and doubly linked lists can be initialized as a linear graph, and a reference to the head and tail can be stored to append or remove from the list. Stacks and queues follow from this definition. Array based structures will be represented as a graph with vertices and no connecting edges. The associative nature of the graph structure will enable numeric keys for traditional array access in addition to associative access.

In addition to graphs, there will be a numeric, boolean and string primitive types. Types will be inferred whenever possible to enable static typing. We will include a standard library featuring common graph algorithms, such as traversals, topological sorts, dijkstra's, etc. in addition to a basic display capability. These will be extensible to enable precise control over graph/tree/list visualization. Libraries representing other data structures will be trivial to construct. These design choices ensure that the language will be programmer friendly, relatively fast and lightweight.

Please see below for an example of a very simple program solving the traveling salesman problem.

```
graph cities = {
   newyork <> chicago = 714
   newyork <> denver = 1629.23
   newyork <> miami = 1093.57
   chicago <> miami = 1193
   chicago <> denver = 919.23
   denver -> miami = 1727.17
   miami -> denver = 1000
}

function pathLength(graph g) {
   float sum = 0
   foreach (edge e : g) {
      sum += e
   }

   return sum;
}
```

```
function tsp(graph g) {
   return min(brute(g, pathLength))
}

graph shortestRoute = tsp(cities)

display(shortestRoute)
```