DiGr: Directed Graph Processing Language
PLT Fall 2010 Final Project Report

Bryan Oemler (Team Leader)
Ari Golub
Dennis V. Perepelitsa

22 December 2010

Contents

1 Introduction

DiGr (pronounced to rhyme with "tiger") is a compiled, imperative, object oriented language designed to easily create, process and modify directed graphs. Directed graphs are simple yet flexible graph theory concepts which show up in everything from basic computer science data structures to gaming. Fundamental objects and operations in DiGr make it easy to quickly and efficiently define trees and graphs, and then modify, search, traverse and otherwise interact with them. DiGr also provides support for more abstract concepts like tree traversals and value hierarchies.

DiGr is a language in which nodes and edges are the most natural objects. Its syntax allows for the creation of nodes, edges, and entire graph structures with small, concise statements. In DiGr, the user can write the minimum amount of information needed to define the digraph, and the compiler will intelligently fill in the rest of the details. Each node and directed edge efficiently stores any additional amount of user information, allowing for a wide variety of user applications. Where possible, DiGr tries to hide implementation details from the user: for example, undirected graphs are compiled as a special type of directed graph, and tree traversal queues present only a minimal front-end to the user.

In DiGr, it is also easy to crawl and manipulate digraphs. "Crawls" are a special type of function defined in a way convenient for depth-first, breadth-first, or any other type of user-defined traversals of digraphs. That is, the most primitive function in the language is a recursive one that moves from parent to children nodes. Crawls are general enough to be useful in many graph-related applications, but narrowly defined enough to let the user do a lot while writing a little.

When given a start node and a "rule" by the user, crawls use an internal queue to move through and examine or modify a directed graph. The user defines which action, if any, the crawl takes at a given node. The rule guides graph traversal by determining the structure of the queue at each step. For example, three variants depth-first search variants are implemented by changing the order in which the rule adds children to the queue. In a conditional path traversal, the crawl maintains a queue with a single node in it.

1.1 What Problems Can DiGr Solve?

DiGr can be applied to a number of problems which can be modeled and solved using basic graph theory operations and ideas. These span from the academic and abstract to more down to earth applications, from finding the best route between points based on various criteria (distance, cost, time required), to designing and modeling search trees and search algorithms for fast storage and look-up of data (contacts lists, dictionary definitions, computer process trees), to even fancier applications like implement finite state machines (and, by extension, regular expressions). Here is a specific example.

A main concern in commercial shipping is getting the products to their destination in as economical a fashion as possible, be it the economy of time, money, or some other factor. Thus it is very important to have an easy yet sophisticated model of the shipping lanes, factories, and destinations involved. The DiGr language is ideal for describing and manipulating the kind of data that a route planner would deal with.

The factories and destinations are represented as DiGr nodes and the shipping lanes that connect them are DiGr edges. The attributes of the edges could represent weight or importance based on a number of factors including distance, frequency traveled, difficulty of transportation, etc. When shipping lanes are temporarily disabled, say, due to weather, those edges could be represented as "broken"; the connection would still exist, but an attribute would render it inactive. Different choice of rules in specially written crawls could select routes by speed or efficiency (e.g. "take the edge with less financial cost", "take the edge of shorter distance") and return two very different routes. Adding and modifying the network are very natural low-level operations in the DiGr language, as are ways to modify and examine the network.

2 Tutorial

The DiGr programming is designed to make creating, modifying and inspecting directed graphs easy and efficient. It provides tools for constructing trees, and an extensible traversal framework. This tutorial will walk you through the basis of DiGr, but will necessarily leave some details out. For more information, please see the reference manual below.

2.1 Basics

DiGr is built on a C-like base. Each program begins execution in an opt with no arguments called `main`, declared as follows :

```
opt main(){

        ...program code..
}
```

Your running program will start within these brackets.

DiGr has 5 variable types: integers, floating point numbers, strings, nodes and edges. The first 3 types are simple data types, similar to variables in other languages. Nodes and edges are more complex variable types that we will get to later.

First let's start with a simple statement that does something in the DiGr language.

```
opt main(){

        print("Hello World")!
}
```

print is a simple operation call that takes its contents, a string ”hello world” in this case, and displays it on the console. A program consists of multiple statements like this, each ending in an exclamation mark.

To store values for later use we have variables. Declare a new variable in the form:

```
type Variable_name1!
```

Type can be str, flt, int, node or edge. The variable name must start with either a lower or upper case character, and consist of any number of underscores, characters or numbers. The following statement declares a string variable.

```
str MyString1!
```

Set a variable to a certain value with the ’=’ symbol, either while declaring it or any time afterward.

```
str MyString = "DiGr"!
MyString = "Program"!
```

To store a collection of values in one symbol, use an array. An array is declared with a set size. Once declared,the size cannot be changed. The following example illustrates an array of integers with size 5 being declared:

```
int MyIntArray[5]!
```

You can initialize an array on declaration by assigning a bracket enclosed list of values of the appropriate type. Not the length of the list must not exceed the length of the array

```
int myList[5] = {1;2;3;4;5}!
```

2.2 Loops and Conditions

Iteration is handled with the while loops, which take conditional statements like:

- `value == value` : check to see if the values are equal.
- `value != value` : check to see if the values are not equal.

and the scalar comparisons <, > , <= , >=. Anything within the body of the while loop runs while these conditions hold. The following example illustrates the while loop:

```
while (myInt != 1)
{
        myInt = myInt - 1!
}
```

For control logic we have if else statements. Like other languages, we check a condition within the if statement and run the first block if statements if it evaluates to true, and the else block if the statement evaluates to false. The else block is optional. We can see its use in the following lines of code.

```
if(myVal ==5)
{
        newValue = 5!
}
else
{
        newValue = 6!
}
```

2.3 User Defined Operations

You can also declare operations to serve as functions. We declare these with the opt label, name and arguments. Arguments have a direction type, either in or out, a data type, and a name within the scope of the operation. Operation declaration cannot be nested. They are declared outside of opt main and can be referenced in any code before or after the opt declaration. The following is an example of an operation function.

```
opt addThree(in int n; out int return)
{
        return = n + 3!
}
```

This operation takes the value n, adds 3 to it and pass it back to the caller in the return variable. The following code calls it:

```
opt main(){

        int m!
        addThree(3;m)!
        print(m)! : This prints 6
}
```

Note that the argument passed must be a variable, as the operation sets its value.

2.4 Graphs

Now to get to the real strength of DiGr, the node and edge types, and their traversal. Node objects represent vertices in a graph, and an edge object is used to connect them. DiGr can connect two nodes with a simple statement.

```
node1 -> node2!
```

We have now created an edge between node1 and node2. The $->$ indicates that this is a directed edge, out from node1, in to node2. We could have done the reverse with the following statement:

```
node1 <- node2!
```

or

```
node2 -> node1!
```

If we wanted an undirected edge, we could use:

```
node1 -- node2!
```

There is an easier, quicker way to create edges between nodes. Using an array of nodes you can create what we call a Connection Context. In a single line of code we can connect any number of nodes with any type of connections within the array. The following line of code will show you how:

```
int myNodeArray[5] = |0 -> (2->4), 3 |!
```

With this one line of code we have created 3 connections between 4 nodes. The connection context is enclosed within the — — symbols. The numbers here reference the nodes at the array index. You can see the edge type between them. A comma after an integer or parenthesized unit allows us to connect the first node to multiple nodes with the same kind of edge. See the Language reference manual for more details.

We will be able to utilize these edge types with crawl operations on these nodes. Node objects have built in special properties which you can access by `nodevariable.property`.

- `myNode.parent(n)` : gives you the nth node that has a directed edge into myNode
- `myNode.child(n)`: the same thing as parent but the edge direction is reversed
- `myNode.inedges` and `myNode.outedges` both return just the number of those types of edges connected to myNode.
- You can retrieve those edges with the `node.outedge(n)` and `node.inedge(n)` properties, similar to the parent and child properties.

You can also define your own properties for a node with the following line of code.

```
myNode.weight = 5!
```

These attributes must be integers. If you attempt to reference an attribute that has not already been defined, the value will be 0.

Edge objects are similar in a lot of ways to the node object. They are implicitly created when you create a node connection, but can be declared independently in their own variable. Edges are declared with the edge type. They have properties similar to the node, that can access the nodes they are connecting. Additionally, they can be given additional properties in the same way as nodes.

2.5 Graph Traversal

To really make the most use of these connections we can use a crawl. A crawl is defined like a function, with `crawl name(args)` rather than `opt name(args)`. The body of the crawl itself usually only operates on a single node, though implicit in a crawl is a graph traversal function. The crawl moves through nodes that have been connected in the direction of their edges. It uses a queue to determine the order of the traversal, and calls a rule object (see below) to determine what (if anything) to add to the queue.

A crawl has two special key words to handle the traversal. The `current` symbol represents the node that the crawl is currently on. The `call` imperative executes the rule, which may or may not add any additional nodes to the queue.

Here is a very basic crawl which does not make use of its rule:

```
crawl myCrawl(in int compareValue)
{
        if(current.weight == compareValue)
        {
                print("this is the right node")!
        }

}
```

This crawl compares the current node's weight property with the value passed to the crawl. If these matches, it prints a message. To start a crawl, you call it like an opt but with additional special arguments.

```
opt myOpt(){

        myCrawl(5) from myNode with myRule!

}
```

The from-with statement at the end handle two additional arguments. myNode is the starting point, the first node to be processed with the crawl.

myRule is a rule, a special object that guides the crawl. A rule is declared like a function, but has no arguments.

```
rule myRule{
        ..rule code..
}
```

A rule's job is to decide which nodes are queued up for the crawl.

It has some special functions which manage the queue. It also has the `current` handle which points to the node the crawl is at. It can add a node to the queue with the `add(node)` function and add to the front of the queue with the `addFront(node)` function. In each case the argument passed must be a variable of type node.

Now that we have some idea as to what the rule is we can put the crawl and rule together. When the crawl runs and reaches the end of the body of statements, it looks at the first node in the queue. If there is something on the queue, it runs again, with this new node set to the `current` handle. Within the crawl you can add new nodes to the queue by invoking the rule with the `call` command. This adds nodes to the queue according to the rule set. You can also change the rule with the `set` command. The following examples illustrates both of these commands.

```
crawl newCrawl(in int someVar)
{
         call! :add new nodes to the queue, if applicable. :
        set(newRule)! :change the way we add nodes to the queue :
        call! : add new nodes with the new rule :
}
```

And there you have it. Within the crawl we can modify or output variables within a graph. And we use a rule to make traversal of this graph as simple as possible.

To put all the pieces together now, we have 3 code block types, the crawl, rule and opt. Opts are general functions with one main function for the program, crawls are specialized operations with iteration built in, and rules guide the crawls.

Hopefully these tools will be helpful to you in any graph related problem solving. There are subtleties in the language we have glossed over here. For further information, see the language reference manual below.

3 Language Reference Manual

3.1 Lexical conventions

There are 5 kinds of tokens: identifiers, keywords, constants, operators, and separators. Tokens are separated by whitespace or new line characters.

3.1.1 Comments

Comment blocks begin and end with the colon character (:).[1]

3.1.2 Identifiers

An identifier is a sequence of letters, numbers and underscores, that begin with a letter. Upper and lower case letters are considered distinct. Identifiers are at least one character long, but no maximum length. Identifiers cannot start with any reserved DiGr keywords.

3.1.3 Keywords

The following keywords are reserved for use by the language.

---

[1] It is DiGr tradition (but not syntactically required) to follow a starting comment character or lead a closing comment character with a left or right parenthesis to form a smiley or frowny face.

```
add
addby
addfront
addbyfront
call
current
crawl
edge
flt
from
in
int
node
opt
out
print
queue
rule
set
str
while
with
```

3.1.4 Constants

Constants types in DiGr are either `int`s, `flt`s, or `str`s. They will be discussed later.

3.1.5 Operators

The list of operators in DiGr, grouped into orders of precedence from highest to lowest, is below. Note that not all operators act on all DiGr types.

```
* / %
+ -
-> <- --
== !=
<= < >= >
|
&& ||
=
```

Some of these are binary operators, and some have a more specialized use. Their application will be discussed in the relevant section below.

3.1.6 Separators

Semicolons (;) separate arguments in opt definitions. The comma character (,) is used to separate argument in an opt call, node children in a connection context, and initial values in an array declaration. Curly brackets are used to separate blocks of code.

3.1.7 Scoping and Execution

DiGr has a global scope in which `crawls`, `opts` and `rules` (only) may be declared. Every DiGr program must contain an `opt` named `main` which takes no arguments, which is where code execution begins.

DiGr is statically and locally scoped within each crawl, rule or opt, but an important exception is that modifying outgoing variables modifies the corresponding variable in the scope the crawl or opt was called from.

3.1.8 Statements

DiGr is an imperative language. All statements are terminated with the ! symbol.[2] Statements can be grouped into blocks using open curly brace { and closed curly brace }.

3.1.9 The print() opt

`print()` is a built in DiGr opt that prints its argument, which can be any `int`, `str`, or `flt` separated by a comma (,). It is the basic mechanism by which DiGr passes information from a running program.

3.2 Primitive Types

There are five kinds of primitive types: `int`, `flt`, `str`, `node`, `edge`, and several derived types, including `rule` and `crawl`. All primitive types must be declared before they can be assigned or dereferenced. Primitive types are declared with their type name and the name of the bound identifier:

```
type identifiername!
```

All primitive types are assigned by being on the left side of the `=` operator. A primitive type can be assigned as it is declared:

```
type identifiername = initial_value!
```

---

[2] In DiGr, when you write a statement, you must really mean it!

3.2.1 Basic Primitive Types

The `int` (integer) is a signed, base 10 whole number. The range of `ints` is machine-specific.

Example:

```
int magNum = 42!
```

`Flts` (floats) are a representation of real, decimal numbers.

```
flt pi = 3.14!
int pi = 3.14! :( error ):
```

`Strs` (strings) begin and end with a double quote (”). The double quote itself (”) and the backslash (\) must both be escaped with a backslash. (e.g. \\ and \"). Strings are compared lexicographically.

Example:

```
str myName = "Ari"!
str myNumber = 10! :( won’t work, 10 is not a str, it is an int ):
str myNumber = "10"! :) this will work (:
```

The common mathematical operators `+ - * /` have the usual meaning when used between two `ints`, two `flts`, or an `int` and a `flt`. % is defined only between two `ints`. In the case of an `int` and a `flt`, the result will match the argument with the least precision.

Example:

```
int numA = 42!
flt numA1 = 42!
flt numB = 10.5!
flt result = numA + numB!         : result will be 52, not 52.5. :
: This would also be true if result was of type int. :
flt result1 = numA1 + numB!     : result1 will be 52.5 :
```

The addition operator can be used on two strings, and results in concatenation. If the result is not stored anywhere, the concatenation has no effect on the original strings.

Example:

```
str first = "Ari"!
str last = "Golub"!
str fullname = first + " " + last!
print(fullName)!                  : prints "Ari Golub" :
```

3.2.2 Node

The `node` is a primitive type in DiGr that represents a node in directed and undirected graphs, and other abstract objects. `Nodes` are connected to other `nodes` through `edges`. A `node` must be declared before it can be used, unless it is created inside a connection context (see below). A `node` can hold as many *attributes* of any name as the user wishes. Attributes are designed to be a flexible concept, and can be created and modified on the fly with little overhead.

Node opts

Each nodes has built in `opts` (DiGr functions) that can be called by placing a dot (.) after the name of the node followed by the `opt` you wish to call. The functions are `child`, `parent`, `inedge`, `inedges`, `outedge` and `outedges`.

- `(node) node.child(int n)` : Returns the (n+1)th child of the node counted by `node.children()`. If n is not within the inclusive range `(0,node.children()-1)`, this function throws a runtime exception.
- `(node) node.parent(int n)` : Returns the (n+1)th parent of the node counted by `node.parents()`. If n is not within the inclusive range `(0,node.parents()-1)`, this function throws a runtime exception.
- `(int) node.inedges` : Returns the integer number of edges coming in to the node.
- `(int) node.outedges` : Returns the integer number of edges coming out of the node.
- `(edge) node.inedge(int n)` : Returns the (n+1)th edge coming into the node. If n is not within the inclusive range `(0,node.inedges()-1)`, this function throws a runtime error.
- `(edge) node.outedge(int n)` : Returns the (n+1)th edges going out of the node. If n is not within the inclusive range `(0,node.outedges()-1)`, this function throws a runtime error.
- `(int) node.<attributeName>` : Returns the value of the attribute named `<attributeName>`. See below for more information.

Undirected edges qualify as both in and out edges for the purposes of these functions. Thus, the children and parents of the current node can be the same set of nodes.

Example (using connection context language, see below):

```
node tree[5] = | 0->1,(2->4) |!
node head = tree[0]!
print(head.inedges)!          : prints "0" :
print(head.outedges)!         : prints "2" :
edge myEdge = head.inedges(0)!  : myEdge is the edge from 0 -> 1 :
```

Node attributes

Attributes are integer values stored under a variable name within the node. Attributes are defined simply by attempting to assign a value to them. In DiGr, referencing an undefined attribute automatically creates an attribute of that name with a value of 0 in the node!

To get or set the value of an attribute, follow the `node` name with a dot (.) and the name of the attribute. Attributes are declared by treating this as an identifier, and can be normally assigned with (=). Node attribute names cannot start with the names of any built-in node functions, including `inedge`, `outedge`, `child` and `parent`.

Example:

```
node myNode!
myNode.weight = 32!
int twoWeight = myNode.weight * 2!        : twoWeight = 64 :
```

Attributes that are declared but not initialized will have value of zero.

```
node myNode!
int t = myNode.weight!
print(t)!      : prints 0 :
```

Operations on Nodes

The (=) operator between two nodes binds the identifier on the left to the object dereferenced by the identifier on the right.

The (`->`) and (`<-`) operators between two nodes will create an unnamed directed edge from the first node to the second, or the second to the first, respectively. Alternatively, the (`--`) operator will create an unnamed undirected edge.

Example:

```
node tree[5] = | 0 -> 1, (2 -> 4) |!
node head = tree[0]!
head.weight = 10!
node alt = tree[1]!
alt.weight = 20!
alt = head!
print(alt.weight)!    : prints 10 :
node last = tree[4]!  : reference to node number 4 from first line :
head <- last!
: creates a directed edge out of last and into head :
```

3.2.3 Edge

The `edge` is the complementary type to a `node` in DiGr. An edge can be explicitly declared and named, but most often an `edge` object is created anonymously as a result of linking nodes. An edge not bound to an identifier can still be accessed via the `inedge()`/`outedge()` function of a `node`. Like `nodes`, `edges` can be given any number of attributes.

Example:

```
node tree[5] = | 0->1,(2->4) |!
node head = tree[0]!
edge myEdge = head.outedges(0)!
```

Declaring a handle to an edge but not assigning to anything will create two anonymous nodes for the directed edge to point between.

```
edge e!
node nout = edge.innode!
: valid reference since this object exists :
```

Edge opts

Each `edge` has built in `opts` (DiGr functions) that can be called by placing a dot (.) after the name of the node followed by the `opt` you wish to call. The functions are `innode` and `outnode`.

- `(node) edge.innode` : Returns the node this edge is pointing to.
- `(node) edge.outnode` : Returns the node this edge is leaving
- `(int) edge.<attributeName>` : Returns the value of the attribute named `<attributeName>`. See below for more information.

Undirected edges are implemented as two directed edges in both configurations between the two nodes.

Edge attributes

`Edges` have attributes in a manner almost identical to `nodes`. To get or set the value of an attribute, follow the `edge` name with a dot (.) and the name of the attribute. Attributes are declared by treating this as an identifier, and can be normally assigned with (=). Edge attribute names cannot start with the names of any built-in edge functions, including `innode` and `outnode`.

Example:

```
node tree[4] = | 0 -> 1, (2 -> 4) |!
node head = tree[0]!
edge myEdge = head.inedges(1)!
: myEdge points to edge between 0 and 2 :
node three = myEdge.innode!
: three points to node 2 :
myEdge.value = 17!
```

Operations on Edges

The (=) operator between two `edges` binds the identifier on the left to the object dereferenced by the identifier on the left.

Example:

```
node tree[5] = | 0 -> 1, (2 -> 4)|!
node head = tree[0]!
edge e1 = head.outedges(0)!     : e1 is between 0 and 1 :
edge e2 = tree[2].outedges(0)!  : e2 is between 2 and 4 :
e1 = e2!
: the handle e1 now refers to the edge between 2 and 4 :
```

3.3 Derived Types

3.3.1 Arrays

DiGr supports arrays built out of any primitive type. Arrays are allocated by giving a type and an identifier for the array, similar to creating a single instance of that type, but following the identifier with an open and closed bracket and the integer number of elements in the array in between the brackets.

```
type arrayidentifier[number_elements]!
```

Initialization
Alternately, the user can initialize the entire array at declaration by placing the initial value of sequential elements inside curly brackets, separated by commas. Giving the array a different array length than what the DiGr compiler infers from context will cause an error.
Examples:

```
int arr1[3]!
arr1[0] = 10!
int arrDeclared[4] = {1, 2, 3, 4}!
node tree[3] = | 0 -> (1 -> 2)|!
node badIntTree[3] = {17, 41}!
node badNodeTree[2] = |1 -> 2 -> 3|!
: wrong, too many nodes in connection context for array size :
```

Array operations
Array indexing begins at 0, and elements are accessed by appending square brackets with the element index to the end of the array. Trying to index into an array outside the bounds of the array will generate a run-time error.

The (=) operator can be used on individual elements of an array to change the value of that element. It can *not* be used to set one array equal to another.

3.3.2 Opts

`Opts` (operations) are the DiGr functions. `opts` must declare their input and output variables as part of their signature. As a result, there are no return types in DiGr `opts`. When called, ”in” variables can be constants, but ”out” variables must be a previously declared identifier of the proper type.

`Opt`s are declared with a sequence of arguments in a parenthesis block separated by semicolons (;), and with the body of the `opt` inside curly brackets. Each argument is denoted `in` or `out`, its type and is given the local identifier to which the value is bound when the `opt` is called. The body of an `opt` can contain any standard DiGr code except further `opt`/`crawl`/`rule` definitions.

Example:

```
opt myFunc(in int var1; in int var2; out int result) {
      if ( var1 > 10 ) {
           result = var1 * 2!
      }
      else {
           result = var1 + var2!
      }
}

int result!
myFunc(3,12,result)!
print(result)!        : prints 15 :
myFunc(11,7,result)!
print(result)!        : prints 22 :
```

3.3.3 Crawl

A `crawl` is the DiGr type used to traverse a tree. `Crawl`s are similar to an `opt` that will run its code on every node it visits, when given which node to start at and the rules for moving to additional nodes. They are general enough to be used for a variety of purposes, but provide enough built-in functionality to quickly define different traversal behaviors and operations.

When called, a `crawl` creates an internal queue of the next nodes to visit, and visits them one at a time by popping the next node off the front of the queue. It executes its code at each node. Each `crawl` also has a `rule` that controls which nodes to add to the queue at any given moment (usually somehow connected to the current node, see the `rule` section below). A `crawl` only knows about one `rule` at a time, but this rule can be changed dynamically. In this way, DiGr implements a level of abstraction in tree traversal: `crawl`s describe *what* one does at a `node`, while `rule`s describe *where* one goes next. When run, a `crawl` must be given an initial `node` to start at and an initial `rule` for how to move on from there.

Variables passed to crawls persist between iterations of the crawl code being executed on nodes, but variables declared in the crawl are redeclared each time. Crawls may recursively call themselves.

`Crawl`s are defined with an `opt`-like set of in and out variables:

```
crawl myCrawl(in intype1 invar1; ... ; out outtype1 outvar1; ...) {
```

```
  : list of crawl statements
}
```

Crawls are executed as follows:

```
myCrawl(myvar1; myvar2; ...; myoutvar1; ...) from mynode1 with myrule;
```

Where "mynode1" is a `node` that serves as the initial starting point of the tree traversal, and "myrule" is a `rule` that assigns the initial traversal `rule` to this crawl (see below). `from` and `with` are reserved keywords used primarily for readability.

In addition to standard DiGr code, the crawl body can contain the following three operations:

- `(node) current`: Reference to the current node the crawl is visiting. This is the handle the crawl uses to perform local computation on the node.
- `call`: Executes the current `crawl`'s `rule`. There is no return value.
- `set <newrule>`: Here, `<newrule>` is a `rule`. This updates the rule currently executed by the crawl upon a `call` statement. There is no return value.

After executing all statements in the crawl body and reaching the closing parenthesis, the crawl automatically moves on to the next node in the queue and starts again. If there are no more nodes in the queue, the crawl terminates with no return value. Any locally-scoped in and out variables persist between crawl operations on different nodes.

3.3.4 Rule

A `rule` is a special form of `opt` with no arguments that is used by a `crawl` to control and inform tree traversal. This abstraction separates tree traversal from tree modification (which is done in the body of a `crawl`, see above) into two distinct sets of operations. A `rule` can be used in any number of `crawls`, but a `crawl` knows about only one `rule` (but can update which one it is using).

Given a current `node`, a rule determines which `nodes`, if any, should be added to the internal double-ended queue. `Rules` can modify either end of the queue, but `crawls` will always pop the next node off the *front* to decide where to go next. A `rule` is declared as follows:

```
rule myRule {
    : rule body goes here :
}
```

There is one special keyword and four built-in opts that the `rule` uses to manipulate the queue and guide an effective crawl:

- `(node) current`: A reference to the current node the crawl is at.
- `add(node)`: Takes a node as an argument and adds it to the *back* of the queue. This operation does not return a value.
- `addfront(node)`: Takes a node as an argument and adds it to the *front* of the queue. This operation does not return a value.
- `addby(<property>, <ordering>, number_to_add )`: Takes 3 arguments: the property on which to sort, the ordering to use on that property, and how many of the winning nodes to add to the *back* of the queue. The syntax for the first two arguments is described below. If there are fewer children than the amount requested, `addBy` will add as many as it can. Returns nothing.
- `addbyfront(<property>, <ordering>, number_to_add )`: Similar to `addBy()`, but adds the winning nodes to the *front* of the queue.

The first argument to `addby` and `addbyfront` is the property being evaluated to determine queuing order, written as `edge.<attName>` or `node.<attName>`, where `<attName>` is the name of the attribute to be used for selection, and the keyword `node` or `edge` indicates whether the rule is to sort children nodes by their attributes or by the attributes of the edges connecting them to the current node.

The second argument describes how to order nodes or edges using the selected attribute. The special symbols dollar sign ($) and tilde (˜) tell the rule to sort in default ascending or descending order, respectively.

The third argument sets a maximum on the number of nodes to add to the queue. The `rule` will add up to, but not over, this number of nodes to the queue.

Example:

```
rule depthFirst {
    int n = 0!
    while (n < current.outedges) {
          add(current.child(n))!
          n= n+1!
    }
}

rule breadthFirst {
    int n = 0!
    while (n < current.outedges) {
          addfront(current.child(n))!
          n= n+1!
    }
}
```

```
        rule weightFirstThreeMax {,
            addby(node.weight,$,3)!
            : this will add at most three children to the back of the
            queue, starting with the node with the greatest weight  :
        }
```

Note that the only handle to the tree being crawled is `current`, the node that the crawl is at. This is a design choice to enforce the abstraction that a `rule` only does evaluation, and not modification.

3.4 Connection Context

The connection context is the easiest way to create an entire tree of nodes in a single line. The connection context has a special grammar and is only valid inside pipe (`|`) operators. It *must* be on the right-hand side of an assignment to a node array, which has the same size as the number of nodes in the described tree.

3.4.1 Grammar

The grammar of the language used to describe a tree inside the connection context can be formally defined as follows (with `tree` as the starting symbol):

```
tree:               node edge children
edge:               -> | <- | --
children:           child, children | child
child:              node | ( tree )
node:               LIT_INT
```

`LIT_INT` is any integer. Integers are references to the node array that prefixes the connection context, and are 0-indexed. The `->` symbol is somewhat analogous to the standard DiGr operator which is written the same way. It binds the node referenced on the left to the tops of the subtrees listed on the right.

The multiple children of a node are separated by commas, and can be a single node (e.g. `|0->1|`) or a subgraph which is wrapped in parenthesis (e.g. `|0->(1->2)|`, in which case node 1 is connected to node 2), or to multiple nodes (e.g. `|0->1,2,3|`, in which case node 0 is connected to nodes 1 2 and 3). The `--` operator is similar to `->` but creates undirected edges.

The size of the array must be large enough to include all of the nodes listed. If a node isn't listed in the connection context, that element of the array is a free-floating node unconnected to the rest of the tree.

Examples:

```
: create a two node graph with a directed edge between the two:
         node simple[2] = |0 -> 1|!
```

```
: create a three node graph with node 0 pointing to node 1 and
: node 1 pointing to node 2:
        node lineofthree[3] = |0 -> (1 -> 2)|!
: create a three node graph with node 0 pointing to nodes 1 and 2:
        node split[3] = |0 -> 1, 2|!
: create a directed four-cycle:
        node fourcycle[4] = |0 -> (1 -> (2 -> (3 -> 4)))|!
: create a complete 4-graph:
       node fourcomplete[4] = |0--(1--2,3),(2--3),3|!
: create a 6-node bipartite graph (with odds and evens in the two
: partitions, respectively):
        node bipartite[6] = |0--(1--(2--3,5)),(3--(4--5))|!
```

3.5 Logic

3.5.1 Conditional Logic

DiGr uses C-style ”if then else” conditional logic statements. These statements can take the following forms:

```
if (expression) { list_of_statements }
if (expression) { list_of_statements } else { list_of_statements }
```

where ”expression” has integer type (DiGr boolean expressions are equivalent to `ints`), and the statements are standard DiGr statements.

3.5.2 Boolean Logic

DiGr has several boolean logic symbols: `||` (conditional or), `&&` (conditional and), `==` (conditional equality), `!=` (conditional inequality), `<` (less than), `<=` (less than or equal to), `>` (greater than), and `>=` (greater than or equal to). These symbols can be used to create boolean statements of arbitrary complexity for use in while loops or if statements.

Example:

```
node myNode!
myNode.weight = 19!
myNode.id = 21!
while(myNode.weight < 10 || myNode.id == 13) {
        myNode.weight = myNode.weight + 1!
        if((myNode.weight % 20) == 0) {
                myNode.id == myNode.id * 2!
        }
}
print(myNode.id)!
: this outputs 42 :
```

3.6 Control Logic

All looping in DiGr is done in `while` loops. A `while` loop begins with a logical boolean evaluation; if the evaluation results in true, the body of the while block is executed. If it is false, the block is skipped. Once the body is executed, the `while` statement is re-evaluated to check if it should run again. The syntax of the loop is C-like: the condition to be tested follows the keyword `while` in parenthesis, which is then followed by a statement block.

There are no ”break” or ”exit” commands to escape a while loop without violating the while condition. To exit a while loop, the condition must evaluate to false by the end of the while block. An example of a while loop:

```
int fact = 1!
int n = 5!
while ( n > 1 ) {
       fact = fact * n!
       n = n - 1!
}
print(fact)! : prints 120
```

4 Project Plan

Our weekly meetings with our TA Hemanth helped us greatly in our planning process. At the early stages, while we were still figuring out what DiGr was really about, we had weekly meetings on Monday nights to discuss a plan of attack for the rest of the project. We set several deadlines, some of which we were able to meet and some of which had to be pushed back due to heavy courseloads and the loss of a teammate. Around the midway point of the semester we started to diverge in the work we were doing: Ari and Bryan focused on the frontend while Dennis began the process of writing an airtight backend. At this point, tasks were atomic enough that the team could split up and each person could implement his part of the design contract. As the semester progressed, meetings became more frequent but less formal in how often they would occur or how long they would last. As the semester wound down and reading week began, the team met almost every night to work on the project.

4.1 Project Timeline

Our ideal timeline is outlined below. As is common, there was more of a crunch towards the end of the project than we expected, as unknown unknowns came up.

- 11/14: Begin scanner/parser/ast development in parallel.
- 11/22: Begin first stage of interpreter development (namespace issues, scoping).
- 12/1: First tests ran, at the syntactic/semantic level.
- 12/3: 90% completion of the core of the DiGr front-end.
- 12/5: Begin development of C++ AST + compiler in parallel, while also working on the C++ backend and translator. Front-end is stable but occasional changes in the language are written in.
- 12/10: 95% completion on C++ backend and compiler.
- 12/11: 95% completion on translator. First complete pipeline from DiGr code to executable output.
- 12/15: First run of entire test suite. Many errors.
- 12/15-12/22: Finishing implementation, test battery, writing documentation.

4.2 Style Guide

The focus of our style plan was to break up the OCaml code into its logical pieces with a tabbing and newline scheme. All statements under the let statement that defines a function is given an additional level of tabbing. In a match statement, all the values are indented and the match comparisons form a single column. If the result section of each match spills past the readable length of our text editor, it was moved to the next line and an additional level of tabbing was added.

`If` statements were lined up in a single column as follows:

```
if condition
then statement
else statement
```

Any nested statements within these clauses is indented. When a single line of code gets long, we break it up over multiple indented lines, usually by the various arguments being passed to a method.

In the compiler and translator, our naming scheme for bound functions was to make explicit what the inputs and outputs are. For example, `cexpr_from_expr` took a single DiGr AST `expr` as an argument and returned a single C++ AST `cexpr`. This saved some significant time looking up the formatting of various functions when dealing with crawling the typed abstract syntax trees.

4.3 Team Member Roles and Responsibilities

Due partially to the small size of our team, and partially to a need to develop quickly, all three team members made at least nominal changes to every part of the compiler. With that in mind, the *main* duties of each team member were as follows:

- **Bryan** (Team Leader): scanner / parser front-end, type checking / static semantic checking in the interpreter, C++ and DiGr AST development, team organization, language white paper
- **Dennis**: initial symbol table / static semantic checking in the interpreter, translation involving DiGr objects and opt/crawl/rules, compilation work, C++ backend, documentation structure.
- **Ari**: scanner / parser front-end, translation work involving connection contexts and arrays, some compilation work, testing suite and test paradigm writeup.

The initial language design, as well as the language reference manual, was a team responsibility.

4.4 Software Development Environment

The DiGr compiler itself is written in OCaml, and the scanner and parser use the OCaml lex and yacc extensions. The backend is written in C++ with the use of a handful of specific standard libraries (vector, algorithm, iostream, etc.). The documentation is written in LaTeX, `make` was used for build management, a `subversion` repository hosted by Google Code was used for version control, and some flowcharts in the documentation were made with GraphViz `dot`. `bash` shell scripting was used to run out test suite, and a `python` script formatted the commit logs and actual code base for inclusion into the final report.

The development tools used varied among team members. Dennis used plain old `emacs` and the command line. Ari used `gedit` with Ocaml syntax highlighting and command line. Bryan used `gedit` as well with cygwin to compile all Ocaml code.

4.5 Project Log

```
r214 | dennis.v.perepelitsa | 2010-12-22 23:45:06 -0500 (Wed, 22 Dec 2010)
    spell-checking all final paper modules, ready for turn in!
r213 | dennis.v.perepelitsa | 2010-12-22 23:38:58 -0500 (Wed, 22 Dec 2010)
    mile reformatting of some code to fix LaTeX overfull h boxes
r212 | dennis.v.perepelitsa | 2010-12-22 23:33:24 -0500 (Wed, 22 Dec 2010)
    testing section done. almost there...
r211 | dennis.v.perepelitsa | 2010-12-22 22:44:38 -0500 (Wed, 22 Dec 2010)
    folding in tutorial and updating to present TeX standard...
r210 | dennis.v.perepelitsa | 2010-12-22 22:14:02 -0500 (Wed, 22 Dec 2010)
    integrating several project plan sections
r209 | oemlerb | 2010-12-22 18:29:27 -0500 (Wed, 22 Dec 2010)
    A very brief style plan write up....  Pretty basic.
r208 | oemlerb | 2010-12-22 18:03:04 -0500 (Wed, 22 Dec 2010)
    Small change
r207 | oemlerb | 2010-12-22 18:02:28 -0500 (Wed, 22 Dec 2010)
    Style,  might not be perfect, but its all I can stomach at the moment
r206 | dennis.v.perepelitsa | 2010-12-22 14:47:41 -0500 (Wed, 22 Dec 2010)
    final cut presentation
r205 | AriGolub | 2010-12-22 14:40:52 -0500 (Wed, 22 Dec 2010)
    runtime test added
r204 | oemlerb | 2010-12-22 14:26:35 -0500 (Wed, 22 Dec 2010)
    fixed typos
r203 | AriGolub | 2010-12-22 14:20:28 -0500 (Wed, 22 Dec 2010)
    projectplan
r202 | AriGolub | 2010-12-22 14:09:13 -0500 (Wed, 22 Dec 2010)
    anything?
r201 | oemlerb | 2010-12-22 13:50:44 -0500 (Wed, 22 Dec 2010)
    Some slight changes
r200 | dennis.v.perepelitsa | 2010-12-22 13:47:14 -0500 (Wed, 22 Dec 2010)
    presentation so far (stealing commit 200 from Bryan)
r199 | oemlerb | 2010-12-22 13:32:56 -0500 (Wed, 22 Dec 2010)
    Made it work with variables
r198 | AriGolub | 2010-12-22 13:30:42 -0500 (Wed, 22 Dec 2010)
    changed add/addfront/crawl to variable
r197 | dennis.v.perepelitsa | 2010-12-22 13:20:23 -0500 (Wed, 22 Dec 2010)
    passing 0 to addby and addbyfront DTRT
r196 | oemlerb | 2010-12-22 13:03:55 -0500 (Wed, 22 Dec 2010)
    Getting rid of patronizing messages.
r195 | oemlerb | 2010-12-22 12:51:06 -0500 (Wed, 22 Dec 2010)
    Removed todos, added some checking
r194 | AriGolub | 2010-12-22 12:37:40 -0500 (Wed, 22 Dec 2010)
    cleaner
```

```
r193 | oemlerb | 2010-12-22 06:38:55 -0500 (Wed, 22 Dec 2010)
    Think i covered enough.  Thats it for now.
r192 | oemlerb | 2010-12-22 04:52:01 -0500 (Wed, 22 Dec 2010)
    Committing in case this computer dies.  Not quite done
r191 | oemlerb | 2010-12-22 04:24:03 -0500 (Wed, 22 Dec 2010)
    Committing in case this computer dies.  Not quite done
r190 | dennis.v.perepelitsa | 2010-12-22 02:21:03 -0500 (Wed, 22 Dec 2010)
    squashing last few bugs. all tests pass
r189 | dennis.v.perepelitsa | 2010-12-22 01:56:00 -0500 (Wed, 22 Dec 2010)
    cleaning up Ocaml warnings and final test polishing...
r188 | AriGolub | 2010-12-22 01:45:37 -0500 (Wed, 22 Dec 2010)
    whoops
r187 | oemlerb | 2010-12-22 01:31:13 -0500 (Wed, 22 Dec 2010)
    Added nodeChild and nodeParent
r186 | oemlerb | 2010-12-22 01:18:08 -0500 (Wed, 22 Dec 2010)
    Updated with current and edge attributes
r185 | dennis.v.perepelitsa | 2010-12-22 01:09:06 -0500 (Wed, 22 Dec 2010)
    last bit of pipeline for static semantic verification!
r184 | oemlerb | 2010-12-22 00:47:47 -0500 (Wed, 22 Dec 2010)
    Added a whole lot of checking and a whole lot of love
r183 | dennis.v.perepelitsa | 2010-12-22 00:47:40 -0500 (Wed, 22 Dec 2010)
    folding in Ari's test plan. I am anal and will probably tweak grammar later
r182 | AriGolub | 2010-12-22 00:40:32 -0500 (Wed, 22 Dec 2010)
    more tests
r181 | dennis.v.perepelitsa | 2010-12-22 00:40:03 -0500 (Wed, 22 Dec 2010)
    intro and LRM are good enough to push
r180 | AriGolub | 2010-12-22 00:10:20 -0500 (Wed, 22 Dec 2010)
    more in testplan
r179 | AriGolub | 2010-12-22 00:06:55 -0500 (Wed, 22 Dec 2010)
    stuff that happened
r178 | dennis.v.perepelitsa | 2010-12-22 00:06:16 -0500 (Wed, 22 Dec 2010)
    now with correct tree ordering
r177 | AriGolub | 2010-12-21 23:38:31 -0500 (Tue, 21 Dec 2010)
    changed the word varible to variable
r176 | AriGolub | 2010-12-21 23:36:20 -0500 (Tue, 21 Dec 2010)
    what did i do again... oh right, variable stuff
r175 | dennis.v.perepelitsa | 2010-12-21 22:55:24 -0500 (Tue, 21 Dec 2010)
    fixed weird crawl argument ordering
r174 | dennis.v.perepelitsa | 2010-12-21 22:46:13 -0500 (Tue, 21 Dec 2010)
    oops, typo
r173 | dennis.v.perepelitsa | 2010-12-21 22:40:28 -0500 (Tue, 21 Dec 2010)
    basic run-time error handling in child, parent, inedge, outedge
r172 | dennis.v.perepelitsa | 2010-12-21 22:17:14 -0500 (Tue, 21 Dec 2010)
    child() and parent() built-in opts work
r171 | dennis.v.perepelitsa | 2010-12-21 22:03:45 -0500 (Tue, 21 Dec 2010)
    anonymous edges are no longer null pointers
r170 | dennis.v.perepelitsa | 2010-12-21 21:55:44 -0500 (Tue, 21 Dec 2010)
    connection contexts now 0-index into tree nodes
r169 | dennis.v.perepelitsa | 2010-12-21 21:36:42 -0500 (Tue, 21 Dec 2010)
    fucking awesome in-order and post-order demo
r168 | dennis.v.perepelitsa | 2010-12-21 21:16:59 -0500 (Tue, 21 Dec 2010)
```

```
    fixing things until depth first works!
r167 | AriGolub | 2010-12-21 21:16:49 -0500 (Tue, 21 Dec 2010)
    test plan (updated)
r166 | AriGolub | 2010-12-21 21:16:07 -0500 (Tue, 21 Dec 2010)
    test plan (test/testplan.txt)
r165 | AriGolub | 2010-12-21 20:29:07 -0500 (Tue, 21 Dec 2010)
    negative numbers
r164 | AriGolub | 2010-12-21 19:40:57 -0500 (Tue, 21 Dec 2010)
    changed name of test script
r163 | AriGolub | 2010-12-21 19:27:36 -0500 (Tue, 21 Dec 2010)
    fixed test programs
r162 | oemlerb | 2010-12-21 18:15:27 -0500 (Tue, 21 Dec 2010)
    Start of the tutorial.  WORK IN PROGRESS
r161 | oemlerb | 2010-12-21 14:20:30 -0500 (Tue, 21 Dec 2010)
    Implemented a few more things.  Now returns false if there is an error
r160 | AriGolub | 2010-12-21 01:39:24 -0500 (Tue, 21 Dec 2010)
    fixing with ; instead of ,
r159 | dennis.v.perepelitsa | 2010-12-21 01:22:05 -0500 (Tue, 21 Dec 2010)
    added project log to final report...
r158 | oemlerb | 2010-12-21 01:20:54 -0500 (Tue, 21 Dec 2010)
    A few more things for the todo, or at least just to consider
r157 | oemlerb | 2010-12-21 01:04:18 -0500 (Tue, 21 Dec 2010)
    My short blurb
r156 | AriGolub | 2010-12-21 00:54:40 -0500 (Tue, 21 Dec 2010)
    added my environment
r155 | dennis.v.perepelitsa | 2010-12-21 00:53:33 -0500 (Tue, 21 Dec 2010)
    creating code-included appendix
r154 | oemlerb | 2010-12-21 00:43:33 -0500 (Tue, 21 Dec 2010)
    Lesson learned
r153 | AriGolub | 2010-12-21 00:32:58 -0500 (Tue, 21 Dec 2010)
    what i learned
r152 | dennis.v.perepelitsa | 2010-12-21 00:27:37 -0500 (Tue, 21 Dec 2010)
    finished architecture writeup, initial Who Did What section
r151 | oemlerb | 2010-12-20 18:32:31 -0500 (Mon, 20 Dec 2010)
    Properly checking crawls
r150 | dennis.v.perepelitsa | 2010-12-20 17:04:23 -0500 (Mon, 20 Dec 2010)
    small changes
r149 | dennis.v.perepelitsa | 2010-12-20 16:47:35 -0500 (Mon, 20 Dec 2010)
    up before LRM.4.0, skipped arrays...
r148 | dennis.v.perepelitsa | 2010-12-20 16:21:19 -0500 (Mon, 20 Dec 2010)
    TODOs for myself and implementation changes to match LRM
r147 | dennis.v.perepelitsa | 2010-12-20 16:19:15 -0500 (Mon, 20 Dec 2010)
    commit changes through LRM.2.4
r146 | oemlerb | 2010-12-20 14:35:44 -0500 (Mon, 20 Dec 2010)
    Fixed order in which we were evaluating nested statements
r145 | dennis.v.perepelitsa | 2010-12-20 12:16:35 -0500 (Mon, 20 Dec 2010)
    stubbing project plan, adding lesson learned, folding in LRM, some introduction editi
    ng
r144 | dennis.v.perepelitsa | 2010-12-20 11:20:13 -0500 (Mon, 20 Dec 2010)
    starting work on architecture writeup
r143 | oemlerb | 2010-12-20 04:34:36 -0500 (Mon, 20 Dec 2010)
```

```
    Error message was off
r142 | oemlerb | 2010-12-20 04:27:27 -0500 (Mon, 20 Dec 2010)
    Cant stop, wont stop, changed order of binding for and ors,
    better type checking with different operations.  Rocking on
r141 | AriGolub | 2010-12-20 03:39:24 -0500 (Mon, 20 Dec 2010)
    working script for testing
r140 | AriGolub | 2010-12-20 03:10:35 -0500 (Mon, 20 Dec 2010)
    removed stupid .txt files in test folder
r139 | AriGolub | 2010-12-20 03:04:07 -0500 (Mon, 20 Dec 2010)
    new tester files
r138 | AriGolub | 2010-12-20 02:13:37 -0500 (Mon, 20 Dec 2010)
    keep ignoring, but not FOR LONG
r137 | oemlerb | 2010-12-20 01:47:16 -0500 (Mon, 20 Dec 2010)
    Function argument checking
r136 | AriGolub | 2010-12-20 01:15:56 -0500 (Mon, 20 Dec 2010)
    ignore
r135 | AriGolub | 2010-12-20 01:12:56 -0500 (Mon, 20 Dec 2010)
    ignore
r134 | AriGolub | 2010-12-19 23:51:08 -0500 (Sun, 19 Dec 2010)
    ignore these commits, i have to sync between laptop and cunix and i need to push tiny
    , tiny changes. enjoy
r133 | AriGolub | 2010-12-19 23:45:52 -0500 (Sun, 19 Dec 2010)
    more test cases
r132 | AriGolub | 2010-12-19 23:39:41 -0500 (Sun, 19 Dec 2010)
    test cases
r131 | oemlerb | 2010-12-19 19:11:37 -0500 (Sun, 19 Dec 2010)
    Checking proper argument passing ..  WOOOT WOOOH know what im sayin
r130 | AriGolub | 2010-12-19 18:06:00 -0500 (Sun, 19 Dec 2010)
    ok, gonna start testing now
r129 | oemlerb | 2010-12-19 18:03:40 -0500 (Sun, 19 Dec 2010)
    Updated array tests
r128 | oemlerb | 2010-12-19 17:56:06 -0500 (Sun, 19 Dec 2010)
    Indexed arrays being evaluated properly
r127 | oemlerb | 2010-12-19 17:37:19 -0500 (Sun, 19 Dec 2010)
    Actual checking imp
r126 | AriGolub | 2010-12-19 17:34:56 -0500 (Sun, 19 Dec 2010)
    doin' work
r125 | oemlerb | 2010-12-19 16:34:47 -0500 (Sun, 19 Dec 2010)
    Ever closer
r124 | oemlerb | 2010-12-19 14:50:15 -0500 (Sun, 19 Dec 2010)
    type checks almost working
r123 | dennis.v.perepelitsa | 2010-12-19 14:49:29 -0500 (Sun, 19 Dec 2010)
    starting documentation push...
r122 | dennis.v.perepelitsa | 2010-12-19 14:03:24 -0500 (Sun, 19 Dec 2010)
    fixing compiler error and edge types in connection contexts
r121 | dennis.v.perepelitsa | 2010-12-19 13:56:13 -0500 (Sun, 19 Dec 2010)
    oops, now grammar back to unambiguous (but we have to type check the static arrays)
r120 | dennis.v.perepelitsa | 2010-12-19 13:49:34 -0500 (Sun, 19 Dec 2010)
    connection contexts --> sequence of statements about arrays
r119 | AriGolub | 2010-12-19 12:31:57 -0500 (Sun, 19 Dec 2010)
    so close to working concon, but no
```

```
r118 | AriGolub | 2010-12-18 22:22:06 -0500 (Sat, 18 Dec 2010)
    beginnings of working connection context; 1->(2->(3->4) works, but nested does not
r117 | dennis.v.perepelitsa | 2010-12-18 20:28:45 -0500 (Sat, 18 Dec 2010)
    filling in the last rule implementation
r116 | dennis.v.perepelitsa | 2010-12-18 20:22:02 -0500 (Sat, 18 Dec 2010)
    STABLE BUILD with addby, addbyfront but no nested dot operations (for now)
r115 | oemlerb | 2010-12-18 20:03:54 -0500 (Sat, 18 Dec 2010)
    Started added type checking.  Changed Nodefunctions to work with variables
r114 | dennis.v.perepelitsa | 2010-12-18 18:47:08 -0500 (Sat, 18 Dec 2010)
    a swarm of translator/AST pattern matching fixes
r113 | dennis.v.perepelitsa | 2010-12-18 16:41:29 -0500 (Sat, 18 Dec 2010)
    fixing formal ordering & testing out variable pass-by-reference
r112 | AriGolub | 2010-12-18 16:34:56 -0500 (Sat, 18 Dec 2010)
    beginnings of working connection context backend
r111 | dennis.v.perepelitsa | 2010-12-18 16:27:17 -0500 (Sat, 18 Dec 2010)
    twiggling with test framework
r110 | dennis.v.perepelitsa | 2010-12-18 16:04:25 -0500 (Sat, 18 Dec 2010)
    finally, a working crawl() example :)
r109 | dennis.v.perepelitsa | 2010-12-18 15:56:52 -0500 (Sat, 18 Dec 2010)
    more cleaning up node/edge handles & pointers
r108 | dennis.v.perepelitsa | 2010-12-18 15:23:30 -0500 (Sat, 18 Dec 2010)
    small fixes everywhere. no basic crawling yet .. but SOON!
r107 | AriGolub | 2010-12-18 15:01:06 -0500 (Sat, 18 Dec 2010)
    added dynamic array indexing
r106 | dennis.v.perepelitsa | 2010-12-18 14:50:13 -0500 (Sat, 18 Dec 2010)
    inedge(), outedge(), innode, outnode properly return handles!
r105 | dennis.v.perepelitsa | 2010-12-18 14:08:45 -0500 (Sat, 18 Dec 2010)
    attribute getting and setting
r104 | AriGolub | 2010-12-18 14:07:12 -0500 (Sat, 18 Dec 2010)
    added dynamic array indexing
r103 | dennis.v.perepelitsa | 2010-12-18 13:50:21 -0500 (Sat, 18 Dec 2010)
    change to underlying attribute representation
r102 | dennis.v.perepelitsa | 2010-12-18 13:45:24 -0500 (Sat, 18 Dec 2010)
    i _believe_ queueing in crawls works properly now
r101 | AriGolub | 2010-12-18 13:25:23 -0500 (Sat, 18 Dec 2010)
    array indexing
r100 | AriGolub | 2010-12-18 04:39:34 -0500 (Sat, 18 Dec 2010)
    beginnings of connection context interpretation
r99 | AriGolub | 2010-12-18 02:51:23 -0500 (Sat, 18 Dec 2010)
    made assignment a statement intsead of expression, implemented ability to assign thin
    gs to variables
r98 | dennis.v.perepelitsa | 2010-12-17 23:24:01 -0500 (Fri, 17 Dec 2010)
    ... I believe that add() and addByFront() works?! MAYBE
r97 | dennis.v.perepelitsa | 2010-12-17 23:07:04 -0500 (Fri, 17 Dec 2010)
    more changes to crawl/rule model
r96 | dennis.v.perepelitsa | 2010-12-17 22:11:12 -0500 (Fri, 17 Dec 2010)
    more crawl + rule functionality!
r95 | dennis.v.perepelitsa | 2010-12-17 21:00:31 -0500 (Fri, 17 Dec 2010)
    proper (compilable) crawl/rule C++ formation
r94 | dennis.v.perepelitsa | 2010-12-11 05:41:31 -0500 (Sat, 11 Dec 2010)
    trying to fix crawl and rule types and arguments
```

```
r93 | dennis.v.perepelitsa | 2010-12-11 04:50:28 -0500 (Sat, 11 Dec 2010)
    example I want to show off :)
r92 | dennis.v.perepelitsa | 2010-12-11 04:46:16 -0500 (Sat, 11 Dec 2010)
    way too much awesome stuff
r91 | dennis.v.perepelitsa | 2010-12-11 04:11:48 -0500 (Sat, 11 Dec 2010)
    more work!
r90 | dennis.v.perepelitsa | 2010-12-11 03:34:42 -0500 (Sat, 11 Dec 2010)
    dealing with main() and C++ includes
r89 | dennis.v.perepelitsa | 2010-12-11 02:43:05 -0500 (Sat, 11 Dec 2010)
    no more errors?!
r88 | dennis.v.perepelitsa | 2010-12-11 02:27:55 -0500 (Sat, 11 Dec 2010)
    TEMPORARY COMMIT ONLY
r87 | oemlerb | 2010-12-11 02:21:19 -0500 (Sat, 11 Dec 2010)
    Alright since you asked for it.  I am not seeing the problem
    but you might.  If its causing to many problems, just comment it out and continue
r86 | dennis.v.perepelitsa | 2010-12-11 01:04:35 -0500 (Sat, 11 Dec 2010)
    fixing C++ errors so this now compiles
r85 | dennis.v.perepelitsa | 2010-12-11 01:04:24 -0500 (Sat, 11 Dec 2010)
    more compiler + translator work
r84 | dennis.v.perepelitsa | 2010-12-11 00:31:49 -0500 (Sat, 11 Dec 2010)
    making more translation + compilation work
r83 | dennis.v.perepelitsa | 2010-12-10 23:45:56 -0500 (Fri, 10 Dec 2010)
    unbroke the build. DO NOT COMMIT THINGS UNLESS make clean; make WORKS!
r82 | AriGolub | 2010-12-09 14:32:36 -0500 (Thu, 09 Dec 2010)
    more work on rules in parser/scanner
r81 | oemlerb | 2010-12-09 14:27:56 -0500 (Thu, 09 Dec 2010)
    Updated c ast
r80 | AriGolub | 2010-12-09 13:59:59 -0500 (Thu, 09 Dec 2010)
    extended attributes in backend
r79 | dennis.v.perepelitsa | 2010-12-09 01:48:20 -0500 (Thu, 09 Dec 2010)
    C++ hashed attribute objects
r78 | AriGolub | 2010-12-09 00:01:21 -0500 (Thu, 09 Dec 2010)
    just kidding, this is the scanner/parser with rules
r77 | AriGolub | 2010-12-08 23:01:50 -0500 (Wed, 08 Dec 2010)
    added rules to scanner/parser
r76 | dennis.v.perepelitsa | 2010-12-08 22:24:50 -0500 (Wed, 08 Dec 2010)
    very beginning of backend
r75 | dennis.v.perepelitsa | 2010-12-08 21:30:58 -0500 (Wed, 08 Dec 2010)
    test I've been using
r74 | dennis.v.perepelitsa | 2010-12-08 21:22:03 -0500 (Wed, 08 Dec 2010)
    my list of major TODO items
r73 | dennis.v.perepelitsa | 2010-12-08 10:12:23 -0500 (Wed, 08 Dec 2010)
    more backend work, fixes here and there
r72 | dennis.v.perepelitsa | 2010-12-08 09:38:11 -0500 (Wed, 08 Dec 2010)
    code generation pipeline in place!
r71 | dennis.v.perepelitsa | 2010-12-08 09:15:09 -0500 (Wed, 08 Dec 2010)
    committing stub files for translation to CAST
r70 | dennis.v.perepelitsa | 2010-12-06 17:04:41 -0500 (Mon, 06 Dec 2010)
    now parsing global opts with signatures now ; cleaning up some other stuff
r69 | dennis.v.perepelitsa | 2010-12-06 04:42:42 -0500 (Mon, 06 Dec 2010)
    misc TODO
```

```
r68 | dennis.v.perepelitsa | 2010-12-06 04:33:19 -0500 (Mon, 06 Dec 2010)
    comparators now go into symbol table; other miscellany
r67 | dennis.v.perepelitsa | 2010-12-06 04:24:33 -0500 (Mon, 06 Dec 2010)
    symbol table now keeping track of types (but not doing type checking yet)
r66 | dennis.v.perepelitsa | 2010-12-06 03:50:55 -0500 (Mon, 06 Dec 2010)
    i forget what i did but it was important
r65 | dennis.v.perepelitsa | 2010-12-06 03:42:55 -0500 (Mon, 06 Dec 2010)
    fixing misc parser errors
r64 | dennis.v.perepelitsa | 2010-12-06 03:27:35 -0500 (Mon, 06 Dec 2010)
    symbol table getting better
r63 | dennis.v.perepelitsa | 2010-12-06 02:18:39 -0500 (Mon, 06 Dec 2010)
    starting symbol table checks
r62 | dennis.v.perepelitsa | 2010-12-05 18:56:45 -0500 (Sun, 05 Dec 2010)
    start very crude automated test suite
r61 | dennis.v.perepelitsa | 2010-12-05 18:36:57 -0500 (Sun, 05 Dec 2010)
    TeX the LRM as Makefile option
r60 | dennis.v.perepelitsa | 2010-12-05 18:34:55 -0500 (Sun, 05 Dec 2010)
    starting C++ backend implementation (very ugly prototyping for now)
r59 | dennis.v.perepelitsa | 2010-12-05 18:34:23 -0500 (Sun, 05 Dec 2010)
    interpreter beginning to crawl AST!
r58 | oemlerb | 2010-12-04 00:24:43 -0500 (Sat, 04 Dec 2010)
    Simplified arrays a bit by making them into kinds of variables and
    connection contexts and actual lists into expressions.
r57 | dennis.v.perepelitsa | 2010-12-03 19:23:25 -0500 (Fri, 03 Dec 2010)
    accepts properly formatted comparator constructors!
r56 | dennis.v.perepelitsa | 2010-12-03 18:24:57 -0500 (Fri, 03 Dec 2010)
    notes to update connection context definition slightly
r55 | dennis.v.perepelitsa | 2010-12-03 17:09:23 -0500 (Fri, 03 Dec 2010)
    adding some very simple tests to cat & pipe into ./digr
r54 | dennis.v.perepelitsa | 2010-12-03 16:54:28 -0500 (Fri, 03 Dec 2010)
    connection contexts accepted by interpreter!!
r53 | dennis.v.perepelitsa | 2010-12-03 16:49:09 -0500 (Fri, 03 Dec 2010)
    connection contexts no longer throw shift/reduce conflicts!
r52 | oemlerb | 2010-12-03 16:42:42 -0500 (Fri, 03 Dec 2010)
    Added arguments notation
r51 | dennis.v.perepelitsa | 2010-12-03 15:49:35 -0500 (Fri, 03 Dec 2010)
    I keep forgetting to commit the LRM
r50 | dennis.v.perepelitsa | 2010-12-03 15:49:18 -0500 (Fri, 03 Dec 2010)
    while/if work with just stmt and stmt_lists!
r49 | AriGolub | 2010-12-01 13:57:10 -0500 (Wed, 01 Dec 2010)
    added line 126 to parser, check it out to make sure it makes sense. basically, wasn't
     accepting |1-->(2--3),4| so i added a new rule to accept it
r48 | AriGolub | 2010-12-01 13:09:26 -0500 (Wed, 01 Dec 2010)
    update todo with more c++ classes
r47 | AriGolub | 2010-12-01 12:38:08 -0500 (Wed, 01 Dec 2010)
    added print function
r46 | dennis.v.perepelitsa | 2010-12-01 12:33:15 -0500 (Wed, 01 Dec 2010)
    updating Makefile clean
r45 | dennis.v.perepelitsa | 2010-12-01 12:32:37 -0500 (Wed, 01 Dec 2010)
    adding two TODOs for me
r44 | dennis.v.perepelitsa | 2010-12-01 12:29:50 -0500 (Wed, 01 Dec 2010)
```

```
    killing 'new' version files
r43 | oemlerb | 2010-11-23 19:32:26 -0500 (Tue, 23 Nov 2010)
    Made program functional
r42 | oemlerb | 2010-11-23 19:21:31 -0500 (Tue, 23 Nov 2010)
    Added block and block list along with 22 shift reduce errors
r41 | oemlerb | 2010-11-23 03:22:34 -0500 (Tue, 23 Nov 2010)
    Had some old code from calculator
r40 | oemlerb | 2010-11-23 01:38:14 -0500 (Tue, 23 Nov 2010)
    a functional interpret
r39 | AriGolub | 2010-11-18 14:44:01 -0500 (Thu, 18 Nov 2010)
    this works
r38 | dennis.v.perepelitsa | 2010-11-18 14:15:17 -0500 (Thu, 18 Nov 2010)
    oops. minor bugs in new parser
r37 | dennis.v.perepelitsa | 2010-11-18 14:05:10 -0500 (Thu, 18 Nov 2010)
    minor fix in arguments of comparator constructor
r36 | AriGolub | 2010-11-18 03:03:21 -0500 (Thu, 18 Nov 2010)
    i realized version control is the point of not having to rename all these files but t
    oo late. i need to sleep
r35 | oemlerb | 2010-11-18 00:02:29 -0500 (Thu, 18 Nov 2010)
    The beginnings of a new parser.  Still in process.   Dont know if it even compiles
r34 | oemlerb | 2010-11-17 23:45:05 -0500 (Wed, 17 Nov 2010)

    some big changes to ast.  wanted to commit them separately so it doesnt mess up anyth
    ing
r33 | oemlerb | 2010-11-17 23:08:23 -0500 (Wed, 17 Nov 2010)
    Added dollar sign for greatest value statement
r32 | AriGolub | 2010-11-17 22:58:46 -0500 (Wed, 17 Nov 2010)
    added brackets to comparator
r31 | AriGolub | 2010-11-17 22:51:51 -0500 (Wed, 17 Nov 2010)
    added comparator
r30 | AriGolub | 2010-11-17 19:51:51 -0500 (Wed, 17 Nov 2010)
    bunch of fixes, works right now, 1 shift/reduce conflit, also added TODO.txt that con
    tains what needs to be don and by who
r29 | AriGolub | 2010-11-17 18:32:35 -0500 (Wed, 17 Nov 2010)
    this works, but not perfect
r28 | AriGolub | 2010-11-17 18:05:20 -0500 (Wed, 17 Nov 2010)
    changes
r27 | AriGolub | 2010-11-16 17:54:01 -0500 (Tue, 16 Nov 2010)
    some parser, some ast
r26 | dennis.v.perepelitsa | 2010-11-16 14:33:04 -0500 (Tue, 16 Nov 2010)
    committing different id types
r25 | AriGolub | 2010-11-16 14:25:23 -0500 (Tue, 16 Nov 2010)
    ast stuff
r24 | AriGolub | 2010-11-16 13:53:46 -0500 (Tue, 16 Nov 2010)
    ast stuff
r23 | dennis.v.perepelitsa | 2010-11-16 13:44:11 -0500 (Tue, 16 Nov 2010)
    basic clean build
r22 | dennis.v.perepelitsa | 2010-11-16 13:33:50 -0500 (Tue, 16 Nov 2010)
    temp fixing ID and strings
r21 | AriGolub | 2010-11-16 13:30:50 -0500 (Tue, 16 Nov 2010)
    tree fixed
```

```
r20 | oemlerb | 2010-11-16 13:24:26 -0500 (Tue, 16 Nov 2010)
    Changed connection to tree
r19 | dennis.v.perepelitsa | 2010-11-16 13:24:12 -0500 (Tue, 16 Nov 2010)
    adding Makefile for project and basic interpreter testbed
r18 | dennis.v.perepelitsa | 2010-11-16 13:22:26 -0500 (Tue, 16 Nov 2010)
    parser builds(\?\!\?\!)
r17 | dennis.v.perepelitsa | 2010-11-16 13:16:58 -0500 (Tue, 16 Nov 2010)
    ast compiles
r16 | AriGolub | 2010-11-16 13:12:43 -0500 (Tue, 16 Nov 2010)
    ast
r15 | dennis.v.perepelitsa | 2010-11-16 13:10:26 -0500 (Tue, 16 Nov 2010)
    fixing ocamlyacc formatting
r14 | dennis.v.perepelitsa | 2010-11-16 13:05:38 -0500 (Tue, 16 Nov 2010)
    starting to fix some bugs; want to make this compile
r13 | oemlerb | 2010-11-15 23:56:04 -0500 (Mon, 15 Nov 2010)
    Added or functionality
r12 | AriGolub | 2010-11-15 23:19:46 -0500 (Mon, 15 Nov 2010)
    more ast
r11 | AriGolub | 2010-11-15 22:31:24 -0500 (Mon, 15 Nov 2010)
    more ast
r10 | AriGolub | 2010-11-15 21:35:12 -0500 (Mon, 15 Nov 2010)
    fixed ast, i think
r9 | oemlerb | 2010-11-15 19:07:09 -0500 (Mon, 15 Nov 2010)
    Adding handlers for or and and.  Brackets
r8 | oemlerb | 2010-11-15 18:47:01 -0500 (Mon, 15 Nov 2010)
    Added abstract syntax tree.  Modified version of microC
r7 | AriGolub | 2010-11-14 17:33:27 -0500 (Sun, 14 Nov 2010)
    started parser
r6 | AriGolub | 2010-11-14 15:54:01 -0500 (Sun, 14 Nov 2010)
    more scanner
r5 | oemlerb | 2010-11-14 14:48:49 -0500 (Sun, 14 Nov 2010)
    Just did some copy and pasting, converting pdf characters to
    normal characters.  Added a few symbols
r4 | oemlerb | 2010-11-14 14:01:53 -0500 (Sun, 14 Nov 2010)
    test test
r3 | AriGolub | 2010-11-14 13:58:05 -0500 (Sun, 14 Nov 2010)
    ari push test
r2 | oemlerb | 2010-11-14 13:42:34 -0500 (Sun, 14 Nov 2010)
    test
r1 | (no author) | 2010-09-26 15:07:20 -0400 (Sun, 26 Sep 2010)
    Initial directory structure.
```

5 Architectural Design

The DiGr compiler pipeline consists of five major modules along with a final execution stage, and three backend/abstract syntax tree definitions and libraries. A block diagram of the flow of information and dependencies is pictured in Figure 1.

5.1 DiGr Compiler Modules

The **scanner** processes a stream of DiGr code and returns tokens. If the input is not lexically correct DiGr code, the scanner fails. At this stage, only the presence of unrecognizable tokens will stop compilation.

The **parser** then uses the grammar defined in the **DiGr AST Definition** to turn the token sequence into an instance of the DiGr AST. The AST is a recursive, typed OCaml tree of tuples. If the token stream is not a syntactically correct DiGr program, compilation fails at the parser stage.

The **interpreter** performs static semantic type checking, scoping and other consistency checks on the DiGr AST. If the AST does not represent a semantically sensible DiGr program, compilation fails at the interpreter stage. Unlike the first two modules, the interpreter does not modify its input (the DiGr AST), but simply accepts or rejects it. The interpreter generates symbol tables for the global and all local scopes, but these do not remain after the interpreter stage.

The **translator** turns the DiGr AST into an instance of the AST described in the **C++ AST Definition**. Much of the translation, especially for the C++-like elements of the language, occurs in a recursive, depth-first manner and is straightforward. The higher-level elements of the DiGr are turned into significantly longer or more complicated sequences of C++ statements. The translator does no further semantic checking of its own, and this module always generates a valid instance of a C++ abstract syntax tree. This is because any problems encountered by the translator reflect either an incomplete or inconsistent DiGr AST definition, or a failure of the interpreter to properly validate the DiGr AST.

In terms of the block diagram, one could argue that the interpreter and translator stages could be combined, since the interpreter does not modify its input. However, we felt that separating the semantic type checking (which can be thought of as part of the compiler front-end) from the beginning of the compilation back-end was a good abstraction. This way, development could be focused on either module, since they perform non-overlapping tasks.

The **compiler** recursively walks the C++ AST and outputs a C++ program. In effect, the compiler takes the semantic meaning of the C++ AST and turns it into a compilable program will all syntactic details included. The compiler is blind to the semantic correctness (or incorrectness) of the actual resulting program.

The sixth stage before program execution is compiling with **g++** against the **DiGr C++ Backend** and running the resulting binary program, but this is not a formal DiGr module. The C++ AST is constrained so as to generate only syntactically valid C++, and the interpreter and translator ensure that the output is *semantically* correct and will compile.

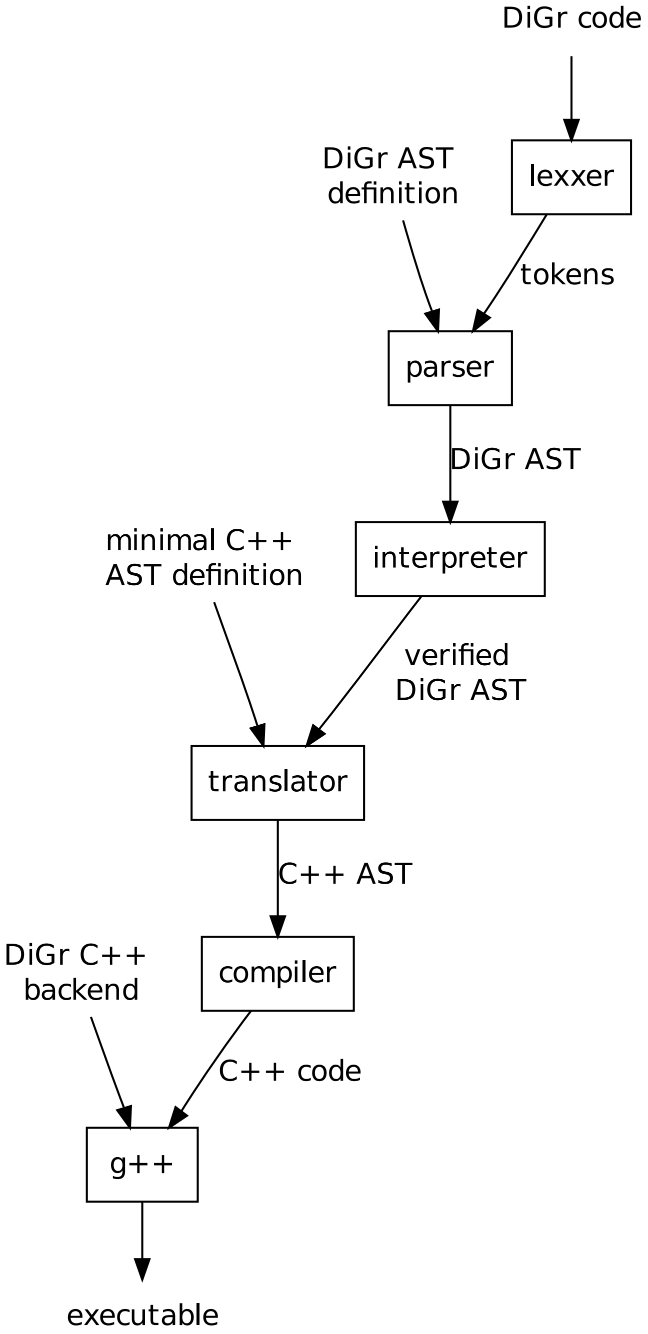

Figure 1: DiGr compiler block diagram

Our test examples have been checked against g++ version 4.4.3, but it is likely that any ISO compliant C++ compiler will compile the DiGr output.

Unfortunately, there are errors it is impossible to check at compile time, and difficult to handle gracefully at runtime. These include segmentation faults from out of bounds accesses of C++ arrays and are an unfortunate consequences of the

5.2 Definitions and Libraries

Several stages of the compiler use the **DiGr AST Definition**, which represents a DiGr program with all syntactic details stripped away. The AST definition was designed to split the difference between being in a form easily constructed by the parser, and easily interpreted and translated later.

The **C++ AST Definition** implements the small but flexible subset of the C++ language needed to output compiled DiGr code. The definition takes some shortcuts (for example, there is no support for shifting (`>>`, `<<`) operators or streams with the single exception of using `std::cout <<` to implement the DiGr `print()` opt), and is meant to be lightweight to make compilation easy. The C++ AST definition has no concept of semantic correctness.

Finally, the **DiGr C++ Backend** is the engine against which compiled DiGr C++ code can be turned into a binary executable. The backend was written to have a simple interface to make the compilation step efficient and clean, and also be short enough so that the overhead in a DiGr binary program is relatively small. The backend does a small amount of runtime error catching.

6 Test Plan

Our test strategy consisted of writing short to medium length DiGr programs which would typically print information to the screen, and creating by hand a ”gold standard” of what the output should be according to the language designers. Additionally, for some test programs we examined the output in the target language by hand to check our code.

To run the test suite, we compile and execute every test program and compare its output to the gold standard. Sometimes, programs would fail at the front-end level (implying the parser or static semantic checking was improperly implemented. Sometimes, programs would fail at the back-end level (output programs in the target language would fail to compile, or throw a run-time exception, or output something different from the gold standard). The stage at which the error occurred allowed us to narrow down bugs along the DiGr compilation pipeline.

Some tests focused on testing atomic features of the language, from basic concepts like fundamental types, arithmetic, opt calls, and so forth, to high-level concepts like graphs, attributes, creation contexts, etc. Other tests were designed to be complicated and integrate a wide cross-section of language features.

The test suite was run after every significant change to the parser, translator, or compiler, to ensure that development had not broken any previous work. A few tests were written to ensure that necessary errors at compile time and run time were in fact caught.

Although all team members contributed test programs and ideas for test programs, and used the test battery to track and fix bugs, Ari was the member responsible for the upkeep of the suite. The complete list of test programs (in alphabetical order in our directory) and what functionality they are designed to test:

- `anonedge` : proper creation of anonymous edges without runtime errors
- `arrays` : creating, accessing and modifying arrays
- `attributes` : proper creation and access of node and edge attributes, both implicitly and explicitly
- `basiccontext` : proper parsing of complicated tree definition in a connection context
- `basiccrawl` : a crawl test that integrates many DiGr features
- `binops` : testing binary operators
- `blockorder` : proper handling of control flow (if, if/else, while)
- `comments` : very simple comment parsing test
- `contexts` : in-depth test of proper connection context compilation and edge assignment between nodes

- `crawlargs` : proper indexing and C++ typing of in and out variables in function signatures
- `depthfirsts` : high-concept test of a breadth-first (the name of the test is misleading) and an iterative depth-first search
- `edgetest` : proper manipulation nodes by traversing edges
- `factorial` : test of a simple recursive function with in/out variables
- `fast` : test of fencepost while loop iteration
- `func` : more complicated test of proper scoping for in/out variables
- `globals` : testing the 'call' function, changing rules inside crawls, and proper compilation with respect to global namespaces in general
- `indexattr` : accessing attributes of elements of an array via indexing into the array
- `nodetest` : creating nodes and edges, plus basic node functionality
- `opttest` : simple opt-calling test to check proper in/out variable binding
- `recursivecrawl` : high-concept test with two depth first searches and, specifically, the ability to call a crawl within a crawl
- `ruleaddby` : proper use of advanced 'addby' feature in a rule
- `runtime` : check to see that a run-time exception catches illegal indexing
- `scope` : proper scoping of similarly named variables inside different local scopes

6.1 basiccrawl test

Here is an example which integrates edges, nodes, crawls and rules. The DiGr source code is

```
rule addMarkedChildren {
  int n = 0!
  while (n < current.outedges) {
     edge tmp_edge = current.outedge(n)!
     if (tmp_edge.mark == 1) {
       node destination = tmp_edge.innode!
       add(destination)!
     }
     n = n + 1!
  }
```

```
}

crawl printId() {
  print (current.id)!
  call!
}


opt main() {

  node n1!
  node n2!
  node n3!
  node n4!
  n1.id = 1!
  n2.id = 2!
  n3.id = 3!
  n4.id = 4!
  n1 -> n2!
  n2 -> n3!
  n2 -> n4!
  edge tmp_edge = n1.outedge(0)!
  tmp_edge.mark = 1!
  tmp_edge = n2.outedge(1)!
  tmp_edge.mark = 1!

  printId() from n1 with addMarkedChildren!

}
```

This is a simple program which creates a tree by connecting nodes, marks some edges with an attribute, and then runs a crawl which prints the `id` attribute of the current node, while only following edges which are marked. The output in the target language is (there is normally a symbol table dump and static semantic checking information output in the header of the program. In this example, we leave it in):

```
/*begin formal AST verification
==================
global signature dump:
main:
==================
Starting
unimplemented expression
n assigned value
n1 assigned value
n2 assigned value
```

```
n3 assigned value
n4 assigned value
tmp_edge assigned value
tmp_edge assigned value
tmp_edge assigned value
==================
symbol table dump:
--> n1: node
--> n2: node
--> n3: node
--> n4: node
--> tmp_edge: edg
==================
symbol table dump:
--> current: node
==================
symbol table dump:
--> current: node
--> n: int
no error!
begin translation to CAST
passed static semantic checking, begin code generation
====================== */

#include ‘‘digr.h’’
#include <iostream>
/* actual definition of C++ functions */
void addMarkedChildren(DiGrNode *current, deque<DiGrNode*> *returnQueue) {
int n =  0 ;
while(n < current->OutEdges())
{DiGrEdge *tmp_edge = current->getOutEdge(n);
if(tmp_edge->getAttribute(‘‘mark’’) ==  1 )
{DiGrNode *destination = tmp_edge->inNode();
returnQueue->push_back(destination);
}
else{}
n=n +  1 ;
}
}

void printId(DiGrNode *current, void (*rule)(DiGrNode*, deque<DiGrNode*>*)) {
deque<DiGrNode*> *queue = new deque<DiGrNode*>();
queue->push_back(current);
do {
current=queue->front();
```

```
queue->pop_front();
std::cout << current->getAttribute(``id'') << std::endl;
rule(current, queue);
} while (queue->size() >  0 );

}

int main() {
try{
DiGrNode *n1 = new DiGrNode();
DiGrNode *n2 = new DiGrNode();
DiGrNode *n3 = new DiGrNode();
DiGrNode *n4 = new DiGrNode();
n1->setAttribute(``id'',  1 );
n2->setAttribute(``id'',  2 );
n3->setAttribute(``id'',  3 );
n4->setAttribute(``id'',  4 );
new DiGrEdge(n1, n2);
new DiGrEdge(n2, n3);
new DiGrEdge(n2, n4);
DiGrEdge *tmp_edge = n1->getOutEdge( 0 );
tmp_edge->setAttribute(``mark'',  1 );
tmp_edge=n2->getOutEdge( 1 );
tmp_edge->setAttribute(``mark'',  1 );
printId(n1, addMarkedChildren);
}
catch(const char *e) {
std::cout << e << std::endl;
}

}
```

The simple DiGr crawls and rules and implicit queues and references to nodes are turned into explicit and careful function signatures and a system of pointers in the target language. When executed, this should outputs the first node (with an id of 1), follow the marked edge to node 2, and the follow the marked edge to node 4. Sure enough, the output is

```
1
2
4
```

6.2 recursivecrawl test

Another sophisticated example is a test that implements post-order and in-order depth-first traversals of a tree. It accomplishes this by leaving the queue empty (in fact, even assigning a blank rule), and simply recursively calling itself on its children before printing. The DiGr source code is:

```
rule blankRule {

}

crawl recurse_to_children_and_print() {

      int n = 0!
      while (n < current.outedges) {
        edge tmp_edge = current.outedge(n)!
        node tmp_node = tmp_edge.innode!
        recurse_to_children_and_print() from tmp_node with blankRule!
        n = n + 1!
     }

     print(current.name)!
}

crawl recurse_inorder() {

      int n = 0!
      while (n < current.outedges) {
        edge tmp_edge = current.outedge(n)!
        node tmp_node = tmp_edge.innode!
        if (tmp_node.name < current.name) {
            recurse_inorder() from tmp_node with blankRule!
        }
        n = n + 1!
     }

     print(current.name)!

     n = 0!
      while (n < current.outedges) {
        edge tmp_edge = current.outedge(n)!
        node tmp_node = tmp_edge.innode!
        if (tmp_node.name > current.name) {
            recurse_inorder() from tmp_node with blankRule!
        }
        n = n + 1!
     }

}
```

```
opt main() {

   node binTree[8] = |4 -> (2 -> 1,3), (6 -> 5,7)|!

   node tmp_node = binTree[1]!
   tmp_node.name = 1!
   tmp_node = binTree[2]!
   tmp_node.name = 2!
   tmp_node = binTree[3]!
   tmp_node.name = 3!
   tmp_node = binTree[4]!
   tmp_node.name = 4!
   tmp_node = binTree[5]!
   tmp_node.name = 5!
   tmp_node = binTree[6]!
   tmp_node.name = 6!
   tmp_node = binTree[7]!
   tmp_node.name = 7!

   node start = binTree[4]!

   print (‘‘post-order!’’)!
   recurse_to_children_and_print() from start with blankRule!

   print (‘‘in-order!’’)!
   recurse_inorder() from start with blankRule!

}
```

This compiles to (leaving out the verbose static semantic output and the symbol table dump) :

```
#include ‘‘digr.h’’
#include <iostream>
/* actual definition of C++ functions */
void blankRule(DiGrNode *current, deque<DiGrNode*> *returnQueue) {

}

void recurse_to_children_and_print(DiGrNode *current, void (*rule)(DiGrNode*, deque<DiGrNode*>
deque<DiGrNode*> *queue = new deque<DiGrNode*>();
queue->push_back(current);
do {
current=queue->front();
queue->pop_front();
int n =  0 ;
```

```
while(n < current->OutEdges())
{DiGrEdge *tmp_edge = current->getOutEdge(n);
DiGrNode *tmp_node = tmp_edge->inNode();
recurse_to_children_and_print(tmp_node, blankRule);
n=n +  1 ;
}std::cout << current->getAttribute(‘‘name’’) << std::endl;
} while (queue->size() >  0 );

}

void recurse_inorder(DiGrNode *current, void (*rule)(DiGrNode*, deque<DiGrNode*>*)) {
deque<DiGrNode*> *queue = new deque<DiGrNode*>();
queue->push_back(current);
do {
current=queue->front();
queue->pop_front();
int n =  0 ;
while(n < current->OutEdges())
{DiGrEdge *tmp_edge = current->getOutEdge(n);
DiGrNode *tmp_node = tmp_edge->inNode();
if(tmp_node->getAttribute(‘‘name’’) < current->getAttribute(‘‘name’’))
{recurse_inorder(tmp_node, blankRule);
}
else{}
n=n +  1 ;
}std::cout << current->getAttribute(‘‘name’’) << std::endl;
n= 0 ;
while(n < current->OutEdges())
{DiGrEdge *tmp_edge = current->getOutEdge(n);
DiGrNode *tmp_node = tmp_edge->inNode();
if(tmp_node->getAttribute(‘‘name’’) > current->getAttribute(‘‘name’’))
{recurse_inorder(tmp_node, blankRule);
}
else{}
n=n +  1 ;
}} while (queue->size() >  0 );

}

int main() {
try{
DiGrNode* binTree[8];
 binTree[0]=new DiGrNode();
 binTree[1]=new DiGrNode();
 binTree[2]=new DiGrNode();
```

```
 binTree[3]=new DiGrNode();
 binTree[4]=new DiGrNode();
 binTree[5]=new DiGrNode();
 binTree[6]=new DiGrNode();
 binTree[7]=new DiGrNode();
new DiGrEdge(binTree[4], binTree[2]);
new DiGrEdge(binTree[4], binTree[6]);
new DiGrEdge(binTree[6], binTree[5]);
new DiGrEdge(binTree[6], binTree[7]);
new DiGrEdge(binTree[2], binTree[1]);
new DiGrEdge(binTree[2], binTree[3]);
DiGrNode *tmp_node =  binTree[1];
tmp_node->setAttribute(‘‘name’’,  1 );
tmp_node= binTree[2];
tmp_node->setAttribute(‘‘name’’,  2 );
tmp_node= binTree[3];
tmp_node->setAttribute(‘‘name’’,  3 );
tmp_node= binTree[4];
tmp_node->setAttribute(‘‘name’’,  4 );
tmp_node= binTree[5];
tmp_node->setAttribute(‘‘name’’,  5 );
tmp_node= binTree[6];
tmp_node->setAttribute(‘‘name’’,  6 );
tmp_node= binTree[7];
tmp_node->setAttribute(‘‘name’’,  7 );
DiGrNode *start =  binTree[4];
std::cout << ‘‘post-order!’’ << std::endl;
recurse_to_children_and_print(start, blankRule);
std::cout << ‘‘in-order!’’ << std::endl;
recurse_inorder(start, blankRule);
}
catch(const char *e) {
std::cout << e << std::endl;
}

}
```

When executed, the output is

```
post-order!
1
3
2
5
7
6
```

```
4
in-order!
1
2
3
4
5
6
7
```

7 Lessons Learned

Dennis

One of the most painfully leaned lessons for me during this project was the importance of a consistent and carefully thought about contract between different modules. A strong enough architecture model, and an eye towards dependencies means that each developer can handle the internal implementation of different parts of the project without having to constantly be aware of small changes in the details of somebody else's work. We got worse at following this rule as the project went along. Towards the end of the project, as the code evolved more and more towards completion, a single change very early on in the architecture model (say, a new keyword in the parser), had to implemented all the way down the line to the compilation stage. Particularly annoying was the fact that there were about six different abstract stages at which an error could propagate. This made last minute features (or, features we did not plan on when we created the DiGr AST) slightly exasperating. In an ideal world, I think our two ASTs and the backend would have been written first, and then the modules worked on independently. In reality, development was concurrent and intertwined.

Ari

As the project neared its completion and we found ourselves testing out the language, the thing that hit me the most was that its easier to come up with a simple idea, implement it perfectly, and then build upon it. This contrasts with the approach that we took: we had the great idea with all the different features and bells and whistles for the user, but had to keep dropping one thing or another because the things that were really necessary, the most basic parts, weren't rock solid because of the bells and whistles. Basically, I learned that its better to set your dream small and build bigger rather than dream big and build smaller. There were a lot of good ideas I wish we would have had the time– or working infrastructure– necessary to build. On a lower level, I learned that Ocaml is exceedingly frustrating but also very gratifying when it works. The slide at the beginning of the year, "never have i done so much writing so little," now makes too much sense.

Bryan

I realized, perhaps too late, that languages like Ocaml require their own coding style standards. Before getting into writing the bulk of the code it would have been helpful to nail down a set Programming style. Ocaml's structure deviates greatly from most other languages I have used. A consistent style would have made Ocaml, a language that was new to all of us, more understandable.

8 Appendix

8.1 scanner.mll

```
{ open Parser }

rule token = parse
[' ' '\t' '\r' '\n']    { token lexbuf }         (* Whitespace *)
| ":"   { comment lexbuf }       (* Comments changed *)
| '('   { LPAREN }
| ')'   { RPAREN }       (* punctuation *)
| '{'   { LBRACE }
| '}'   { RBRACE }
| '!'   { EXC }
| ','   { COMMA }
| ';'   { SEMI }
| '~'   { NEG }
| '+'   { PLUS }
| '-'   { MINUS }
| '*'   { TIMES }
| "\""  { QUOTE }
| '/'   { DIVIDE }
| '='   { ASSIGN }
| '%'   { MOD }
| "=="  { EQ }
| "!="  { NEQ }
| '<'   { LT }
| "<="  { LEQ }
| '>'   { GT }
| ">="  { GEQ }
| '|'   { CNCT }
| "->"  { REDGE }
| "<-"  { LEDGE }
| "--"  { UEDGE }
| "["   {LBRACK}
| "]"   {RBRACK}
| "||"  {OR}
| "&&"  {AND}
| '.' {DOT}
| "add"         { ADD }
| "addBy"       { ADDBY }
| "addFront"    { ADDFRONT }
| "call"        { CALL }
| "set"         { SET }
| "addByFront"  { ADDBYFRONT }
| "call"        { CALL }
| "crawl"       { CRAWL }
| "edge"        { EDGE }
| "else"        { ELSE }
| "for"         { FOR }
| "flt"         { FLOAT }
```

```
| "from"        { FROM }
| "in"          { IN }
| "int"         { INT }
| "if"          { IF }
| "node"        { NODE }
| "opt"         { OPT }
| "order"       { ORDER }
| "out"         { OUT }
| "print"       { PRINT }
| "queue"       { QUEUE }
| "rule"        { RULE }
| "str"         { STR }
| "while"       { WHILE }
| "with"        { WITH }
| "$"           { DOLR }
| "child"       { CHILD }
| "parent"      { PARENT }
| "inedges"   { INEDGES }
| "outedges"  { OUTEDGES }
| "inedge"      { INEDGE }
| "outedge"     { OUTEDGE }
| "innode"      { INNODE }
| "outnode"     { OUTNODE }

| eof           { EOF }
| ['0'-'9']+ as lxm                     { LITINT(int_of_string lxm) }
| ['0'-'9']*'.'['0'-'9']+ as lxm        { LITFLT(float_of_string lxm) }
| '\"' [^'\"']* '\"' as lxm             { LITSTR(lxm) }

| ['a'- 'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }

| _ as char     { raise (Failure("illegal character " ^ Char.escaped char) ) }

and comment = parse
":" { token lexbuf }
| _ { comment lexbuf }
```

8.2 parser.mly

```
%{ open Ast %}

/* TODO: rules are not implemented, like, at all */

%token CHILDREN PARENTS CHILD PARENT INEDGES OUTEDGES INEDGE OUTEDGE INNODE
OUTNODE
%token EXC LPAREN RPAREN LBRACE RBRACE COMMA SEMI NEG PLUS MINUS TIMES DIVIDE
MOD
%token LBRACK RBRACK OR AND EOF DOT QUOTE DOLR
%token ASSIGN EQ NEQ LT LEQ GT GEQ CNCT REDGE LEDGE UEDGE
%token ADD ADDBY ADDFRONT ADDBYFRONT COMP CRAWL EDGE ELSE
%token FOR FLOAT FROM IN INT IF NODE OPT ORDER OUT PRINT QUEUE RULE STR CALL
%token CALL SET
%token WHILE WITH
%token <int> LITINT
%token <float> LITFLT
%token <string> ID
%token <string> LITSTR

%nonassoc NOELSE
%nonassoc ELSE
%nonassoc NOPAREN
%right ASSIGN
%left AND OR
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE MOD

%start program
%type <Ast.program> program

%%

/*
program:
                                        { [], [] }
|       program fdecl                           { fst $1, ($2 :: snd $1) }
*/

program:
        /* nothing */                           { [] }
|       program fdecl                           { $2 :: $1 }

fdecl:
        OPT ID LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE
```

```
        { { func_type = "opt"; fname = $2; formals = $4; body = List.rev $7 } }
|       CRAWL ID LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE
        { { func_type = "crawl"; fname = $2; formals = $4; body = List.rev $7 }
}
|       RULE ID LBRACE stmt_list RBRACE
        { { func_type = "rule"; fname = $2; formals = []; body = List.rev $4 }
}

/* dvp: the List.rev is from how we untangle the formals in
formal_list */

formals_opt:
        /* nothing */                           { [] }
|       formal_list                             { List.rev $1 }




formal:
        OUT INT ID                              { Validate(Out,Int,$3) }
|       OUT NODE ID                             { Validate(Out,Node,$3) }
|       OUT EDGE ID                             { Validate(Out,Edg,$3) }
|       OUT STR ID                              { Validate(Out,Str,$3) }
|       OUT FLOAT ID                            { Validate(Out,Flt,$3) }
|       IN INT ID                               { Validate(In,Int,$3) }
|       IN NODE ID                              { Validate(In,Node,$3) }
|       IN EDGE ID                              { Validate(In,Edg,$3) }
|       IN STR ID                               { Validate(In,Str,$3) }
|       IN FLOAT ID                             { Validate(In,Flt,$3) }



formal_list:
        formal                                  { [$1] }
|       formal_list SEMI formal                 { $3 :: $1 }

stmt_list:
        /* nothing */                           { [] }
|       stmt_list stmt                          { $2 :: $1 }

variable:
        ID                                      { VarId($1) }
|       variable DOT ID                         { RecVar($1, $3) }
|       ID LBRACK LITINT RBRACK                 { ArrayIndStat($1,$3)}
|       ID LBRACK variable RBRACK               { ArrayIndDyn($1,$3)}

stmt:
        expr EXC                                { Expr($1) }
```

```
|       IF LPAREN expr RPAREN LBRACE stmt_list RBRACE %prec NOELSE { If($3,
List.rev $6,[]) }
|       IF LPAREN expr RPAREN LBRACE stmt_list RBRACE ELSE LBRACE stmt_list
RBRACE { If($3, List.rev $6, List.rev $10) }
|       WHILE LPAREN expr RPAREN LBRACE stmt_list RBRACE                {
While($3, List.rev $6) }
|       INT variable EXC                                 { Declare_Only(Int, $2)
}
|       NODE variable EXC                                { Declare_Only(Node,
$2) }
|       EDGE variable EXC                                { Declare_Only(Edg, $2)
}
|       STR variable EXC                                 { Declare_Only(Str, $2)
}
|       FLOAT variable EXC                               { Declare_Only(Flt, $2)
}
|       INT variable ASSIGN expr EXC                     { Declare(Int, $2, $4)
}
|       NODE variable ASSIGN expr EXC                    { Declare(Node, $2, $4)
}
|       EDGE variable ASSIGN expr EXC                    { Declare(Edg, $2, $4)
}
|       STR variable ASSIGN expr EXC                     { Declare(Str, $2, $4)
}
|       FLOAT variable ASSIGN expr EXC                   { Declare(Flt, $2, $4)
}
|  NODE variable ASSIGN CNCT tree CNCT EXC      { CreateGraph($2, $5) }
|       ID LPAREN actuals_opt RPAREN EXC                 { Call($1, $3) }
|       PRINT LPAREN actuals_opt RPAREN EXC              { Print($3) }

|       ID LPAREN actuals_opt RPAREN FROM variable WITH ID EXC {Crawl($1, $3, $6, $8)}
|       variable LEDGE variable EXC {EdgeCreation($1, Ledge, $3) }
|       variable REDGE variable EXC {EdgeCreation($1, Redge, $3) }
|       variable UEDGE variable EXC {EdgeCreation($1, Uedge, $3) }
|       variable ASSIGN expr EXC                         { Assign($1, $3) }
|       CALL EXC                                 { CallRule }
|       SET ID EXC              { SetRule($2) }
|       ADD LPAREN variable RPAREN EXC                   { RAdd($3) }
```

```
|       ADDFRONT LPAREN variable RPAREN EXC                           { RAddFront($3) }
|       ADDBY LPAREN NODE DOT ID COMMA DOLR COMMA LITINT RPAREN EXC{ RAddBy($5, AddByNode, Dolr, $9) }
|       ADDBY LPAREN NODE DOT ID COMMA NEG COMMA LITINT RPAREN EXC      { RAddBy($5, AddByNode, Tilde, $9) }
|       ADDBYFRONT LPAREN NODE DOT ID COMMA DOLR COMMA LITINT RPAREN EXC{ RAddByFront($5, AddByNode, Dolr, $9) }
|       ADDBYFRONT LPAREN NODE DOT ID COMMA NEG COMMA LITINT RPAREN EXC{ RAddByFront($5, AddByNode, Tilde, $9) }


expr:
        LPAREN expr RPAREN                          { $2 }
|       plainString                                 { Lit_Str($1) }
|       LITINT                                      { Lit_Int($1) }
|       LITFLT                                      { Lit_Flt($1) }
|       expr PLUS expr                              { Binop($1, Add, $3) }
|       expr MINUS expr                             { Binop($1, Sub, $3) }
|       expr TIMES expr                             { Binop($1, Mult, $3) }
|       expr DIVIDE expr                            { Binop($1, Div, $3) }
|       expr EQ expr                                { Binop($1, Equal, $3) }
|       expr NEQ expr                               { Binop($1, Neq, $3) }
|       expr LT expr                                { Binop($1, Less, $3) }
|       expr LEQ expr                               { Binop($1, Leq, $3) }
|       expr GT expr                                { Binop($1, Greater, $3) }
|       expr GEQ expr                               { Binop($1, Geq, $3) }
|       expr AND expr                               { Binop($1, And, $3 ) }
|       expr OR expr                                { Binop($1, Or, $3 ) }
|       expr MOD expr                               { Binop($1, Mod, $3) }
|       LBRACE actuals_list RBRACE                  { Actuals($2)}
|       variable DOT OUTEDGE LPAREN expr RPAREN         { NodeOutEdge($1,$5) }
|       variable DOT INEDGE LPAREN expr RPAREN          { NodeInEdge($1,$5) }
|       variable DOT CHILD LPAREN expr RPAREN         { NodeChild($1,$5) }
|       variable DOT PARENT LPAREN expr RPAREN          { NodeParent($1,$5) }
|       variable DOT OUTEDGES                             { NodeOutEdges($1) }
|       variable DOT INEDGES                              { NodeInEdges($1) }
|       variable DOT INNODE                               { EdgeInNode($1) }
|       variable DOT OUTNODE                              { EdgeOutNode($1) }
|       variable                                    { Variable($1) }

tree:
   headnode                     {Leaf($1)}
|  headnode REDGE children      {SubTree($1, Redge, $3)}
|  headnode LEDGE children      {SubTree($1, Ledge, $3)}
|  headnode UEDGE children      {SubTree($1, Uedge, $3)}

headnode:
  LITINT {$1}

children:
```

```
  nodetree {[$1]}
| nodetree COMMA children {$1 :: $3}

nodetree:
  LITINT {Leaf($1)}
| LPAREN tree RPAREN {$2}

plainString:
        LITSTR                                  { $1 }

actuals_opt:
        /* nothing */                           { [] }
|       actuals_list                            { List.rev $1 }

actuals_list:
        expr                                    { [$1] }
|       actuals_list SEMI expr                  { $3 :: $1 }
```

8.3 ast.ml

```
type op = Add | Sub | Mult | Div | Equal | Neq | Less | Leq | Greater | Geq |
And | Or | Mod

type typ = Node | Int | Flt | Str | Edg
type edg =  Ledge | Redge | Uedge
type paren = Rparen | Lparen
type dir = In | Out
type ruleProp = Dolr | Tilde

type variable =
        VarId of string
(* all recvars are attributes! *)
|       RecVar of variable * string
|       ArrayIndDyn of string * variable
|       ArrayIndStat of string * int

(* these are for inside connection contexts ONLY *)
type tree = Leaf of int | SubTree of int * edg * tree list

type expr =
        Lit_Flt of float
|       Lit_Str of string
|       Lit_Int of int
|       Variable of variable
|       Binop of expr * op * expr
|       Actuals of expr list
|       NodeInEdge of variable * expr
|       NodeOutEdge of variable * expr
|       NodeInEdges of variable
|       NodeOutEdges of variable
|       EdgeInNode of variable
|       EdgeOutNode of variable
|       NodeChild of variable * expr
|       NodeParent of variable * expr


type conObj =
        Lit_Int_Con of int
|       Edge of edg
|       Paren of paren

type addByType = AddByNode | AddByEdge

type stmt =
        Expr of expr
|       EdgeCreation of variable * edg * variable
|       Declare_Only of typ * variable
|       Declare of typ * variable * expr
```

```
|       Call of string * expr list
|       CallRule
|       Crawl of string * expr list * variable * string
|       CreateGraph of variable * tree
|       Print of expr list
|       If of expr * stmt list * stmt list
|       While of expr * stmt list
|       Assign of variable * expr
|       SetRule of string
|       RAdd  of variable
|       RAddFront of variable
|       RAddBy of string * addByType * ruleProp * int
|       RAddByFront of string * addByType * ruleProp * int

type formal =
        Validate of dir * typ * string

type func_decl = {
        func_type : string;
        fname : string;
        formals : formal list;
        body : stmt list;
}

type program = func_decl list
```

8.4 interpret.ml

```
let verbose = false

open Ast

module ST = Map.Make(String)
let error= [| false |]


let operation_role (o : op) = match o with
          Add -> "add"
        | Equal -> "any"
        | Neq -> "any"
        | Less -> "basic"
        | Leq ->"basic"
        | Greater -> "basic"
        | Geq ->"basic"
        | And -> "int"
        | Or ->"int"
        | Mod ->"int"
        | Sub -> "num"
        | Mult ->"num"
        | Div ->"num"

let dir2str (d : dir) = match d with
          In ->"in"
        | Out ->"out"
let type2str (t : typ) = match t with
          Node -> "node"
        | Int -> "int"
        | Flt -> "flt"
        | Str -> "str"
        | Edg -> "edg"


let rec var2str (v : variable) = match v with
          VarId s -> s
        | RecVar (v,s) -> var2str v
        | ArrayIndDyn (s,v) -> s
        | ArrayIndStat (s,i) -> s



let drop_arr s =
        let substr = String.sub s 0 3 in
        match substr with
         "nod" -> "node"
        | "edg" -> "edg"
        |_-> substr

```

```
let rec get_variable_type map (v : variable) = match v with
          VarId(s) ->

                if ST.mem s map then ST.find s map else "error"

        | RecVar(v,s1) ->
                let vtyp = (get_variable_type map v) in
                if  vtyp = "node" || vtyp ="edg"
                then "int"
                else "error"
        | ArrayIndDyn (s,v) ->
                if ST.mem s map && (get_variable_type map v ) = "int"
                then drop_arr (ST.find s map)
                else "error"
        | ArrayIndStat (s,i) ->
                if ST.mem  s map
                then drop_arr (ST.find s map)
                else "error"

let check_node (v: variable) map =
        if  (get_variable_type map v ) = "node"
        then true
        else (error.(0) <- true; print_endline ("Argument is not a node");false)

let check_edge (v: variable) map =
        if ((get_variable_type map v) = "edg")
        then true
        else (error.(0) <- true; print_endline ("Argument is not a edge");false)

let check_index v map =
        (get_variable_type map v) = "int"

let addVar ( v: variable) (t: typ) map = match v with
          VarId(s) -> (ST.add s (type2str t) map)
        | ArrayIndStat (s,i) ->( ST.add s ((type2str t)^"arr") map)
        | ArrayIndDyn (s,v) ->  if check_index v map
                                then (ST.add s ((type2str t)^"arr") map)
                                else (print_endline ("Array size not int"); map)
        | _-> map



let check_con_var (v : variable) map=match v with

          ArrayIndStat (s,i) ->
                        if ( ST.mem s  map)
                        then ( error.(0) <- true; print_endline (s ^ " already declared"); map)
```

```
                        else (print_endline (s ^ " declared with connection context");
                                ST.add s "nodearr" map )



        | ArrayIndDyn (s,v) ->
                        if ( ST.mem s  map) || not (check_index v map)
                        then ( error.(0) <- true;print_endline (s ^ " problem with connection context"); map)
                        else (print_endline (s ^ " declared with connection context");
                                ST.add s "nodearr" map)
        |_->    (error.(0) <- true;
                        print_endline ((var2str v) ^ " not proper variable for connection context");
                        map)




let op_check typ1 typ2 optyp  =
        match optyp with
         "basic" -> typ1 = "str" || typ1 = "int" || typ1 = "flt"
        | "num" ->  typ1 = "flt"   || typ1 = "int"
        | "int" -> typ1 = "int" && typ2 = "int"
        | "add" -> typ1 = "str" || typ1 = "int" || typ1 = "flt"
        | "any" -> true
        | _ -> true



let rec get_expr_type map (e : expr) = match e with
          Lit_Flt f -> "flt"
        | Lit_Int i -> "int"
        | Lit_Str s -> "str"
        | Variable v -> (get_variable_type map v)
        | Binop (e1, op, e2) ->
                        let optyp = operation_role op in
                        let typ1 = get_expr_type map e1 in
                        let typ2 = get_expr_type map e2 in
                        if      (op_check typ1 typ2 optyp)
                                && (typ1 = typ2
                                || ( (typ1 = "int" || typ1 ="flt")
                                        && (typ2 = "int" || typ2 = "flt")
                                        ))
                        then    if optyp = "basic" || optyp = "any"
                                then "int"
                                else typ1

                        else (  error.(0) <- true;
```

```
                                 print_endline("cannot operate on " ^ typ1 ^ " and " ^ typ2 ^ " with " ^ optyp ^ " operation"); "error" )



        | Actuals(el) ->
                 List.fold_left  (fun tp tc-> if (get_expr_type map tc) = tp
                                              then tp
                                              else "error")
                                (get_expr_type map (List.hd el))
                                el
        | NodeInEdge(v,e)->
                        if  (check_node v map)
                        then    if (get_expr_type map e) = "int"
                                then "edg"
                                else (error.(0) <- true;print_endline("Inedge indexed with non int"); "error")
                        else (error.(0) <- true;print_endline("Cannot call InEdge on variable");"error")
        | NodeOutEdge (v,e)->
                        if  (check_node v map)
                        then    if (get_expr_type map e) = "int"
                                then "edg"
                                else (error.(0) <- true;print_endline("Outedge indexed with non int"); "error")
                        else (error.(0) <- true;print_endline("Cannot call outedge on variable");"error")
        | NodeInEdges(v)->
                        if  (check_node v map)
                        then "int"
                        else (error.(0) <- true;print_endline("Cannot call inedges on variable");"error")
        | NodeOutEdges (v)->
                        if(check_node v map)
                        then "int"
                        else (error.(0) <- true;print_endline("Cannot call outedges on variable");"error")
        | EdgeInNode (v)->
                        if (check_edge v map)
                        then "node"
                        else (error.(0) <- true;print_endline("Cannot call innode on variable");"error")
        |  EdgeOutNode(v)->
                        if(check_edge v map)
                        then "node"
                        else (error.(0) <- true;print_endline("Cannot call outnode on variable");"error")
```

```
        | NodeChild(v,e) ->
                        if  (check_node v map)
                        then    if (get_expr_type map e) = "int"
                                then "node"
                                else (  error.(0) <- true;
                                        print_endline("Nodechild indexed with non int");
                                        "error")
                        else (error.(0) <- true;print_endline("Cannot call Nodechild on variable");"error")
        | NodeParent(v,e) ->
                        if  (check_node v map)
                        then    if (get_expr_type map e) = "int"
                                then "node"
                                else (error.(0) <- true;
                                        print_endline("NodeParent indexed with non int");
                                        "error")
                        else (error.(0) <- true;print_endline("Cannot call NodeParent on variable");"error")


(* keep track of the type as well as the variable name *)
let get_formals_from_fdecl formals =
        let m (f : formal) =
                        match f with Validate(d, t, s) -> (s, (type2str t))
        in List.map m formals

let extract_type_from_formal (f : formal) =
        match f with Validate(d, t, s) -> ((dir2str d) ,(type2str t))


let get_tuple_from_fdecl (f : func_decl) =
        (f.fname, (List.fold_right (fun a b -> (extract_type_from_formal a)::b) f.formals []))


let assign_method (fdecl : func_decl) crawlh ruleh opth =
        match fdecl.func_type with
          "rule" ->
                (fun a -> Hashtbl.add ruleh (fst a) (snd a) )
                (get_tuple_from_fdecl fdecl)
        | "opt" ->
                (fun a -> Hashtbl.add opth (fst a) (snd a) )
                (get_tuple_from_fdecl fdecl)
        | "crawl" ->
                (fun a -> Hashtbl.add crawlh (fst a) (snd a) )
                (get_tuple_from_fdecl fdecl)
        |      _ -> ( error.(0) <- true; print_endline "cannot identify method type")
```

```


let use_var name hash =
        if      ST.mem  name hash
        then    true
        else    false

let check_assign  vtyp  (e : expr) map =
        let exprtyp = (get_expr_type map e) in
                 exprtyp = vtyp
                || (exprtyp = "int" &&  vtyp = "flt")
                || (exprtyp = "flt" &&  vtyp = "int")

let add_special_var mtyp map =
        if mtyp = "rule" || mtyp = "crawl"
        then ST.add "current" "node" map
        else map



let check_argument map (e : expr) (dir, typ) =
        if dir = "out"
        then match e with Variable(v) ->
                if (get_variable_type map v ) = typ
                then true
                else false
        |       _-> false
        else
                if (get_expr_type map e) = typ
                then true
                else false


let rec check_args map explist arglist =
        if  (List.length explist = List.length arglist
                && ((List.length explist) =0
                        || (check_argument map (List.hd explist)  (List.hd arglist))
                ))
        then
                if ((List.length explist) = 0)
                then    true
                else check_args map (List.tl explist) (List.tl arglist)
        else
                false


let make_table f g crawlh ruleh=

        let formals_st =
                let addtomap smap word =
```

```
                        match word with
                        (s, t) ->
                                if not (ST.mem s smap)
                                then    (print_endline ("adding opt argument to symbol table: " ^ s);
                                    ST.add s t smap)
                                else (error.(0) <- true;print_endline ("Argument name " ^ s ^ " used multiple times");
                                        smap)
                in
                List.fold_left addtomap (add_special_var f.func_type ST.empty)
                (get_formals_from_fdecl f.formals)
        in
        let checkvar map (v : variable) =
                if ST.mem (var2str v) map
                then map
                else
                        if Hashtbl.mem g (var2str v)
                        then map
                        else    (error.(0) <- true;
                                print_endline ("ERROR: undeclared variable: " ^ (var2str v));
                                map)
        in

let rec checkexp map (e : expr) =
        match e with
          Lit_Flt f -> map
        | Lit_Str s -> map
        | Lit_Int i -> map
        | Actuals a -> List.fold_right (fun m n -> checkexp n m ) a map
        | Variable v -> checkvar map v
        | Binop (e1, o, e2) -> checkexp (checkexp map e1) e2
        | _ ->  (print_endline "unimplemented expression"; map)
in

let rec checkstmt map (s : stmt) =
      match s with
      (* check declarations *)
          Declare_Only (t, v) ->

                if ST.mem (var2str v) map
                then
                        (error.(0) <- true; print_endline ("ERROR: duplicate local declaration: " ^ (var2str v));
                        map)
                else
                        if Hashtbl.mem g (var2str v)
                        then (error.(0) <- true;
```

```
                              print_endline ("ERROR: duplicate GLOBAL declaration: " ^ (var2str v));
                              map)
                       else addVar v t map

      | Declare (t, v, e) ->
              if ST.mem (var2str v) map
              then
                      (error.(0) <- true;
                      print_endline ("ERROR: duplicate local declaration: " ^ (var2str v));
                      map)
              else
                   if Hashtbl.mem g (var2str v) then
                       (error.(0) <- true;print_endline ("ERROR: duplicate GLOBAL declaration: " ^ (var2str v));
                        map)
                   else

              if check_assign  (type2str t) e map
              then addVar v t map
              else
                      (error.(0) <- true;
                      print_endline("Expression not of type "^(type2str t)^", variable not declared "  );
                      map)

      | Assign (v, e) ->
              if ST.mem (var2str v) map
              then
                      if check_assign  (drop_arr (get_variable_type map v )) e map
                      then    (print_endline((var2str v)^ " assigned value");
                              map)
                      else    (error.(0) <- true;
                              print_endline((var2str v)^"'s type did not match type");
                              map)
              else
                      (error.(0) <- true;
                      (print_endline ((var2str v)^ " not defined. Cannot assign value"));
                      map)
    |CreateGraph (v,t) -> check_con_var v map

    (* check expressions *)
    | Expr (e) ->
              checkexp map e
    (* check when we call functions? *)
```

```
        | Call (c, elist) ->

                if Hashtbl.mem g c
                then    (print_string ("calling opt: " ^ c ^" :");
                        (let argtypes = Hashtbl.find g c
                        in
                        if check_args map elist argtypes
                        then print_endline(c ^" call passed with proper arguments")
                        else    (error.(0) <- true;
                                print_endline (c ^ " call passed with incorrect arguments")));
                                map)


                else
                        (error.(0) <- true;
                        print_endline ("ERROR: undefined opt: " ^ c);
                        map)
        (* if/while *)
          | While (e, sl) ->
                (if not ((get_expr_type map e) = "int")
                then    (error.(0) <- true;
                         print_endline ("While expression is not evaluating to an int"))
                else ();
                let _ = (List.fold_left checkstmt (checkexp map e) (sl)) in map)

          | If(e,sl1,sl2) ->
                (if not ((get_expr_type map e) = "int")
                 then   (error.(0) <- true;
                        print_endline ("If expression is not evaluating to an int"))
                 else print_string (""));
                        (let _ = (List.fold_left  checkstmt   (checkexp map e) sl1) in
                         let _ = (List.fold_left  checkstmt   (checkexp map e) sl2) in
                         map)
          | Crawl (cn , el, no ,ru)  ->
                if Hashtbl.mem crawlh cn
                then
                let argtypes = Hashtbl.find crawlh cn
                in
                if check_args map el argtypes
                then    if (check_node no map) && (Hashtbl.mem ruleh ru)
                        then map
                        else (error.(0) <- true;
                                print_endline("Wrong arguments passed to " ^cn^ " crawl in from-where clause");
```

```
                        map)
        else  (error.(0) <- true;
                print_endline("Crawl " ^cn^ " called with improper
                    arguments"); map)
        else  (error.(0) <- true;
                print_endline("Crawl " ^cn^ " undefined");
                map)
| CallRule ->
        if f.func_type = "crawl"
        then     map
        else (print_endline(f.fname ^" is not a crawl.  Cannot use
            call"); map)
| Print(el) ->
        List.fold_left (fun m e-> let etyp = get_expr_type m e in
                                match etyp with
                                "edg" ->(error.(0) <- true;
                                   print_endline
                                ("Edges cannot be printed"); m)
                                |"node" ->(error.(0) <- true;
                                   print_endline
                                ("Nodes cannot be printed"); m)
                                |"error" ->(error.(0) <- true;
                                   print_endline
                                ("Expression could not be printed
                                   "); m)
                                |_-> m
                        )
                        map
                        el
| SetRule(rl)->
        if f.func_type = "crawl"
        then    if Hashtbl.mem ruleh rl
                then map
                else(error.(0) <- true;print_endline(rl ^" is not
                   a declared rule"); map )
        else (error.(0) <- true;print_endline(f.fname ^" is not a
           crawl.  Cannot use SetRule");
                map)
| RAdd(v)  ->   if check_node v map
                then map
                else (error.(0) <- true;print_endline("Adding
                   variable not of type node"); map)
| RAddFront(v) -> if check_node v map
                then map
                else (error.(0) <- true;print_endline("Adding
                   variable not of type node"); map)
| RAddBy(s,a,rp,i)-> map
| RAddByFront(s,a,rp,i)-> map
| EdgeCreation (v1, edg, v2) -> if check_node v1 map  &&
   check_node v2 map
                then map
```

```
                        else (error.(0) <- true;print_endline("Non nodes passed to variables");
                        map)
        in
        List.fold_left checkstmt formals_st f.body

let dump_table t =
        print_endline "=================";
        print_endline "symbol table dump: ";
        ST.fold (fun k v l -> print_endline ("--> " ^ k ^ ": " ^ v)) t ()

let dump_tuple t =
        (
        print_endline ("---> " ^ (fst t) ^ ": " ^ (string_of_int (List.length (snd t))));
        List.map print_endline (snd t);
        )

let dump_hash h =
        print_endline "=================";
        print_endline "global signature dump: ";
        Hashtbl.iter (fun a b -> ( print_string (a ^ ": ");
                                List.fold_right
                                        (fun c d -> print_string ((fst c) ^"" ^(snd c) ^ " ")) b ();
                                print_endline ""; ) )
        h;

        print_endline "================="

let check_ast (p : program) =
        match p with
        (fdecllist) ->
                let funcHash = Hashtbl.create 100 in
                let crawlHash =Hashtbl.create 50 in
                let ruleHash = Hashtbl.create 50 in
                (  List.fold_right
                        (fun a b -> assign_method a crawlHash ruleHash funcHash  )
                        (fdecllist)
                        () ;
                dump_hash funcHash;
        (
                if not (Hashtbl.mem funcHash "main")
                then  (error.(0) <- true; print_endline("No main function declared"))
                else (print_endline("Starting")) );
                List.fold_right (fun a b -> dump_table a)
                        (List.map       (fun a -> make_table a funcHash crawlHash ruleHash)
                                        (List.rev fdecllist))
```

```
                      ();
              if error.(0)
              then print_endline "an error!"
              else print_endline "no error!";
              error.(0)
       )
```

8.5 translate.ml

```
open Ast
open Cast

let rec varname_from_variable v = match v with
        VarId s -> s
|       RecVar (v,s) -> (varname_from_variable v) ^ "." ^ s
|       ArrayIndDyn (s,v) -> s ^ "[" ^ (varname_from_variable v) ^ "]"
|       ArrayIndStat (s,i) -> s ^ "[" ^ (string_of_int i) ^ "]"


let addtoend l e = List.rev (e :: (List.rev l))


let cop_from_op (o : op ) = match o with
          Add -> CAdd
        | Sub -> CSub
        | Mult -> CMult
        | Div -> CDiv
        | Equal -> CEqual
        | Neq -> CNeq
        | Less -> CLess
        | Leq -> CLeq
        | Greater-> CGreater
        | Geq -> CGeq
        | And -> CAnd
        | Or -> COr
        | Mod -> CMod

let trans_dir d = match d with
          Redge -> CRedge
        | Ledge -> CLedge
        | Uedge -> CUedge

let num_from_leaf f = match f with
          Leaf p -> p
        | _ -> -1


let gethead ( s : tree ) treename = match s with
         SubTree (i,e,tl) -> CId (Cvar (treename ^ "[" ^ (string_of_int i) ^ "]"))
        | Leaf (i) -> CId (Cvar (treename ^ "[" ^ (string_of_int i) ^ "]"))

let rec cstmtlist_of_edge_declarations name size =
        if size = 0
        then []
        else
```

```
                (CAssign(CArrayStat(name, size - 1), CCallNew("DiGrEdge",[]) ) )
                :: (cstmtlist_of_edge_declarations name (size-1))

let rec cstmtlist_of_tree_declarations name size =
        if size = 0
        then []
        else
           ( CAssign(CArrayStat(name, size - 1), CCallNew("DiGrNode",[]) ) ) ::
        (cstmtlist_of_tree_declarations name (size - 1))

let cstmtlist_from_tree tree treename =
        let rec treefold element stmt_list treename = match element with
             Leaf (i) -> stmt_list
           | SubTree (i,e,tl) ->
                (
                match e with
                        Redge ->
                                List.fold_right (fun b a -> treefold b a treename)
                                                 tl
                                                (stmt_list @ (List.map
                                                        (fun f -> CExpr (CCallNew ("DiGrEdge",
[CId (Cvar (treename ^ "[" ^ (string_of_int i) ^ "]"));
(gethead f treename)])) )  tl))

                      | Ledge ->
                                 List.fold_right (fun b a -> treefold b a treename)
                                                 tl
                                                (stmt_list @ (List.map
(fun f -> CExpr (CCallNew
                ("DiGrEdge",[(gethead f treename);
                CId (Cvar (treename ^ "[" ^ (string_of_int i) ^ "]"))])) ) tl))

                      | Uedge ->
                                 List.fold_right (fun b a -> treefold b a treename)
                                tl
                                (stmt_list @ (List.map (fun f -> CExpr (CCallNew
                                ("DiGrEdge",[CId (Cvar (treename ^ "[" ^ (string_of_int i) ^ "]"));
                                (gethead f treename); CLiteral_String("true")])) )  tl))
        )
        in
```

```
        treefold tree [] treename


let rec cvar_from_var v = match v with
          VarId (s) -> Cvar(s)
        | ArrayIndStat (name, index) -> CArrayStat(name, index)
        | ArrayIndDyn (name, index) -> CArrayDyn(name, (cvar_from_var index))
        |  RecVar (v, s) -> (print_endline "ERROR: this should never be called?!";
        Cvar(s))

let rec cexpr_from_expr ( e : expr ) = match e with
          Lit_Flt f -> CLiteral_Float(f)
        | Lit_Str s -> CLiteral_String(s)
        | Lit_Int i -> CLiteral_Int(i)
        | Actuals a ->
                        (print_endline ("ERROR: can't assign list to single object");
                        CNoexpr)
        | Binop (e1, o, e2) -> CBinop (cexpr_from_expr e1, cop_from_op o, cexpr_from_expr e2)
        | NodeInEdge (v,e) ->
                        CObjCall(Cvar (varname_from_variable v), "getInEdge", [cexpr_from_expr e])
        | NodeOutEdge (v,e) ->
                        CObjCall(Cvar (varname_from_variable v), "getOutEdge", [cexpr_from_expr e])
        | NodeChild (v,e) ->
                CObjCall(Cvar (varname_from_variable v), "getChild", [cexpr_from_expr e])
        | NodeParent (v,e) ->
                  CObjCall(Cvar (varname_from_variable v), "getParent", [cexpr_from_expr e])
                | NodeInEdges v -> CObjCall(Cvar (varname_from_variable v), "InEdges", [])
                | NodeOutEdges v -> CObjCall(Cvar (varname_from_variable v), "OutEdges", [])
                | EdgeInNode v -> CObjCall(Cvar (varname_from_variable v), "inNode", [])
                | EdgeOutNode v -> CObjCall(Cvar (varname_from_variable v), "outNode", [])
        | Variable v ->
                (
                    match v with
                          VarId s -> CId (Cvar s)
                        | RecVar (v,s) ->
                                CObjCall(Cvar(varname_from_variable v), "getAttribute", [CId(Cvar("\"" ^ s ^ "\""))])
                        | ArrayIndStat (s,i) -> CId(CArrayStat(s,i))
```

```
                        | ArrayIndDyn (s,v) -> CId(CArrayDyn(s, Cvar(
                           varname_from_variable v)))
                   )

let rec cexprlist_from_actualsexpr e = match e with
                Actuals a -> CActuals(List.map (fun m -> cexpr_from_expr m
                   ) a)
        | _ -> CActuals([])

let ctype_from_typ (t : typ) = match t with
           Node -> CDiGrNode
        |  Int -> CInt
        |  Flt -> CFloat
        |  Str -> CString
        |  Edg -> CDiGrEdge

let rec cstmt_from_stmt (s : stmt ) = match s with
           Print l -> CPrint(List.map cexpr_from_expr l)
        |  Call (s, l) -> CExpr(CCall (s, (List.map cexpr_from_expr l)))
        |  CallRule -> CExpr(CCall ("rule", [CId(Cvar("current"));
        CId(Cvar("queue"))]))
        |  SetRule r -> CAssignRule("rule", CId(Cvar(r)))
        |  Assign (v, e) -> (
                                match v with
                                          VarId (s) ->
                                                CAssign(Cvar(s),
                                                   cexpr_from_expr e)
                                        | ArrayIndStat (name, index) ->
                                                CAssign(CArrayStat(name,
                                                   index), cexpr_from_expr
                                                    e)
                                        | ArrayIndDyn (name, index) ->
                                                CAssign(CArrayDyn(name, (
                                                   cvar_from_var index)),
                                                   cexpr_from_expr e)
                                        | RecVar (v, s) -> CExpr(CObjCall(
Cvar(varname_from_variable v),
"setAttribute" ,
[CId(Cvar("\"" ^ s ^ "\""));
                                                        cexpr_from_expr e]
                                                              ))
                        )
        |  RAdd n -> CExpr(CObjCall(Cvar("returnQueue"),"push_back", [CId(
           Cvar((varname_from_variable n))]))
        |  RAddFront n -> CExpr(CObjCall(       Cvar("returnQueue"),
                                                "push_front",
                                                [CId(Cvar((
                                                   varname_from_variable n
                                                   )))]))

        |  RAddBy (s, t, rp, i) -> CExpr(
```

```
       CCall ("DiGrAddBy", [CId(Cvar("current")); CId(Cvar("returnQueue"));
        CId(Cvar("BACK"));
          (match t with AddByNode -> CId(Cvar("ADDBY_NODE")) | AddByEdge ->
        CId(Cvar("ADDBY_EDGE")));
          CLiteral_String("\"" ^ s ^ "\"");
          ( match rp with Dolr -> CId(Cvar("DESCENDING")) | Tilde ->
        CId(Cvar("ASCENDING")));
          CLiteral_Int(i)]
             ))
|  RAddByFront (s, t, rp, i) -> CExpr(
       CCall ( "DiGrAddBy",
                [CId(Cvar("current")); CId(Cvar("returnQueue"));
        CId(Cvar("FRONT"));
          (match t with AddByNode -> CId(Cvar("ADDBY_NODE")) | AddByEdge ->
                CId(Cvar("ADDBY_EDGE")));
          CLiteral_String("\"" ^ s ^ "\"");
          ( match rp with Dolr -> CId(Cvar("DESCENDING"))
                        | Tilde -> CId(Cvar("ASCENDING")));
                         CLiteral_Int(i)]
             ))
|  Crawl (s, el, a1, a2) -> CExpr(CCall (s,
   [CId(Cvar((varname_from_variable a1))); CId(Cvar(a2))] @ (List.map cexpr_from_expr el)
   ) )
|       CreateGraph (variable, tree) ->
(
match variable with
           RecVar (v1, v2) ->
                        ( print_endline "ERROR: only arrays can be assigned to a connection context";
                         CExpr(CNoexpr) )
        |  VarId s->
                         ( print_endline "ERROR: only arrays can be assigned to a connection context" ;
                        CExpr(CNoexpr) )
        | ArrayIndDyn (s,i) ->
                ( print_endline "ERROR: only statically-sized arrays can be assigned to a connection context";
                 CExpr(CNoexpr) )
        | ArrayIndStat (name, size) ->
            CBlock(
              CDeclare(CSigArr(CTypePointer(CDiGrNode),CArrayStat(name,size)))
              :: (
                 (List. rev (cstmtlist_of_tree_declarations name size)) @
```

```
                 (cstmtlist_from_tree tree name)
                 )
            )
        )
        |  Declare_Only (t, v) ->
        (
           match t with
           Node ->
           (
              match v with
                VarId (s) ->  CDeclareAssign (CSigPtr(CDiGrNode,s), CCallNew("DiGrNode",[]))
                        | ArrayIndStat (name, size) -> CBlock(

                                CDeclare(CSigArr(CTypePointer(CDiGrNode),CArrayStat(name,size)))
                                :: (
                                (List.rev (cstmtlist_of_tree_declarations name size))
                                )
                        )
      | ArrayIndDyn (s, e) ->
                ( print_endline ("ERROR: cannot declare a type for an element of array" ^ s);
                CExpr(CNoexpr) )
      | RecVar (v, s) -> ( print_endline ("ERROR: cannot declare a type for an attribute" ^
                        (varname_from_variable v));
                        CExpr(CNoexpr) )
   )

| Edg ->
   (
      match v with

        VarId (s) ->
                CDeclareAssign (CSigPtr(CDiGrEdge,s), CCallNew("DiGrEdge",[]))
      | ArrayIndStat (name, size) -> CBlock(
                        CDeclare(CSigArr(CTypePointer(CDiGrEdge),CArrayStat(name,size)))
                        :: (
                                (List.rev (cstmtlist_of_edge_declarations name size))
                        )
                        )
      | ArrayIndDyn (s, e) ->
```

```
                        ( print_endline ("ERROR: cannot declare a type for an element of array" ^ s);
                         CExpr(CNoexpr) )
              | RecVar (v, s) -> ( print_endline ("ERROR: cannot declare a type for an attribute" ^
(varname_from_variable v));
                                 CExpr(CNoexpr) )
           )
           | _ ->
           (
              match v with

                VarId (s) ->  CDeclare (CSigVar(ctype_from_typ t,s))
                | ArrayIndStat (s,i) -> CDeclare(CSigArr(ctype_from_typ t, CArrayStat(s, i)))
                | ArrayIndDyn (s, i) ->
                        CDeclare(CSigArr(ctype_from_typ t, CArrayDyn(s, cvar_from_var i)))
                        | _ ->  CExpr(CNoexpr)
           )
        )
        | Declare(t, v, e) ->
           (
                        match t with
                        Node ->
                        (
                        match v with
                                VarId (s) ->  (
                                        match e with
                                                Variable a -> CDeclareAssign
                                                        (CSigPtr(CDiGrNode,s), cexpr_from_expr e)
                                                | EdgeInNode a ->
                                                        CDeclareAssign ( CSigPtr( CDiGrNode,s), cexpr_from_expr e)
                                                | EdgeOutNode a ->
                                                        CDeclareAssign ( CSigPtr( CDiGrNode,s), cexpr_from_expr e)
                                                | NodeChild(_,_) -> CDeclareAssign (CSigPtr (CDiGrNode,s), cexpr_from_expr e)
```

```
                                                | NodeParent(_,_) ->
                                                   CDeclareAssign (CSigPtr
                                                   (CDiGrNode,s),
                                                   cexpr_from_expr e)
                                                | _ ->   CExpr(CNoexpr)
                                        )
                                | ArrayIndStat(name,size) ->
                                        CDeclareAssign(CSigArr(
                                           CTypePointer(CDiGrNode),
                                                                CArrayStat
                                                                   (name,
                                                                   size)),
cexprlist_from_actualsexpr e)
                        | _ ->  CExpr(CNoexpr)
                        )
                        | Edg ->
                        (
                        match v with
                                VarId (s) ->  (
                                        match e with
                                                  Variable a ->
                                                     CDeclareAssign (
                                                     CSigPtr(CDiGrEdge,s),
                                                      cexpr_from_expr e)
                                                | NodeOutEdge(_,_) ->
CDeclareAssign (CSigPtr(CDiGrEdge,s), cexpr_from_expr e)
                                                | NodeInEdge(_,_) ->
                                                        CDeclareAssign (
                                                           CSigPtr(
                                                           CDiGrEdge,s),
                                                           cexpr_from_expr
                                                            e)
                                                | _ -> CExpr(CNoexpr)
                                        )

                                | ArrayIndStat(s,z) ->
                                        CDeclareAssign (CSigArr(CDiGrEdge,
                                                                CArrayStat
                                                                   (s,z)),
cexprlist_from_actualsexpr e)
                        | _ ->  CExpr(CNoexpr)
                        )
                        | _ ->
                        (
                                match v with
                                 VarId (s) ->
                                                CDeclareAssign(CSigVar(
                                                   ctype_from_typ t, s),
                                                   cexpr_from_expr e)
                                | ArrayIndStat(s,z) ->
```

```
                                        CDeclareAssign (CSigArr(
                                           ctype_from_typ t,
                                                                CArrayStat
                                                                   (s,z)),
cexprlist_from_actualsexpr e)
                                |       _ -> CExpr(CNoexpr)
                        )
           )
                | Expr e -> CExpr (cexpr_from_expr e)
                | EdgeCreation (s1, e, s2) ->
                   (
                      match e with
                        Redge ->
                        CExpr(CCallNew ("DiGrEdge",[CId (Cvar (
                           varname_from_variable s1));
                                CId (Cvar (varname_from_variable s2))]))
                      | Ledge ->
                                CExpr(CCallNew ("DiGrEdge",[CId (Cvar (
                                   varname_from_variable s2));
                                CId (Cvar (varname_from_variable s1))]))
                      | Uedge -> CExpr(CCallNew ("DiGrEdge",[CId (Cvar (
                         varname_from_variable s1));
                                CId (Cvar (varname_from_variable s2));
                                   CLiteral_String("true")]))
                   )
                | If (e, sl1, sl2) ->
           CIf( cexpr_from_expr e, List.map cstmt_from_stmt sl1, List.map
        cstmt_from_stmt sl2)
        | While (e, sl) ->
           CWhile( cexpr_from_expr e, List.map cstmt_from_stmt sl)

let auto_crawl_formals = [CSigPtr(CDiGrNode,"current");
        CFuncFormal(    CVoid,
                        "rule",
                        [CTypePointer(CDiGrNode); CTypePointer(CVector(
                           CTypePointer(CDiGrNode)))]
                )]
let auto_rule_formals = [CSigPtr(CDiGrNode,"current");
        CSigPtr(CVector(CTypePointer(CDiGrNode)),"returnQueue")]

let csigvar_from_formal f = match f with
        Validate (d, t, s) -> (match d with
            In -> (match t with
                  Int -> CSigVar(CInt,s)
                | Flt -> CSigVar(CFloat,s)
                | Str -> CSigVar(CString,s)
                | Node -> CSigPtr(CDiGrNode, s)
                | Edg -> CSigPtr(CDiGrEdge, s)
                    )
        |    Out -> (match t with
                  Int -> CSigRef(CInt,s)
```

```
                | Flt -> CSigRef(CFloat,s)
                | Str -> CSigRef(CString,s)
                | Node -> CSigPtr(CDiGrNode, s)
                | Edg -> CSigPtr(CDiGrEdge, s)
                    )
        )

let add_to_list e l = e :: l
let merge_two_lists l1 l2 = List.fold_right add_to_list l1 l2

let cfdecl_from_fdecl (f : func_decl) =
        (if f.func_type = "opt"
        then
          (
                if f.fname = "main"
                then
                     { cfname = f.fname;
                       creturntype = CInt;
                       cformals = (List.map csigvar_from_formal f.formals);
                       cbody = [CTryCatchBlock(List.map cstmt_from_stmt f.body)] }
                else
                     { cfname = f.fname;
                       creturntype = CVoid;
                       cformals = (List.map csigvar_from_formal f.formals);
                       cbody = (List.map cstmt_from_stmt f.body) }
                  )
        else    if f.func_type = "crawl"
                then
                  { cfname = f.fname;
                    creturntype = CVoid;
                    cformals = auto_crawl_formals @ (List.map csigvar_from_formal f.formals);

                 cbody =
                        [
                               CDeclareAssign(CSigPtr(CVector(CTypePointer(CDiGrNode)),"queue"),
                        CCallNew("deque<DiGrNode*>",[])) ;
                               CExpr(CObjCall(Cvar("queue"),"push_back",[CId(Cvar("current"))])) ;
                               CDoWhile(CBinop(CId(Cvar("queue->size()")), CGreater, CLiteral_Int(0)),
                                       (
                                         CAssign(Cvar("current"), CId(Cvar("queue->front()"))) ::
                                       (
                                           CExpr(CId(Cvar("queue->pop_front()"))) ::
```

```
                                           (List.map cstmt_from_stmt f.body)) )
                        )]


                 }
                else
                  { cfname = f.fname;
                    creturntype = CVoid;
                    cformals = auto_rule_formals;
                    cbody =
                       (List.map cstmt_from_stmt f.body)
                   }
        )


let cast_from_ast (p : program) = match p with
        (fdecllist) ->
                List.map cfdecl_from_fdecl (List.rev fdecllist)
```

8.6 cast.ml

```
type cop = CAdd | CSub | CMult | CDiv | CEqual | CNeq | CLess | CLeq | CGreater | CGeq | CAnd | COr | CMod
type cdirection = CLedge | CRedge | CUedge
type ctype = CVoid | CInt | CFloat | CString | CDiGrNode | CDiGrEdge | CVector of ctype | CTypePointer of ctype

type cvar =
  Cvar of string
| CArrayStat of string * int
| CArrayDyn of string * cvar
| CPointer of string

type cexpr =
    CLiteral_Int of int
  | CLiteral_Float of float
  | CLiteral_String of string
  | CActuals of cexpr list
  | CId of cvar
  | CBinop of cexpr * cop * cexpr
  | CCallNew of string * cexpr list
  | CCall of string * cexpr list
  | CObjCall of cvar * string * cexpr list
  | CIdAddr of string
  | CNoexpr

type csigvar =
  CSigVar of ctype * string
| CSigVect of ctype * string
| CSigPtr of ctype * string
| CSigRef of ctype * string
| CSigArr of ctype * cvar
| CFuncFormal of ctype * string * ctype list

type cstmt =
    CTryCatchBlock of cstmt list
  | CBlock of cstmt list
  | CExpr of cexpr
  | CDeclare of csigvar
  | CDeclareAssign of csigvar * cexpr

  | CAssign of cvar * cexpr
  | CAssignRule of string * cexpr
  | CReturn of cexpr
  | CIf of cexpr * cstmt list * cstmt list
  | CWhile of cexpr * cstmt list
  | CDoWhile of cexpr * cstmt list
  | CPrint of cexpr list

type cfunc_decl = {
```

```
    creturntype : ctype;
    cfname : string;
    cformals : csigvar list;
    cbody : cstmt list;
  }

type cprogram = cfunc_decl list
```

8.7 compile.ml

```
open Cast

let string_of_cop o = match o with
          CAdd -> "+"
        | CSub -> "-"
        | CMult -> "*"
        | CDiv -> "/"
        | CEqual -> "=="
        | CNeq -> "!="
        | CLess -> "<"
        | CLeq -> "<="
        | CGreater -> ">"
        | CGeq -> ">="
        | CAnd -> "&&"
        | COr -> "||"
        | CMod -> "%"

let rec string_of_ctype t = match t with
          CVoid -> "void"
        | CInt -> "int"
        | CFloat -> "double"
        | CString -> "string"
        | CDiGrNode -> "DiGrNode"
        | CDiGrEdge -> "DiGrEdge"
        | CTypePointer p -> string_of_ctype p ^ "*"
        | CVector v -> "deque<" ^ string_of_ctype v ^ ">"


let string_type_of_formal (s : csigvar) = match s with
          CSigVar (t, n) -> (string_of_ctype t)
        |  CSigVect (t, n) -> "vector<" ^ (string_of_ctype t) ^ ">"
        |  CSigPtr (t, n) -> (string_of_ctype t) ^ "*"
        |  CSigRef (t, n) -> "&" ^ (string_of_ctype t)
        |  CSigArr (t, n) -> string_of_ctype t
        | CFuncFormal (t, n, a) -> string_of_ctype t

let rec string_of_cvar v = match v with
          Cvar s-> s
        | CArrayStat (n, i) -> " " ^ n ^ "[" ^ string_of_int i ^ "]"
        | CArrayDyn (n, i) -> " " ^ n ^ "[" ^ (string_of_cvar i) ^ "]"
        | CPointer s -> " *" ^ s



let rec string_of_csigvar ( s : csigvar) = match s with
          CSigVar (t, s) -> (string_of_ctype t) ^ " " ^ s
        | CSigPtr (t ,s) -> (string_of_ctype t) ^ " *" ^ s
        | CSigArr (t, a) -> (string_of_ctype t) ^ " " ^
        (
```

```
                match a with
                Cvar s -> s
                | CArrayStat (s, i) -> s ^ "[" ^ string_of_int i ^ "]"
                | CArrayDyn (s, c) -> s ^ "[" ^ string_of_cvar c ^ "]"
                | CPointer (n) -> (print_endline ("ERROR: tried to make
                   pointer to indexed array" ^ n); "BAD")
        )
        | CSigRef (t, s) -> (string_of_ctype t) ^ " &" ^ s
        | CSigVect (t, s) -> "vector<" ^ (string_of_ctype t) ^ "> " ^ s
        | CFuncFormal (t, s, a) -> (string_of_ctype t) ^ " (*" ^ s ^")(" ^
                (
                if ((List.length a) > 0) then
                    (List.fold_left (fun b c -> b ^ ", " ^ (
                       string_of_ctype c)) (string_of_ctype (List.hd a)) (
                       List.tl a) )
                else ""
                ) ^

                ")"

let array_name v = match v with
          CArrayStat (n, i) -> n
        | CArrayDyn (n, i) -> n
        | Cvar (n) -> ( print_endline ("ERROR! " ^ n ^ " is a variable and
            not an array!"); "BAD")
        | CPointer (n) -> ( print_endline ("ERROR! " ^ n ^ " is a pointer
           and not an array!"); "BAD")

let array_size v = match v with
          CArrayStat (n, i) -> i
        | CArrayDyn (n, i) -> int_of_string (string_of_cvar i)
        | Cvar (n) -> ( print_endline ("ERROR! " ^ n ^ " is a variable and
            not an array!"); -1)
        | CPointer (n) -> ( print_endline ("ERROR! " ^ n ^ " is a pointer
           and not an array!"); -1)

let signature_of_fdecl f =
        (string_of_ctype f.creturntype) ^ " " ^ f.cfname ^ "(" ^
                (
                if ((List.length f.cformals) > 0)
                then
                    (List.fold_right    (fun a b -> b ^ ", " ^ (
                       string_type_of_formal a))
                                        (List.tl f.cformals)
                                        (string_type_of_formal (List.hd f.
                                           cformals))
                        )
                else ""
                )
                ^");\n\n"

```

```
let rec string_of_cexpr  e = match e with
          CLiteral_Int i -> " "^string_of_int i^" "
        | CLiteral_Float f-> " "^string_of_float f^ " "
        | CLiteral_String s -> s
        | CActuals a ->
                "{" ^ (
                        List.fold_left  (fun b a -> (string_of_cexpr a) ^
                           ", " ^ b)
                                        (string_of_cexpr (List.hd a))
                                        (List.tl a)
                ) ^ "}"
        | CId s -> (string_of_cvar s)
        | CBinop (e1, o, e2) -> (string_of_cexpr e1) ^ " " ^ (
           string_of_cop o) ^ " " ^ (string_of_cexpr e2)
        | CCall (s, l) ->
                s ^ "(" ^
                (
                if ((List.length l) > 0)
                then
                        (List.fold_left (fun b a -> b ^ ", " ^ (
                           string_of_cexpr a))
                                        (string_of_cexpr (List.hd l))
                                        (List.tl l))
                else ""
                ) ^ ")"
        | CCallNew (s, l) -> "new " ^ string_of_cexpr (CCall (s,l))
        | CObjCall (os, s, l) -> (string_of_cvar os) ^ "->" ^
           string_of_cexpr (CCall(s,l))
        | CNoexpr -> "/* caught a NOEXPR! */"
        | CIdAddr s -> " &" ^ s ^ " "

let rec init_nodes name size =
        if size == 0 then name ^ "[" ^ (string_of_int size) ^ "] = new
           DiGrNode();\n"
        else name ^ "[" ^ (string_of_int size) ^ "] = new DiGrNode();\n" ^
            (init_nodes name (size - 1))

let rec string_of_cstmt s = match s with
        CTryCatchBlock stmtlist -> "try{\n" ^
        (
                if List.length stmtlist > 0 then
                   (List.fold_left      (fun b a -> b ^ (string_of_cstmt a
                      ))
                                        (string_of_cstmt (List.hd stmtlist
                                           ))
                                        (List.tl stmtlist) )
                else ""
        ) ^ "}\ncatch(const char *e) {\nstd::cout << e << std::endl;\n}\n"

        | CPrint l -> "std::cout << " ^ (List.fold_right (fun a b -> (
           string_of_cexpr a) ^ " << " ^ b ) l "std::endl;\n")
```

```
        | CExpr e -> (string_of_cexpr e) ^ ";\n"
        | CDeclare s -> (string_of_csigvar s) ^ ";\n"
        | CAssignRule (s, e) -> s ^ " = " ^ (string_of_cexpr e) ^ ";\n"
        | CAssign (s, e) -> (string_of_cvar s) ^ "=" ^ (string_of_cexpr e) ^ ";\n"
        | CWhile (e, s) ->
                "while(" ^ (string_of_cexpr e) ^ ")\n{" ^
                (
                if List.length s > 0
                then
                   (List.fold_left (fun b a -> b ^ (string_of_cstmt a))
                                        (string_of_cstmt (List.hd s))
                                        (List.tl s) )
                else ""
                ) ^ "}"
                | CIf (e,s1,s2 ) -> "if("  ^ (string_of_cexpr e) ^ ")\n{" ^
                (
                        if List.length s1 > 0
                        then
                           (List.fold_left      (fun b a -> b ^ ( string_of_cstmt a))
                                                (string_of_cstmt (List.hd s1))
                                                (List.tl s1) )
                        else ""
                )
                 ^ "}\n"
                        ^ "else{" ^
                (
                if List.length s2 > 0
                then
                        (List.fold_right        (fun a b -> ( string_of_cstmt a) ^ b)
                                                (List.tl s2)
                                                (string_of_cstmt (List.hd s2)) )
                else ""
                )
                 ^"}\n"
                | CBlock s ->
                    ( if List.length s > 0 then
                      (List.fold_left (fun b a -> b ^ (string_of_cstmt a)) (string_of_cstmt (List.hd s)) (List.tl s) )
                else ""
                   )
        | CReturn e -> "return " ^ (string_of_cexpr e) ^";\n"
        | CDeclareAssign (s,e) -> (string_of_csigvar s) ^ " = " ^ ( string_of_cexpr e) ^ ";\n"
        | CDoWhile (e, s) -> "do {\n" ^
                (
```

```
                if List.length s > 0
                then
                      (List.fold_left   (fun b a -> b ^ (string_of_cstmt a))
                                        (string_of_cstmt (List.hd s))
                                        (List.tl s))
                else ""
                ) ^ "} while (" ^ (string_of_cexpr e) ^ ");\n"



let string_of_c_fdecl cf =
        (string_of_ctype cf.creturntype) ^ " " ^ cf.cfname ^ "(" ^
        (
                if ((List.length cf.cformals) > 0)
                then
                    (List.fold_left     (fun b a -> b ^ ", " ^ (string_of_csigvar a))
                                        (string_of_csigvar (List.hd cf.cformals))
                                        (List.tl cf.cformals) )
                else ""
        )
        ^") {\n" ^
           (List.fold_right     (fun a b -> (string_of_cstmt a) ^ b)
                                cf.cbody
                                "" )
        ^ "\n}\n\n"

let string_c_includes = "#include \"digr.h\"\n#include <iostream>\n"


let string_of_c_program ( p : cprogram) =
        match p with
        (cfdecllist) ->
                "/* actual definition of C++ functions */\n" ^
                (List.fold_right (fun a b -> (string_of_c_fdecl a) ^ b) cfdecllist "" )

let _ =
        print_endline "/*begin formal AST verification";
        let lexbuf = Lexing.from_channel stdin in
        let program = Parser.program Scanner.token lexbuf in
        let error = Interpret.check_ast program
        in
        let c_program =
        (

                print_endline "begin translation to CAST";
                Translate.cast_from_ast program
        )
```

```
        in
        if error then
        (
                print_endline "======================";
                print_endline "FAILED STATIC SEMANTIC CHECK";
                print_endline "NO TARGET LANGUAGE OUTPUT */"

        )
                else (
                print_endline "passed static semantic checking, begin code generation";
                print_endline "====================== */";
                print_endline (string_c_includes ^ (string_of_c_program c_program))
        )
```

8.8 digr.h

```
#include <deque>
#include <vector>
#include <string>
#include <algorithm>
#include <exception>
#include <map>

// dvp: boo, is there no clever way to implement unions with strings
// as members?

typedef int AttributeType;

using std::string;
using std::vector;
using std::deque;
using std::sort;

class DiGrEdge;
class DiGrNode;

class DiGrNode {

 private:
  // vector to store pointers to associated edges in
  vector<DiGrEdge*> _inEdges;
  vector<DiGrEdge*> _outEdges;
  std::map<std::string, AttributeType> _attributes;

public:
  DiGrNode();

  void setAttribute(string attrName, AttributeType attrValue);
  AttributeType getAttribute(string attrName);

  void addInEdge(DiGrEdge *e);
  void addOutEdge(DiGrEdge *e);

  DiGrEdge* getInEdge(int index);
  DiGrEdge* getOutEdge(int index);

  DiGrNode* getParent(int index);
  DiGrNode* getChild(int index);

  int InEdges();
  int OutEdges();
  // int Parents();
  // int Children();

};
```

```

class DiGrEdge {
 private:
  // pointers to nodes
  DiGrNode* _inNode;
  DiGrNode* _outNode;
  // hashmap of attributes
  std::map<std::string, AttributeType> _attributes;

public:
  DiGrEdge();
  DiGrEdge(DiGrNode* fromNode, DiGrNode* toNode, bool Uedge = false);
  // attribute setters and accessors
  DiGrNode* inNode();
  DiGrNode* outNode();

  void setAttribute(string attrName, AttributeType attrValue);
  AttributeType getAttribute(string attrName);

};

enum AddByObject {ADDBY_NODE, ADDBY_EDGE};
enum AddByOrder {ASCENDING, DESCENDING};
enum AddByWhere {BACK, FRONT};

void DiGrAddBy(DiGrNode *current, deque<DiGrNode*> *queue, AddByWhere addWhere, AddByObject addObj, string property, AddByOrder order, int max);
```

8.9 digr.cpp

```cpp
#include "digr.h"

///////////////////////////////////////////////////////////

DiGrNode::DiGrNode() {

}

void DiGrNode::addInEdge(DiGrEdge *e) {
  _inEdges.push_back(e);
}

void DiGrNode::addOutEdge(DiGrEdge *e) {
  _outEdges.push_back(e);
}

int DiGrNode::InEdges() {
  return _inEdges.size();
}

int DiGrNode::OutEdges() {
  return _outEdges.size();
}

DiGrEdge* DiGrNode::getInEdge(int index) {
  if (index < 0 || index >= _inEdges.size()) {throw "DiGr run-time error: attempting to index an incoming edge which doesn't exist!"; }
  return _inEdges[index];
}

DiGrEdge* DiGrNode::getOutEdge(int index) {
  if (index < 0 || index >= _outEdges.size()) {throw "DiGr run-time error: attempting to index an outgoing edge which doesn't exist!"; }
  return _outEdges[index];
}

DiGrNode* DiGrNode::getParent(int index) {
  if (index < 0 || index >= _inEdges.size()) {throw "DiGr run-time error: attempting to index a parent node which doesn't exist!"; }
  return (getInEdge(index))->outNode();
}

DiGrNode* DiGrNode::getChild(int index) {
  if (index < 0 || index >= _outEdges.size()) {throw "DiGr run-time error: attempting to index a child node which doesn't exist!"; }
  return (getOutEdge(index))->inNode();
}

void DiGrNode::setAttribute(string attrName, AttributeType attrValue) {
```

```

  if (_attributes.count(attrName) == 0) {
    // create this attribute for the first time
    _attributes.insert(std::pair<std::string,AttributeType>(attrName, attrValue));

  } else {
    // find the attribute and modify it
    _attributes[attrName] = attrValue;
  }
}

AttributeType DiGrNode::getAttribute(string attrName) {

  if (_attributes.count(attrName) == 0) {
    // create this attribute for the first time
    _attributes.insert(std::pair<std::string,AttributeType>(attrName,(AttributeType) 0));
    return 0;

  } else {
    // find the attribute and return it
    return _attributes[attrName];
  }

}

//////////////////////////////////////////////////////////////

DiGrEdge::DiGrEdge(DiGrNode *fromNode, DiGrNode *toNode, bool Uedge) {
  (*fromNode).addOutEdge(this);
  (*toNode).addInEdge(this);
  _inNode = toNode;
  _outNode = fromNode;

  if (Uedge) {
    new DiGrEdge(toNode, fromNode, false);
  }

}

DiGrEdge::DiGrEdge() {
  _inNode = new DiGrNode();
  _outNode = new DiGrNode();
}

DiGrNode* DiGrEdge::inNode() {
  return _inNode;
}

DiGrNode* DiGrEdge::outNode() {
```

```
  return _outNode;
}

void DiGrEdge::setAttribute(string attrName, AttributeType attrValue) {

  if (_attributes.count(attrName) == 0) {
    // create this attribute for the first time
    _attributes.insert(std::pair<std::string,AttributeType>(attrName,
        attrValue));

  } else {
    // find the attribute and modify it
    _attributes[attrName] = attrValue;
  }
}

AttributeType DiGrEdge::getAttribute(string attrName) {

  if (_attributes.count(attrName) == 0) {
    // create this attribute for the first time
    _attributes.insert(std::pair<std::string,AttributeType>(attrName,(
        AttributeType) 0));
    return 0;

  } else {
    // find the attribute and return it
    return _attributes[attrName];
  }

}

string globalProperty;

bool edgeSorterDescending (DiGrEdge *e1, DiGrEdge *e2) {
  return e1->getAttribute(globalProperty) < e2->getAttribute(
      globalProperty);
}
bool edgeSorterAscending (DiGrEdge *e1, DiGrEdge *e2) {
  return e1->getAttribute(globalProperty) > e2->getAttribute(
      globalProperty);
}
bool nodeSorterDescending (DiGrNode *n1, DiGrNode *n2) {
  return n1->getAttribute(globalProperty) < n2->getAttribute(
      globalProperty);
}
bool nodeSorterAscending (DiGrNode *n1, DiGrNode *n2) {
  return n1->getAttribute(globalProperty) > n2->getAttribute(
      globalProperty);
}


```

```
void DiGrAddBy(DiGrNode *current, deque<DiGrNode*> *queue, AddByWhere addWhere, AddByObject addObj, string property, AddByOrder order, int max) {


  // set the global property & check how many to return
  globalProperty = property;
  if (max > current->OutEdges()) max = current->OutEdges();
  if (max == 0) max = current->OutEdges();

  // push back edges and nodes
  vector<DiGrEdge*> allEdges;
  for (int e = 0; e < current->OutEdges(); e++) {
    allEdges.push_back(current->getOutEdge(e));
  }

  vector<DiGrNode*> allNodes;
  for (int n = 0; n < current->OutEdges(); n++) {
    allNodes.push_back(current->getChild(n));
  }

  // sort pointers as appropriate
  if (addObj == ADDBY_NODE) {
    if (order == DESCENDING) sort(allNodes.begin(), allNodes.end(), nodeSorterDescending);
    else sort(allNodes.begin(), allNodes.end(), nodeSorterAscending);
    for (int n = current->OutEdges() - max; n < current->OutEdges(); n++) {
      if (addWhere == FRONT) queue->push_front(allNodes[n]);
      if (addWhere == BACK) queue->push_back(allNodes[n]);
    }
  }

  if (addObj == ADDBY_EDGE) {
    if (order == DESCENDING) sort(allEdges.begin(), allEdges.end(), edgeSorterDescending);
    else sort(allEdges.begin(), allEdges.end(), edgeSorterAscending);
    for (int n = current->OutEdges() - max; n < current->OutEdges(); n++) {
      if (addWhere == FRONT) queue->push_front(allEdges[n]->inNode());
      if (addWhere == BACK) queue->push_back(allEdges[n]->inNode());
    }
  }

  // TODO: implement reverse sort with reverse iterator


}
```