Mathew Mallett mm4673
Rusty Nelson rnn2102
Guanqi Luo gl2483
ComS 4115 Project Proposal


## Language Description:

We propose a language where vertices (aka nodes) and links are a first class data types. These types are used to construct, manipulate and test an implicit graph structure which is the main store in the language. A user of our language would be able to create and manipulate graphs as easily as they could work with an array in most other languages. The language will provide a way to construct the nodes and links, as well as provide efficient implementations of common operations, like searching and path finding through a standard library. The data structures will be open enough to enable developers to build other graph algorithms in the language as well as support a variety of different graph types from directed/undirected, weighted and restricted depending on the programmers adherence to their own rules. The language will compile to C.

## Language Use:

The language can be used to quickly build up and work with graphs. It will be designed to make common graph operations easy and efficient. For example, a user could build up a graph data structure from a file, using our graph primitive to make the task easier, then use the graph primitive's methods to quickly build an algorithm to determine if the graph is connected. More specific examples would be to solve pathing, connectedness, ordering, sequencing, max flow and min-cut problems.

Mathew Mallett mm4673
Rusty Nelson rnn2102
Guanqi Luo gl2483
ComS 4115 Project Proposal

## Language Parts:

- Graph is implicitly defined and ready to manipulate as soon as the programmer's code starts running.
- Data nodes and links types are primary data types
- Anonymous links as well as indexable links
- Weighing of links
- Complex arithmetic type link and node indexes
- Complex link and node types with fields
- Pattern matching in function definitions
- Function are stored as nodes and can be queried as such
- Links and Nodes can be queried from the graph by their key types
- Standard library to implement common graph needs (Neighbors, Degree, Children, etc) for different graph types

## Sample Code 0:

```
/* Directionally link a node defined as “a” to a node defined as “b” */
“a” --> “b”

/* Directionally link an existing node defined as “a” and to a node defined
as “c” */
( “a” ) --> “c”

/* Another directional link */
( “a” ) --> “d”

/* Bidirectionally link an existing node defined as “b” and to a node
defined as “e” */
( “b” ) <--> “e”

/* Find a node defined as “a” and get all of it’s children */
( “a” ).out
>> [ b, c, d ]

/* A define a degree function which takes an argument of type node and adds
the number of outgoing links and the number of incoming links and returns
the accumulation.  Note that the function is a node as well. */
degree node:() => node.in.length + node.out.length

/* Find a node defined as “a” and pass it to the degree function */
( degree ) ( “a” )
>> 3
```

```
/* Find a node defined as “b” and pass it to the degree function */
( degree ) ( “b” )
>> 2

/* Find a node defined as “a” and find a node defined as “e” and test if
the “e” node is a child of “a” */
( “a” ) -?> ( “e” )
>> false

/* Find a node defined as “a” and find a node defined as “e” and find a
path between them */
( “a” ) -...> ( “e” )
>> a -> b -> e
```

## Sample Code 1:

```
“A” -{3}-> “B”
( “B” ) -{4}-> “C”
( “C” ) -{2}-> “D”
( “D” ) -{1}-> “E”

// Tail recursive function call with cases and function treated as node.
weighted_depth NONE init:int => {
      init
} | link:-- init:int {
      // Still having issues showing order of operations v.
      ( weighted_depth ) link.end init + link.weight
} | node:() init:int => {
      ( weighted_depth ) node.out[0] init
}
```