n2n
A Relational Graphing Language

Project Manager: Elisheva Aeder (ea2621)
Language Guru: Nicholas Ray Falba (nrf2118)
System Architect: Johan M. Mena (jmm2371)
Verification & Validation: Jialun Liu (jl4347)

Motivation

In today's world there is an escalating interest in relationships. Be it the connection of people on a social networking platform, of family members in a family tree, of variables in a mathematical equation, or the connection of trains in a city subway system—we are constantly trying to find networks of people and things and analyze how they interrelate. Most often, relationships are implemented in a programming setting via graph data structures containing sets of nodes and edges that define connections between the nodes. In standard programming languages, however, graphs can be tedious to create and manipulate, requiring the creation of separate classes for nodes and edges, and burdening the programmer to keep track of their graphs manually. Finding and analyzing relationships between nodes of a network can be challenging. We therefore created n2n, a language that provides high-level abstraction to create, maintain, and manipulate graphs, with a specific focus on the relationship between the nodes.

n2n has nodes, edges and graphs as built-in data types to assist a programmer in the simplistic construction and maintenance of graphs. When one node is connected to another, the relationship between them is named and specific to those nodes. The graphs can be bidirectional in that there can be two different relationships between two nodes depending on the direction of the relationship. For example, node A has the relationship of Parent to node B, while node B has the relationship of Child with node A. The storage and maintenance of the connections and node data occurs under the hood, without the programmer's need to create a data structure with which to implement the graph. This alleviates the difficulties of developing networks of nodes and eases the creation of algorithms to manipulate them.

Syntax

Data Types

Basic types

Int	data type representing a positive or negative integer
String	data type representing a plain text word, phrase, or sentence using ASCII
Bool	data type that can only take two values, true or false
Double	data type representing a floating point number

Complex Types

Data	represents a grouping of basic and / or other complex types. The number of fields, and the diversity of their types are up to the programmer
Node	represents a node in a graph. It must contain at least some Data. They can be instantiated either explicitly, or implicitly (see below)
Relationship	represents a relationship between two nodes. They can be instantiated explicitly or implicitly (see below)
Graph	represents a collection of nodes with relationships. This is the main thing our language is supposed to create and manipulate

Collections (colls)

List	an ordered collection of elements
Map	a key-value store

Language Keywords

If/else	used to indicate certain conditions under which a block of code ought to be executed. Of form :
	
	if(Bool expression){
	…
	} else {
	…
	}

let	Used to indicated a declaration of a variable. Of form:

	let var_name: data_type

Operators

+, -, *, /	Arithmetic
>, <, ==, !=, >=, <=	Comparative
&&, ||, !	Logical
=	Assignment
:	Used after a variable name in instantiation to indicate the data type that the variable is.
.	Used to access a field in a "grouping".

Built-in functions

We have a few built-in functions as part of the standard library of our language.

node()	Takes an argument that is one of the basic data types or a grouping and spits back a node containing that data.
rel()	Takes a node, a basic data type, and another node and returns a relationship between the first node and another node.
ins()	Takes a graph, and a relationship. There are two cases: The relationship contains two nodes that already exist in the graph, in which case, the function will simply insert a new relationship between the two nodes in the graph; One of the nodes in the relationship doesn't exist. In which case the function will create an empty node implicitly and define a relationship between the existing node and the new node.
rem()	Takes a graph, and either a relationship or a node. If the second argument is a node, then the function removes the node from the graph and all the relationships associated with it (from nodes to it/and from it to nodes). If the second argument is a relationship, only the relationship is removed.
neighbors()	Takes a graph and a node. Returns a list of all the direct neighbors of that node. In particular, this will return only those neighbors for which the node has an edge directed toward (not from).
addField()	Takes two arguments: a basic data type, and a variable name. This tells the compiler to add a field into one of our basic data types, sort of like adding an instance variable to a Java class. This is useful for adding different types of data to Nodes later in the program.

Operations on collections

- map(coll, fn) ; returns an array of the results of running fn on each elem of coll;
- reduce(coll, fn, init) ; combines all elements of coll by applying fn to two args at a time, starting from init;

- each(coll, fn) ; applies fn to each elem of coll, returns the original coll;
- filter(coll, pred) ; yields every element in coll to a predicate pred;

Useful native syntax

Line termination by new line. No semicolons.

Native definitions of graphs and graph elements:

Defining a relationship

let r1: Relationship = [A 4 B]
Where A and B are connected by an edge with weight 4. If A or B are not nodes already, these nodes are created implicitly.

Defining a graph

```
let graph: Graph = {A 4 B
                    B 5 C
                    C 3 A
                    D 4 A}
```

This creates a graph with 4 empty nodes, A, B, C, and D. Edges will be created from A to B, B to C, C to A, and D to A, with relationships defined by the integers 4, 5, 3, and 4, respectively.

Comments

Our language only supports multi-line comments. Use ; to start and ; to end.

Sample Code

```
; declare some data ;
data SwimmingPool {
  let length: Int
  let size: Double
}

; declaring a relationship with one attribute ;
data Connected [
  let isConnected: Bool
]

; another relationship;
data sortaConneted [
  let isSortaConnected: Bool
]
```

```
; declaring a Swimming Pool Graph that contains SwimmingPool nodes related through some relationships;
let spg: Graph = { p1 Connected p2
                   p1 Connected p4
                     p1 sortaConnected p3
                   p2 Connected p3
                   p3 sortaConnected p4 }

; filter direct neighbors by relationship, nodes connected to node p1 either directly or indirectly would be returned;
let connected-neighbors: List = neighbors(p1 Connected)

; get direct neighbors ;
let p1Neighbors: List = neighbors(p1)

; inserting a node into the 'spg' graph ;
ins(spg {p6 Connected p7})

; This removes the edge from A to D, and D as well if there are no more relationships associated with it (pointing to/coming from) ;
rem(spg {p6 Connected p7})

; This removes the Node D and all edges associated with it (pointing to/coming from) ;
rem(spg D)
```

Depth-first search comparison n2n vs. Java

n2n

```
addField(Node, visited: Boolean) ; Node now has a boolean field called visited;

fn visited(n: Node) -> void { n.visited = true }

fn visitAllNodes(g: Graph, n: Node) -> void {
        if (! n.visited) {
                visited(n)
                each(neighbors(n), { node in visitAllNodes(g, node) })
        }
}

; Dollar sign indicates start of main method ;
$
        let node1: Node = node(Visited)
        let node2: Node = node(Visited)
        let node3: Node = node(Visited)
        let g:Graph = { node1 1 node2
                        node2 2 node1
                        node3 3 node1 }
        dfs(g, node1)
```

Java

```
import java.util.ArrayList;
import java.util.List;

class Main {
  private static class Node {
    private boolean visited;
    private List<Node> neighbors;

    public Node() {
      this.visited = false;
      this.neighbors = new ArrayList<Node>();
    }

    public boolean isVisited() {
      return this.visited;
    }
    public void setVisited(boolean visited){
      this.visited = visited;
    }

    public void addNeighbor(Node node){
      this.neighbors.add(node);
```

```
    }

    public List<Node> getNeighbors() {
      return this.neighbors;
    }
  }

  private static class Graph {
    List<Node> nodes;
    public Graph() {
      this.nodes = new ArrayList<Node>();
    }

    public Graph(List<Node> nodes) {
      this();
      for(Node node : nodes) {
        this.nodes.add(node);
      }
    }

    public Node getFirst() throws Exception {
      if (nodes.isEmpty()) {
        throw new Exception();
      } else {
        return nodes.get(0);
      }
    }
  }

  public static void main(String[] args) throws Exception {
    Node n1 = new Node();
    Node n2 = new Node();
    n2.addNeighbor(n1);
    n1.addNeighbor(n2);
    List<Node> nodes = new ArrayList<Node>();
    nodes.add(n1);
    nodes.add(n2);
    Graph g = new Graph(nodes);
    dfs(g, g.getFirst());
    System.out.println("Done");
  }

  public static void visitAllNodes(Graph g, Node n) {
    if(n.isVisited()) {
      return;
    }
    n.setVisited(true);
    for (Node node : n.getNeighbors()) {
      dfs(g, node);
    }

  }
}
```