Final Project Report- GAL

Anton: ain2108, Donovan: dc3095, Macrina: mml2204, Andrew: af2849

August 12, 2016

Contents

1 Introduction

Graph Application Language or GAL is designed with the end goal in mind that graph operations and manipulations can be simplified. Many real world problems can be modeled using graphs and algorithms can be implemented using GAL to solve them. Currently available mainstream languages such as C, java, and python do not provide sufficient graph orientated packages to facilitate the creation of graphs and the implementation of graph algorithms.

With the end goal of creating a full-fetched language that is centered around providing the user with numerous graph operations and built in functions that will facilitate graph programming, GAL will contain special data structures and semantics to allow the user to easily interact with graphs and special data structures with syntax that is similar to the familiar C programming language. The language will have a compiler written in OCAML and compiles down to LLVM.

1.1 Summary

GAL simplifies many graph operations such as adding a node or an edge to an existing graph. Graph creation has also been made more convenient by shrinking the number of lines of code required to create one. Under the hood, GAL represents graphs as a list of edges. This means that with a simple line of code, users can create a complex that will take numerous lines of code to achieve in other generic programming languages. This is in hopes that with the removal of the complexity of representing graphs in code, the user can focus more on building and testing graph algorithms.

1.2 Key Features of GAL

- **Graph Declaration**: Graph declarations are basically placing edges or nodes into a list structure and that basically defines the entire topology of the graph.
- **User Defined Functions**: Much like any other generic programming language, GAL offers users the ability to create their own functions to facilitate algorithm implementation.
- **Control Flow**: GAL also has the complete suite of control flow operations such as while and for loops.

2 Setup

The following set of instructions set up the GAL compiler.

2.1 Installation

1. Unpack the GAL compiler tarbell
2. Run the make file by entering: `make`

   This creates the `gal.native` file which allows `.gal` files to be compiled.

2.2 Running the Compiler

This requires 2 steps

1. Writing a `.gal` source file and storing that file in the same directory as the `gal.native` as mentioned above in the installation.

2. Run the following command in the console:

```
>./gal.native < test.gal > test.ll
```

3. Finally, create the executable file:

```
>lli test.ll
```

3 Writing the First GAL Program

STEP 1: Creating the `.gal` source code:
Create a new file called firstGAL.gal in the directory of desire and open it with the preferred text editor.

STEP 2: Defining Functions
Functions that are being called in the main program have to be defined here.

```
edge build_edge(string src, int w, string dst){

    edge e1;
    e1 = |src, w, dest|;
    return e1;
}
```

STEP 3: Writing the `main` Function in the Program
GAL requires a main function of the form.

```
int main(){
}
```

STEP 4: Declaring and Assigning Variables
Variables must be declared first before assignment can take place.

```
/*DECLARATION OF VARIABLES*/
string src_e1;
int weight_e1;
string dst_e1;

/*ASSIGNMENT OF VARIABLES*/
src_e1 = "A";
weight_e1 = 2;
dst_e1 = "B";
```

STEP 5: Declaring and Assigning an Edge

```
edge e2;

e2 = |"A", 10, "C"|;
```

STEP 6: Declaring and Assigning a Graph
Remember that a graph in GAL is implemented as a list of edges.

```
elist l1;

l1 = [e2];
```

STEP 7: Function Calls

```
edge e1;

e1 = build_edge(src_e1,weight_e1,dst_e1);
```

STEP 8: Graph Operator adding an Edge to a Graph

```
l1 = eadd(e1,l1);
```

STEP 9: Printing

```
print_str("This is a test print of a string");
print_endline();
print_str("This now prints an integer");
print_endline();
print_int(weight_e1);
```

STEP 9: Final `firstGAL.gal` source code

The final code when put together should look like this

```
edge build_edge(string src, int w, string dst){
  edge e1;
    e1 = |src, w, dest|;
    return e1;
}

int main(){

  string src_e1;
  int weight_e1;
  string dst_e1;

  src_e1 = "A";
  weight_e1 = 2;
  dst_e1 = "B";

  edge e2;
  e2 = |"A", 10, "C"|;

  elist l1;
  l1 = [e2];

  edge e1;
  e1 = build_edge(src_e1,weight_e1,dst_e1);

  l1 = eadd(e1,l1);

  print_str("This is a test print of a string");
  print_endline();
  print_str("This now prints an integer");
  print_endline();
  print_int(weight_e1);

}
```

4 Language Reference Manual

4.1 Lexical Conventions

Six type of tokens exist in GAL: identifiers, keywords, constants, strings, expression operators and other forms of separators. Common keystrokes such as blanks, tabs and newlines are ignored and used to separate tokens. At least one of these common keystrokes are required to separate adjacent tokens.

4.1.1 Comments

The characters `/*`introduce a comment which terminates with the characters `*/`. There are no single line comments (such as // in C).

4.1.2 Code Line Termination

Lines of code in statement blocks or expressions must be terminated with the semicolon `;`

4.1.3 Identifiers (Names)

An identifier is a sequence of letters and digits; the first character must be alphabetic. The underscore counts as alphabetic. Upper and lower case letters are considered different. Identifiers used in function names may not be used in other function names or as variable names except in the following case

4.1.4 Keywords

The following identifiers are reserved for use as keywords and may not be used otherwise:

1. int
2. elist
3. slist
4. ilist
5. nlist
6. string
7. while
8. if
9. else
10. for
11. return
12. node
13. edge

4.1.5 String

A string is a sequence of ASCII characters surrounded by double quotes i.e. one set of double quotes `"` begins the string and another set `"` ends the string. For example, `"GAL"` represents a string. Individual characters of the string cannot be accessed. There are no escape characters within strings.

4.1.6 Constants

2 distinct constant types are present in GAL:

1. Integer Constants: This is a sequence of decimal digits, the limit of which corresponds to the memory space of the machine it is running on

2. String Constants: This is of type string, strings can be both an identifier or a constant.

4.2 Scoping and Derived Data Types

All identifiers in GAL are local to the function in which the identifier is defined in. 2 fundamental types exist in GAL- integers and strings, and GAL defines several derived data types which comprise the 2 fundamental types which are shown below. Both derived and fundamental types are referred to as "type" in the rest of the manual.

1. List: They comprise several items of other types such as integers, strings, edges and nodes which are list of list of edges. These explicit types of lists are implemented in GAL as a prefix to the word list. For example, `ilist` is a list of integers, `elist` is a list of edges and `slist` is a list of strings. However, a list cannot contain functions. All objects in a list must be of the same type. For example, a list can contain all edges. Graphs in GAL are essentially a list of edges. Lists which are created but not yet defined have no values because they have not been initialised, errors occur if undefined lists are referenced.

2. Node: It encodes all the information present in a graph vertex. It contains the string name of the source vertex and the set of all vertices and their corresponding weights of those edges that the source is connected to.

3. Edge: It contains three elements namely two strings corresponding to the two vertices the edge connects and an integer representing its weight.

4. Function: It takes one or more input objects of node, edge, list, integer or string type and returns a single object of a given type, namely, node, edge, list or integer. Functions cannot return other functions.

4.3 Expressions denoted by *expr*

Expressions described below are listed in decreasing level of precedence. The expressions in the same subsection have the same level of precedence. Operators that can act on the expressions are also described.

4.3.1 Primary Expressions

Primary expressions are expressions that include identifiers, strings, constants, nodes, edges, parenthesized expressions of any type and subscripts. Primary expression involving subscripts are left associative.

4.3.2 Identifiers and Constants

Identifiers and constants are both primary expressions of the previously defined form. These are denoted in the manual as *identifier* and *constant*

4.3.3 Node denoted by *node*

A node is a primary expression of the form

```
|string:integer,string,integer,string)(integer,string)
.....(integer,string)|
```

The *string* and *integer* may be constants and/or identifiers of string and integer types respectively. The first *string* denotes the vertex represented by a source node and the (integer, string) pair denote a weighted edged that the source node is connected to. This syntax facilitates the creation of graphs with a single source node and multiple connections, for example the code shown below can be used to generate the graph shown in Figure 1.

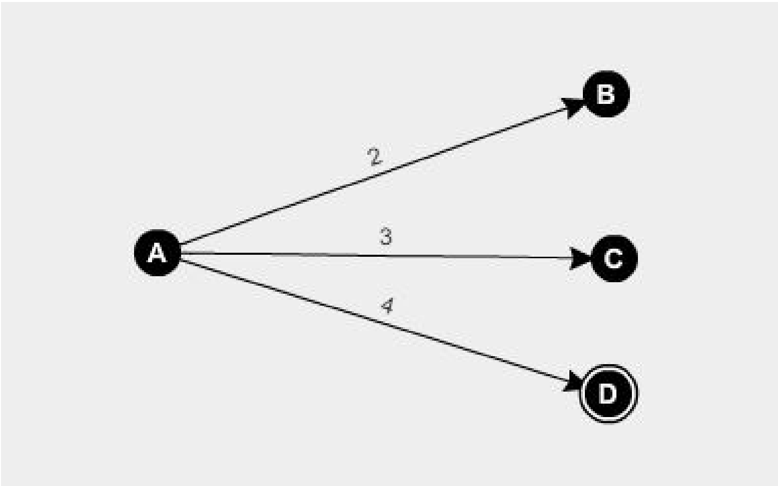

Figure 1: Graph generated

```
node a = |"A":2,"B",3,"C",4,"D"|
```

This is synonymous with creating an `elist`.

4.3.4 Edge denoted by *edge*

An edge is a primary expression of the form

```
    |string,integer,string|
```

The *string* and *integer* may be constants and/or identifiers of string and integer types respectively. The first *string* denotes the source vertex followed by the *integer* weight and the destination vertex representing an edge in the graph. An expression evaluating to an integer is not permitted in place of *integer* in equation (2).

Thus when our language encounters a |, it checks for the subsequent pattern and accordingly decides if an edge or node is being defined.

4.3.5 Parenthesized expressions

Any expression in GAL can be parenthesized. The format is

```
( expr )
```

Parenthesis cannot be inserted or removed from within the node or edge definitions.

4.3.6 Subscripts

The form is

```
identifier[constant]
```

The *identifier* must be of type list and the *constant* must be an integer greater than or equal to 0. Subscript expressions output the *constant*th element of the list. Lists are indexed from 0, an error will occur if the element that is being accessed is greater than the length of the list.

4.3.7 Function calls

Functions previously defined may be called. This takes the form of:

```
identifier(expr_opt)
```

The `expr_opt` denotes a comma separated set of inputs to the function and may be absent if the function is defined as containing no inputs. Any expression is acceptable as long as it evaluates to the required type as mentioned in the function definition. Thus nested function calls can exist. All inputs of the functions must be explicitly listed in the function call.

Thus an identifier followed by open parenthesis matching the requirements above is a function. If it is previously undefined or does not meet the above requirements an error is returned. If the defined function has no input arguments, the called function must also not have any. All function parameters are passed by value i.e. changes to the input parameters within the function will not be reflected in the calling function unless the parameter is returned.

4.3.8 Unary Operators

Our language has two unary operators - unary minus and logical negation. They are right associative.

4.3.9 Unary minus

The form is

```
-expr
```

The primary expression *expr* must evaluate into an Integer type.

4.3.10 Logical negation

This expression is of the form

```
!(expr)
```

The primary expression contained within the parenthesis has to be an explicit comparison. For example:

```
!(2 == 3)
```

In the above code, this would evaluate into a 1. GAL does not have boolean types. The negation operator returns an output that is opposite to that within the parenthesized *expr*.

4.3.11 Multiplicative Binary Operators

The operators of this type are * and / They are left associative.

4.3.12 Binary Multiplication

```
    expr * expr
```

Both the expressions in the above must evaluate to an integer type.

4.3.13 Binary Division

```
    expr / expr
```

Both the expressions in the above must evaluate to an integer type.

4.3.14 Additive Binary Operators

The operators of this type are + and - and are left associative.

Addition:

```
expression+expression
```

Subtraction:

```
expression-expression
```

All expressions must evaluate to integers for the operations to be valid.

4.3.15 Binary Operators

These are left associative. Each expression of this type evaluates to integer 1 if true and integer 0 is false. The expressions on both sides of the operator must evaluate to integers. The operators are of the following types:

```
expr < expr
```

```
expr > expr
```

```
expr <= expr
```

```
expr >= expr
```

```
expr == expr
```

The operators `<`(less than), `>`(greater than), `<=`(less than equal) and `>=`(greater than equal) all return a 0 if the comparison is false and a 1 if the comparison is true. The same is true for the `==` operator.

4.3.16 Graph Equality Operator

This is left associative. Each expression of this type evaluates to integer 1 if true and integer 0 is false. Its form is:

```
expr ==.expr
```

Equality of the graph is when every edge in the graph is identical. Similar to a binary equality operator, if the result of the comparison is false, the corresponding output is 0, the converse is true if the comparison is true.

4.3.17 AND Operator

It is of the form

```
expr && expr
```

It is valid only if the left and right side expressions both evaluate to integers. First the left hand expression is evaluated. If it returns, a non-zero integer, then the right side is evaluated. If that too returns a non-zero integer the AND operator expression evaluates to integer 1. If the left side expression evaluates to integer 0, the right side expression is not evaluated and the AND operator expression evaluates to 0.

4.3.18 OR Operator

It is of the form

```
expr || expr
```

It is valid only if the left and right side expressions contain an explicit comparison. For example:

```
(2==3) || (4==4)
```

This evaluates into 1 since 4 is equal to 4.

4.3.19 Assignment Operator

This is right associative and is of the form

```
exprA = exprB
```

`exprA` must an identifier, a subscript expression, a parenthesized identifier or a parenthesized subscript expression. *exprB* may be an expression of any type. Both sides of the assignment must have be evaluated to identical types for the assignment to be valid.

4.4 Declarations

Only variables need to be declared at the top of every function, including the main function.

4.4.1 Variable Declaration

All variables used in a function must be declared at the start of the function. They may be (re)assigned at any point within the function in which they are declared in which the variable takes the value of the new assignment. The scope of the variable is limited to the function in which it is declared. Variables cannot be declared or defined outside functions. variables can be of type integer, string, list, node or edge. The type of every identifier within a function does not change throughout the function. If the contents of any declared variable are printed before definition, a random value is printed.

- Variables of type integer, string, node, list and edge are declared as follows:

  ```
  type identifier;
  ```

  `type` assigns a type from among integer, string, node ,list or edge to the identifier.

4.5 Definitions

These are of two types:

4.5.1 Function Definition

This takes the following form:

```
type-specifier identifier(type1 input1, type2 input2 ... typen
    inputn){
/*first declare the variables and initialise them*/

/*set of simple and compound statements*/

/*return (return_value);*/
}
```

The `def` keyword is used to define the function. The return statement can occur anywhere within the function provided it is the last statement within the function according to its control flow. Statements occurring after return in the control flow will cause errors. Functions have to be defined at the beginning of the program to be successfully called in the `main()` program

4.5.2 Variable Definition

A variable definition is simply an assignment as shown earlier in section 4.10.

4.6 Statements denoted by *statement*

Execution of statements are carried out in order unless specified otherwise. There are several types of statements:

4.6.1 Expression Statement

Most statements are expression statements which have the form

```
    expression;
```

Usually expression statements are assignment or function calls.

4.6.2 Compound statement

Several statements *statement* of any statement type may be enclosed in a block beginning and ending with curly braces as follows:

```
{statement-list};
```

The entire block (along with the curly braces) is called a compound statement. Statement-list can comprise a single statement (including the null statement) or a set of statements of any statement type. Thus compound statements may be nested.

4.6.3 Conditional Statement

The form is:

```
if (expr) {
  statement-list
}
else{
  statement-list
}
```

This entire form is called a conditional statement. Every `if` must be followed by an expression and then a compound statement. The `else` keyword must be present or an error will occur. The `expression` must be an explicitly comparison and code like `if(1)` will break, it has to be written as `if(1==1)`. If the comparison evaluates to 1, the compound statement immediately after if is evaluated and the block following else is not executed and the flow proceeds to the next statement(following the conditional statement). If the comparison following *if* evaluates to 0, the else block is executed. Thus conditional statements may be nested.

4.6.4 For Loop Statement

The statement has the form:

```
for(int expr1;int expr2;int expr3){
  statement-list
}
```

None of the expression statements can be omitted. Identical to C, the first expression specifies the initialization of the loop, the second specifies a test made before each iteration such that the loop is exited when the expression evaluates to 0; the third expression specifies an increment or decrement which is performed after each iteration.

4.6.5 While Loop Statement

This conditional loop has the form:

```
while(expr){
      statement-list
      }
```

The statement list executes for as long as the `expr` within the parenthesis evaluates to a non-zero integer, this `expr` has to be an explicit comparison. The `expr` is evaluated before the execution of the `statement-list`.

4.6.6 Return Statement

The *return* statement is a function return to the caller. Every function must have a return value. This return value may or may not be collected by the calling function depending on the statement containing the function call. The format is:

```
    return (expression);
```

In the above, *expression* must evaluate to the same type as that in the function definition it is present in.

4.6.7 Null Statement

This has the form

```
    /*nothing*/;
```

4.7 Built-In Functions

GAL has six built in functions:

4.8 Printing of Integers, Strings, Newlines and String Comparisons

Some inbuilt functions for printing integers, strings and entering newlines onto the output console.

4.8.1 `print_int`

```
    print_int(expr);
```

`print_int` takes in an `expr` that must evaluate into an integer.

4.8.2 `print_str`

```
    print_str("string");
```

`print_str` prints anything that is enclosed within `"` as a string.

4.8.3 `print_endline`

```
    print_endline();
```

This prints a newline onto the console.

4.8.4 `streq`

```
    streq(string1,string2);
```

This built in function compares on the first character of each string and returns 0 if they are equal and -1 if they are not equal.

4.9 Built-ins for Operations on Lists

Figure 2 shows the way in which the built in functions for list operations work in GAL. As mentioned above in how lists are being implemented in GAL, each

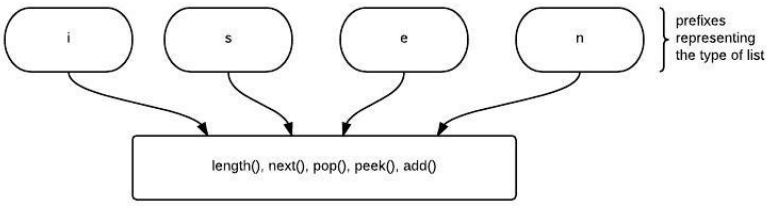

Figure 2: Flowchart Showing Structure of Built-In

corresponding built in function has a prefix to it for each corresponding list type that is it working on. The examples shown below are for integer list, `ilist` but work in exactly the same way for all other list types.

4.9.1 `length()`

```
identifierA = ilength(identifierB);
```

`identifierA` is of type integer, `identifierB` is of type `ilist` and the function operation `ilength` on `identifierB` will result in the length of the integer list.

4.9.2 `next()`

```
identifierA = inext(identifierB);
```

The `next()` function returns the list with the head of the list being the next element in the list. In this case, `identifierA` now contains a list with the head being the next element on the list contained in `identifierB`. Cycling through can list can be done with the following code:

```
identifierB = inext(identifierB);
```

4.9.3 `pop()`

```
identifierA = ipop(identifierB);
```

The `pop()` function returns a new list stored in `identifierA` without the first element that is present in `identifierB`. `pop()` destroys the head that is being popped.

4.9.4 `peek()`

```
identifierA = ipeek(identifierB);
```

The `peek()` function returns the first element at the head of the list.

4.9.5 `add()`

```
identifierA = iadd(2,identifierA);
```

The `add()` function takes a list of its corresponding type and an element of its corresponding type and adds it to the head of the list. This will be the new head of the list. For example, taking the above graph created in Figure 1:

```
a = eadd([|"B",5,"E"|,a);
```

The above code listing will create the graph as shown in Figure 3.

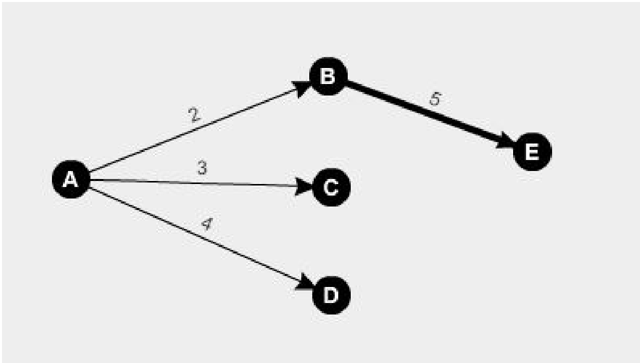

Figure 3: New Graph with added edge

4.9.6 Finding source vertex source()

This computes the source vertex in an edge. Its format is

```
identifier = source(expr);
```

Where `identifier` is an identifier of type string and `expr` is an identifier or expression of type edge.

4.9.7 Finding destination vertex dest()

This computes the destination vertex in an edge. Its format is

```
identifier = dest(expr);
```

Where `identifier` is an identifier of type string and `expr` is an identifier or expression of type edge.

4.9.8 Finding weight of an edge `weight()`

This the weight of an edge. Its format is

```
identifier = weight(expr);
```

Where `identifier` is an identifier of type int and `expr` is an identifier or expression of type edge.

No identifier can have the names of any of the above mentioned built-in functions.

4.10 Standard Library Functions

There are several printing functions and lots of basic functions on graphs which commonly occur in most applications. We have put some of these in the standard library and described them below

4.10.1 Finding node with the most number of edges

This computes the node in a list of nodes with the most number of edges. The output is a list of edges i.e. a node since a node is implemented internally as a list of edges. It is called using

```
elist_id = get_most_edges_node(nlist_id);
```

where `elist_id` is an identifier to a list of edges and `nlist_id` is an identifier/-constant/expression evaluating to a list of nodes.

4.10.2 Finding the outgoing edge with highest weight

This finds the outgoing edge with the highest weight in a node. The output is an edge and input is a node. It is called using

```
edge_id = get_heaviest_edge(node_id);
```

where `edge_id` is an edge identifier and `node_id` is an identifier/constant/ expression evaluating to a node.

4.10.3 Finding the heaviest edge in a list of nodes

This finds the edge with the highest weight in a list of nodes. The input is a list of nodes and the output is an edge. It is called using

```
edge_id = get_heavist_graph_edge(nlist_id);
```

where `edge_id` is an edge identifier and `nlist_id` is an identifier/constant/expression evaluating to a list of nodes.

4.10.4 printing text in a line

This prints the text followed by a new line character. It is called using

```
print_line(string_ip);
```

Where `string_ip` is an identifier/constant/expression evaluating to a string. It returns an integer which may/ may not be captured by the calling function.

4.10.5 printing the number of strings a list of strings

This prints the integer number of strings in a list of strings. It is called using

```
print_sl_len(slist_id);
```

where `slist_id` is a constant/ identifier/ expression evaluating to a list of strings. It returns an integer which may/ may not be captured by the calling function.

4.10.6 printing a list of strings

This prints the strings present in the input list as follows

```
->string1::string2::....::stringn
```

where `stringk for k = 1 to n` is a string as printed by `print_str` It is called using

```
print_slist(slist_id);
```

where `slist_id` is a constant/ identifier/ expression evaluating to a list of strings. It returns an integer which may/ may not be captured by the calling function.

4.10.7 printing an edge

This prints an edge in the form

```
|source,weight,dest|
```

where source and dest are constants/ identifiers/ expressions evaluating to a string and weight is the integer weight of the edge. It is called using

```
print_edge(edge_id);
```

where `edge_id` is an identifier/constant/expression evaluating to an edge. It returns an integer which may/ may not be captured by the calling function.

4.10.8 printing a list of edges

This prints a list of edges in the form

```
->edge1::edge2::......::edgen
```

where `edgek for k = 1 to n` is an edge as printed by `print_edge`. It is called using

```
print_elist(elist_id);
```

where `elist_id` is an identifier/constant/expression evaluating to a list of edges. It returns an integer which may/ may not be captured by the calling function.

4.10.9 printing a list of integers

This prints the integers in the input list in the following format

```
->int1::int2::int3...intn
```

where `intk for k = 1 to n` is an integer as printed by `print_int` It is called using

```
print_ilist(ilist_id);
```

where `ilist_id` is an identifier/constant/expression evaluating to a list of integers. It returns an integer which may/ may not be captured by the calling function.

4.10.10 printing a list of nodes

This prints the nodes in the list in the following format

```
->nlist1::nlist2::...::nlistn
```

where `nlistk for k = 1 to n` is a node i.e. a list of edges as printed by `print_elist`. This is also the reason why we don't require a `print_node` function. It is called using

```
print_nlist(nlist_id);
```

where `nlist_id` is an identifier/constant/expression evaluating to a list of nodes. It returns an integer which may/ may not be captured by the calling function.

4.10.11 reversing a list of integers

This reverses the input integer list. It returns an integer list with the order of elements the reverse of its input. It is called using

```
ilist_id2 = irev(ilist_id1)
```

where `ilist_id1` is an identifier/constant/expression evaluating to a list of integers. It returns an integer list which is captured in the above with the ilist identifier `ilist_id2`.

4.10.12 reversing a list of strings

This reverses the input list of strings. It returns a list of strings with the order of list elements the reverse of its input. It is called using

```
slist_id2 = srev(slist_id1)
```

where `slist_id1` is an identifier/constant/expression evaluating to a list of strings. It returns a list of strings which is captured in the above with the slist identifier `slist_id2`.

4.10.13 reversing a list of edges

This reverses the input list of edges. It returns a list of edges with the order of list elements the reverse of its input. It is called using

```
elist_id2 = erev(elist_id1)
```

where `elist_id1` is an identifier/constant/expression evaluating to a list of edges. It returns a list of edges which is captured in the above with the elist identifier `elist_id2`.

4.10.14 reversing a list of nodes

This reverses the input list of nodes. It returns a list of nodes with the order of list elements the reverse of its input. It is called using

```
nlist_id2 = nrev(nlist_id1)
```

where `nlist_id1` is an identifier/constant/expression evaluating to a list of nodes. It returns a list of nodes which is captured in the above with the nlist identifier `nlist_id2`.

Our built-ins `iadd, sadd, eadd, nadd` new the new element of the appropriate type to the start of the corresponding list. The following four functions `iadd_back, sadd_back, eadd_back, nadd_back` perform the same operations respectively but the appending is done at the end of the list instead of at the start.

4.10.15 appending to the end of a list of integers

This is called using

```
ilist_id2 = iadd_back(ilist_id1, int_id);
```

which returns an integer list with the `int_id` element appending to the end of the input integer list `ilist_id1`. This returned integer list is captured in the above with the ilist identifier `ilist_id2`. `ilist_id1` is an identifier/constant/-expression evaluating to an integer list while `int_id` is an identifier/constant/-expression evaluating to an integer.

4.10.16 appending to the end of a list of strings

This is called using

```
slist_id2 = sadd_back(slist_id1, string_id);
```

which returns a list of strings with the `string_id` element appending to the end of the input list of strings `slist_id1`. This returned list of strings is captured in the above with the slist identifier `slist_id2`. `slist_id1` is an identifier/constant/expression evaluating to a list of strings while `string_id` is an identifier/constant/expression evaluating to a string.

4.10.17 appending to the end of a list of edges

This is called using

```
elist_id2 = eadd_back(elist_id1, edge_id);
```

which returns a list of edges with the `edge_id` element appending to the end of the input list of edges `elist_id1`. This returned list of edges is captured in the above with the elist identifier `elist_id2`. `elist_id1` is an identifier/constant/-expression evaluating to a list of edges while `edge_id` is an identifier/constan-t/expression evaluating to an edge.

4.10.18 appending to the end of a list of nodes

This is called using

```
nlist_id2 = nadd_back(nlist_id1, node_id);
```

which returns a list of nodes with the `nodes_id` element appending to the end of the input list of nodes `nlist_id1`. This returned list of nodes is captured in the above with the nlist identifier `nlist_id2`. `nlist_id1` is an identifier/constant/expression evaluating to a list of nodes while `node_id` is an identifier/constant/expression evaluating to a node.

While, the above four functions appended a single element of the appropriate type to a list of the same type, the following 4 functions append the contents of the second input list to those of the first input list and return the list obtained provided the input lists are of the same type.

4.10.19 Concatenating two integer lists

This is called using

```
ilist_id3 = iconcat(ilist_id1, ilist_id2);
```

where `ilist_id1, ilist_id2` are constants/identifiers/expressions evaluating to list of integers. The function returns a list of integers as mentioned above which is captured by `ilist_id3`.

4.10.20 Concatenating two string lists

This is called using

```
slist_id3 = sconcat(slist_id1, slist_id2);
```

where `slist_id1, slist_id2` are constants/identifiers/expressions evaluating to list of strings. The function returns a list of strings as mentioned above which is captured by `slist_id3`.

4.10.21 Concatenating two edge lists

This is called using

```
elist_id3 = econcat(elist_id1, elist_id2);
```

where `elist_id1, elist_id2` are constants/identifiers/expressions evaluating to a list of edges. The function returns a list of edges as mentioned above which is captured by `elist_id3`.

4.10.22 Concatenating two node lists

This is called using

```
nlist_id3 = nconcat(nlist_id1, nlist_id2);
```

where `nlist_id1, nlist_id2` are constants/identifiers/expressions evaluating to a list of nodes. The function returns a list of nodes as mentioned above which is captured by `nlist_id3`.

5 Project Plan

5.1 Planning

The group planned for weekly meetings on Tuesday as well as Monday to discuss and consolidate ideas and progression on the project. The work was divided into who was more interested into doing what and GitHub was used as the main source of version control as well as storage for project files. Weekly meeting with the TA were really helpful towards solving issues with llvm and implementing numerous features in the language.

5.2 Communication and Synchronization

GitHub proved to be a highly valuable asset towards the development of the project. Version control and automatic merges of source files aided in the efficiency at which code was being written. Slack was also used for communication within the team. Different channels in slack were used to various purposes such as implementation and general discussion. The neat thing about slack is that GitHub can be added as a module onto the slack communication tool so that all members are aware of the commits and pushes that are made by any one on the team.

5.3 Project Development

5.4 Development Tools

The compiler was written using the following tools:

- OCaml
- OCamllex
- OCamlyacc
- llvm module in OCaml

Tests were written in our own language and uses a testall.sh shell file to test all parts of the compiler.
The language compiles down to LLVM and therefore the final step would be to use a LLVM compiler to create the executable.

5.5 Programming Style Guide

1. Comment out sections of the code explaining its function.
2. Code indentation enforced to ensure easy debugging as well as identifying nested functions
3. Long lines of code are entered on a new line with an indentation.

5.6 Project Log

This is a screen shot of the workload graph on the GitHub repository that the group uses for version control.

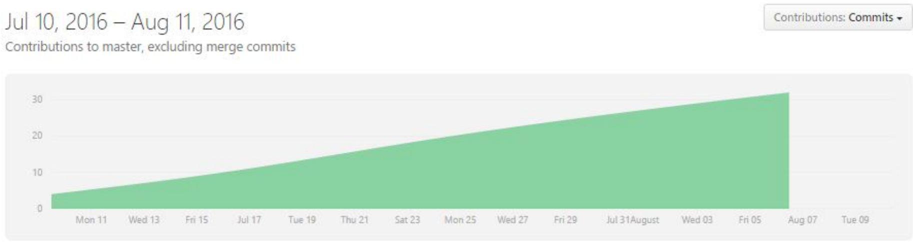

Figure 4: Graph Showing the GitHub Workload

5.7 Roles and Responsibilities

The roles and responsibilities of the group were divided before the project commenced, except for Andrew who joined the group at a later date.

Name	Role and Responsibility
Anton	Language Guru: He is the man when it comes to designing the language, makes the syntax judgments and decides what is possible to implement and what is not. Also obsessively crazy about functional programming.
Andrew	Test Suite: The devil's advocate that writes the entire test suite that tries to break the language and checks for failures. This facilitates the writing of new code and to prove that the language is working
Donovan	Manager: The slave driver that worries the most about deadlines and how the project is progressing.
Macrina	Standard Library: Obsessed with writing a multitude of functions to make every GAL programmer's life a breeze.

Table 1: An example table.

6 Architectural Design

The following sub sections show how GAL was designed using block diagrams. The scanner and parser was done by Anton and Donovan. Implementation of the semantic checker was done by Anton. Codegen was done by Donovan and Anton.

6.1 Scanning

OCamllex was used to tokenize the source code into parsable tokens that the parser will take and process. Similar to most programs, the scanner ignores comments, tabs, newlines and space characters.

Figure 5: Flowchart showing the architectural design of GAL

6.2 Parsing And the Abstract Syntax Tree(AST)

OCamlyacc was used to parse the scanned tokens that were produced by the **scanner**. This was done by parsing the tokens into an AST.

6.3 Semantic Checking

The semantic checking was written in **OCaml** and it's primary role is to check a parsed AST for various semantic errors. It checks for correct function definitions, any reference to undefined functions or uninitialised variables, scoping issues, type mismatches in expressions. This is done by using a `StringMap` to store all function names and variable assignments. This produces the semantically-checked-AST (SAST) that the code generation will take in for further processing.

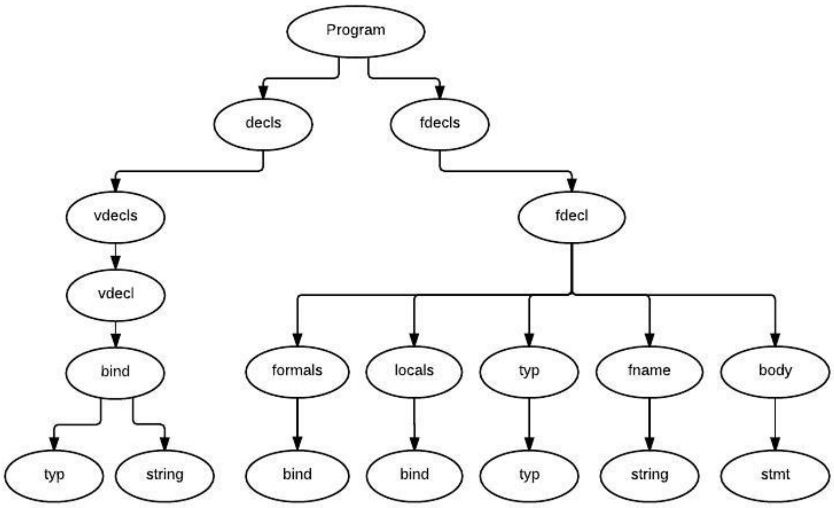

Figure 6: Flowchart Showing the AST

6.4 Code Generation

This is written in **OCaml** with the LLVM module opened. Code generation basically translates the SAST into LLVM code which the LLVM compiler will compile into machine executable code.

7 Test Plan

7.1 Test Cases

Our test cases focused on checking that the semantic checker would validate or invalidate GAL-specific syntax. Initially, this was a set of tests that would try to trick the interpreter with small errors or incorrect assignments. After the basics of the language finished and code generation began, we began using tests to check program output of simple print statements and basic algorithms in order to make sure our lists, edges and nodes were working properly.

7.2 Testing Automation

Testing was automated with a bash shell script that walks through the ”tests” directory and runs each program through an operation based on whether it starts with ”test” or ”fail.” This allowed us to keep all of the tests in the same folder as well as add a few individual tests for individual problems that would not be picked up by the script. The output of those programs is then compared with their comparable files in the test suite. These are labeled either ”testname.out” or ”testname.err” depending on whether they should have output or fail.

7.3 Test Source Files

Please see Appendix for a full list test source files.

7.4 Who Did What

Andrew Feather put together the test suite and set up the script to work through the files in the test suite. Anton Nefedenkov and Donovan Chan also added some individual tests while working on the language compiler, which Andrew would later add into the main set of tests. Macrina Lobo formalized the language with the reference manual and wrote the standard library functions.

8 Lessons Learned

8.1 Andrew Feather

I learned how languages truly come together. In addition to the theory we learned in class, there is a lot that goes into constructing a usable syntax and transferring that syntax into code. Luckily, I had some great teammates who took the reigns and got a working parse tree and code generation relatively early on. I also learned that there is still no substitute for meeting in person in regard to keeping everyone on schedule and up-to-date with a project that can move and change as quickly as this one.

Advice: Agree on a syntax early. Testing a language with changing syntax is like chasing a moving target. Whether or not you can prove it's the optimal choice, it is important to make choices on syntax and stick with them.

8.2 Donovan Chan

I learned that creating a programming language has a lot more to it than what it seen on the surface. What seems like a very simple `"hello world"` program has many things going on under the hood. I have also learn that when time is a factor, many things that seemed to sound really good on paper is actually not feasible when given a strict time line. Having only 3 weeks to actually conceive an idea and then put it all together takes great coordination and effort from everyone in the team.

Advice: Try to focus on what the programming language is set out to achieve before diving into the nitty gritty details of things. The focus on details will lead to many lengthy discussions that might be completely irrelevant when changes to the project direction occur.

8.3 Anton

1. Functional programming!!! Never before did I enjoy writing code so much. 2. Sometimes you fail. And you have to settle to a dumb option, because you are out of time. You make mistakes in the parser, logical ones, that only surface during code generation. I fell on my face in the codegen, and we ended up with 4 different lists and a function for each of them. Very very stupid. 3. Some things don't have an entire SO devoted to them, so you gonna be stuck on your own, with weird C++ interface references. Like for example with Ocaml LLVM bindings... 4. Typecheking is beautiful, and it is surprising how much the compiler can deduce, how much checking you can do during static semantic check.

Advice: Think about types and your builtins. Make sure you are not asking OCaml LLVM module to figure our the types of your lists. Find a way around that. Codegen is going to be weird. You are still writing it in OCaml, but its not really functional code anymore. Earlier you figure out how the bindings work, the better you'll be equipped for anything you want to implement.

8.4 Macrina

I had never heard of OCAML and functional programming was just a meaningless phrase for me before this course. While, I won't let my programming life revolve around OCAML after this course, learning it gave me a new and interesting perspective. I found the stages of compilation very interesting - I still can't wrap my head around the fact that simple (or rather, not exceedingly complex) steps when coupled together can be using to compile a language. I had never heard of LLVM before this course either. Team work! I am highly opiniated - the team spirit helped cure me of this (to some extent).

Advice: The milestones set for the project are invaluable. Stick to them. Actively discuss with the assigned advisor and brainstorm within the team. Keep an open mind while presenting or receiving ideas. The scanner, parser are similar to those in the microC compiler discussed in class. Make use of this and don't try to reinvent the wheel. Take time to write code in your language for a variety of relevant algorithms for the proosal itself and plan accordingly. Polymorphism, pointers and other seemingly simple operations become complex in the codegen so give them sufficient time. Try to compile down to LLVM at least - the feeling of accomplishment is worth the pain.

9 Appendix

9.1 ast.ml

```
(* Authors: Donovan Chan, Andrew Feather, Macrina Lobo,
      Anton Nefedenkov
   Note: This code was writte on top of Prof. Edwards's
```

```
        microc code. We hope this is acceptable. *)

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

(* List and Edge here are different from below *)
type uop = Not

type typ = Int | String |  Edge | Void
       | EListtyp | SListtyp | IListtyp | NListtyp
       | EmptyListtyp | Nothing


type bind = typ * string

type expr = Litint of int
     | Litstr of string
     | Id of string
     | Binop of expr * op * expr
     | Assign of string * expr
     | Noexpr
     | Unop of uop * expr
     | Call of string * expr list
     | Edgedcl of expr * expr * expr
     | Listdcl of expr list
     (*  Localdecl of typ * string  *)
(* Added to support local decls *)
(*MIGHT HAVE ISSUES HERE, alternative expr list*)

type stmt =
       | Localdecl of typ * string
       | Block of stmt list
       | Expr of expr
       | If of expr * stmt * stmt  (*MIGHT NOT NEED ELSE ALL THE
    TIME*)
       | For of expr * expr * expr * stmt
       | While of expr * stmt
       | Return of expr


type func_decl = {

  typ   : typ;
  fname   : string;
  formals : bind list;
  locals  : bind list;
  body  : stmt list;
}

type program = bind list * func_decl list
```

9.2 scanner.mll

```
(*Ocamllex scanner for GAL*)
(* Authors: Donovan Chan, Andrew Feather, Macrina Lobo,
      Anton Nefedenkov
   Note: This code was writte on top of Prof. Edwards's
        microc code. We hope this is acceptable. *)

{ open Parser }

rule token = parse
```

```
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"      { comment lexbuf }           (* Comments *)
| '('       { LPAREN }
| ')'       { RPAREN }
| '['     { LSQBRACE }
| ']'       { RSQBRACE }
| '{'       { LBRACE }
| '}'       { RBRACE }
| '|'       { BAR }
| ':'       { COLON }
| ';'       { SEMI }
| ','       { COMMA }
| '+'       { PLUS }
| '-'       { MINUS }
| '*'       { TIMES }
| '/'       { DIVIDE }
| '='       { ASSIGN }
| "::"      { LISTSEP }
| "=="      { EQ }
| "!="      { NEQ }
| '<'       { LT }
| "<="      { LEQ }
| '>'       { GT }
| ">="      { GEQ }
| "&&"      { AND }
| "||"      { OR }
| "!"       { NOT }
| "while"   { WHILE }
| "if"      { IF }
| "else"    { ELSE }
| "for"     { FOR }
| "return"  { RETURN }
| "slist"   { SLISTT }
| "node"    { ELISTT }
| "ilist"   { ILISTT }
| "elist"   { ELISTT }
| "nlist"   { NLISTT }
| "edge"    { EDGE }
| "int"     { INT }
| "string"  { STRING }
| ['0'-'9']+ as lxm { LITINT(int_of_string lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm)
  }
| '"' (([' ' '-'!' '#'-'[' ']'-'~']*) as s ) '"' { LITSTR(s) }    (*
  * or + we have no idea*)
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped
  char)) }

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

9.3 parser.mly

```
%{
(*  Authors: Donovan Chan, Andrew Feather, Macrina Lobo,
      Anton Nefedenkov
    Note: This code was write on top of Prof. Edwards's
      microc code. We hope this is acceptable. *)
    open Ast
    open Help
```

```
    let build_edge ~src (weight, dst) =
      Edgedcl(src, weight, dst)
%}

%token SEMI LPAREN RPAREN LSQBRACE RSQBRACE LBRACE RBRACE BAR COLON
     LISTSEP COMMA
%token EPLUS EMINUS PLUS MINUS TIMES DIVIDE ASSIGN NOT
%token EQ LT LEQ GT GEQ AND OR NEQ
%token RETURN IF ELSE FOR INT STRING EDGE SLISTT NLISTT ELISTT
    ILISTT DEFINE WHILE
%token <int> LITINT
%token <string> ID
%token <string> LITSTR
%token EOF


%right ASSIGN
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE
%right NOT

%start program
%type <Ast.program> program

%%

program: decls EOF { $1 }

decls: /*nothing */ {[],[]}
  | decls vdecl { ($2 :: fst $1), snd $1 }
  | decls fdecl { fst $1, ($2 :: snd $1) }

vdecl: typ ID SEMI { ($1, $2) }

fdecl:
  typ ID LPAREN formals_opts RPAREN LBRACE func_body RBRACE
  {{ typ = $1; fname = $2; formals = $4;
    locals = Help.get_vardecls [] $7;
    body = $7 }}

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

formal_list: typ ID     { [($1,$2)] }
  | formal_list COMMA typ ID  { ($3,$4) :: $1 }

typ:
    INT      { Int      }
  | STRING   { String   }
  | SLISTT   { SListtyp   }
  | EDGE     { Edge     }
  | NLISTT   { NListtyp   }
  | ELISTT   { EListtyp   }
  | ILISTT   { IListtyp   }

func_body:
```

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


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

                    /*DOESNT ALLOW RETURN of Nothing*/
stmt:
    typ ID SEMI              { Localdecl($1, $2)}
    | expr SEMI                { Expr $1 }
  | RETURN expr SEMI             { Return $2  }
  | LBRACE stmt_list RBRACE         { Block(List.rev $2) }
  | IF LPAREN expr RPAREN stmt ELSE stmt  { If($3, $5, $7) }
  | FOR LPAREN expr_opt SEMI expr SEMI expr_opt RPAREN stmt
                      { For($3,$5,$7,$9) }
  | WHILE LPAREN expr RPAREN stmt        { While($3, $5) }


list_list: /*nothing*/ { [] }
  | listdecl { List.rev $1 }

listdecl:
    expr          { [$1] }
  | listdecl LISTSEP expr { $3 :: $1 }

node_syntax:
    expr COLON w_dst_list { List.map (build_edge ~src:$1) $3}

w_dst_list:
    expr COMMA expr      {[($1, $3)]}
  | expr COMMA expr COMMA w_dst_list {($1, $3)::$5}

expr:
      /* typ ID              { Localdecl($1, $2)} */
    | BAR node_syntax BAR {Listdcl($2)}
    | LITINT      { Litint($1) }
  | ID        { Id($1) }
  | LITSTR      { Litstr($1) }
  | BAR expr COMMA expr COMMA expr BAR  { Edgedcl($2,$4,$6) }
  | LSQBRACE list_list RSQBRACE       { Listdcl($2) }
  | 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) }
    | NOT expr         { Unop(Not, $2) }
    | ID ASSIGN expr   { Assign($1, $3) }
    | LPAREN expr RPAREN { $2 }
    | ID LPAREN actuals_opt RPAREN  { Call($1, $3)}

expr_opt: /*nothing*/ { Noexpr }
  | expr        { $1 }
```

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

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

9.4 semant.ml

```
(* Authors: Donovan Chan, Andrew Feather, Macrina Lobo,
      Anton Nefedenkov
   Note: This code was writte on top of Prof. Edwards's
        microc code. We hope this is acceptable. *)

open Ast;;

module StringMap = Map.Make(String);;
let m = StringMap.empty;;

(* Error messages of the exceptions *)
let dup_global_exp = " duplicate global ";;
let dup_local_exp = " duplicate local ";;
let dup_formal_exp = " duplicate formal arg ";;
let dup_func_exp = " duplicate function name ";;
let builtin_decl_exp = " cannot redefine ";;
let main_undef_exp = " main not defined ";;

(* Names of built in functions can be added below *)
let builtins_list =
  ["print_int"; "print_str";
   "length"; "source"; "dest"; "pop"; "weight"; "print_endline"; "
    peek"];;

(* Built in decls *)
let print_int_fdcl =
  { typ = Int; fname = "print_int"; formals = [(Int, "a")];
    locals = []; body = []};;

let print_str_fdcl =
  { typ = String; fname = "print_str"; formals = [(String, "a")];
    locals = []; body = []};;

let slength_fdcl =
  { typ = Int; fname = "slength"; formals = [(SListtyp, "a")];
    locals = []; body = []};;

let elength_fdcl =
  { typ = Int; fname = "elength"; formals = [(EListtyp, "a")];
    locals = []; body = []};;

let ilength_fdcl =
  { typ = Int; fname = "ilength"; formals = [(IListtyp, "a")];
    locals = []; body = []};;

let nlength_fdcl =
  { typ = Int; fname = "nlength"; formals = [(NListtyp, "a")];
    locals = []; body = []};;

let dest_fdcl =
  { typ = String; fname = "dest"; formals = [(Edge, "a")];
    locals = []; body = []};;

```

```
let source_fdcl =
  { typ = String; fname = "source"; formals = [(Edge, "a")];
    locals = []; body = []};;

let weight_fdcl =
  { typ = Int; fname = "weight"; formals = [(Edge, "a")];
    locals = []; body = []};;

let print_endline_fdcl =
  { typ = Int; fname = "print_endline"; formals = [];
    locals = []; body = []};;

(* This function needs discussion *)
let spop_fdcl =
  { typ = SListtyp; fname = "spop"; formals = [(SListtyp, "a")];
    locals = []; body = []};;

let ipop_fdcl =
  { typ = IListtyp; fname = "ipop"; formals = [(IListtyp, "a")];
    locals = []; body = []};;

let epop_fdcl =
  { typ = EListtyp; fname = "epop"; formals = [(EListtyp, "a")];
    locals = []; body = []};;

let npop_fdcl =
  { typ = NListtyp; fname = "npop"; formals = [(NListtyp, "a")];
    locals = []; body = []};;

let speek_fdcl =
  { typ = String; fname = "speek"; formals = [(SListtyp, "a")];
    locals = []; body = []};;

let ipeek_fdcl =
  { typ = Int; fname = "ipeek"; formals = [(IListtyp, "a")];
    locals = []; body = []};;

let epeek_fdcl =
  { typ = Edge; fname = "epeek"; formals = [(EListtyp, "a")];
    locals = []; body = []};;

let npeek_fdcl =
  { typ = EListtyp; fname = "npeek"; formals = [(NListtyp, "a")];
    locals = []; body = []};;

let snext_fdcl =
  { typ = SListtyp; fname = "snext"; formals = [(SListtyp, "a")];
    locals = []; body = []};;

let enext_fdcl =
  { typ = EListtyp; fname = "enext"; formals = [(EListtyp, "a")];
    locals = []; body = []};;

let inext_fdcl =
  { typ = IListtyp; fname = "inext"; formals = [(IListtyp, "a")];
    locals = []; body = []};;

let nnext_fdcl =
  { typ = NListtyp; fname = "nnext"; formals = [(NListtyp, "a")];
    locals = []; body = []};;

let sadd_fdcl =
```

```
  { typ = SListtyp; fname = "sadd"; formals = [(String, "b"); (SListtyp, "a")];
    locals = []; body = []};;

let eadd_fdcl =
  { typ = EListtyp; fname = "eadd"; formals = [(Edge, "b"); (EListtyp, "a")];
    locals = []; body = []};;

let iadd_fdcl =
  { typ = IListtyp; fname = "iadd"; formals = [(Int, "b"); (IListtyp, "a")];
    locals = []; body = []};;

let nadd_fdcl =
  { typ = NListtyp; fname = "nadd"; formals = [(EListtyp, "b"); (NListtyp, "a")];
    locals = []; body = []};;

let str_comp_fdcl =
  { typ = Int; fname = "streq"; formals = [(String, "a"); (String, "b")];
    locals = []; body = []};;


let builtin_fdcl_list =
  [ print_int_fdcl; print_str_fdcl; slength_fdcl; dest_fdcl;
    source_fdcl; spop_fdcl; weight_fdcl; print_endline_fdcl;
    speek_fdcl; ipeek_fdcl; epeek_fdcl; snext_fdcl; elength_fdcl;
    enext_fdcl; inext_fdcl; ilength_fdcl; nnext_fdcl; npeek_fdcl;
    nlength_fdcl; sadd_fdcl; eadd_fdcl; iadd_fdcl; nadd_fdcl;
    str_comp_fdcl; ipop_fdcl; epop_fdcl; npop_fdcl ];;


(* Static semantic checker of the program. Will return void
   on success. Raise an exception otherwise. Checks first the
   globals, then the functions. *)


(* Reports if duplicates present duplicates. *)
let report_duplicate exception_msg list func_name =
  (* Helper that build a list of duplicates *)
  let rec helper dupls = function
    [] -> List.rev dupls;
    | n1 :: n2 :: tl when n1 = n2 -> helper (n2::dupls) tl
    | _ :: tl -> helper dupls tl

  (* Another helper, that uniq's the duples
    (if not already uniq) Works on sorted lists! *)
  in let rec uniq result = function
      [] -> result
      | hd::[] -> uniq (hd::result) []
    | hd1::(hd2::tl as tail) ->
        if hd1 = hd2 then uniq result tail
          else uniq (hd1::result) tail

  (* Get a list of duplicates *)
  in let dupls = uniq [] (helper [] (List.sort compare list))

  (* If the list is not an empty list *)
  in if dupls <> [] then
    match func_name with
```

```
      | "" ->
      (exception_msg ^ (String.concat " " dupls) )
      | _ ->
      (exception_msg ^ (String.concat " " dupls) ^ " in " ^ func_name )

  else ""


(* Returns a list of lists of locals *)
let rec extract_locals local_vars = function
  [] -> List.rev local_vars
  | hd::tl -> extract_locals
    (( hd.fname, (List.map snd hd.locals))::local_vars) tl
;;

(* Extracts formal arguments *)
let rec extract_formals formals = function
  [] -> List.rev formals
  | hd::tl -> extract_formals
    (( hd.fname, (List.map snd hd.formals))::formals) tl

(* Helper functions extracts good stuff from list of funcs *)
let rec func_duplicates exp_msg exception_list = function
  | []          -> List.rev exception_list
  | (name, var_list)::tail  ->
  func_duplicates
    exp_msg
    ((report_duplicate exp_msg var_list name)::exception_list)
    tail
;;

(* Function get rid of empty string in exception list *)
let rec purify_exp_list result = function
  [] -> List.rev result
  | hd::tl when hd <> "" -> purify_exp_list (hd::result) tl
  | _::tl -> purify_exp_list result tl
;;

(* List of built ins is the implicit argument here*)
let rec check_builtins_defs exp_list expmsg funcs = function
  [] -> List.rev exp_list
  | hd::tl ->
    if (List.mem hd funcs) then
      let exp = expmsg ^ hd in
      check_builtins_defs (exp::exp_list) expmsg funcs tl
    else
      check_builtins_defs exp_list expmsg funcs tl
;;

(* Helper function to print types *)
let string_of_typ asttype = match asttype with
  | Int -> " int "
  | String -> " string "
  | SListtyp -> " slist "
  | Edge -> " edge "
  | Void -> " (bad expression) "
  | EListtyp -> " elist "
    | NListtyp -> " nlist "
    | IListtyp -> " ilist "

(* Function checks bunch of fun stuff in the function structure *)
```

```
let check_func exp_list globs_map func_decl funcs_map =

  (* Function returns the type of the identifier *)
  let get_type_of_id exp_list vars_map id =
    (* StringMap.iter
      (fun name typname -> (print_string (name ^ "\n")) )
      vars_map;  *)
    try (StringMap.find id vars_map, exp_list)
    with Not_found ->
      (Void, (" in " ^ func_decl.fname ^ " var: " ^
          " unknown identifier " ^ id)::exp_list)

  (* Helper will return a list of exceptions *)
  in let rec get_expression_type vars_map exp_list = function
    | Litstr(_) -> (String, exp_list)
    | Litint(_) -> (Int, exp_list)
    | Id(name)  -> get_type_of_id exp_list vars_map name
    | Binop(e1, op, e2) (* as e *) ->
      let (v1, exp_list) = get_expression_type vars_map exp_list e1 in
      let (v2, exp_list) = get_expression_type vars_map exp_list e2
      in (match op with
        (* Integer operators *)
        | Add | Sub | Mult | Div | Equal | Less | Leq
        | Greater | Geq | And | Or | Neq
        when (v1 = Int && v2 = Int) -> (Int, exp_list)
        (* List operators *)
        (* | Eadd | Esub when v1 = Listtyp && v2 = Listtyp -> (Listtyp, exp_list) *)
        | _ -> (Void, ( " in " ^ func_decl.fname ^ " expr: " ^
                " illegal binary op ")::exp_list) )
    | Unop(op, e1) -> get_expression_type vars_map exp_list e1
    | Noexpr -> (Void, exp_list) (* Need to check how Noexp is used *)
    | Assign(var, e) (* as ex *) ->
        (* print_string (" assignment to " ^ var ^ "\n"); *)
      let (lt, exp_list) = get_type_of_id exp_list vars_map var in
      let (rt, exp_list) = get_expression_type vars_map exp_list e
        in if (lt <> rt && rt <> EmptyListtyp) || rt = Void then
        (Void,(" in " ^ func_decl.fname ^ " expr: " ^
            " illegal assignment to variable " ^ var)::exp_list)
        else (rt, exp_list)
    | Edgedcl(e1, e2, e3) ->
      let (v1, exp_list) = get_expression_type vars_map exp_list e1 in
      let (v2, exp_list) = get_expression_type vars_map exp_list e2 in
      let (v3, exp_list) = get_expression_type vars_map exp_list e3 in
        if v1 = String && v3 = String && v2 = Int then
          (Edge, exp_list)
        else
          (Void, ( " in " ^ func_decl.fname ^ " edge: " ^
                " bad types ")::exp_list)
    | Listdcl(elist) ->
      (* Get the type of the first element of the list *)
      let get_elmt_type decl_list = match decl_list with
      | [] -> Nothing
      | hd::tl ->
        let (v1, exp_list) = get_expression_type vars_map exp_list hd in
        v1
```

```
      in

      (* Get the type of the list *)
      let get_list_type elmt_type = match elmt_type with
        | Nothing   -> EmptyListtyp
        | Edge    -> EListtyp
        | String  -> SListtyp
        | Int     -> IListtyp
        | EListtyp  -> NListtyp
        | _     -> raise (Failure("in list decl process"))

      in

      let rec check_list exp_list = function
      [] -> List.rev exp_list
      | hd::[] ->
        let (v1, exp_list) = get_expression_type vars_map exp_list hd in
        check_list exp_list []
      | hd1::(hd2::tl as tail) ->
        let (v1, exp_list) = get_expression_type vars_map exp_list hd1 in
        let (v2, exp_list) = get_expression_type vars_map exp_list hd2 in
        if v1 <> v2 then
          check_list
          ((" in " ^ func_decl.fname ^ " list: " ^
                " bad types of expressions ")::exp_list)
          []
        else
          check_list exp_list tail
      in

      let list_exp_list = check_list [] elist
      in if list_exp_list <> [] then
        (Void, (exp_list @ list_exp_list))
      else
        let elmt_type = get_elmt_type elist in
        let list_typ  = get_list_type elmt_type in
        (list_typ, exp_list)


    (* CARE HERE, NOT FINISHED AT ALL *)
    | Call(fname, actuals) ->
      try let fd = StringMap.find fname funcs_map
      in if List.length actuals <> List.length fd.formals then
        (Void, (
          " in " ^ func_decl.fname ^ " fcall: " ^
          fd.fname ^ " expects " ^
          (string_of_int (List.length fd.formals)) ^
           " arguments " )::exp_list)
      else
        (* Helper comparing actuals to formals *)
        let rec check_actuals formals exp_list = function
          [] -> List.rev exp_list
          | actual_name::tla -> match formals with
            | []       -> raise (Failure(" bad. contact me "))
            | hdf::tlf ->
              let (actual_typ, exp_list) = get_expression_type
              vars_map exp_list actual_name in
              let (formal_typ, _) = hdf in
              if formal_typ = actual_typ then
```

```
                check_actuals tlf exp_list tla
              else
                (" in " ^ func_decl.fname ^
                " fcall: wrong argument type in " ^
                fname ^ " call ")::exp_list

        in let exp_list = check_actuals
          (fd.formals)
          exp_list
          actuals
        in (fd.typ, exp_list)

      with Not_found ->
        (Void, (" in " ^ func_decl.fname ^ " fcall:" ^
            " function " ^ fname ^ " not defined ")::exp_list)

    | _ -> (Void, exp_list)

(* In short, helper walks through the ast checking all kind of things *)
  in let rec helper vars_map exp_list = function
    | [] -> List.rev exp_list
    |  hd::tl -> (match hd with
      | Localdecl(typname, name) ->
          (* print_string ("locvar " ^ name ^ " added \n"); *)
          helper (StringMap.add name typname vars_map) exp_list tl
      | Expr(e) ->
          (* print_string " checking expression "; *)
          let (typname, exp_list) = get_expression_type vars_map exp_list e in
          helper vars_map exp_list tl
      | If(p, s1, s2) ->
          let (ptype, exp_list) = get_expression_type vars_map exp_list p in
            if ptype <> Int then

              helper vars_map
              (( " in " ^ func_decl.fname ^
                 " if: predicate of type " ^ string_of_typ ptype )
              ::(helper vars_map (helper vars_map exp_list [s1]) [s2]))
              tl
            else
              helper vars_map
              (helper vars_map (helper vars_map exp_list [s1]) [s2])
              tl
      | For(e1, e2, e3, s) ->
          let (e1_typ, exp_list) = get_expression_type vars_map exp_list e1 in
          let (e2_typ, exp_list) = get_expression_type vars_map exp_list e2 in
          let (e3_typ, exp_list) = get_expression_type vars_map exp_list e3 in
          if e1_typ = e3_typ && e2_typ = Int then
            helper vars_map (helper vars_map exp_list [s]) tl
          else
            helper vars_map
            ((" in " ^ func_decl.fname ^
             " for loop: bad types of expressions. Type * Int * Type expected. ")
            ::exp_list)
```

```
          tl
      | While(cond, loop) ->
          let (cond_typ, exp_list) = get_expression_type vars_map exp_list cond in
          if cond_typ = Int then
            helper vars_map (helper vars_map exp_list [loop]) tl
          else
            helper vars_map
            ((" in " ^ func_decl.fname ^
             " while loop: bad type of conditional expression ")
            ::exp_list)
            tl

      | Block(sl) -> (match sl with
          | [Return(_) as s] ->
            helper vars_map (helper vars_map exp_list [s]) tl
          | Return(_)::_ ->
            helper vars_map
            ((" in " ^ func_decl.fname ^ " ret: nothing can come after return" ^
            " in a given block")::exp_list)
            tl
          | Block(sl)::ss ->
            helper vars_map
            (helper vars_map exp_list (sl @ ss))
            tl
          | s::sl as stl-> helper vars_map
            (helper vars_map exp_list stl)
            tl
          | [] -> helper vars_map exp_list tl

        )

      (* Make sure that tl is an empty list at this point, otherwise throgw exception *)
      | Return(e) -> let (rettyp, exp_list) = get_expression_type vars_map exp_list e
          in if rettyp = func_decl.typ then
              helper vars_map exp_list tl
          else (func_decl.fname ^ " ret: expected return type " ^
              (string_of_typ func_decl.typ) ^ " but expression is of type " ^
              (string_of_typ rettyp))::exp_list
      | _ -> helper vars_map exp_list [] (* Placeholder *)

    )

  in let globs_forms_map = List.fold_left
    (fun m (typname, name) -> StringMap.add name typname m)
    globs_map
    func_decl.formals

  in helper globs_forms_map exp_list (List.rev func_decl.body)


let rec check_functions exp_list globs_map funcs_map = function
  | [] -> List.rev exp_list
  | hd::tl -> check_functions
      (check_func exp_list globs_map hd funcs_map)
      globs_map
      funcs_map
      tl
```

```
(* The thing that does all the checks *)
let check (globals, funcs) =

  (* Check duplicate globals *)
  let global_dup_exp =
    report_duplicate dup_global_exp (List.map snd globals) ""

  (* Check the local variables *)
  in let exp = global_dup_exp::
  ((func_duplicates dup_local_exp []
            (extract_locals [] funcs)))

  (* Check the formal arguments *)
  in let exp = func_duplicates
    dup_formal_exp
    exp
    (extract_formals [] funcs)

  (* Check for func name duplicates *)
  in let exp = (report_duplicate
    dup_func_exp
    (List.map (fun n -> n.fname) funcs)
    "")::exp

  (* Check if built ins were redefined *)
  in let exp = (check_builtins_defs
    exp
    builtin_decl_exp
    (List.map (fun n -> n.fname) funcs)
    builtins_list)

  (* Add builtins to the map *)
  in let builtin_decls = List.fold_left
    (fun m fd -> StringMap.add fd.fname fd m)
    StringMap.empty
    builtin_fdcl_list

  (* Add user declared functions to the map *)
  in let fdecl_map = List.fold_left
    (fun m fd -> StringMap.add fd.fname fd m)
    builtin_decls
    funcs

  (* Check if main was properly declared *)
  in let exp =
      try ignore (StringMap.find "main" fdecl_map); exp
      with Not_found -> main_undef_exp :: exp

  (* Get a map of globals for future use in symbol table
    composition for each function *)
  in let globs_map = List.fold_left
    (fun m (typname, name) -> StringMap.add name typname m)
    StringMap.empty
    globals

  in let exp = check_functions exp globs_map fdecl_map funcs


  (* Get rid of elements containing empty sstring *)
  in purify_exp_list [] exp
```

```
  (*in exp::List.map (report_duplicate dup_local_exp)
    (extract_locals [] funcs) *)

;;
```

9.5 codegen.ml

```
(* Authors: Donovan Chan, Andrew Feather, Macrina Lobo,
      Anton Nefedenkov
   Note: This code was write on top of Prof. Edwards's
      microc code. We hope this is acceptable. *)


module A = Ast
module L = Llvm
module P = Printf
module StringMap = Map.Make(String)


let translate (globals, functions) =

  let the_funcs_map = StringMap.empty in
  let the_funcs_map =
    List.fold_left
    (fun map fdecl -> StringMap.add fdecl.A.fname fdecl.A.typ map)
    the_funcs_map
    functions
  in

    (* Holding global string constants *)
    let glob_str_const_hash = Hashtbl.create 200 in

  (* Build a context and the module *)
  let context = L.global_context () in
  let the_module = L.create_module context "GAL"

  (* Few helper functions returning the types *)
  and i32_t = L.i32_type context    (* Integer *)
  and i8_t = L.i8_type context      (* Char    *)
  and i1_t = L.i1_type context      (* Needed for predicates *)

    in let i8_p_t = L.pointer_type i8_t    (* Pointer *)
  in let edge_t = L.struct_type context   (* Edge type *)
        (Array.of_list [i8_p_t; i32_t; i8_p_t])

    in let one = L.const_int i32_t 1

  in let empty_node_t = L.named_struct_type context "empty" in
  L.struct_set_body empty_node_t (Array.of_list [L.pointer_type
    empty_node_t; L.pointer_type i1_t; i32_t ])  true;

  let node_t = L.named_struct_type context "node" in
    L.struct_set_body node_t (Array.of_list [L.pointer_type node_t;
     i8_p_t; i32_t ])  true;

    let e_node_t = L.named_struct_type context "enode" in
    L.struct_set_body e_node_t (Array.of_list [L.pointer_type
    e_node_t; L.pointer_type edge_t; i32_t ])  true;

  let i_node_t = L.named_struct_type context "inode" in
    L.struct_set_body i_node_t (Array.of_list [L.pointer_type
```

```
    i_node_t; i32_t; i32_t ])  true;

  let n_node_t = L.named_struct_type context "nnode" in
    L.struct_set_body n_node_t (Array.of_list [L.pointer_type
    n_node_t; L.pointer_type e_node_t; i32_t ])  true;

  (* Pattern match on A.typ returning a llvm type *)
  let ltype_of_typ ltyp = match ltyp with
    | A.Int    -> i32_t
    | A.Edge  -> L.pointer_type edge_t
    | A.String  -> i8_p_t
    | A.EmptyListtyp -> L.pointer_type empty_node_t
    | A.SListtyp -> L.pointer_type node_t
    | A.EListtyp -> L.pointer_type e_node_t
    | A.IListtyp -> L.pointer_type i_node_t
    | A.NListtyp -> L.pointer_type n_node_t
    | _   -> raise (Failure ("Type not implemented\n"))


  in let list_type_from_type ocaml_type = match ocaml_type with
    | A.Int      -> i_node_t
    | A.String     -> node_t
    | A.Edge    -> e_node_t
    | A.EListtyp  -> n_node_t
    | _ -> raise (Failure("such lists are not supported "))

  (* Global variables *)
  in let global_vars =
    let global_var m (t, n) =
      (* Initialize the global variable to 000...000 *)
      let init = L.const_int (ltype_of_typ t) 0
    (* Bind the gloabal to its name and its lglobal *)
    in StringMap.add n (L.define_global n init the_module) m
  in List.fold_left global_var StringMap.empty globals

  (************ In built functions below ***********)

  (* Function llvm type *)
  in let printf_t = L.var_arg_function_type i32_t [| L.pointer_type
     i8_t |]
  (* Function declaration *)
  in let printf_func = L.declare_function "printf" printf_t
    the_module

    (* Builds a user defined function *)
  in let function_decls =
    let function_decl map fdecl = (
      (* Get the types of the formals in a list *)
      let formal_types =
        Array.of_list
        (List.map (fun (t, _) -> ltype_of_typ t) fdecl.A.formals)

      (* Get the llvm function type with known return and formals
    types *)
      in let ftype =
        L.function_type
        (ltype_of_typ fdecl.A.typ)
        formal_types

      (* Bind the name of the function to (llvm function, ast
    function) *)
      in StringMap.add fdecl.A.fname
```

```
      (L.define_function fdecl.A.fname ftype the_module, fdecl)
      map)
    (* Populate the map by folding the list of functions *)
    in List.fold_left function_decl StringMap.empty functions

  (* Builds the function body in the module *)
  in let build_function_body fdecl =

    let ocaml_local_hash = Hashtbl.create 100 in
    let local_hash = Hashtbl.create 100 in

    (* Get the llvm function from the map *)
    let (the_function, _) = StringMap.find fdecl.A.fname
    function_decls in


    (* Direct the builder to the right place *)
    let builder = L.builder_at_end context (L.entry_block
    the_function) in

    (* BFotmat string needed for printing. *)
     (*   Will put format string into %tmt in global area *)
    let int_format_string = L.build_global_stringptr "%d" "ifs"
    builder in
    let string_format_string = L.build_global_stringptr "%s" "sfs"
    builder in
    let endline_format_string = L.build_global_stringptr "%s\n" "
    efs" builder in

    let _ =

      let rec enumerate i enumed_l = function
          | [] -> List.rev enumed_l
          | hd::tl -> enumerate (i + 1) ((hd, i)::enumed_l) tl
      in

      let add_formal (t, n) (p, _) =
        L.set_value_name n p;
        let local = L.build_alloca (ltype_of_typ t) n builder in
        ignore (L.build_store p local builder);
        Hashtbl.add local_hash n local;
        Hashtbl.add ocaml_local_hash n t;
      in

      let params = enumerate 0 [] (Array.to_list (L.params
    the_function))

      in List.iter2 add_formal fdecl.A.formals params

    in let add_local builder (t, n) =
        let local_var = L.build_alloca (ltype_of_typ t) n builder
        in Hashtbl.add local_hash n local_var
(*
    in let add_local_list builder ltype n =
        let local_var = L.build_alloca (ltype) n builder
        in Hashtbl.add local_hash n local_var *)

    in let lookup name =
      try Hashtbl.find local_hash name
      with Not_found -> StringMap.find name global_vars

    in let rec get_node_type expr = match expr with
```

```
      | A.Litint(_) -> i_node_t
      | A.Litstr(_) -> node_t
      | A.Listdcl(somelist) ->
        if somelist = [] then
          raise (Failure("empty list decl"))
        else
          let hd::_ = somelist in get_node_type hd
      | A.Binop(e1, _, _) -> get_node_type e1
      | A.Edgedcl(_)  -> e_node_t
      | A.Id(name)  ->
        let ocaml_type = (Hashtbl.find ocaml_local_hash name)
        in  list_type_from_type ocaml_type
      | A.Call("iadd", _) | A.Call("inext", _) ->
        i_node_t
      | A.Call("eadd", _) | A.Call("enext", _) ->
        e_node_t
      | A.Call("sadd", _) | A.Call("snext", _) ->
        node_t
      | A.Call("nadd", _) | A.Call("nnext", _) ->
        n_node_t
      | A.Call("ilength", _) | A.Call("slength", _) | A.Call("nlength", _) | A.Call("elength", _) ->
        i_node_t
      | A.Call(fname, _) ->
        let ftype = StringMap.find fname the_funcs_map in
        ltype_of_typ ftype
        (* try let fdecl = List.find
        (fun fdecl -> if fdecl.A.fname = fname then true else false)
        functions
        in (ltype_of_typ fdecl.A.typ) with Not_found -> in  *)
      | _ -> raise (Failure(" type not supported in list "))


    (* We can now describe the action to be taken on ast traversal *)
    (* Going to first pattern match on the list of expressions *)
    in let rec expr builder e =

      (* Helper to add element to the list *)
      let add_element head_p new_node_p =
        let new_node_next_field_pointer =
          L.build_struct_gep new_node_p 0 "" builder in
          ignore (L.build_store head_p
          new_node_next_field_pointer builder);
          new_node_p

      in let add_payload node_p payload_p =
        let node_payload_pointer =
          L.build_struct_gep node_p 1 "" builder in
          ignore (L.build_store payload_p
          node_payload_pointer builder);
          node_p

      in let build_node node_type payload =
        let alloc = L.build_malloc node_type ("") builder in
        let payload_p = expr builder payload in
        add_payload alloc payload_p

      in match e with
      | A.Litint(i) -> L.const_int i32_t i
      | A.Litstr(str) ->
```

```
        let s = L.build_global_stringptr str str builder in
        let zero = L.const_int i32_t 0 in
        let lvalue = L.build_in_bounds_gep s [|zero|] str builder
    in
        let lv_str = L.string_of_llvalue s in
        (* P.fprintf stderr "%s\n" lv_str; *)

        Hashtbl.add glob_str_const_hash lvalue str;
        s
      | A.Edgedcl(src, w, dst) ->
        let src_p = expr builder src
        and w =  expr builder w
        and dst_p = expr builder dst
        in let alloc = L.build_malloc edge_t ("") builder

        in let src_field_pointer =
          L.build_struct_gep alloc 0 "" builder
        and weight_field_pointer =
          L.build_struct_gep alloc 1 "" builder
        and dst_field_pointer =
          L.build_struct_gep alloc 2 "" builder

        in
          ignore (L.build_store src_p src_field_pointer builder);
          ignore (L.build_store dst_p dst_field_pointer builder);
          ignore (L.build_store w weight_field_pointer builder);
          L.build_in_bounds_gep alloc [|(L.const_int i32_t 0)|] ""
    builder

      | A.Listdcl(elist) ->
        let elist = List.rev elist in

        if (elist = []) then
          L.const_pointer_null (L.pointer_type empty_node_t)
          (* raise (Failure("empty list assignment")) *)
        else
          let (hd::tl) = elist in
          let good_node_t = get_node_type hd in
          let head_node = build_node (good_node_t) hd in
          let head_node_len_p = L.build_struct_gep head_node 2 ""
    builder in
          let head_node_next_p =  L.build_struct_gep head_node 0 ""
     builder in
          ignore (L.build_store (L.undef (L.pointer_type
    good_node_t)) head_node_next_p builder);
          ignore (L.build_store (expr builder (A.Litint(1)))
    head_node_len_p builder);

          let rec build_list the_head len = function
            | [] -> the_head
            | hd::tl ->(
              let len = len + 1 in
              let new_node = build_node good_node_t hd in
              let new_head = add_element the_head new_node in
              let new_head_len_p = L.build_struct_gep new_head 2 ""
     builder in
              ignore (L.build_store (expr builder (A.Litint(len)))
    new_head_len_p builder);
              build_list new_head (len) tl)

          in (build_list head_node 1 tl)

```

```
    | A.Id(name) -> L.build_load (lookup name) name builder
    | A.Assign(name, e) ->
      let loc_var = lookup name in
      let e' = (expr builder e) in

      (* Cant add it like this. Need a different comparison. And need to remove
        old var form the hash map  *)
      if ((L.pointer_type empty_node_t) = (L.type_of e')) then
        (

        (* This is the ocaml type of the variable *)
        let list_type = Hashtbl.find ocaml_local_hash name in

        (* Cant get to the right type for store instruction, so this: *)
        let get_llvm_node_type ocaml_type = match ocaml_type with
          | A.SListtyp -> node_t
          | A.IListtyp -> i_node_t
          | A.NListtyp -> n_node_t
          | A.EListtyp -> e_node_t
          | _          -> raise (Failure("list type not supported"))
        in

        let llvm_node_t = get_llvm_node_type list_type in
        let dummy_node = L.build_malloc llvm_node_t ("") builder in
        let dummy_node_len_p = L.build_struct_gep dummy_node 2 "" builder in
        ignore (L.build_store (expr builder (A.Litint(0))) dummy_node_len_p builder);
        ignore (L.build_store dummy_node loc_var builder);
        e' )
      else
        (ignore (L.build_store e' (lookup name) builder); e')

    (* Calling builtins below *)
    | A.Call("print_int", [e]) ->
      L.build_call printf_func
      [| int_format_string; (expr builder e)|]
      "printf"
      builder
    | A.Call("print_str", [e]) ->
      L.build_call printf_func
      [| string_format_string; (expr builder e)|]
      "printf"
      builder
    | A.Call("print_endline", []) ->
      L.build_call printf_func
      [| endline_format_string; (expr builder (A.Litstr("")))|]
      "printf"
      builder
    | A.Call("source", [e]) ->
      let src_field_pointer = L.build_struct_gep (expr builder e) 0 "" builder
      in L.build_load src_field_pointer "" builder
    | A.Call("weight", [e]) ->
      let weight_field_pointer = L.build_struct_gep (expr builder e) 1 "" builder
      in L.build_load weight_field_pointer "" builder
    | A.Call("dest", [e]) ->
```

```
    let dest_field_pointer = L.build_struct_gep (expr builder e) 2 "" builder
    in L.build_load dest_field_pointer "" builder
  | A.Call("spop", [e]) | A.Call("epop", [e]) | A.Call("ipop", [e]) | A.Call("npop", [e])->
    let head_node_p = (expr builder e) in
    let head_node_next_node_pointer = L.build_struct_gep head_node_p 0 "" builder in
    ignore (L.build_free head_node_p builder);
    L.build_load head_node_next_node_pointer "" builder
  | A.Call("speek", [e]) | A.Call("ipeek", [e]) | A.Call("epeek", [e]) | A.Call("npeek", [e])->
    let head_node_p = (expr builder e) in
    (* Trying to make the crash graceful here 565jhfdshjgq2 *)
    if  head_node_p = (L.const_pointer_null (L.type_of head_node_p)) then
      raise (Failure("nothing to peek at, sorry"))
    else
    let head_node_payload_pointer = L.build_struct_gep head_node_p 1 "" builder in
    L.build_load head_node_payload_pointer "" builder
  | A.Call("snext", [e]) | A.Call("enext", [e]) | A.Call("inext", [e]) | A.Call("nnext", [e])->
    let head_node_next_p = L.build_struct_gep (expr builder e) 0 "" builder in
    L.build_load head_node_next_p "" builder
  | A.Call("slength", [e]) | A.Call("elength", [e]) | A.Call("ilength", [e]) | A.Call("nlength", [e]) ->
    let head_node = expr builder e in
    if (L.pointer_type empty_node_t) = (L.type_of head_node) then
      L.const_int i32_t 0
    else

      let head_node_len_p =  L.build_struct_gep (head_node) 2 "" builder in
      L.build_load head_node_len_p  "" builder

  | A.Call("sadd", [elmt; the_list]) | A.Call("iadd", [elmt; the_list])
  | A.Call("nadd", [elmt; the_list]) | A.Call("eadd", [elmt; the_list]) ->

    (* Build the new node *)
    (* let elmt = (expr builder the_list) in *)
    let the_head = (expr builder the_list) in
    let good_node_t = get_node_type elmt in
    let new_node = build_node (good_node_t) elmt in

    (* To accomodate for calls that take an empty list in (?) *)
    if (L.pointer_type empty_node_t) = (L.type_of the_head) then
      let new_node_len_p = L.build_struct_gep new_node 2 "" builder in
      ignore (L.build_store (L.const_int i32_t 1) new_node_len_p builder);
      new_node
    else

      (* If the length is 0, we should detect this in advance *)
```

```
        let head_node_len_p =  L.build_struct_gep the_head 2 "" builder in
        let llength_val = L.build_load head_node_len_p  "" builder in

        if (L.is_null llength_val) then
          let new_node_len_p = L.build_struct_gep new_node 2 "" builder in
          ignore (L.build_store (L.const_int i32_t 1) new_node_len_p builder);
          new_node

        else

          (* Get the lenght of the list *)
          let old_length = L.build_load (L.build_struct_gep the_head 2 "" builder) "" builder in
          let new_length = L.build_add old_length one "" builder in

          (* Store the lenght of the list *)
          let new_node_len_p = L.build_struct_gep new_node 2 "" builder in
          ignore (L.build_store new_length new_node_len_p builder);

          (* Attach the new head to the old head *)
          add_element the_head new_node
    | A.Call("streq", [s1;s2]) ->
      let v1 = (expr builder s1) and v2 = expr builder s2 in
      let v1value = L.build_load (L.build_load  (L.global_initializer v1)  "" builder) "" builder in
      let v2value = L.build_load (L.build_load  (L.global_initializer v2)  "" builder) "" builder in

      let str = L.string_of_lltype (L.type_of v2value) in

      let result = (L.build_icmp L.Icmp.Eq v1value v2value "" builder) in
      let result = L.build_not result "" builder in
      let result = L.build_intcast result i32_t "" builder in
      result
    (*
*)
    | A.Call(fname, actuals) ->

      (* Will clean up later *)
      let bitcast_actuals (actual, _) =
        let lvalue = expr builder actual in
        lvalue
      in

      let rec enumerate i enumed_l = function
        | [] -> List.rev enumed_l
        | hd::tl -> enumerate (i + 1) ((hd, i)::enumed_l) tl
      in

      let actuals = (enumerate 0 [] actuals) in
      let (fdef, _) = StringMap.find fname function_decls in
      let actuals = List.rev (List.map bitcast_actuals (List.rev actuals)) in
      let result = fname ^ "_result" in
```

```
        L.build_call fdef (Array.of_list actuals) result builder
      | A.Binop(e1, op, e2) ->
        let v1 = expr builder e1 and v2 = expr builder e2 in
        let value =
        (match op with
            A.Add    -> L.build_add
          | A.Sub    -> L.build_sub
          | A.Mult   -> L.build_mul
          | A.Div    -> L.build_sdiv
          | A.And    -> L.build_and
          | A.Or     -> L.build_or
          | A.Equal    -> L.build_icmp L.Icmp.Eq
          | A.Neq    -> L.build_icmp L.Icmp.Ne
            | A.Less     -> L.build_icmp L.Icmp.Slt
            | A.Leq      -> L.build_icmp L.Icmp.Sle
            | A.Greater -> L.build_icmp L.Icmp.Sgt
            | A.Geq      -> L.build_icmp L.Icmp.Sge)
        v1 v2 "tmp" builder  in value
      | A.Unop(op, e) ->
        let e' = expr builder e in
        (match op with
          | A.Not -> L.build_not) e' "tmp" builder
      | _ -> raise (Failure("expr not supported"))


    in let add_terminal builder f =
      match L.block_terminator (L.insertion_block builder) with
      | Some _ -> ()
      | None -> ignore (f builder)

    in let rec stmt builder = function
      | A.Localdecl(t, n) -> (Hashtbl.add ocaml_local_hash n t; ignore (add_local builder (t, n)); builder)
      | A.Block(sl)      -> List.fold_left stmt builder sl
      | A.Expr(e)        -> ignore (expr builder e); builder
      | A.Return(e)      -> ignore (L.build_ret (expr builder e) builder); builder
      | A.If(p, then_stmt, else_stmt) ->
        (* Get the boolean *)
        let bool_val = (expr builder p)

        (* Add the basic block *)
        in let merge_bb   = L.append_block context "merge" the_function
        in let then_bb      = L.append_block context "then" the_function
        in let else_bb     = L.append_block context "else" the_function

        (* Write the statements into their respective blocks, build conditional branch*)
        in
          add_terminal (stmt (L.builder_at_end context then_bb) then_stmt) (L.build_br merge_bb);
          add_terminal (stmt (L.builder_at_end context else_bb) else_stmt) (L.build_br merge_bb);
          ignore (L.build_cond_br bool_val then_bb else_bb builder);

        (* Return the builder *)
        L.builder_at_end context merge_bb
      | A.While(predicate, body) ->
```

```

        let pred_bb = L.append_block context "while" the_function in
          ignore (L.build_br pred_bb builder);

          let body_bb = L.append_block context "while_body" the_function in
          add_terminal (stmt (L.builder_at_end context body_bb) body)
          (L.build_br pred_bb);

          let pred_builder = L.builder_at_end context pred_bb in
          let bool_val = (* bool_of_int *) (expr pred_builder predicate) in

          let merge_bb = L.append_block context "merge" the_function in
          ignore (L.build_cond_br bool_val body_bb merge_bb pred_builder);
          L.builder_at_end context merge_bb

      | _            -> raise (Failure("statement not implemented"))

    in let builder = stmt builder (A.Block (List.rev fdecl.A.body)) in

    add_terminal builder (L.build_ret (L.const_int (ltype_of_typ A.Int) 0))


  in List.iter build_function_body functions;
  the_module
```

9.6 gal.ml

```
type action = Ast | LLVM_IR | Compile;;

module P = Printf;;

let _ = (*
  let action = if Array.length Sys.argv > 1 then
      List.assoc Sys.argv.(1)
        [("-a", Ast); ("-l", LLVM_IR); ("-c", Compile) ]
    else *)
      Compile
    in

  (* Standard Library Functions *)
  let stdlib_file               = "stdlib_code.gal" in
  let stdlib_in                 = open_in stdlib_file in
  let stdlib_lexbuf             = Lexing.from_channel stdlib_in in
  let (std_globs, std_funcs)    = Parser.program Scanner.token stdlib_lexbuf in

  (* The input program *)
  let lexbuf                    = Lexing.from_channel stdin in
  let (globs, funcs)            = Parser.program Scanner.token lexbuf in

  let ast = (std_globs @ globs, std_funcs @ funcs) in

  (* P.fprintf stderr "%s" "ast built\n"; *)
```

```
  let exp_list = Semant.check ast in
  if exp_list <> [] then
    raise (Failure ("\n" ^ (String.concat "\n" exp_list)))

  else
    (* P.fprintf stderr "%s" "ast checked\n"; *)
    let m = Codegen.translate ast in
    Llvm_analysis.assert_valid_module m;
    print_string (Llvm.string_of_llmodule m);
    (* P.fprintf stderr "%s" "code generated\n"; *)
```

9.7 stdlib_code.gal

```
elist get_most_edges_node(nlist graph){

    int len;
    len = nlength(graph);
    int i;
    i = 0;

    ilist lengths;
    lengths = [];

    nlist temp;
    temp = graph;

    /* Get the number of edges */
    while( (i < len) && (len > 0)){
        lengths = iadd(elength(npeek(temp)), lengths);
        temp = nnext(temp);
        i = i + 1;
    }
    lengths = irev(lengths);

    len = ilength(lengths);
    i = 0;
    int longest;
    longest = 0;
    int order;
    order = 1;

    while( (i < len) && (len > 0)){
        if(longest < ipeek(lengths)){
            longest = ipeek(lengths);
            order = i + 1;
        }else{}
        lengths = inext(lengths);
        i = i + 1;
    }

    temp = graph;
    elist result;
    result = [];

    while(order > 1){
        temp = nnext(temp);
    }

    result = npeek(temp);
    return result;
}
```

```
edge get_heaviest_graph_edge(nlist l1){

    int len;
    len = nlength(l1);
    int i;
    i = 0;
    int heaviest_w;
    heaviest_w = 0;

    edge heaviest;
    heaviest = |"EMPTY", 0, "EMPTY"|;

    elist temp;
    temp = [];

    while( (i < len) && (len > 0) ){

        /* Get the head of the list and move forward */
        temp = npeek(l1);
        l1 = nnext(l1);

        /* Get the weight of the element */
        if( heaviest_w < weight(get_heaviest_edge(temp)) ){
            heaviest_w = weight(get_heaviest_edge(temp));
            heaviest = get_heaviest_edge(temp);
        }else{}

        /* Increment */
        i = i + 1;
    }

    return heaviest;
}

/* Function will return the ehaviest edge of the node */
edge get_heaviest_edge(node n1){

    int len;
    len = elength(n1);
    int i;
    i = 0;

    int heaviest_w;
    heaviest_w = 0;

    edge heaviest;
    heaviest = |"EMPTY", 0, "EMPTY"|;

    edge temp;
    temp = |"", 0, ""|;

    /* Iterate through the list, compare weights of edges */
    while( (i < len) && (len > 0) ){

        /* Get the head of the list and move forward */
        temp = epeek(n1);
        n1 = enext(n1);

        /* Get the weight of the element */
        if( heaviest_w < weight(temp) ){
            heaviest_w = weight(temp);
            heaviest = temp;
```

```
        }else{}

        /* Increment */
        i = i + 1;
    }

    return heaviest;
}


int print_line(string str){
    print_str(str);
    print_endline();
    return 0;
}


int print_sl_len(slist lister){
    print_int(slength(lister));
    print_endline();
    return 0;
}

int print_slist(slist l1){
    int len;
    len = slength(l1);
    slist tmp;
    tmp = l1;
    int i;
    i = 0;
    print_str("->");
    while (i < len) {
          print_str(speek(tmp));
          print_str("::");
          tmp = snext(tmp);
          i = i + 1;
    }
    print_endline();

    return 1;
}

int print_edge(edge e){
    print_str("|");
    print_str(source(e));
    print_str(", ");
    print_int(weight(e));
    print_str(", ");
    print_str(dest(e));
    print_str("|");
    return 0;
}

int print_elist(elist l1){
    int len;
    len = elength(l1);
    elist tmp;
    tmp = l1;
    int i;
    i = 0;
    print_str("->");
    while (i < len) {
```

```
            print_edge(epeek(tmp));
            print_str("::");
            tmp = enext(tmp);
            i = i + 1;
    }
    print_endline();

    return 1;
}

int print_ilist(ilist l1){
    int len;
    len = ilength(l1);
    ilist tmp;
    tmp = l1;
    int i;
    i = 0;
    print_str("->");
    while (i < len) {
            print_int(ipeek(tmp));
            print_str("::");
            tmp = inext(tmp);
            i = i + 1;
    }
    print_endline();

    return 1;
}

int print_nlist(nlist l1){
    int len;
    len = nlength(l1);
    nlist tmp;
    tmp = l1;
    int i;
    i = 0;
    print_str("->");
    while (i < len) {
            print_elist(npeek(tmp));
            print_str("::");
            tmp = nnext(tmp);
            i = i + 1;
    }
    print_endline();

    return 1;
}

ilist irev(ilist l1){

    int len_l1;
    len_l1 = ilength(l1);
    ilist temp_l1;
    temp_l1 = [];
    int temp_element;

    while(!(len_l1 ==0)){

        /*adds the first element of the list l1 to temp_l1*/
        temp_element = ipeek(l1);
        temp_l1 = iadd(temp_element,temp_l1);

```

```
        /*advances the head of the list*/
        l1 = inext(l1);

        len_l1 = len_l1 - 1;

    }
    return temp_l1;
}

slist srev(slist l1){

    int len_l1;
    len_l1 = slength(l1);
    slist temp_l1;
    temp_l1 = [];
    string temp_element;

    while(!(len_l1 ==0)){

        /*adds the first element of the list l1 to temp_l1*/
        temp_element = speek(l1);
        temp_l1 = sadd(temp_element,temp_l1);

        /*advances the head of the list*/
        l1 = snext(l1);

        len_l1 = len_l1 - 1;

    }
    return temp_l1;
}

elist erev(elist l1){

    int len_l1;
    len_l1 = elength(l1);
    elist temp_l1;
    temp_l1 = [];
    edge temp_element;

    while(!(len_l1 ==0)){

        /*adds the first element of the list l1 to temp_l1*/
        temp_element = epeek(l1);
        temp_l1 = eadd(temp_element,temp_l1);

        /*advances the head of the list*/
        l1 = enext(l1);

        len_l1 = len_l1 - 1;

    }
    return temp_l1;
}

nlist nrev(nlist l1){

    int len_l1;
    len_l1 = nlength(l1);
    nlist temp_l1;
    temp_l1 = [];
    node temp_element;
```

```
    while(!(len_l1 ==0)){

        /*adds the first element of the list l1 to temp_l1*/
        temp_element = npeek(l1);
        temp_l1 = nadd(temp_element,temp_l1);

        /*advances the head of the list*/
        l1 = nnext(l1);

        len_l1 = len_l1 - 1;

    }
    return temp_l1;
}


ilist iadd_back(ilist l1,int i){


    l1 = irev(l1);
    l1 = iadd(i,l1);
    l1 = irev(l1);
    return l1;

}

slist sadd_back(slist l1,string i){


    l1 = srev(l1);
    l1 = sadd(i,l1);
    l1 = srev(l1);
    return l1;

}

elist eadd_back(elist l1,edge i){


    l1 = erev(l1);
    l1 = eadd(i,l1);
    l1 = erev(l1);
    return l1;

}

nlist nadd_back(nlist l1,node i){


    l1 = nrev(l1);
    l1 = nadd(i,l1);
    l1 = nrev(l1);
    return l1;
}

ilist iconcat(ilist l1, ilist l2){

    l1 = irev(l1);
    int len_l2;
    len_l2 = ilength(l2);
    int temp_element;
```

```
    while(!(len_l2==0)){

        temp_element = ipeek(l2);
        l1 = iadd(temp_element,l1);
        l2 = inext(l2);

        len_l2 = len_l2 - 1;
    }

    l1 = irev(l1);
    return l1;
}

slist sconcat(slist l1, slist l2){

    l1 = srev(l1);
    int len_l2;
    len_l2 = slength(l2);
    string temp_element;

    while(!(len_l2==0)){

        temp_element = speek(l2);
        l1 = sadd(temp_element,l1);
        l2 = snext(l2);

        len_l2 = len_l2 - 1;
    }

    l1 = srev(l1);
    return l1;
}

elist econcat(elist l1, elist l2){

    l1 = erev(l1);
    int len_l2;
    len_l2 = elength(l2);
    edge temp_element;

    while(!(len_l2==0)){

        temp_element = epeek(l2);
        l1 = eadd(temp_element,l1);
        l2 = enext(l2);

        len_l2 = len_l2 - 1;
    }

    l1 = erev(l1);
    return l1;
}

nlist nconcat(nlist l1, nlist l2){

    l1 = nrev(l1);
    int len_l2;
    len_l2 = nlength(l2);
    node temp_element;

    while(!(len_l2==0)){
```

```
        temp_element = npeek(l2);
        l1 = nadd(temp_element,l1);
        l2 = nnext(l2);

        len_l2 = len_l2 - 1;
    }

    l1 = nrev(l1);
    return l1;
}
```

9.8 help.ml

```
open Ast

(* I hope this function is not too broken, still needs testing
  works for if conditionals, do not know about for loops.
  What it does is it goes through the body of the function
  extracting all local variables, returnning a list of locals *)

let rec get_vardecls vars = function
  | [] -> List.rev vars
  | hd::tl -> (match hd with
    | Localdecl(typname, name) ->
      (* print_string " Expr"; *)
      get_vardecls ((typname, name)::vars) tl
    | Block(slist) -> (* print_string " Block"; *)
             (match slist with
      | [] -> get_vardecls vars tl
      | hd1::tl1 ->
        get_vardecls
          (get_vardecls (get_vardecls vars [hd1]) tl1)
          tl1)
    | If(e, s1, s2) ->
      (* print_string " If"; *)
      get_vardecls (get_vardecls (get_vardecls vars [s1]) [s2]) tl
    | For(_, _, _, s) ->
      get_vardecls (get_vardecls vars [s]) tl
    | While(e, s) ->
      get_vardecls (get_vardecls vars [s]) tl
    | _ -> get_vardecls vars tl )

;;
```

9.9 Sample Code: dfs.gal

```
int dfs(nlist graph,string A)
{

  int found;
  found =0;

  slist visited;
  slist stack;
  stack = ["A"];

  elist v;
  int s_counter;
  string temp_str;
  string node_name;
  int node_found;
```

```
nlist temp;
temp = graph;
int graph_length;
graph_length = nlength(graph);
int i;
i = 0;
int count;
count = 0;
string v_dest;

string v_source;
visited = [""];
int streq_val;
string top_of_stack;
elist use_node;
elist temp_node;
string temp_source;
string temp_dest;
slist stack_temp;
elist use_node_temp;

stack_temp = stack;
int count_loop;
count_loop = 0;
string temp_visited;

while(count<7)
{
  if(i >= graph_length)
  {
    return found;
  }
  else
  {

  }
  if(count>0){
    /*print_str(speek(stack_temp));*/
    stack_temp = snext(stack);
    stack = snext(stack);

  }
  else{

  }

  top_of_stack = speek(stack_temp);
  visited = sadd_back(visited,top_of_stack);
  /*this might give us issues*/

  /*Iterate through graph to find correct edge*/
  i = 0;
  temp = graph;
  while(i < graph_length)
  {
    temp_node = npeek(temp);
    temp_source = source(epeek(temp_node));

    streq_val = streq(temp_source,top_of_stack);
    if(streq_val == 0)
    {
```

```
        use_node = temp_node;
      }
      else
      {

      }
      temp = nnext(temp);
      i = i + 1;
    }

    /*temp = nnext(temp);*/



    /*v_source = source(epeek(use_node));
    v_dest = dest(epeek(use_node));
    visited = sadd_back(visited, v_source);*/

    i = 0;

    use_node_temp = use_node;

    while(i<elength(use_node))
    {

      temp_dest = dest(epeek(use_node_temp));
      use_node_temp = enext(use_node_temp);
      stack = sadd_back(stack,temp_dest);


      i = i + 1;
    }


    count = count+1;
  }


  while(count_loop<slength(visited)){

    temp_visited = speek(visited);
    visited = snext(visited);
    if(streq(temp_visited,A)==0){
      found = 1;
    }
    else{

    }

  }
  return found;

}

int main()
{

  int isfound;

  /* Declare our nodes above */
  node n1;
    n1 = |"A": 2, "B", 4, "C"|;
```

```
    node n2;
    n2 = |"B": 11, "E", 12, "F"|;
    node n3;
    n3 = |"C": 5, "G", 16, "H"|;

  nlist new_graph;
    new_graph = [n1::n2::n3];

    isfound = dfs(new_graph,"Z");
    if(isfound == 1){
      print_str("NODE IS FOUND USING DFS");

    }
    else{
      print_str("NODE IS NOT FOUND");
    }
    /*ABEFCGH*/
    print_endline();

}
```

9.10 Sample Code: demo.gal

```
int main(){

  print_endline();

  /* Declare our nodes above */
  node n1;
    n1 = |"A": 2, "B", 11, "C", 4, "D", 14, "E"|;
    node n2;
    n2 = |"B": 7, "C", 3, "A", 20, "D"|;
    node n3;
    n3 = |"C": 5, "D", 5, "A", 16, "E"|;
    node n4;
    n4 = |"D": 20, "A", 7, "B"|;

    print_line("Lets print them to see what we got:");
    print_elist(n1);
    print_elist(n2);
    print_elist(n3);
    print_elist(n4);

    print_endline();
    print_endline();

    /* Lets declare another node. But using diffrent syntax */
    elist n5;
    n5 = [|"E", 24, "D"|::|"E", 13, "B"|];

    print_line("We can also print them as a graph:");
    nlist graph;
    graph = [n1::n2::n3::n4::n5];

    /* We can use a different function to print this graph */
    print_nlist(graph);
    print_endline();


    graph = npop(graph);
    print_nlist(graph);
    graph = npop(graph);
```

```
    print_nlist(graph);
    graph = npop(graph);
    print_nlist(graph);


    slist testpops;
    testpops = ["A"::"B"::"C"];
    print_slist(testpops);
    testpops = spop(testpops);
    print_slist(testpops);
    testpops = spop(testpops);
    print_slist(testpops);





    print_line("Lets get the heaviest edge of the node n1:");
    edge heaviest;
    heaviest = get_heaviest_edge(n1);
    print_edge(heaviest);
    print_endline();

    print_line("How about the heaviest edge in our graph? Sure:");
    heaviest = get_heaviest_graph_edge(graph);
    print_edge(heaviest);
    print_endline();

    print_line("Lets get the node that has the most edges");
    node important;
    important = get_most_edges_node(graph);
    print_line(source(epeek(important)));

    return 0;
}
```

9.11 testall.sh

```
#!/bin/sh

# Regression testing script for MicroC
# Step through a list of files
#  Compile, run, and check the output of each expected-to-work test
#  Compile and check the error of each expected-to-fail test

# Path to the LLVM interpreter
LLI="lli"
#LLI="/usr/local/opt/llvm/bin/lli"

# Path to the microc compiler.  Usually "./microc.native"
# Try "_build/microc.native" if ocamlbuild was unable to create a
    symbolic link.
GAL="./gal.native"
#GAL="_build/microc.native"

# Set time limit for all operations
ulimit -t 30

globallog=testall.log
rm -f $globallog
error=0
globalerror=0
```

```

keep=0

Usage() {
    echo "Usage: testall.sh [options] [.gal files]"
    echo "-k    Keep intermediate files"
    echo "-h    Print this help"
    exit 1
}

SignalError() {
    if [ $error -eq 0 ] ; then
    echo "FAILED"
    error=1
    fi
    echo "  $1"
}

# Compare <outfile> <reffile> <difffile>
# Compares the outfile with reffile.  Differences, if any, written to difffile
Compare() {
    generatedfiles="$generatedfiles $3"
    echo diff -b $1 $2 ">" $3 1>&2
    diff -b "$1" "$2" > "$3" 2>&1 || {
    SignalError "$1 differs"
    echo "FAILED $1 differs from $2" 1>&2
    }
}

# Run <args>
# Report the command, run it, and report any errors
Run() {
    echo $* 1>&2
    eval $* || {
    #SignalError "$1 failed on $*"
    return 1
    }
}

# RunFail <args>
# Report the command, run it, and expect an error
RunFail() {
    echo $* 1>&2
    eval $* && {
    SignalError "failed: $* did not report an error"
    return 1
    }
    return 0
}

Check() {
    error=0
    basename=`echo $1 | sed 's/.*\\///
                             s/.gal//'`
    reffile=`echo $1 | sed 's/.gal$//'`
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

    echo -n "$basename..."

    echo 1>&2
    echo "###### Testing $basename" 1>&2
```

```

    generatedfiles=""

    generatedfiles="$generatedfiles ${basename}.ll ${basename}.out" &&
    Run "$GAL" "<" $1 ">" "${basename}.ll" &&
    Run "$LLI" "${basename}.ll" ">" "${basename}.out"
    Compare ${basename}.out ${reffile}.out ${basename}.diff

    # Report the status and clean up the generated files

    if [ $error -eq 0 ] ; then
    if [ $keep -eq 0 ] ; then
        rm -f $generatedfiles
    fi
    echo "OK"
    echo "###### SUCCESS" 1>&2
    else
    echo "###### FAILED" 1>&2
    globalerror=$error
    fi
}

CheckFail() {
    error=0
    basename=`echo $1 | sed 's/.*\\///
                             s/.gal//'`
    reffile=`echo $1 | sed 's/.gal$//'`
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

    echo -n "$basename..."

    echo 1>&2
    echo "###### Testing $basename" 1>&2

    generatedfiles=""

    generatedfiles="$generatedfiles ${basename}.err ${basename}.diff" &&
    RunFail "$GAL" "<" $1 "2>" "${basename}.err" ">>" $globallog &&
    Compare ${basename}.err ${reffile}.err ${basename}.diff

    # Report the status and clean up the generated files

    if [ $error -eq 0 ] ; then
    if [ $keep -eq 0 ] ; then
        rm -f $generatedfiles
    fi
    echo "OK"
    echo "###### SUCCESS" 1>&2
    else
    echo "###### FAILED" 1>&2
    globalerror=$error
    fi
}

while getopts kdpsh c; do
    case $c in
    k) # Keep intermediate files
        keep=1
        ;;
    h) # Help
```

```
        Usage
        ;;
    esac
done

shift `expr $OPTIND - 1`

LLIFail() {
  echo "Could not find the LLVM interpreter \"$LLI\"."
  echo "Check your LLVM installation and/or modify the LLI variable
     in testall.sh"
  exit 1
}

which "$LLI" >> $globallog || LLIFail


if [ $# -ge 1 ]
then
    files=$@
else
    files="../tests/test_*.gal ../tests/fail_*.gal"
fi

for file in $files
do
    case $file in
    *test_*)
        Check $file 2>> $globallog
        ;;
    *fail_*)
        CheckFail $file 2>> $globallog
        ;;
    *)
        echo "unknown file type $file"
        globalerror=1
        ;;
    esac
done

exit $globalerror
```

9.12 fail_assignment_edge2.gal

```
int main()
{
  edge e1;
  e1 = |5,2,"B"|;
}
```

9.13 fail_assignment_int_to_string.gal

```
int main()
{
  string a;
  a = 5;
}
```

9.14 fail_assignment_string_to_int.gal

```
int main()
{
```

```
  int a;
  a = "This";
}
```

9.15 fail_binary_addition1.gal

```
int main()
{
  int a;

  a = 5 + "hello";

}
```

9.16 fail_binary_addition2.gal

```
int main()
{
  int a;
  string b;

  b = "this";

  a = 5 + b;

}
```

9.17 fail_binary_division.gal

```
int main()
{
  int a;
  string b;

  b = "this";

  a = 5 + b;

}
```

9.18 fail_binary_multiplucation1.gal

```
int main()
{
  int a;

  a = 5 * "hello";

}
```

9.19 fail_duplicate_assignint.gal

```
int main()
{
  int a;
  int b;

  a = 5;
  b = 5;

  int a;
}
```

9.20 Fail_duplicate_formal_identifiers.gal

```
int a;

int main(int a, int a)
{
  int b;
}
```

9.21 fail_duplicate_function_names.gal

```

int main(int x, int y)
{


}

int this()
{


}

int this()
{


}
```

9.22 fail_duplicate_global_assignment.gal

```

int a;
int a;

int main()
{


}
```

9.23 Fail_function_doesnt_exist.gal

```

int main()
{

  test();

}
```

9.24 Fail_incorrect_argument_types.gal

```
int main()
{
  string b;
  int a;

  b = "hello";
  a = 2;
```

```

  test(b,a);

}

int test(int x, int y)
{

  int z;

}
```

9.25 fail_incorrect_number_function_arguments.gal

```
int main()
{
  string b;
  int a;

  b = "hello";
  a = 2;

  test(b,a);

}

int test(int x, int y)
{

  int z;

}
```

9.26 Fail_incorrect_number_function_arguments2.gal

```
int main()
{

  int x;
  int y;

  x = 5;
  y = 7;

  test(x,y);

}

int test()
{
    int c;
    c = 7;
}
```

9.27 Fail_main_nonexistent.gal

```

int x()
{
  int a;
  int b;
  int c;
```

```

  c = a + b;

}
```

9.28 Fail_no_id_before_usage_int.gal

```
int main()
{
  a = 5;

}
```

9.29 Fail_redefine_builtin_edge.gal

```
int main()
{
  string edge;
}
```

9.30 fail_redefine_builtin_int.gal

```
int main()
{
  string int;
}
```

9.31 fail_redefine_builtin_list.gal

```
int main()
{
  int slist;
}
```

9.32 Fail_redefine_existing_function.gal

```
int print_int() {}

int main()
{
  return 0;
}
```

9.33 Test_assignment_list1.gal

```
int main()
{
  edge e1;
  edge e2;
  edge e3;

  e1 = |"A",5,"B"|;
  e2 = |"B",7,"C"|;
  e3 = |"C",2,"A"|;

  elist l1;
  l1 = [e1];

  return 1;

}
```

9.34 test_boolean_false.gal

```
int main()
{
  if(!(1==1))
  {
    print_str("This is true");
  }
  else
  {
    print_str("This is NOT true");
  }
  return 1;
}
```

9.35 Test_boolean_true.gal

```
int main()
{
  if(1==1)
  {
    print_str("This is true");
  }
  else
  {}
  return 1;

}
```

9.36 test_create_edge.gal

```
int main(){

  edge e1;
  return 1;
}
```

9.37 Test_print_ilist.gal

```
int main()
{

  ilist x;

  x = [1];
  x = iadd(2,x);
  x = iadd(100,x);


  print_ilist(x);
  return 1;
}
```

9.38 Test_print_ilist_rev.gal

```
int main()
{

  ilist x;
    ilist rev_x;

```

```
  x = [11];
  x = iadd(10,x);
  x = iadd(9,x);
  x = iadd(8,x);
  x = iadd(7,x);
  x = iadd(1,x);
  x = iadd(2,x);
  x = iadd(3,x);

  x = irev(x);

  print_ilist(x);
  return 1;
}
```

9.39 test_print_int.gal

```
int main(){
    print_int(1);
    return 1;
}
```

9.40 Test_print_int1.gal

```
int main(){

  int a;
  a = 5;

    print_int(a);
    return 1;
}
```

9.41 Test_print_order.gal

```
int main(){
  print_int(6);
  print_endline();
  print_str("Hello");
  print_endline();
  print_int(903);
  print_endline();
    return 1;
}
```