

# Final Project Report- GAL

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

August 12, 2016

## Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	Summary . . . . .	5
1.2	Key Features of GAL . . . . .	5
<b>2</b>	<b>Setup</b>	<b>5</b>
2.1	Installation . . . . .	5
2.2	Running the Compiler . . . . .	6
<b>3</b>	<b>Writing the First GAL Program</b>	<b>6</b>
<b>4</b>	<b>Language Reference Manual</b>	<b>8</b>
4.1	Lexical Conventions . . . . .	8
4.1.1	Comments . . . . .	8
4.1.2	Code Line Termination . . . . .	8
4.1.3	Identifiers (Names) . . . . .	8
4.1.4	Keywords . . . . .	8
4.1.5	String . . . . .	8
4.1.6	Constants . . . . .	8
4.2	Scoping and Derived Data Types . . . . .	9
4.3	Expressions denoted by <i>expr</i> . . . . .	9
4.3.1	Primary Expressions . . . . .	9
4.3.2	Identifiers and Constants . . . . .	9
4.3.3	Node denoted by <i>node</i> . . . . .	10
4.3.4	Edge denoted by <i>edge</i> . . . . .	10
4.3.5	Parenthesized expressions . . . . .	10
4.3.6	Subscripts . . . . .	11
4.3.7	Function calls . . . . .	11
4.3.8	Unary Operators . . . . .	11
4.3.9	Unary minus . . . . .	11
4.3.10	Logical negation . . . . .	11
4.3.11	Multiplicative Binary Operators . . . . .	12
4.3.12	Binary Multiplication . . . . .	12
4.3.13	Binary Division . . . . .	12
4.3.14	Additive Binary Operators . . . . .	12
4.3.15	Binary Operators . . . . .	12
4.3.16	Graph Equality Operator . . . . .	13
4.3.17	AND Operator . . . . .	13

4.3.18	OR Operator . . . . .	13
4.3.19	Assignment Operator . . . . .	13
4.4	Declarations . . . . .	13
4.4.1	Variable Declaration . . . . .	14
4.5	Definitions . . . . .	14
4.5.1	Function Definition . . . . .	14
4.5.2	Variable Definition . . . . .	14
4.6	Statements denoted by <i>statement</i> . . . . .	14
4.6.1	Expression Statement . . . . .	14
4.6.2	Compound statement . . . . .	15
4.6.3	Conditional Statement . . . . .	15
4.6.4	For Loop Statement . . . . .	15
4.6.5	While Loop Statement . . . . .	15
4.6.6	Return Statement . . . . .	16
4.6.7	Null Statement . . . . .	16
4.7	Built-In Functions . . . . .	16
4.8	Printing of Integers, Strings, Newlines and String Comparisons . .	16
4.8.1	<code>print_int</code> . . . . .	16
4.8.2	<code>print_str</code> . . . . .	16
4.8.3	<code>print_endline</code> . . . . .	16
4.8.4	<code>streq</code> . . . . .	16
4.9	Built-ins for Operations on Lists . . . . .	16
4.9.1	<code>length()</code> . . . . .	17
4.9.2	<code>next()</code> . . . . .	17
4.9.3	<code>pop()</code> . . . . .	17
4.9.4	<code>peek()</code> . . . . .	17
4.9.5	<code>add()</code> . . . . .	17
4.9.6	Finding source vertex <code>source()</code> . . . . .	18
4.9.7	Finding destination vertex <code>dest()</code> . . . . .	18
4.9.8	Finding weight of an edge <code>weight()</code> . . . . .	18
4.10	Standard Library Functions . . . . .	18
4.10.1	Finding node with the most number of edges . . . . .	19
4.10.2	Finding the outgoing edge with highest weight . . . . .	19
4.10.3	Finding the heaviest edge in a list of nodes . . . . .	19
4.10.4	printing text in a line . . . . .	19
4.10.5	printing the number of strings a list of strings . . . . .	19
4.10.6	printing a list of strings . . . . .	19
4.10.7	printing an edge . . . . .	20
4.10.8	printing a list of edges . . . . .	20
4.10.9	printing a list of integers . . . . .	20
4.10.10	printing a list of nodes . . . . .	20
4.10.11	reversing a list of integers . . . . .	21
4.10.12	reversing a list of strings . . . . .	21
4.10.13	reversing a list of edges . . . . .	21
4.10.14	reversing a list of nodes . . . . .	21
4.10.15	appending to the end of a list of integers . . . . .	21
4.10.16	appending to the end of a list of strings . . . . .	22
4.10.17	appending to the end of a list of edges . . . . .	22
4.10.18	appending to the end of a list of nodes . . . . .	22
4.10.19	Concatenating two integer lists . . . . .	22

4.10.20 Concatenating two string lists . . . . .	23
4.10.21 Concatenating two edge lists . . . . .	23
4.10.22 Concatenating two node lists . . . . .	23
<b>5 Project Plan</b>	<b>24</b>
5.1 Planning . . . . .	24
5.2 Communication and Synchronization . . . . .	24
5.3 Project Development . . . . .	24
5.4 Development Tools . . . . .	24
5.5 Programming Style Guide . . . . .	24
5.6 Project Log . . . . .	24
5.7 Roles and Responsibilities . . . . .	25
<b>6 Architectural Design</b>	<b>26</b>
6.1 Scanning . . . . .	26
6.2 Parsing And the Abstract Syntax Tree(AST) . . . . .	27
6.3 Semantic Checking . . . . .	27
6.4 Code Generation . . . . .	27
<b>7 Test Plan</b>	<b>27</b>
7.1 Test Cases . . . . .	27
7.2 Testing Automation . . . . .	28
7.3 Test Source Files . . . . .	28
7.4 Who Did What . . . . .	28
<b>8 Lessons Learned</b>	<b>28</b>
8.1 Andrew Feather . . . . .	28
8.2 Donovan Chan . . . . .	28
8.3 Anton . . . . .	29
8.4 Macrina . . . . .	29
<b>9 Appendix</b>	<b>29</b>
9.1 ast.ml . . . . .	29
9.2 scanner.mll . . . . .	30
9.3 parser.mly . . . . .	31
9.4 semant.ml . . . . .	34
9.5 codegen.ml . . . . .	43
9.6 gal.ml . . . . .	52
9.7 stdlib.code.gal . . . . .	53
9.8 help.ml . . . . .	60
9.9 Sample Code: dfs.gal . . . . .	60
9.10 Sample Code: demo.gal . . . . .	63
9.11 testall.sh . . . . .	64
9.12 fail_assignment_edge2.gal . . . . .	67
9.13 fail_assignment_int_to_string.gal . . . . .	67
9.14 fail_assignment_string_to_int.gal . . . . .	67
9.15 fail_binary_addition1.gal . . . . .	68
9.16 fail_binary_addition2.gal . . . . .	68
9.17 fail_binary_division.gal . . . . .	68
9.18 fail_binary_multiplication1.gal . . . . .	68

9.19 fail_duplicate_assignint.gal . . . . .	68
9.20 Fail_duplicate_formal_identifiers.gal . . . . .	69
9.21 fail_duplicate_function_names.gal . . . . .	69
9.22 fail_duplicate_global_assignment.gal . . . . .	69
9.23 Fail_function_doesnt_exist.gal . . . . .	69
9.24 Fail_incorrect_argument_types.gal . . . . .	69
9.25 fail_incorrect_number_function_arguments.gal . . . . .	70
9.26 Fail_incorrect_number_function_arguments2.gal . . . . .	70
9.27 Fail_main_nonexistent.gal . . . . .	70
9.28 Fail_no_id_before_usage_int.gal . . . . .	71
9.29 Fail_redefine_builtin_edge.gal . . . . .	71
9.30 fail_redefine_builtin_int.gal . . . . .	71
9.31 fail_redefine_builtin_list.gal . . . . .	71
9.32 Fail_redefine_existing_function.gal . . . . .	71
9.33 Test_assignment_list1.gal . . . . .	71
9.34 test_boolean_false.gal . . . . .	72
9.35 Test_boolean_true.gal . . . . .	72
9.36 test_create_edge.gal . . . . .	72
9.37 Test_print_ilist.gal . . . . .	72
9.38 Test_print_ilist_rev.gal . . . . .	72
9.39 test_print_int.gal . . . . .	73
9.40 Test_print_int1.gal . . . . .	73
9.41 Test_print_order.gal . . . . .	73

# 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 tarball
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:

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

3. Finally, create the executable file:

```
1 >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.

```
1 edge build_edge( string src , int w, string dst){  
2  
3     edge e1;  
4     e1 = |src , w, dest|;  
5     return e1;  
6 }
```

### STEP 3: Writing the `main` Function in the Program

GAL requires a main function of the form.

```
1 int main(){  
2 }
```

### STEP 4: Declaring and Assigning Variables

Variables must be declared first before assignment can take place.

```
1 /*DECLARATION OF VARIABLES*/  
2 string src_e1;  
3 int weight_e1;  
4 string dst_e1;  
5  
6 /*ASSIGNMENT OF VARIABLES*/  
7 src_e1 = "A";  
8 weight_e1 = 2;  
9 dst_e1 = "B";
```

### STEP 5: Declaring and Assigning an Edge

```
1 edge e2;  
2  
3 e2 = | "A" , 10 , "C" |;
```

### STEP 6: Declaring and Assigning a Graph

Remember that a graph in GAL is implemented as a list of edges.

```
1 eelist l1;
2
3 l1 = [e2];
```

### STEP 7: Function Calls

```
1 edge e1;
2
3 e1 = build_edge(src_e1, weight_e1, dst_e1);
```

### STEP 8: Graph Operator adding an Edge to a Graph

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

### STEP 9: Printing

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

### STEP 9: Final firstGAL.gal source code

The final code when put together should look like this

```
1 edge build_edge(string src, int w, string dst){
2     edge e1;
3     e1 = |src, w, dest|;
4     return e1;
5 }
6
7 int main(){
8
9     string src_e1;
10    int weight_e1;
11    string dst_e1;
12
13    src_e1 = "A";
14    weight_e1 = 2;
15    dst_e1 = "B";
16
17    edge e2;
18    e2 = |"A", 10, "C"|;
19
20    eelist l1;
21    l1 = [e2];
22
23    edge e1;
24    e1 = build_edge(src_e1, weight_e1, dst_e1);
25
26    l1 = eadd(e1, l1);
27
28    print_str("This is a test print of a string");
29    print_endline();
30    print_str("This now prints an integer");
31    print_endline();
32    print_int(weight_e1);
33
34 }
```

## 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   | 6. string | 11. return |
| 2. elist | 7. while  |            |
| 3. slist | 8. if     | 12. node   |
| 4. ilist | 9. else   |            |
| 5. nlist | 10. for   | 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

- 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.

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

```
1 | string :integer ,string ,integer ,string )(integer ,string )
2 ....( 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.

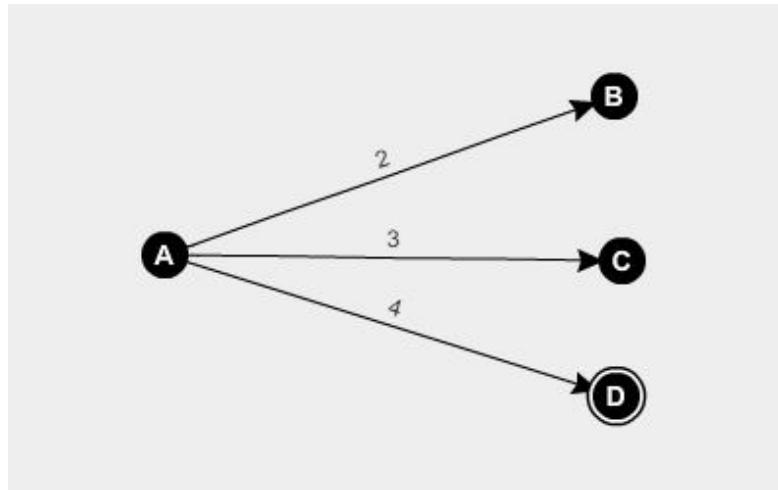


Figure 1: Graph generated

```
1 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

```
1 | 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

```
1 | ( expr )
```

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

#### 4.3.6 Subscripts

The form is

```
1 | 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:

```
1 | 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

```
1 | -expr
```

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

#### 4.3.10 Logical negation

This expression is of the form

```
1 | !( expr )
```

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

```
1 | !(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

```
1 |     expr * expr
```

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

#### 4.3.13 Binary Division

```
1 |     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:

```
1 |     expression+expression
```

Subtraction:

```
1 |     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:

```
1 |     expr < expr
```

```
1 |     expr > expr
```

```
1 |     expr <= expr
```

```
1 |     expr >= expr
```

```
1 |     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:

```
1 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

```
1 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

```
1 expr || expr
```

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

```
1 (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

```
1 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:

```
1 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:

```
1 type-specifier identifier(type1 input1, type2 input2 ... typen
                           inputn){
2 /*first declare the variables and initialise them*/
3
4 /*set of simple and compound statements*/
5
6 /*return (return_value);*/
7 }
```

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

```
1 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:

```
1 {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:

```
1 if (expr) {
2     statement-list
3 }
4 else{
5     statement-list
6 }
```

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:

```
1 for (int expr1; int expr2; int expr3){
2     statement-list
3 }
```

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:

```
1 while(expr){
2     statement-list
3 }
```

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:

```
1   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

```
1   /*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

```
1   print_int(expr);
```

*print\_int* takes in an *expr* that must evaluate into an integer.

### 4.8.2 print\_str

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

*print\_str* prints anything that is enclosed within " as a string.

### 4.8.3 print\_endline

```
1   print_endline();
```

This prints a newline onto the console.

### 4.8.4 strcmp

```
1   strcmp(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

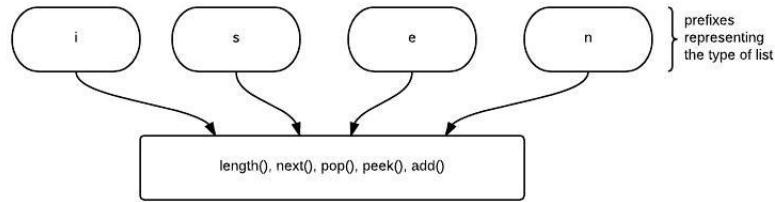


Figure 2: Flowchart Showing Structure of Built-In

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()`

```
1 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()`

```
1 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:

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

#### 4.9.3 `pop()`

```
1 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()`

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

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

#### 4.9.5 `add()`

```
1 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:

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

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

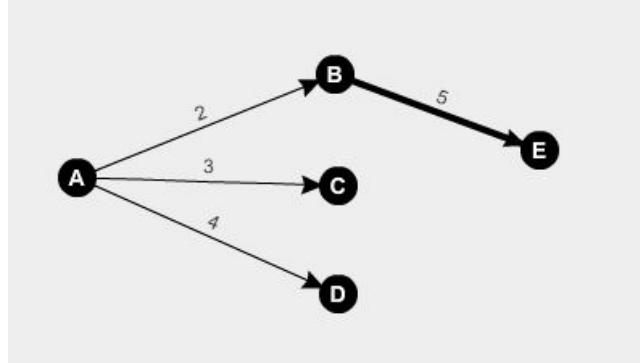


Figure 3: New Graph with added edge

#### 4.9.6 Finding source vertex `source()`

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

```
1 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

```
1 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

```
1 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

```
1  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

```
1  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

```
1  edge_id = get_heaviest_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

```
1  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

```
1  print_slen(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

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

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

```
1  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

```
1 | 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

```
1 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

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

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

```
1 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

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

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

```
1 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

```
1 ->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

```
1 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

```
1 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 list 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

```
1 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 list 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

```
1 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 list 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

```
1 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 list 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

```
1 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

```
1 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

```
1 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/constant/expression evaluating to an edge.

#### 4.10.18 appending to the end of a list of nodes

This is called using

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

which returns a list of nodes with the `node_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

```
1 ilist_id3 = concat(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

```
1 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

```
1 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

```
1 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.

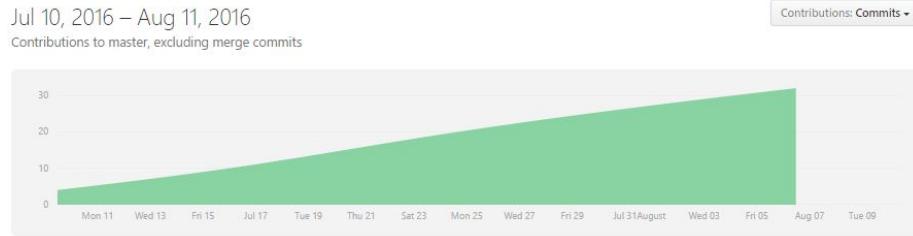


Figure 4: Graph Showing the GitHub Workload

### 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.

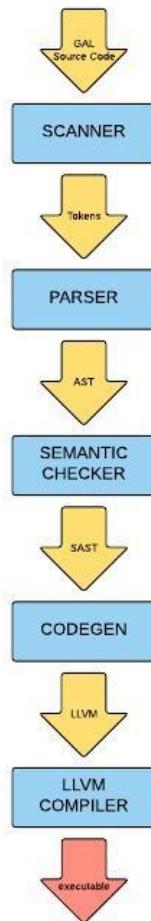


Figure 5: Flowchart showing the architectural design of GAL

## 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 uninitialized 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.

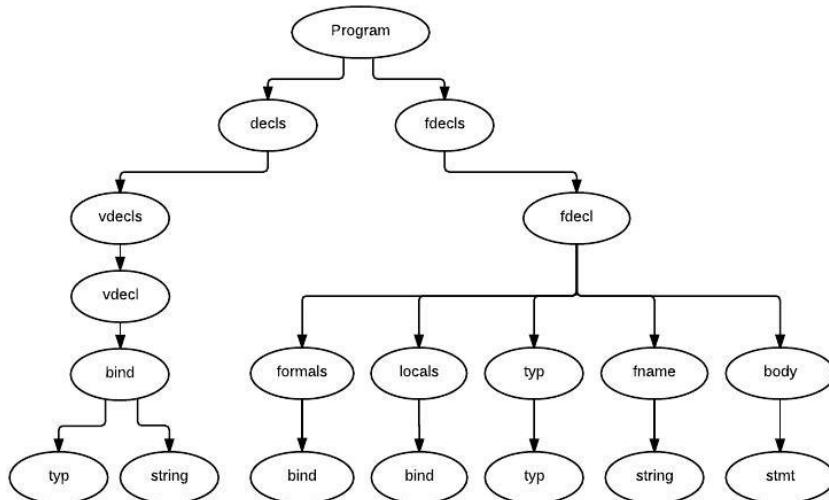


Figure 6: Flowchart Showing the AST

## 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 opinionated - 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

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

```

4           microc code. We hope this is acceptable. *)
5
6 type op = Add | Sub | Mult | Div | Equal | Neq |
7     Less | Leq | Greater | Geq | And | Or
8
9 (* List and Edge here are different from below *)
10 type uop = Not
11
12 type typ = Int | String | Edge | Void
13     | EListtyp | SListtyp | IListtyp | NListtyp
14     | EmptyListtyp | Nothing
15
16
17 type bind = typ * string
18
19 type expr = Litint of int
20     | Litstr of string
21     | Id of string
22     | Binop of expr * op * expr
23     | Assign of string * expr
24     | Noexpr
25     | Unop of uop * expr
26     | Call of string * expr list
27     | Edgedcl of expr * expr * expr
28     | Listdcl of expr list
29     (* Localdecl of typ * string *)
30 (* Added to support local decls *)
31 (*MIGHT HAVE ISSUES HERE, alternative expr list*)
32
33 type stmt =
34     | Localdecl of typ * string
35     | Block of stmt list
36     | Expr of expr
37     | If of expr * stmt * stmt (*MIGHT NOT NEED ELSE ALL THE
TIME*)
38     | For of expr * expr * expr * stmt
39     | While of expr * stmt
40     | Return of expr
41
42
43 type func_decl = {
44
45     typ : typ;
46     fname : string;
47     formals : bind list;
48     locals : bind list;
49     body : stmt list;
50 }
51
52 type program = bind list * func_decl list

```

## 9.2 scanner.mll

```

1 (*Ocamlex scanner for GAL*)
2 (* Authors: Donovan Chan, Andrew Feather, Macrina Lobo,
3    Anton Nefedenkov
4    Note: This code was writte on top of Prof. Edwards's
5         microc code. We hope this is acceptable. *)
6
7 { open Parser }
8
9 rule token = parse

```

```

10 [ , ' \t ' '\r' '\n'] { token lexbuf } (* Whitespace *)
11 /*/* { comment lexbuf } (* Comments *)
12 '(' { LPAREN }
13 ')' { RPAREN }
14 '[' { LSQBRACE }
15 ']' { RSQBRACE }
16 '{' { LBRACE }
17 '}' { RBRACE }
18 '|' { BAR }
19 ':' { COLON }
20 ';' { SEMI }
21 ',' { COMMA }
22 '+' { PLUS }
23 '-' { MINUS }
24 '*' { TIMES }
25 '/' { DIVIDE }
26 '=' { ASSIGN }
27 ":" { LISTSEP }
28 "==" { EQ }
29 "!=" { NEQ }
30 '<' { LT }
31 "<=" { LEQ }
32 '>' { GT }
33 ">=" { GEQ }
34 "&&" { AND }
35 "||" { OR }
36 "!" { NOT }
37 "while" { WHILE }
38 "if" { IF }
39 "else" { ELSE }
40 "for" { FOR }
41 "return" { RETURN }
42 "slist" { SLISTT }
43 "node" { ELISTT }
44 "ilist" { ILISTT }
45 "elist" { ELISTT }
46 "nlist" { NLISTT }
47 "edge" { EDGE }
48 "int" { INT }
49 "string" { STRING }
50 ['0'-'9']+ as lxm { LITINT(int_of_string lxm) }
51 ['a'-'z' 'A'-'Z'][ 'a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
52 "''" (([ ' ' - '!' '#' - '[' ']' - '^' ]*) as s ) '"" { LITSTR(s) } (*
   * or + we have no idea*)
53 eof { EOF }
54 _ as char { raise (Failure("illegal character " ^ Char.escaped
   char)) }
55
56 and comment = parse
57 /*/* { token lexbuf }
58 _ { comment lexbuf }

```

### 9.3 parser.mly

```

1 %{
2 (* Authors: Donovan Chan, Andrew Feather, Macrina Lobo,
3    Anton Nefedenkov
4 Note: This code was writte on top of Prof. Edwards's
5       microc code. We hope this is acceptable. *)
6 open Ast
7 open Help

```

```

8      let build_edge ~src (weight, dst) =
9          Edgedcl(src, weight, dst)
10         %}
11
12 %token SEMI LPAREN RPAREN LSQBRACE RSQBRACE LBRACE RBRACE BAR COLON
13             LISTSEP COMMA
14 %token EPLUS EMINUS PLUS MINUS TIMES DIVIDE ASSIGN NOT
15 %token EQ LT LEQ GT GEQ AND OR NEQ
16 %token RETURN IF ELSE FOR INT STRING EDGE SLISTT NLISTT ELISTT
17             ILISTT DEFINE WHILE
18 %token <int> LITINT
19 %token <string> ID
20 %token <string> LITSTR
21 %token EOF
22
23
24 %right ASSIGN
25 %left OR
26 %left AND
27 %left EQ NEQ
28 %left LT GT LEQ GEQ
29 %left PLUS MINUS
30 %left TIMES DIVIDE
31 %right NOT
32
33 %start program
34 %type <Ast.program> program
35 %%
36
37 program: decls EOF { $1 }
38
39 decls: /*nothing */ { [] ,[] }
40     | decls vdecl { ($2 :: fst $1), snd $1 }
41     | decls fdecl { fst $1, ($2 :: snd $1) }
42
43 vdecl: typ ID SEMI { ($1, $2) }
44
45 fdecl:
46     typ ID LPAREN formals_opts RPAREN LBRACE func_body RBRACE
47     {{ typ = $1; fname = $2; formals = $4;
48       locals = Help.get_vardecls [] $7;
49       body = $7 }}
50
51 formals_opts:
52     /* nothing */ { [] }
53     | formal_list { List.rev $1 }
54
55 formal_list: typ ID { [($1,$2)] }
56     | formal_list COMMA typ ID { ($3,$4) :: $1 }
57
58 typ:
59     INT { Int }
60     | STRING { String }
61     | SLISTT { SListtyp }
62     | EDGE { Edge }
63     | NLISTT { NListtyp }
64     | ELISTT { EListtyp }
65     | ILISTT { IListtyp }
66
67 func_body:

```

```

68     /*nothing*/      { [] }
69     | func_body stmt    { $2 :: $1 }
70
71
72 stmt_list:
73     /*nothing*/ { [] }
74     | stmt_list stmt { $2 :: $1 }
75
76             /*DOESNT ALLOW RETURN of Nothing*/
77 stmt:
78     typ ID SEMI           { Localdecl($1, $2)}
79     | expr SEMI            { Expr $1 }
80     | RETURN expr SEMI    { Return $2 }
81     | LBRACE stmt_list RBRACE { Block(List.rev $2) }
82     | IF LPAREN expr RPAREN stmt ELSE stmt { If($3, $5, $7) }
83     | FOR LPAREN expr_opt SEMI expr SEMI expr_opt RPAREN stmt
84             { For($3,$5,$7,$9) }
85     | WHILE LPAREN expr RPAREN stmt { While($3, $5) }
86
87
88 list_list: /*nothing*/ { [] }
89     | listdecl { List.rev $1 }
90
91 listdecl:
92     expr { [$1] }
93     | listdecl LISTSEP expr { $3 :: $1 }
94
95 node_syntax:
96     expr COLON w_dst_list { List.map (build_edge ~src:$1) $3}
97
98 w_dst_list:
99     expr COMMA expr { [($1, $3)]}
100    | expr COMMA expr COMMA w_dst_list {($1, $3)::$5}
101
102 expr:
103     /* typ ID           { Localdecl($1, $2)} */
104     | BAR node_syntax BAR {Listdcl($2)}
105     | LITINT   { Litint($1) }
106     | ID       { Id($1) }
107     | LITSTR   { Litstr($1) }
108     | BAR expr COMMA expr COMMA expr BAR { Edgedcl($2,$4,$6) }
109     | LSQBRACE list_list RSQBRACE { Listdcl($2) }
110     | expr PLUS  expr { Binop($1, Add, $3) }
111     | expr MINUS expr { Binop($1, Sub, $3) }
112     | expr TIMES expr { Binop($1, Mult, $3) }
113     | expr DIVIDE expr { Binop($1, Div, $3) }
114     | expr EQ    expr { Binop($1, Equal, $3) }
115     | expr NEQ   expr { Binop($1, Neq, $3) }
116     | expr LT    expr { Binop($1, Less, $3) }
117     | expr LEQ   expr { Binop($1, Leq, $3) }
118     | expr GT    expr { Binop($1, Greater, $3) }
119     | expr GEQ   expr { Binop($1, Geq, $3) }
120     | expr AND   expr { Binop($1, And, $3) }
121     | expr OR    expr { Binop($1, Or, $3) }
122     | NOT expr { Unop(Not, $2) }
123     | ID ASSIGN expr { Assign($1, $3) }
124     | LPAREN expr RPAREN { $2 }
125     | ID LPAREN actuals_opt RPAREN { Call($1, $3) }
126
127 expr_opt: /*nothing*/ { Noexpr }
128     | expr { $1 }
129

```

```

130  actuals_opt: /*nothing*/ { [] }
131  | actuals_list { List.rev $1 }
132
133  actuals_list:
134    expr { [$1] }
135  | actuals_list COMMA expr { $3 :: $1 }

```

## 9.4 semant.ml

```

1 (* Authors: Donovan Chan, Andrew Feather, Macrina Lobo,
2      Anton Nefedenkov
3      Note: This code was written on top of Prof. Edwards's
4            microc code. We hope this is acceptable. *)
5
6 open Ast;;
7
8 module StringMap = Map.Make(String);;
9 let m = StringMap.empty;;
10
11 (* Error messages of the exceptions *)
12 let dup_global_exp = " duplicate global ";;
13 let dup_local_exp = " duplicate local ";;
14 let dup_formal_exp = " duplicate formal arg ";;
15 let dup_func_exp = " duplicate function name ";;
16 let builtin_decl_exp = " cannot redefine ";;
17 let main_undef_exp = " main not defined ";;
18
19 (* Names of built in functions can be added below *)
20 let builtins_list =
21   [ "print_int"; "print_str";
22     "length"; "source"; "dest"; "pop"; "weight"; "print_endline";
23     "peek" ];;
24
25 (* Built in decls *)
26 let print_int_fdcl =
27   { typ = Int; fname = "print_int"; formals = [(Int, "a")];
28     locals = []; body = [] };;
29
30 let print_str_fdcl =
31   { typ = String; fname = "print_str"; formals = [(String, "a")];
32     locals = []; body = [] };;
33
34 let slength_fdcl =
35   { typ = Int; fname = "slength"; formals = [(SListtyp, "a")];
36     locals = []; body = [] };;
37
38 let elength_fdcl =
39   { typ = Int; fname = "elength"; formals = [(EListtyp, "a")];
40     locals = []; body = [] };;
41
42 let ilength_fdcl =
43   { typ = Int; fname = "ilength"; formals = [(IListtyp, "a")];
44     locals = []; body = [] };;
45
46 let nlength_fdcl =
47   { typ = Int; fname = "nlength"; formals = [(NListtyp, "a")];
48     locals = []; body = [] };;
49
50 let dest_fdcl =
51   { typ = String; fname = "dest"; formals = [(Edge, "a")];
52     locals = []; body = [] };;

```

```

53 let source_fdcl =
54   { typ = String; fname = "source"; formals = [(Edge, "a")];
55     locals = []; body = []};;
56
57 let weight_fdcl =
58   { typ = Int; fname = "weight"; formals = [(Edge, "a")];
59     locals = []; body = []};;
60
61 let print_endline_fdcl =
62   { typ = Int; fname = "print_endline"; formals = [];
63     locals = []; body = []};;
64
65 (* This function needs discussion *)
66 let spos_fdcl =
67   { typ = SListtyp; fname = "spos"; formals = [(SListtyp, "a")];
68     locals = []; body = []};;
69
70 let ipop_fdcl =
71   { typ = IListtyp; fname = "ipop"; formals = [(IListtyp, "a")];
72     locals = []; body = []};;
73
74 let epop_fdcl =
75   { typ = EListtyp; fname = "epop"; formals = [(EListtyp, "a")];
76     locals = []; body = []};;
77
78 let npop_fdcl =
79   { typ = NListtyp; fname = "npop"; formals = [(NListtyp, "a")];
80     locals = []; body = []};;
81
82 let speek_fdcl =
83   { typ = String; fname = "speek"; formals = [(SListtyp, "a")];
84     locals = []; body = []};;
85
86 let ipeek_fdcl =
87   { typ = Int; fname = "ipeek"; formals = [(IListtyp, "a")];
88     locals = []; body = []};;
89
90 let epeek_fdcl =
91   { typ = Edge; fname = "epeek"; formals = [(EListtyp, "a")];
92     locals = []; body = []};;
93
94 let npeek_fdcl =
95   { typ = EListtyp; fname = "npeek"; formals = [(NListtyp, "a")];
96     locals = []; body = []};;
97
98 let snext_fdcl =
99   { typ = SListtyp; fname = "snext"; formals = [(SListtyp, "a")];
100    locals = []; body = []};;
101
102 let enext_fdcl =
103   { typ = EListtyp; fname = "enext"; formals = [(EListtyp, "a")];
104    locals = []; body = []};;
105
106 let inext_fdcl =
107   { typ = IListtyp; fname = "inext"; formals = [(IListtyp, "a")];
108    locals = []; body = []};;
109
110 let nnext_fdcl =
111   { typ = NListtyp; fname = "nnext"; formals = [(NListtyp, "a")];
112    locals = []; body = []};;
113
114 let sadd_fdcl =

```

```

115 { typ = SListtyp; fname = "sadd"; formals = [(String, "b")]; (
116   SListtyp, "a"));
117   locals = []; body = [] };;
118 let eadd_fdcl =
119 { typ = EListtyp; fname = "eadd"; formals = [(Edge, "b")]; (
120   EListtyp, "a"));
121   locals = []; body = [] };;
122 let iadd_fdcl =
123 { typ = IListtyp; fname = "iadd"; formals = [(Int, "b")]; (
124   IListtyp, "a"));
125   locals = []; body = [] };;
126 let nadd_fdcl =
127 { typ = NListtyp; fname = "nadd"; formals = [(EListtyp, "b")]; (
128   NListtyp, "a"));
129   locals = []; body = [] };;
130 let str_comp_fdcl =
131 { typ = Int; fname = "streq"; formals = [(String, "a"); (String,
132   "b")];
133   locals = []; body = [] };;
134
135 let builtin_fdcl_list =
136 [ print_int_fdcl; print_str_fdcl; slength_fdcl; dest_fdcl;
137   source_fdcl; spop_fdcl; weight_fdcl; print_endline_fdcl;
138   speek_fdcl; ipeek_fdcl; epeek_fdcl; snext_fdcl; elength_fdcl;
139   enext_fdcl; inext_fdcl; ilength_fdcl; nnexxt_fdcl; npeek_fdcl;
140   nlengtth_fdcl; sadd_fdcl; eadd_fdcl; iadd_fdcl; nadd_fdcl;
141   str_comp_fdcl; ipop_fdcl; epop_fdcl; npop_fdcl ];;
142
143 (* Static semantic checker of the program. Will return void
144  on success. Raise an exception otherwise. Checks first the
145  globals, then the functions. *)
146
147
148 (* Reports if duplicates present duplicates. *)
149 let report_duplicate_exception_msg list func_name =
150 (* Helper that build a list of duplicates *)
151 let rec helper dupls = function
152   [] -> List.rev dupls;
153   | n1 :: n2 :: tl when n1 = n2 -> helper (n2 :: dupls) tl
154   | _ :: tl -> helper dupls tl
155
156 (* Another helper, that uniq's the duples
157  (if not already uniq) Works on sorted lists! *)
158 in let rec uniq result = function
159   [] -> result
160   | hd :: [] -> uniq (hd :: result) []
161   | hd1 :: (hd2 :: tl as tail) ->
162     if hd1 = hd2 then uniq result tail
163     else uniq (hd1 :: result) tail
164
165 (* Get a list of duplicates *)
166 in let dupls = uniq [] (helper [] (List.sort compare list))
167
168 (* If the list is not an empty list *)
169 in if dupls <> [] then
170   match func_name with

```

```

172 | "" ->
173   (exception_msg ^ (String.concat " " dupls) )
174 | _ ->
175   (exception_msg ^ (String.concat " " dupls) ^ " in " ^ func_name
176   )
177 else ""
178
179
180 (* Returns a list of lists of locals *)
181 let rec extract_locals local_vars = function
182   [] -> List.rev local_vars
183   | hd:: tl -> extract_locals
184     (( hd.fname, (List.map snd hd.locals))::local_vars) tl
185 ;;
186
187 (* Extracts formal arguments *)
188 let rec extract_formals formals = function
189   [] -> List.rev formals
190   | hd:: tl -> extract_formals
191     (( hd.fname, (List.map snd hd.formals))::formals) tl
192
193 (* Helper functions extracts good stuff from list of funcs *)
194 let rec func_duplicates exp_msg exception_list = function
195   [] -> List.rev exception_list
196   | (name, var_list):: tail ->
197     func_duplicates
198       exp_msg
199       ((report_duplicate exp_msg var_list name)::exception_list)
200     tail
201 ;;
202
203 (* Function get rid of empty string in exception list *)
204 let rec purify_exp_list result = function
205   [] -> List.rev result
206   | hd:: tl when hd <> "" -> purify_exp_list (hd::result) tl
207   | _:: tl -> purify_exp_list result tl
208 ;;
209
210 (* List of built ins is the implicit argument here*)
211 let rec check_builtins_defs exp_list expmsg funcs = function
212   [] -> List.rev exp_list
213   | hd:: tl ->
214     if (List.mem hd funcs) then
215       let exp = expmsg ^ hd in
216       check_builtins_defs (exp::exp_list) expmsg funcs tl
217     else
218       check_builtins_defs exp_list expmsg funcs tl
219 ;;
220
221 (* Helper function to print types *)
222 let string_of_typ asttype = match asttype with
223   | Int -> " int "
224   | String -> " string "
225   | SListtyp -> " slist "
226   | Edge -> " edge "
227   | Void -> " (bad expression) "
228   | EListtyp -> " elist "
229   | NListtyp -> " nlist "
230   | IListtyp -> " ilist "
231
232 (* Function checks bunch of fun stuff in the function structure *)

```

```

233 let check_func exp_list glob_map func_decl funcs_map =
234
235 (* Function returns the type of the identifier *)
236 let get_type_of_id exp_list vars_map id =
237   (* StringMap.iter
238     (fun name typename -> (print_string (name ^ "\n")) ) )
239   vars_map; *)
240 try (StringMap.find id vars_map, exp_list)
241 with Not_found ->
242   (Void, (" in " ^ func_decl.fname ^ " var: " ^
243           " unknown identifier " ^ id):: exp_list)
244
245 (* Helper will return a list of exceptions *)
246 in let rec get_expression_type vars_map exp_list = function
247 | Litstr(_) -> (String, exp_list)
248 | Litint(_) -> (Int, exp_list)
249 | Id(name) -> get_type_of_id exp_list vars_map name
250 | Binop(e1, op, e2) (* as e *) ->
251   let (v1, exp_list) = get_expression_type vars_map exp_list e1
252   in
253     let (v2, exp_list) = get_expression_type vars_map exp_list e2
254     in (match op with
255         (* Integer operators *)
256         | Add | Sub | Mult | Div | Equal | Less | Leq
257         | Greater | Geq | And | Or | Neq
258         when (v1 = Int && v2 = Int) -> (Int, exp_list)
259         (* List operators *)
260         (* | Eadd | Esub when v1 = Listtyp && v2 = Listtyp -> (
261             Listtyp, exp_list) *)
262         | _ -> (Void, (" in " ^ func_decl.fname ^ " expr: " ^
263                     " illegal binary op "): exp_list)
264     | Unop(op, e1) -> get_expression_type vars_map exp_list e1
265     | Noexpr -> (Void, exp_list) (* Need to check how Noexp is used *)
266
267 | Assign(var, e) (* as ex *) ->
268   (* print_string (" assignment to " ^ var ^ "\n"); *)
269   let (lt, exp_list) = get_type_of_id exp_list vars_map var in
270   let (rt, exp_list) = get_expression_type vars_map exp_list e
271   in if (lt <> rt && rt <> EmptyListtyp) || rt = Void then
272     (Void, (" in " ^ func_decl.fname ^ " expr: " ^
273             " illegal assignment to variable " ^ var): exp_list)
274   else (rt, exp_list)
275 | Edgedcl(e1, e2, e3) ->
276   let (v1, exp_list) = get_expression_type vars_map exp_list e1
277   in
278     let (v2, exp_list) = get_expression_type vars_map exp_list e2
279   in
280     let (v3, exp_list) = get_expression_type vars_map exp_list e3
281     in
282       if v1 = String && v3 = String && v2 = Int then
283         (Edge, exp_list)
284       else
285         (Void, (" in " ^ func_decl.fname ^ " edge: " ^
286                 " bad types "): exp_list)
287 | Listdcl(elist) ->
288   (* Get the type of the first element of the list *)
289   let get_elmt_type decl_list = match decl_list with
290   | [] -> Nothing
291   | hd::tl ->
292     let (v1, exp_list) = get_expression_type vars_map exp_list
293     hd in
294     v1

```

```

288   in
289
290   (* Get the type of the list *)
291   let get_list_type elmt_type = match elmt_type with
292   | Nothing    -> EmptyListtyp
293   | Edge       -> EListtyp
294   | String     -> SListtyp
295   | Int        -> IListtyp
296   | EListtyp   -> NListtyp
297   | _          -> raise (Failure("in list decl process"))
298
299   in
300
301   let rec check_list exp_list = function
302   [] -> List.rev exp_list
303   | hd::[] ->
304     let (v1, exp_list) = get_expression_type vars_map exp_list
305     hd in
306     check_list exp_list []
307   | hd1::(hd2::tl as tail) ->
308     let (v1, exp_list) = get_expression_type vars_map exp_list
309     hd1 in
310     let (v2, exp_list) = get_expression_type vars_map exp_list
311     hd2 in
312     if v1 <> v2 then
313       check_list
314         ((`in `^ func_decl.fname `^ " list: " ^
315           " bad types of expressions "):exp_list)
316     []
317   else
318     check_list exp_list tail
319   in
320
321   let list_exp_list = check_list [] elist
322   in if list_exp_list <> [] then
323     (Void, (exp_list @ list_exp_list))
324   else
325     let elmt_type = get_elmt_type elist in
326     let list_typ = get_list_type elmt_type in
327     (list_typ, exp_list)
328
329
330   (* CARE HERE, NOT FINISHED AT ALL *)
331   | Call(fname, actuals) ->
332     try let fd = StringMap.find fname funcs_map
333     in if List.length actuals <> List.length fd.formals then
334       (Void, (
335         " in " ^ func_decl.fname ^ " fcall: " ^
336         fd.fname ^ " expects " ^
337         (string_of_int (List.length fd.formals)) ^
338         " arguments " ):exp_list)
339     else
340       (* Helper comparing actuals to formals *)
341       let rec check_actuals formals exp_list = function
342       [] -> List.rev exp_list
343       | actual_name::tla -> match formals with
344         | [] -> raise (Failure(" bad. contact me "))
345         | hdf::tlf ->
346           let (actual_typ, exp_list) = get_expression_type
347             vars_map exp_list actual_name in
348           let (formal_typ, _) = hdf in
349           if formal_typ = actual_typ then

```

```

347         check_actualls tlf exp_list tla
348     else
349         (" in " ^ func_decl.fname ^
350          " fcall: wrong argument type in " ^
351          fname ^ " call "):: exp_list
352
353     in let exp_list = check_actualls
354       (fd.formals)
355       exp_list
356       actualls
357     in (fd.typ, exp_list)
358
359     with Not_found ->
360       (Void, (" in " ^ func_decl.fname ^ " fcall;" ^
361              " function " ^ fname ^ " not defined "):: exp_list)
362
363   | _ -> (Void, exp_list)
364
365 (* In short, helper walks through the ast checking all kind of
366    things *)
367 in let rec helper vars_map exp_list = function
368   | [] -> List.rev exp_list
369   | hd::tl -> (match hd with
370     | Localdecl(typname, name) ->
371         (* print_string ("locvar " ^ name ^ " added \n"); *)
372         helper (StringMap.add name typname vars_map) exp_list tl
373     | Expr(e) ->
374         (* print_string " checking expression "; *)
375         let (typename, exp_list) = get_expression_type vars_map
376         exp_list e in
377           helper vars_map exp_list tl
378     | If(p, s1, s2) ->
379         let (ptype, exp_list) = get_expression_type vars_map
380         exp_list p in
381           if ptype <> Int then
382             helper vars_map
383             (( " in " ^ func_decl.fname ^
384               " if: predicate of type " ^ string_of_typ ptype )
385               ::(helper vars_map (helper vars_map exp_list [s1]) [
386                 s2])) tl
387           else
388             helper vars_map
389             (helper vars_map (helper vars_map exp_list [s1]) [s2])
390           tl
391   | For(e1, e2, e3, s) ->
392     let (e1_typ, exp_list) = get_expression_type vars_map
393     exp_list e1 in
394     let (e2_typ, exp_list) = get_expression_type vars_map
395     exp_list e2 in
396     let (e3_typ, exp_list) = get_expression_type vars_map
397     exp_list e3 in
398       if e1_typ = e3_typ && e2_typ = Int then
399         helper vars_map (helper vars_map exp_list [s]) tl
400       else
401         helper vars_map
402         (( " in " ^ func_decl.fname ^
403           " for loop: bad types of expressions. Type * Int * "
404             Type expected. ")
405             :: exp_list)

```

```

400           tl
401     | While(cond, loop) ->
402       let (cond_typ, exp_list) = get_expression_type vars_map
403     exp_list cond in
404       if cond_typ = Int then
405         helper vars_map (helper vars_map exp_list [loop]) tl
406       else
407         helper vars_map
408         ((`in` ^ func_decl.fname ^ `while loop: bad type of conditional expression`)
409          :: exp_list)
410         tl
411
412     | Block(sl) -> (match sl with
413       | [Return(_) as s] ->
414         helper vars_map (helper vars_map exp_list [s]) tl
415       | Return(_) :: _ ->
416         helper vars_map
417         ((`in` ^ func_decl.fname ^ `ret: nothing can come
418           after return` ^ `in a given block`)):: exp_list)
419         tl
420     | Block(sl)::ss ->
421       helper vars_map
422       (helper vars_map exp_list (sl @ ss))
423       tl
424     | s::sl as stl-> helper vars_map
425       (helper vars_map exp_list stl)
426       tl
427     | [] -> helper vars_map exp_list tl
428
429   )
430
431   (* Make sure that tl is an empty list at this point,
432 otherwise throw exception *)
433   | Return(e) -> let (rettytyp, exp_list) = get_expression_type
434     vars_map exp_list e
435       in ifrettytyp = func_decl.typ then
436         helper vars_map exp_list tl
437       else (func_decl.fname ^ `ret: expected return type` ^
438             (string_of_typ func_decl.typ) ^ `but expression is
439             of type` ^
440             (string_of_typrettytyp)):: exp_list
441       | _ -> helper vars_map exp_list [] (* Placeholder *)
442
443   )
444
445   in let glob_forms_map = List.fold_left
446     (fun m (typename, name) -> StringMap.add name typename m)
447     glob_forms_map
448     func_decl.formals
449
450   in helper glob_forms_map exp_list (List.rev func_decl.body)
451
452 let rec check_functions exp_list glob_forms_map funcs_map = function
453   | [] -> List.rev exp_list
454   | hd::tl -> check_functions
455     (check_func exp_list glob_forms_map hd funcs_map)
456     glob_forms_map
457     funcs_map
458     tl

```

```

457 (* The thing that does all the checks *)
458 let check (globals, funcs) =
459
460   (* Check duplicate globals *)
461   let global_dup_exp =
462     report_duplicate dup_global_exp (List.map snd globals) ""
463
464   (* Check the local variables *)
465   in let exp = global_dup_exp :::
466     ((func_duplicates dup_local_exp []
467       (extract_locals [] funcs)))
468
469   (* Check the formal arguments *)
470   in let exp = func_duplicates
471     dup_formal_exp
472     exp
473     (extract_formals [] funcs)
474
475   (* Check for func name duplicates *)
476   in let exp = (report_duplicate
477     dup_func_exp
478     (List.map (fun n -> n.fname) funcs)
479     "") :: exp
480
481   (* Check if built ins were redefined *)
482   in let exp = (check_builtins_defs
483     exp
484     builtin_decl_exp
485     (List.map (fun n -> n.fname) funcs)
486     builtins_list)
487
488   (* Add builtins to the map *)
489   in let builtin_decls = List.fold_left
490     (fun m fd -> StringMap.add fd.fname fd m)
491     StringMap.empty
492     builtin_fdcl_list
493
494   (* Add user declared functions to the map *)
495   in let fdecl_map = List.fold_left
496     (fun m fd -> StringMap.add fd.fname fd m)
497     builtin_decls
498     funcs
499
500   (* Check if main was properly declared *)
501   in let exp =
502     try ignore (StringMap.find "main" fdecl_map); exp
503     with Not.found -> main_undef_exp :: exp
504
505   (* Get a map of globals for future use in symbol table
506      composition for each function *)
507   in let glob_map = List.fold_left
508     (fun m (typename, name) -> StringMap.add name typename m)
509     StringMap.empty
510     globals
511
512   in let exp = check_functions exp glob_map fdecl_map funcs
513
514
515   (* Get rid of elements containing empty sstring *)
516   in purify_exp_list [] exp
517
518

```

```
519
520 (*in exp::List.map ( report_duplicate dup_local_exp )
521   (extract_locals [] funcs) *)
522
523;;
```

## 9.5 codegen.ml

```

1 (* Authors: Donovan Chan, Andrew Feather, Macrina Lobo,
2    Anton Nefedenkov
3 Note: This code was written on top of Prof. Edwards's
4       microc code. We hope this is acceptable. *)
5
6
7 module A = Ast
8 module L = LLVM
9 module P = Printf
10 module StringMap = Map.Make(String)
11
12
13 let translate (globals, functions) =
14
15   let the_funcs_map = StringMap.empty in
16   let the_funcs_map =
17     List.fold_left
18       (fun map fdecl -> StringMap.add fdecl.A.fname fdecl.A.typ map)
19       the_funcs_map
20       functions
21
22 in
23
24 (* Holding global string constants *)
25 let glob_str_const_hash = Hashtbl.create 200 in
26
27 (* Build a context and the module *)
28 let context = L.global_context () in
29 let the_module = L.create_module context "GAL"
30
31 (* Few helper functions returning the types *)
32 and i32_t = L.i32_type context (* Integer *)
33 and i8_t = L.i8_type context (* Char *)
34 and i1_t = L.i1_type context (* Needed for predicates *)
35
36   in let i8_p_t = L.pointer_type i8_t (* Pointer *)
37   in let edge_t = L.struct_type context (* Edge type *)
38       (Array.of_list [i8_p_t; i32_t; i8_p_t])
39
40   in let one = L.const_int i32_t 1
41
42   in let empty_node_t = L.named_struct_type context "empty" in
43   L.struct_set_body empty_node_t (Array.of_list [L.pointer_type
44         empty_node_t; L.pointer_type i1_t; i32_t]) true;
45
46 let node_t = L.named_struct_type context "node" in
47   L.struct_set_body node_t (Array.of_list [L.pointer_type node_t;
48         i8_p_t; i32_t]) true;
49
50 let e_node_t = L.named_struct_type context "enode" in
51   L.struct_set_body e_node_t (Array.of_list [L.pointer_type
52         e_node_t; L.pointer_type edge_t; i32_t]) true;
53
54 let i_node_t = L.named_struct_type context "inode" in
55   L.struct_set_body i_node_t (Array.of_list [L.pointer_type
56         i_node_t; L.pointer_type edge_t; i32_t]) true;

```

```

    i_node_t; i32_t; i32_t ]) true;
52
53 let n_node_t = L.named_struct_type context "nnode" in
54   L.struct_set_body n_node_t (Array.of_list [L.pointer_type
55     n_node_t; L.pointer_type e_node_t; i32_t ]) true;
56
57 (* Pattern match on A.typ returning a llvm type *)
58 let ltype_of_typ ltyp = match ltyp with
59   | A.Int    -> i32_t
60   | A.Edge   -> L.pointer_type edge_t
61   | A.String  -> i8_p_t
62   | A.EmptyListtyp -> L.pointer_type empty_node_t
63   | A.SListtyp -> L.pointer_type node_t
64   | A.EListtyp -> L.pointer_type e_node_t
65   | A.IListtyp -> L.pointer_type i_node_t
66   | A.NListtyp -> L.pointer_type n_node_t
67   | _ -> raise (Failure ("Type not implemented\n"))
68
69 in let list_type_from_type ocaml_type = match ocaml_type with
70   | A.Int      -> i_node_t
71   | A.String    -> node_t
72   | A.Edge      -> e_node_t
73   | A.EListtyp  -> n_node_t
74   | _ -> raise (Failure("such lists are not supported "))
75
76 (* Global variables *)
77 in let global_vars =
78   let global_var m (t, n) =
79     (* Initialize the global variable to 000...000 *)
80     let init = L.const_int (ltype_of_typ t) 0
81     (* Bind the gloabal to its name and its lglobal *)
82     in StringMap.add n (L.define_global n init the_module) m
83   in List.fold_left global_var StringMap.empty globals
84
85 (***** In built functions below *****)
86
87 (* Function llvm type *)
88 in let printf_t = L.var_arg_function_type i32_t [| L.pointer_type
89   i8_t |]
90 (* Function declaration *)
91 in let printf_func = L.declare_function "printf" printf_t
92   the_module
93
94 (* Builds a user defined function *)
95 in let function_decls =
96   let function_decl map fdecl =
97     (* Get the types of the formals in a list *)
98     let formal_types =
99       Array.of_list
100      (List.map (fun (t, _) -> ltype_of_typ t) fdecl.A.formals)
101
102     (* Get the llvm function type with known return and formals
103     types *)
104     in let ftype =
105       L.function_type
106       (ltype_of_typ fdecl.A.typ)
107       formal_types
108
109     (* Bind the name of the function to (llvm function , ast
110     function) *)
111     in StringMap.add fdecl.A.fname

```

```

108     (L.define_function fdecl.A.fname ftype the_module , fdecl)
109     map)
110 (* Populate the map by folding the list of functions *)
111 in List.fold_left function_decl StringMap.empty functions
112
113 (* Builds the function body in the module *)
114 in let build_function_body fdecl =
115
116   let ocaml_local_hash = Hashtbl.create 100 in
117   let local_hash = Hashtbl.create 100 in
118
119   (* Get the llvm function from the map *)
120   let (the_function, _) = StringMap.find fdecl.A.fname
121   function_decls in
122
123   (* Direct the builder to the right place *)
124   let builder = L.builder_at_end context (L.entry_block
125   the_function) in
126
127   (* BFotmat string needed for printing. *)
128   (* Will put format string into %tmt in global area *)
129   let int_format_string = L.build_global_stringptr "%d" "ifs"
130   builder in
131   let string_format_string = L.build_global_stringptr "%s" "sfs"
132   builder in
133   let endline_format_string = L.build_global_stringptr "%s\n" "efs"
134   builder in
135
136   let _ =
137
138     let rec enumerate i enumed_l = function
139       | [] -> List.rev enumed_l
140       | hd::tl -> enumerate (i + 1) ((hd, i)::enumed_l) tl
141     in
142
143     let add_formal (t, n) (p, _) =
144       L.set_value_name n p;
145       let local = L.build_alloca (ltype_of_typ t) n builder in
146       ignore (L.build_store p local builder);
147       Hashtbl.add local_hash n local;
148       Hashtbl.add ocaml_local_hash n t;
149     in
150
151     let params = enumerate 0 [] (Array.to_list (L.params
152     the_function))
153
154     in List.iter2 add_formal fdecl.A.formals params
155
156   in let add_local builder (t, n) =
157     let local_var = L.build_alloca (ltype_of_typ t) n builder
158     in Hashtbl.add local_hash n local_var
159 (*
160   in let add_local_list builder ltype n =
161     let local_var = L.build_alloca (ltype) n builder
162     in Hashtbl.add local_hash n local_var *)
163
164   in let lookup name =
165     try Hashtbl.find local_hash name
166     with Not_found -> StringMap.find name global_vars
167
168   in let rec get_node_type expr = match expr with

```

```

164 | A.Litint(_)->i_node_t
165 | A.Litstr(_)->node_t
166 | A.Listdcl(somelist)->
167 |   if somelist = [] then
168 |     raise (Failure("empty list decl"))
169 |   else
170 |     let hd::_ = somelist in get_node_type hd
171 | A.Binop(e1, _, _)->get_node_type e1
172 | A.Edgecl(_)->e_node_t
173 | A.Id(name)->
174 |   let ocaml_type = (Hashtbl.find ocaml_local_hash name)
175 |   in list_type_from_type ocaml_type
176 | A.Call("iadd", _)|A.Call("inext", _)->
177 |   i_node_t
178 | A.Call("eadd", _)|A.Call("enext", _)->
179 |   e_node_t
180 | A.Call("sadd", _)|A.Call("snext", _)->
181 |   node_t
182 | A.Call("nadd", _)|A.Call("nnext", _)->
183 |   n_node_t
184 | A.Call("ilength", _)|A.Call("slength", _)|A.Call("nlengt
185 |   h", _)|A.Call("elength", _)->
186 |   i_node_t
187 | A.Call(fname, _)->
188 |   let ftype = StringMap.find fname the_funcs_map in
189 |   ltype_of_typ ftype
190 |   (* try let fdecl = List.find
191 |      (fun fdecl -> if fdecl.A.fname = fname then true else false
192 |    )
193 |      functions
194 |      in (ltype_of_typ fdecl.A.typ) with Not_found -> in *)
195 |   | _ -> raise (Failure(" type not supported in list "))
196
197 (* We can now describe the action to be taken on ast traversal
198 *)
199 (* Going to first pattern match on the list of expressions *)
200 in let rec expr_builder e =
201
202   (* Helper to add element to the list *)
203   let add_element head_p new_node_p =
204     let new_node_next_field_pointer =
205       L.build_struct_gep new_node_p 0 "" builder in
206       ignore (L.build_store head_p
207             new_node_next_field_pointer builder);
208       new_node_p
209
210   in let add_payload node_p payload_p =
211     let node_payload_pointer =
212       L.build_struct_gep node_p 1 "" builder in
213       ignore (L.build_store payload_p
214             node_payload_pointer builder);
215       node_p
216
217   in let build_node node_type payload =
218     let alloc = L.build_malloc node_type ("") builder in
219     let payload_p = expr_builder payload in
220     add_payload alloc payload_p
221
222   in match e with
223   | A.Litint(i)->L.const_int i32_t i
224   | A.Litstr(str)->

```

```

223   let s = L.build_global_stringptr str str builder in
224   let zero = L.const_int i32_t 0 in
225   let lvalue = L.build_in_bounds_gep s [| zero |] str builder
226   in
227     let lv_str = L.string_of_llvalue s in
228       (* P.printf stderr "%s\n" lv_str; *)
229
230   Hashtbl.add glob_str_const_hash lvalue str;
231   s
232 | A.Edgecl(src, w, dst) ->
233   let src_p = expr builder src
234   and w = expr builder w
235   and dst_p = expr builder dst
236   in let alloc = L.build_malloc edge_t ("") builder
237
238   in let src_field_pointer =
239     L.build_struct_gep alloc 0 "" builder
240   and weight_field_pointer =
241     L.build_struct_gep alloc 1 "" builder
242   and dst_field_pointer =
243     L.build_struct_gep alloc 2 "" builder
244
245   in
246     ignore (L.build_store src_p src_field_pointer builder);
247     ignore (L.build_store dst_p dst_field_pointer builder);
248     ignore (L.build_store w weight_field_pointer builder);
249     L.build_in_bounds_gep alloc [| (L.const_int i32_t 0) |] ""
250   builder
251
252 | A.Listdcl(elist) ->
253   let elist = List.rev elist in
254
255   if (elist = []) then
256     L.const_pointer_null (L.pointer_type empty_node_t)
257     (* raise (Failure("empty list assignment")) *)
258   else
259     let (hd::tl) = elist in
260     let good_node_t = get_node_type hd in
261     let head_node = build_node (good_node_t) hd in
262     let head_node_len_p = L.build_struct_gep head_node 2 ""
263   builder in
264     let head_node_next_p = L.build_struct_gep head_node 0 ""
265   builder in
266     ignore (L.build_store (L.undefined (L.pointer_type
267     good_node_t)) head_node_next_p builder);
268     ignore (L.build_store (expr builder (A.Litint(1)))
269     head_node_len_p builder);
270
271     let rec build_list the_head len = function
272       | [] -> the_head
273       | hd::tl ->(
274         let len = len + 1 in
275         let new_node = build_node good_node_t hd in
276         let new_head = add_element the_head new_node in
277         let new_head_len_p = L.build_struct_gep new_head 2 ""
278   builder in
279         ignore (L.build_store (expr builder (A.Litint(len)))
280     new_head_len_p builder);
281         build_list new_head (len) tl)
282
283     in (build_list head_node 1 tl)
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1897
1898
1899
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1988
1989
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2088
2089
2089
2090
2091
2092
2093
2094
2095
2096
2097
2097
2098
2099
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2188
2189
2189
2190
2191
2192
2193
2194
2195
2196
2197
2197
2198
2199
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2239
2240
2241
2242
2243
2244
2245
2246
2247
2247
2248
2249
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2288
2289
2289
2290
2291
2292
2293
2294
2295
2296
2296
2297
2298
2299
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2379
2380
2381
2382
2383

```

```

277 | A.Id(name) -> L.build_load (lookup name) name builder
278 | A.Assign(name, e) ->
279   let loc_var = lookup name in
280   let e' = (expr builder e) in
281
282   (* Cant add it like this. Need a different comparison. And
283    need to remove
284     old var from the hash map *)
285   if ((L.pointer_type empty_node_t) = (L.type_of e')) then
286   (
287     (* This is the ocaml type of the variable *)
288     let list_type = Hashtbl.find ocaml_local_hash name in
289
290     (* Cant get to the right type for store instruction, so
291      this: *)
292     let get_llvm_node_type ocaml_type = match ocaml_type with
293       | A.SListtyp -> node_t
294       | A.IListtyp -> i_node_t
295       | A.NListtyp -> n_node_t
296       | A.EListtyp -> e_node_t
297       | _ -> raise (Failure("list type not supported"))
298     in
299
300     let llvm_node_t = get_llvm_node_type list_type in
301     let dummy_node = L.build_malloc llvm_node_t ("") builder
302     in
303       let dummy_node_len_p = L.build_struct_gep dummy_node 2 ""
304         builder in
305         ignore (L.build_store (expr builder (A.Litint(0)))
306 dummy_node_len_p builder);
307         ignore (L.build_store dummy_node loc_var builder);
308         e'
309       else
310         (ignore (L.build_store e' (lookup name) builder); e')
311
312   (* Calling builtins below *)
313   | A.Call("print_int", [e]) ->
314     L.build_call printf_func
315     [| int_format_string; (expr builder e) |]
316     "printf"
317     builder
318   | A.Call("print_str", [e]) ->
319     L.build_call printf_func
320     [| string_format_string; (expr builder e) |]
321     "printf"
322     builder
323   | A.Call("print_endline", []) ->
324     L.build_call printf_func
325     [| endline_format_string; (expr builder (A.Litstr(""))) |]
326     "printf"
327     builder
328   | A.Call("source", [e]) ->
329     let src_field_pointer = L.build_struct_gep (expr builder e)
330     0 "" builder
331     in L.build_load src_field_pointer "" builder
332   | A.Call("weight", [e]) ->
333     let weight_field_pointer = L.build_struct_gep (expr builder
334     e) 1 "" builder
335     in L.build_load weight_field_pointer "" builder
336   | A.Call("dest", [e]) ->

```

```

331   let dest_field_pointer = L.build_struct_gep (expr builder e
332 ) 2 "" builder
333     in L.build_load dest_field_pointer "" builder
334     | A.Call("spop", [e]) | A.Call("epop", [e]) | A.Call("ipop",
335 [e]) | A.Call("npop", [e]) ->
336       let head_node_p = (expr builder e) in
337         let head_node_next_node_pointer = L.build_struct_gep
338           head_node_p 0 "" builder in
339           ignore (L.build_free head_node_p builder);
340           L.build_load head_node_next_node_pointer "" builder
341           | A.Call("speek", [e]) | A.Call("ipeek", [e]) | A.Call("epeek",
342 [e]) | A.Call("npeek", [e]) ->
343             let head_node_p = (expr builder e) in
344               (* Trying to make the crash graceful here 565jhfdshjq2 *)
345               if head_node_p = (L.const_pointer_null (L.type_of
346 head_node_p)) then
347                 raise (Failure("nothing to peek at, sorry"))
348               else
349                 let head_node_payload_pointer = L.build_struct_gep
350                   head_node_p 1 "" builder in
351                     L.build_load head_node_payload_pointer "" builder
352                     | A.Call("snext", [e]) | A.Call("enext", [e]) | A.Call("inext",
353 [e]) | A.Call("nnext", [e]) ->
354                       let head_node_next_p = L.build_struct_gep (expr builder e)
355                         0 "" builder in
356                           L.build_load head_node_next_p "" builder
357                           | A.Call("slength", [e]) | A.Call("elength", [e]) | A.Call("ilength",
358 [e]) | A.Call("nlenth", [e]) ->
359                             let head_node = expr builder e in
360                               if (L.pointer_type empty_node_t) = (L.type_of head_node)
361 then
362   L.const_int i32_t 0
363 else
364
365   let head_node_len_p = L.build_struct_gep (head_node) 2 ""
366   builder in
367     L.build_load head_node_len_p "" builder
368
369     | A.Call("sadd", [elmt; the_list]) | A.Call("iadd", [elmt;
370 the_list])
371     | A.Call("nadd", [elmt; the_list]) | A.Call("eadd", [elmt;
372 the_list]) ->
373
374   (* Build the new node *)
375   (* let elmt = (expr builder the_list) in *)
376   let the_head = (expr builder the_list) in
377   let good_node_t = get_node_type elmt in
378   let new_node = build_node (good_node_t) elmt in
379
380   (* To accomodate for calls that take an empty list in (?) *)
381   *)
382   if (L.pointer_type empty_node_t) = (L.type_of the_head)
383 then
384   let new_node_len_p = L.build_struct_gep new_node 2 ""
385   builder in
386     ignore (L.build_store (L.const_int i32_t 1)
387 new_node_len_p builder);
388     new_node
389   else
390
391   (* If the length is 0, we should detect this in advance *)
392

```

```

375     let head_node_len_p = L.build_struct_gep the_head 2 ""
376     builder in
377     let llength_val = L.build_load head_node_len_p ""
378     builder in
379     if (L.is_null llength_val) then
380       let new_node_len_p = L.build_struct_gep new_node 2 ""
381     builder in
382       ignore (L.build_store (L.const_int i32_t 1)
383       new_node_len_p builder);
384       new_node
385     else
386       (* Get the lenght of the list *)
387       let old_length = L.build_load (L.build_struct_gep
388       the_head 2 "" builder) "" builder in
389       let new_length = L.build_add old_length one "" builder
390       in
391       ignore (L.build_store new_length new_node_len_p builder
392       );
393       (* Attach the new head to the old head *)
394       add_element the_head new_node
395     | A.Call("strcmp", [s1;s2]) ->
396       let v1 = (expr builder s1) and v2 = expr builder s2 in
397       let v1value = L.build_load (L.build_load (L.
398       global_initializer v1) "" builder) "" builder in
399       let v2value = L.build_load (L.build_load (L.
400       global_initializer v2) "" builder) "" builder in
401       let str = L.string_of_lltype (L.type_of v2value) in
402       let result = (L.build_icmp L.Icmp.Eq v1value v2value "") in
403       builder in
404       let result = L.build_not result "" builder in
405       let result = L.build_intcast result i32_t "" builder in
406       result
407     (*
408   | A.Call(fname, actuals) ->
409     (* Will clean up later *)
410     let bitcast_actuals (actual, _) =
411       let lvalue = expr builder actual in
412       lvalue
413     in
414
415     let rec enumerate i enumed_l = function
416       | [] -> List.rev enumed_l
417       | hd::tl -> enumerate (i + 1) ((hd, i)::enumed_l) tl
418     in
419
420     let actuals = (enumerate 0 [] actuals) in
421     let (fdef, _) = StringMap.find fname function_decls in
422     let actuals = List.rev (List.map bitcast_actuals (List.rev
423     actuals)) in
424     let result = fname ^ "_result" in

```

```

425     L.build_call fdef (Array.of_list actuals) result builder
426 | A.Binop(e1, op, e2) ->
427   let v1 = expr builder e1 and v2 = expr builder e2 in
428   let value =
429     (match op with
430      | A.Add    -> L.build_add
431      | A.Sub    -> L.build_sub
432      | A.Mult   -> L.build_mul
433      | A.Div    -> L.build_sdiv
434      | A.And    -> L.build_and
435      | A.Or     -> L.build_or
436      | A.Equal  -> L.build_icmp L.Icmp.Eq
437      | A.Neq    -> L.build_icmp L.Icmp.Ne
438      | A.Less   -> L.build_icmp L.Icmp.Slt
439      | A.Leq    -> L.build_icmp L.Icmp.Sle
440      | A.Greater -> L.build_icmp L.Icmp.Sgt
441      | A.Geq    -> L.build_icmp L.Icmp.Sge)
442   v1 v2 "tmp" builder in value
443 | A.Unop(op, e) ->
444   let e' = expr builder e in
445   (match op with
446     | A.Not -> L.build_not) e' "tmp" builder
447   | _ -> raise (Failure("expr not supported"))
448
449
450 in let add_terminal builder f =
451   match L.block_terminator (L.insertion_block builder) with
452   | Some _ -> ()
453   | None -> ignore (f builder)
454
455 in let rec stmt builder = function
456   | A.Localdecl(t, n) -> (Hashtbl.add ocaml_local_hash n t;
457     ignore (add_local builder (t, n)); builder)
458   | A.Block(sl)       -> List.fold_left stmt builder sl
459   | A.Expr(e)         -> ignore (expr builder e); builder
460   | A.Return(e)       -> ignore (L.build_ret (expr builder e))
461   builder; builder
462   | A.If(p, then_stmt, else_stmt) ->
463     (* Get the boolean *)
464     let bool_val = (expr builder p)
465
466     (* Add the basic block *)
467     in let merge_bb   = L.append_block context "merge"
468     the_function
469     in let then_bb    = L.append_block context "then"
470     the_function
471     in let else_bb    = L.append_block context "else"
472     the_function
473
474     (* Write the statements into their respective blocks, build
475     conditional branch*)
476     in
477       add_terminal (stmt (L.builder_at_end context then_bb)
478 then_stmt) (L.build_br merge_bb);
479       add_terminal (stmt (L.builder_at_end context else_bb)
480 else_stmt) (L.build_br merge_bb);
481       ignore (L.build_cond_br bool_val then_bb else_bb builder)
482 ;
483
484     (* Return the builder *)
485     L.builder_at_end context merge_bb
486   | A.While(predicate, body) ->

```

```

478     let pred_bb = L.append_block context "while" the_function
479     in
480         ignore (L.build_br pred_bb builder);
481
482     let body_bb = L.append_block context "while_body"
483     the_function in
484         add_terminal (stmt (L.builder_at_end context body_bb)
485         body)
486             (L.build_br pred_bb);
487
488     let pred_builder = L.builder_at_end context pred_bb in
489     let bool_val = (* bool_of_int *) (expr pred_builder
490     predicate) in
491
492     let merge_bb = L.append_block context "merge"
493     the_function in
494         ignore (L.build_cond_br bool_val body_bb merge_bb
495         pred_builder);
496         L.builder_at_end context merge_bb
497
498     | _           -> raise (Failure("statement not implemented"))
499
500   in let builder = stmt builder (A.Block (List.rev fdecl.A.body))
501   in
502       add_terminal builder (L.build_ret (L.const_int (ltype_of_typ A.
503       Int) 0))
504
505   in List.iter build_function_body functions;
506   the_module

```

## 9.6 gal.ml

```

1 type action = Ast | LLVM_IR | Compile;;
2
3 module P = Printf;;
4
5 let _ = (
6   let action = if Array.length Sys.argv > 1 then
7     List.assoc Sys.argv.(1)
8     [("-a", Ast); ("l", LLVM_IR); ("-c", Compile) ]
9   else *)
10   Compile
11   in
12
13   (* Standard Library Functions *)
14   let stdlib_file          = "stdlib_code.gal" in
15   let stdlib_in            = open_in stdlib_file in
16   let stdlib_lexbuf        = Lexing.from_channel stdlib_in in
17   let (std_globs, std_funcs) = Parser.program Scanner.token
18   stdlib_lexbuf in
19
20   (* The input program *)
21   let lexbuf               = Lexing.from_channel stdin in
22   let (globs, funcs)        = Parser.program Scanner.token
23   lexbuf in
24
25   let ast = (std_globs @ globs, std_funcs @ funcs) in
26   (* P.printf stderr "%s" "ast built\n"; *)

```

```

26 let exp_list = Semant.check ast in
27 if exp_list <> [] then
28   raise (Failure ("\\n" ^ (String.concat "\\n" exp_list)))
29 else
30   (* P.printf stderr "%s" "ast checked\\n"; *)
31   let m = Codegen.translate ast in
32   Llvm_analysis.assert_valid_module m;
33   print_string (Llvm.string_of_llmodule m);
34   (* P.printf stderr "%s" "code generated\\n"; *)
35

```

## 9.7 stdlib\_code.gal

```

1  elist get_most_edges_node (nlist graph){
2
3    int len;
4    len = nlength(graph);
5    int i;
6    i = 0;
7
8    ilist lengths;
9    lengths = [];
10
11   nlist temp;
12   temp = graph;
13
14   /* Get the number of edges */
15   while( (i < len) && (len > 0)){
16     lengths = iadd(elength(npeek(temp)), lengths);
17     temp = nnex(temp);
18     i = i + 1;
19   }
20   lengths = irev(lengths);
21
22   len = ilength(lengths);
23   i = 0;
24   int longest;
25   longest = 0;
26   int order;
27   order = 1;
28
29   while( (i < len) && (len > 0)){
30     if(longest < ipeek(lengths)){
31       longest = ipeek(lengths);
32       order = i + 1;
33     }else{
34       lengths = inext(lengths);
35       i = i + 1;
36     }
37
38   temp = graph;
39   elist result;
40   result = [];
41
42   while(order > 1){
43     temp = nnex(temp);
44   }
45
46   result = npeek(temp);
47   return result;
48 }
49

```

```

50 edge get_heaviest_graph_edge(nlist l1){
51
52     int len;
53     len = nlength(l1);
54     int i;
55     i = 0;
56     int heaviest_w;
57     heaviest_w = 0;
58
59     edge heaviest;
60     heaviest = |"EMPTY", 0, "EMPTY"|;
61
62     eelist temp;
63     temp = [];
64
65     while( (i < len) && (len > 0) ){
66
67         /* Get the head of the list and move forward */
68         temp = npeek(l1);
69         l1 = nnnext(l1);
70
71         /* Get the weight of the element */
72         if( heaviest_w < weight(get_heaviest_edge(temp)) ){
73             heaviest_w = weight(get_heaviest_edge(temp));
74             heaviest = get_heaviest_edge(temp);
75         } else {}
76
77         /* Increment */
78         i = i + 1;
79     }
80
81     return heaviest;
82 }
83
84 /* Function will return the heaviest edge of the node */
85 edge get_heaviest_edge(node n1){
86
87     int len;
88     len = elength(n1);
89     int i;
90     i = 0;
91
92     int heaviest_w;
93     heaviest_w = 0;
94
95     edge heaviest;
96     heaviest = |"EMPTY", 0, "EMPTY"|;
97
98     edge temp;
99     temp = |" ", 0, " "|;
100
101    /* Iterate through the list , compare weights of edges */
102    while( (i < len) && (len > 0) ){
103
104        /* Get the head of the list and move forward */
105        temp = epeek(n1);
106        n1 = enext(n1);
107
108        /* Get the weight of the element */
109        if( heaviest_w < weight(temp) ){
110            heaviest_w = weight(temp);
111            heaviest = temp;

```

```

112         } else {}
113
114         /* Increment */
115         i = i + 1;
116     }
117
118     return heaviest;
119 }
120
121
122 int print_line(string str){
123     print_str(str);
124     print_endline();
125     return 0;
126 }
127
128
129 int print_sllen(slist lister){
130     print_int(slength(lister));
131     print_endline();
132     return 0;
133 }
134
135 int printslist(slist l1){
136     int len;
137     len = slength(l1);
138     slist tmp;
139     tmp = l1;
140     int i;
141     i = 0;
142     print_str("->");
143     while (i < len) {
144         print_str(speek(tmp));
145         print_str(":");
146         tmp = snext(tmp);
147         i = i + 1;
148     }
149     print_endline();
150
151     return 1;
152 }
153
154 int print_edge(edge e){
155     print_str("|");
156     print_str(source(e));
157     print_str(",");
158     print_int(weight(e));
159     print_str(",");
160     print_str(dest(e));
161     print_str("|");
162     return 0;
163 }
164
165 int print_elist(elist l1){
166     int len;
167     len = elength(l1);
168     elist tmp;
169     tmp = l1;
170     int i;
171     i = 0;
172     print_str("->");
173     while (i < len) {

```

```

174     print_edge(epeek(tmp));
175     print_str("::");
176     tmp = enext(tmp);
177     i = i + 1;
178 }
179 print_endline();
180
181     return 1;
182 }
183
184 int print_ilist(ilist l1){
185     int len;
186     len = ilength(l1);
187     ilist tmp;
188     tmp = l1;
189     int i;
190     i = 0;
191     print_str(">>");
192     while (i < len) {
193         print_int(ipeek(tmp));
194         print_str("::");
195         tmp = inext(tmp);
196         i = i + 1;
197     }
198     print_endline();
199
200     return 1;
201 }
202
203 int print_nlist(nlist l1){
204     int len;
205     len = nlength(l1);
206     nlist tmp;
207     tmp = l1;
208     int i;
209     i = 0;
210     print_str(">>");
211     while (i < len) {
212         print_elist(npeek(tmp));
213         print_str("::");
214         tmp = nnext(tmp);
215         i = i + 1;
216     }
217     print_endline();
218
219     return 1;
220 }
221
222 ilist irev(ilist l1){
223
224     int len_l1;
225     len_l1 = ilength(l1);
226     ilist temp_l1;
227     temp_l1 = [];
228     int temp_element;
229
230     while (!(len_l1 == 0)) {
231
232         /*adds the first element of the list l1 to temp_l1*/
233         temp_element = ipeek(l1);
234         temp_l1 = iadd(temp_element, temp_l1);
235

```

```

236     /*advances the head of the list*/
237     l1 = inext(l1);
238
239     len_l1 = len_l1 - 1;
240
241 }
242 return temp_l1;
243 }
244
245 slist srev(slist l1){
246
247     int len_l1;
248     len_l1 = slength(l1);
249     slist temp_l1;
250     temp_l1 = [];
251     string temp_element;
252
253     while (!(len_l1 == 0)){
254
255         /*adds the first element of the list l1 to temp_l1*/
256         temp_element = speek(l1);
257         temp_l1 = sadd(temp_element, temp_l1);
258
259         /*advances the head of the list*/
260         l1 = snext(l1);
261
262         len_l1 = len_l1 - 1;
263
264     }
265     return temp_l1;
266 }
267
268 elist erev(elist l1){
269
270     int len_l1;
271     len_l1 = elength(l1);
272     elist temp_l1;
273     temp_l1 = [];
274     edge temp_element;
275
276     while (!(len_l1 == 0)){
277
278         /*adds the first element of the list l1 to temp_l1*/
279         temp_element = epeek(l1);
280         temp_l1 = eadd(temp_element, temp_l1);
281
282         /*advances the head of the list*/
283         l1 = enext(l1);
284
285         len_l1 = len_l1 - 1;
286
287     }
288     return temp_l1;
289 }
290
291 nlist nrev(nlist l1){
292
293     int len_l1;
294     len_l1 = nlengh(l1);
295     nlist temp_l1;
296     temp_l1 = [];
297     node temp_element;

```

```

298     while (!(len_l1 == 0)) {
299
300         /*adds the first element of the list l1 to temp_l1*/
301         temp_element = npeek(l1);
302         temp_l1 = nadd(temp_element, temp_l1);
303
304         /*advances the head of the list*/
305         l1 = nnnext(l1);
306
307         len_l1 = len_l1 - 1;
308
309     }
310     return temp_l1;
311 }
312
313
314
315     ilist iadd_back(ilist l1, int i) {
316
317         l1 = irev(l1);
318         l1 = iadd(i, l1);
319         l1 = irev(l1);
320         return l1;
321
322     }
323
324
325     slist sadd_back(slist l1, string i) {
326
327         l1 = srev(l1);
328         l1 = sadd(i, l1);
329         l1 = srev(l1);
330         return l1;
331
332     }
333
334
335     elist eadd_back(elist l1, edge i) {
336
337         l1 = erev(l1);
338         l1 = eadd(i, l1);
339         l1 = erev(l1);
340         return l1;
341
342     }
343
344
345     nlist nadd_back(nlist l1, node i) {
346
347         l1 = nrev(l1);
348         l1 = nadd(i, l1);
349         l1 = nrev(l1);
350         return l1;
351
352     }
353
354     ilist iconcat(ilist l1, ilist l2) {
355
356         l1 = irev(l1);
357         int len_l2;
358         len_l2 = ilength(l2);
359         int temp_element;

```

```

360     while (! ( len_l2 ==0)) {
361
362         temp_element = ipeek(l2);
363         l1 = iadd(temp_element, l1);
364         l2 = inext(l2);
365
366         len_l2 = len_l2 - 1;
367     }
368
369     l1 = irev(l1);
370     return l1;
371 }
372
373     slist sconcat(slist l1, slist l2) {
374
375         l1 = srev(l1);
376         int len_l2;
377         len_l2 = slength(l2);
378         string temp_element;
379
380         while (! ( len_l2 ==0)) {
381
382             temp_element = speek(l2);
383             l1 = sadd(temp_element, l1);
384             l2 = snext(l2);
385
386             len_l2 = len_l2 - 1;
387         }
388
389         l1 = srev(l1);
390         return l1;
391     }
392
393     elist econcat(elist l1, elist l2) {
394
395         l1 = erev(l1);
396         int len_l2;
397         len_l2 = elength(l2);
398         edge temp_element;
399
400         while (! ( len_l2 ==0)) {
401
402             temp_element = epeek(l2);
403             l1 = eadd(temp_element, l1);
404             l2 = enext(l2);
405
406             len_l2 = len_l2 - 1;
407         }
408
409         l1 = erev(l1);
410         return l1;
411     }
412
413     nlist nconcat(nlist l1, nlist l2) {
414
415         l1 = nrev(l1);
416         int len_l2;
417         len_l2 = nlengh(l2);
418         node temp_element;
419
420         while (! ( len_l2 ==0)) {
421

```

```

422     temp_element = npeek(12);
423     l1 = nadd(temp_element, l1);
424     l2 = nnext(l2);
425
426     len_l2 = len_l2 - 1;
427 }
428
429 l1 = nrev(l1);
430 return l1;
431
432 }
```

## 9.8 help.ml

```

1 open Ast
2
3 (* I hope this function is not too broken, still needs testing
4    works for if conditionals, do not know about for loops.
5    What it does is it goes through the body of the function
6    extracting all local variables, returnning a list of locals *)
7
8 let rec get_vardecls vars = function
9   | [] -> List.rev vars
10  | hd::tl -> (match hd with
11    | Localdecl(typname, name) ->
12      (* print_string " Expr"; *)
13      get_vardecls ((typname, name)::vars) tl
14    | Block(slist) -> (* print_string " Block"; *)
15      (match slist with
16       | [] -> get_vardecls vars tl
17       | hd1::tl1 ->
18         get_vardecls
19         (get_vardecls (get_vardecls vars [hd1]) tl1)
20         tl1)
21    | If(e, s1, s2) ->
22      (* print_string " If"; *)
23      get_vardecls (get_vardecls (get_vardecls vars [s1]) [s2]) tl
24    | For(_, _, _, s) ->
25      get_vardecls (get_vardecls vars [s]) tl
26    | While(e, s) ->
27      get_vardecls (get_vardecls vars [s]) tl
28    | _ -> get_vardecls vars tl )
29
30;;
```

## 9.9 Sample Code: dfs.gal

```

1 int dfs(nlist graph, string A)
2 {
3
4   int found;
5   found = 0;
6
7   slist visited;
8   slist stack;
9   stack = ["A"];
10
11  elist v;
12  int s_counter;
13  string temp_str;
14  string node_name;
15  int node_found;
```

```

16
17     nlist temp;
18     temp = graph;
19     int graph_length;
20     graph_length = nlength(graph);
21     int i;
22     i = 0;
23     int count;
24     count = 0;
25     string v_dest;
26
27     string v_source;
28     visited = [""];
29     int streq_val;
30     string top_of_stack;
31     elist use_node;
32     elist temp_node;
33     string temp_source;
34     string temp_dest;
35     slist stack_temp;
36     elist use_node_temp;
37
38     stack_temp = stack;
39     int count_loop;
40     count_loop = 0;
41     string temp_visited;
42
43     while(count < 7)
44     {
45         if(i >= graph_length)
46         {
47             return found;
48         }
49         else
50         {
51
52         }
53         if(count > 0){
54             /*print_str(speek(stack_temp));*/
55             stack_temp = snext(stack);
56             stack = snext(stack);
57
58         }
59         else{
60
61     }
62
63     top_of_stack = speek(stack_temp);
64     visited = sadd_back(visited,top_of_stack);
65     /*this might give us issues*/
66
67     /*Iterate through graph to find correct edge*/
68     i = 0;
69     temp = graph;
70     while(i < graph_length)
71     {
72         temp_node = npeek(temp);
73         temp_source = source(epeek(temp_node));
74
75         streq_val = streq(temp_source,top_of_stack);
76         if(streq_val == 0)
77         {

```

```

78     use_node = temp_node;
79 }
80 else
81 {
82 }
83 temp = nnext(temp);
84 i = i + 1;
85 }
86 /*temp = nnext(temp);*/
87
88
89
90
91
92 /*v_source = source(epeek(use_node));
93 v_dest = dest(epeek(use_node));
94 visited = sadd_back(visited ,v_source);*/
95
96 i = 0;
97
98 use_node_temp = use_node;
99
100 while(i<elength(use_node))
101 {
102
103     temp_dest = dest(epeek(use_node_temp));
104     use_node_temp = enext(use_node_temp);
105     stack = sadd_back(stack ,temp_dest);
106
107
108     i = i + 1;
109 }
110
111
112 count = count+1;
113 }
114
115
116 while(count_loop<slength(visited)){
117
118     temp_visited = speek(visited);
119     visited = snext(visited);
120     if(streq(temp_visited ,A)==0){
121         found = 1;
122     }
123     else{
124
125     }
126
127 }
128 return found;
129 }
130
131 int main()
132 {
133
134     int isfound ;
135
136     /* Declare our nodes above */
137     node n1;
138     n1 = |"A": 2, "B" , 4, "C" |;
139

```

```

140     node n2;
141     n2 = |"B": 11, "E", 12, "F"|;
142     node n3;
143     n3 = |"C": 5, "G", 16, "H"|;
144
145     nlist new_graph;
146     new_graph = [n1::n2::n3];
147
148     isfound = dfs(new_graph, "Z");
149     if(isfound == 1){
150         print_str("NODE IS FOUND USING DFS");
151     }
152     else{
153         print_str("NODE IS NOT FOUND");
154     }
155     /*ABFCGH*/
156     print_endline();
157
158 }
159 }
```

## 9.10 Sample Code: demo.gal

```

1 int main(){
2
3     print_endline();
4
5     /* Declare our nodes above */
6     node n1;
7     n1 = |"A": 2, "B", 11, "C", 4, "D", 14, "E"|;
8     node n2;
9     n2 = |"B": 7, "C", 3, "A", 20, "D"|;
10    node n3;
11    n3 = |"C": 5, "D", 5, "A", 16, "E"|;
12    node n4;
13    n4 = |"D": 20, "A", 7, "B"|;
14
15    print_line("Lets print them to see what we got:");
16    print_elist(n1);
17    print_elist(n2);
18    print_elist(n3);
19    print_elist(n4);
20
21    print_endline();
22    print_endline();
23
24    /* Lets declare another node. But using diffrent syntax */
25    elist n5;
26    n5 = [| "E", 24, "D" | :: | "E", 13, "B" |];
27
28    print_line("We can also print them as a graph:");
29    nlist graph;
30    graph = [n1::n2::n3::n4::n5];
31
32    /* We can use a different function to print this graph */
33    print_nlist(graph);
34    print_endline();
35
36    graph = npop(graph);
37    print_nlist(graph);
38    graph = npop(graph);
```

```

40     print_nlist(graph);
41     graph = npop(graph);
42     print_nlist(graph);
43
44     slist testpops;
45     testpops = ["A" :: "B" :: "C"];
46     print_slist(testpops);
47     testpops = spop(testpops);
48     print_slist(testpops);
49     testpops = spop(testpops);
50     print_slist(testpops);
51
52
53
54
55
56
57     print_line("Lets get the heaviest edge of the node n1:");
58     edge heaviest;
59     heaviest = get_heaviest_edge(n1);
60     print_edge(heaviest);
61     print_endline();
62
63     print_line("How about the heaviest edge in our graph? Sure:");
64     heaviest = get_heaviest_graph_edge(graph);
65     print_edge(heaviest);
66     print_endline();
67
68     print_line("Lets get the node that has the most edges");
69     node important;
70     important = get_most_edges_node(graph);
71     print_line(source(epeek(important)));
72
73     return 0;
74 }
```

## 9.11 testall.sh

```

1 #!/bin/sh
2
3 # Regression testing script for MicroC
4 # Step through a list of files
5 # Compile, run, and check the output of each expected-to-work test
6 # Compile and check the error of each expected-to-fail test
7
8 # Path to the LLVM interpreter
9 LLI="lli"
10 #LLI="/usr/local/opt/llvm/bin/lli"
11
12 # Path to the microc compiler. Usually "./microc.native"
13 # Try "_build/microc.native" if ocamlbuild was unable to create a
14 # symbolic link.
14 GAL="./gal.native"
15 #GAL="_build/microc.native"
16
17 # Set time limit for all operations
18 ulimit -t 30
19
20 globallog=testall.log
21 rm -f $globallog
22 error=0
23 globalerror=0
```

```

24
25 keep=0
26
27 Usage() {
28     echo "Usage: testall.sh [options] [.gal files]"
29     echo "-k      Keep intermediate files"
30     echo "-h      Print this help"
31     exit 1
32 }
33
34 SignalError() {
35     if [ $error -eq 0 ] ; then
36         echo "FAILED"
37         error=1
38     fi
39     echo "$1"
40 }
41
42 # Compare <outfile> <reffile> <difffile>
43 # Compares the outfile with reffile. Differences , if any, written
44 # to difffile
45 Compare() {
46     generatedfiles="$generatedfiles $3"
47     echo diff -b $1 $2 ">" $3 1>&2
48     diff -b "$1" "$2" > "$3" 2>&1 || {
49         SignalError "$1 differs"
50         echo "FAILED $1 differs from $2" 1>&2
51     }
52 }
53
54 # Run <args>
55 # Report the command, run it , and report any errors
56 Run() {
57     echo $* 1>&2
58     eval $* || {
59         #SignalError "$1 failed on $*"
60         return 1
61     }
62 }
63
64 # RunFail <args>
65 # Report the command, run it , and expect an error
66 RunFail() {
67     echo $* 1>&2
68     eval $* && {
69         SignalError "failed: $* did not report an error"
70     }
71     return 0
72 }
73
74 Check() {
75     error=0
76     basename=`echo $1 | sed 's/.*/\//'
77                         s/.gal//,'
78     reffile=`echo $1 | sed 's/.gal//,'
79     basedir="`echo $1 | sed 's/\/[^\/]*$//'`."
80
81     echo -n "$basename..."
82
83     echo 1>&2
84     echo ##### Testing $basename" 1>&2

```

```

85 generatedfiles=""
86
87 generatedfiles="$generatedfiles ${basename}.ll ${basename}.out"
88 &&
89 Run "$GAL" "<" $1 ">" "${basename}.ll" &&
90 Run "$LLI" "${basename}.ll" ">" "${basename}.out"
91 Compare ${basename}.out ${reffile}.out ${basename}.diff
92
93 # Report the status and clean up the generated files
94
95 if [ $error -eq 0 ] ; then
96 if [ $keep -eq 0 ] ; then
97 rm -f $generatedfiles
98 fi
99 echo "OK"
100 echo "##### SUCCESS" 1>&2
101 else
102 echo "##### FAILED" 1>&2
103 globalerror=$error
104 fi
105 }
106
107 CheckFail() {
108 error=0
109 basename='echo $1 | sed 's/*\\//'
110           's/.gal//,'
111 reffile='echo $1 | sed 's/.gal$//'
112 basedir="`echo $1 | sed 's/\/\/*[^\/]*$//`/."
113
114 echo -n "$basename..."
115
116 echo 1>&2
117 echo "##### Testing $basename" 1>&2
118
119 generatedfiles=""
120
121 generatedfiles="$generatedfiles ${basename}.err ${basename}.diff" &&
122 RunFail "$GAL" "<" $1 "2>" "${basename}.err" ">>" $globallog &&
123 Compare ${basename}.err ${reffile}.err ${basename}.diff
124
125 # Report the status and clean up the generated files
126
127 if [ $error -eq 0 ] ; then
128 if [ $keep -eq 0 ] ; then
129 rm -f $generatedfiles
130 fi
131 echo "OK"
132 echo "##### SUCCESS" 1>&2
133 else
134 echo "##### FAILED" 1>&2
135 globalerror=$error
136 fi
137 }
138
139 while getopts kdpshc; do
140 case $c in
141 k) # Keep intermediate files
142     keep=1
143     ;;
144 h) # Help

```

```

145      Usage
146      ;;
147      esac
148 done
149 shift `expr $OPTIND - 1`
151
152 LLIFail() {
153     echo "Could not find the LLVM interpreter \\"$LLI\"."
154     echo "Check your LLVM installation and/or modify the LLI variable
155         in testall.sh"
156     exit 1
157 }
158 which "$LLI" >> $globallog || LLIFail
159
160 if [ $# -ge 1 ]
161 then
162     files=$@
163 else
164     files="../.tests/test_*.gal ../.tests/fail_*.gal"
165 fi
166
167 for file in $files
168 do
169     case $file in
170     *test_*)
171         Check $file 2>> $globallog
172         ;;
173     *fail_*)
174         CheckFail $file 2>> $globallog
175         ;;
176     *)
177         echo "unknown file type $file"
178         globalerror=1
179         ;;
180     esac
181 done
182
183 exit $globalerror

```

## 9.12 fail\_assignment\_edge2.gal

```

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

```

## 9.13 fail\_assignment\_int\_to\_string.gal

```

1 int main()
2 {
3     string a;
4     a = 5;
5 }

```

## 9.14 fail\_assignment\_string\_to\_int.gal

```

1 int main()
2 {

```

```
3     int a;
4     a = "This";
5 }
```

### 9.15 fail\_binary\_addition1.gal

```
1 int main()
2 {
3     int a;
4
5     a = 5 + "hello";
6
7 }
```

### 9.16 fail\_binary\_addition2.gal

```
1 int main()
2 {
3     int a;
4     string b;
5
6     b = "this";
7
8     a = 5 + b;
9
10 }
```

### 9.17 fail\_binary\_division.gal

```
1 int main()
2 {
3     int a;
4     string b;
5
6     b = "this";
7
8     a = 5 + b;
9
10 }
```

### 9.18 fail\_binary\_multiplication1.gal

```
1 int main()
2 {
3     int a;
4
5     a = 5 * "hello";
6
7 }
```

### 9.19 fail\_duplicate\_assignint.gal

```
1 int main()
2 {
3     int a;
4     int b;
5
6     a = 5;
7     b = 5;
8
9     int a;
10 }
```

## 9.20 Fail\_duplicate\_formal\_identifiers.gal

```
1 int a;
2
3 int main(int a, int a)
4 {
5     int b;
6 }
```

## 9.21 fail\_duplicate\_function\_names.gal

```
1
2 int main(int x, int y)
3 {
4
5 }
6
7 int this()
8 {
9
10
11 }
12
13
14 int this()
15 {
16
17 }
```

## 9.22 fail\_duplicate\_global\_assignment.gal

```
1
2 int a;
3 int a;
4
5 int main()
6 {
7
8 }
```

## 9.23 Fail\_function\_doesnt\_exist.gal

```
1
2 int main()
3 {
4
5     test();
6
7 }
```

## 9.24 Fail\_incorrect\_argument\_types.gal

```
1 int main()
2 {
3     string b;
4     int a;
5
6     b = "hello";
7     a = 2;
```

```

8     test(b,a);
9 }
10 }
11 }
12 int test(int x, int y)
13 {
14     int z;
15 }
16 }
```

### 9.25 fail\_incorrect\_number\_function\_arguments.gal

```

1 int main()
2 {
3     string b;
4     int a;
5
6     b = "hello";
7     a = 2;
8
9     test(b,a);
10 }
11 }
12 int test(int x, int y)
13 {
14
15     int z;
16 }
17 }
```

### 9.26 Fail\_incorrect\_number\_function\_arguments2.gal

```

1 int main()
2 {
3
4     int x;
5     int y;
6
7     x = 5;
8     y = 7;
9
10    test(x,y);
11 }
12 }
13 int test()
14 {
15     int c;
16     c = 7;
17 }
```

### 9.27 Fail\_main\_nonexistent.gal

```

1
2 int x()
3 {
4     int a;
5     int b;
6     int c;
```

```
7     c = a + b;  
8  
9 }  
10
```

### 9.28 Fail\_no\_id\_before\_usage\_int.gal

```
1 int main()  
2 {  
3     a = 5;  
4 }  
5
```

### 9.29 Fail\_redefine\_builtin\_edge.gal

```
1 int main()  
2 {  
3     string edge;  
4 }
```

### 9.30 fail\_redefine\_builtin\_int.gal

```
1 int main()  
2 {  
3     string int;  
4 }
```

### 9.31 fail\_redefine\_builtin\_list.gal

```
1 int main()  
2 {  
3     int slist;  
4 }
```

### 9.32 Fail\_redefine\_existing\_function.gal

```
1 int print_int() {}  
2  
3 int main()  
4 {  
5     return 0;  
6 }
```

### 9.33 Test\_assignment\_list1.gal

```
1 int main()  
2 {  
3     edge e1;  
4     edge e2;  
5     edge e3;  
6  
7     e1 = |"A",5,"B"|;  
8     e2 = |"B",7,"C"|;  
9     e3 = |"C",2,"A"|;  
10  
11     elist l1;  
12     l1 = [e1];  
13  
14     return 1;  
15 }  
16
```

### 9.34 test\_boolean\_false.gal

```
1 int main()
2 {
3     if (!(1==1))
4     {
5         print_str("This is true");
6     }
7     else
8     {
9         print_str("This is NOT true");
10    }
11    return 1;
12 }
```

### 9.35 Test\_boolean\_true.gal

```
1 int main()
2 {
3     if(1==1)
4     {
5         print_str("This is true");
6     }
7     else
8     {}
9     return 1;
10}
11 }
```

### 9.36 test\_create\_edge.gal

```
1 int main(){
2
3     edge e1;
4     return 1;
5 }
```

### 9.37 Test\_print\_ilist.gal

```
1 int main()
2 {
3
4     ilist x;
5
6     x = [1];
7     x = iadd(2,x);
8     x = iadd(100,x);
9
10
11    print_ilist(x);
12    return 1;
13 }
```

### 9.38 Test\_print\_ilist\_rev.gal

```
1 int main()
2 {
3
4     ilist x;
5     ilist rev_x;
6 }
```

```

7   x = [11];
8   x = iadd(10,x);
9   x = iadd(9,x);
10  x = iadd(8,x);
11  x = iadd(7,x);
12  x = iadd(1,x);
13  x = iadd(2,x);
14  x = iadd(3,x);
15
16  x = irev(x);
17
18  print_ilist(x);
19  return 1;
20 }
```

### 9.39 test\_print\_int.gal

```

1 int main(){
2     print_int(1);
3     return 1;
4 }
```

### 9.40 Test\_print\_int1.gal

```

1 int main(){
2
3     int a;
4     a = 5;
5
6     print_int(a);
7     return 1;
8 }
```

### 9.41 Test\_print\_order.gal

```

1 int main(){
2     print_int(6);
3     print_endline();
4     print_str("Hello");
5     print_endline();
6     print_int(903);
7     print_endline();
8     return 1;
9 }
```