TLFKAC (BNKAB)

T**he **L**anguage **F**ormerly **K**nown **A**s **CRAWL** (**B**ut **N**ow **K**nown **A**s **BRAWL)

Language Manual and Project Report

Rajesh Venkataraman
(rv2187@columbia.edu)

Amoghavarsha Ramappa
(ar2645@columbia.edu)

Harish JP
(harishjp@gmail.com)

Table of Contents

1. Introduction

1.1.1. Motivation

Graphs play an important role in many applications in Computer Science. The theory of graphs has been successfully exploited in many practical applications, with the most significant ones related to networks. It is important to emphasize the computational and algorithmic aspects of graphs. This emphasis arises from the conviction that whenever graph theory is applied to solving any practical problem, it almost always leads to large graphs – graphs that are virtually impossible to analyze without the aid of a computer. The high-speed digital computer is one of the reasons for the growth of interest in graph theory.

In spite of such widespread interest in Graphs and their properties, all attempts at solving problems in Graph Theory computationally have almost always led to the creation of a new library which can perform certain operations on Graphs. We, as students of Computer Science, believe that there are umpteen advantages to having a language dedicated to describing and manipulating Graphs. Hence we want to create a language which can be used by Graph Theoreticians, Mathematicians and Computer Scientists to solve computational Graph problems.

1.1.2. Description

BRAWL is a language which manipulates Graphs and Nodes as first order primitives. By this, we mean that operations on Graphs and Nodes in BRAWL are treated as fundamental as say C or C++ treats integer and string operations. By this, we hope that people can express their ideas about Graphs and operations on them succinctly and clearly and take advantage of a computer's inherent power in analyzing Graphs.

The BRAWL language is terse and the syntax closely follows the popular pseudocode formats found in algorithms. This makes the language not only easy to develop in but also easy to understand. The BRAWL language has been implemented in the functional programming language ***OCAML***. ***OCAML*** is a beautiful language to program in and offers various features which make it more compelling to use ***OCAML***.

We chose to implement the compiler in the ***OCAML*** as an experiment. This will help us learn the different facets of a functional programming language.

1.1.3. Example

As an example program, we provide the following program which implements the *Depth First Search* algorithm:

```
int visited[5];
edge e_a[0];
int w = 0;
int print(int a){
      return 0;
}
int print_string(string a){
      return 0;
}
int print_endline(){
      return 0;
}
int callback(edge e){
      print (<-e);
      print_endline();
      e_a + e;
      w += sizeof e;
      return 0;
}
int dfs(graph g, int v){
      foreach (e in g->v where visited[<-e] == 0) {
            visited[<-e] = 1;
            callback(e);
            dfs(g, <-e);
      }
}

int main(){
      string t = "\t";
      int i = 0;
      while(i < sizeof visited) {
            visited[i] = 0;
            i += 1;
      }
      graph g = $5, [0, 1, 2]$ + [1, 2, 3] + [0, 2, 1]
      + [1, 3, 7] + [2, 4, 11];
      dfs(g, 0);
      foreach (e in e_a) {
            print (->e), print_string(t), print(<-e),
            print_string(t), print(sizeof e);
            print_endline();
      }
      print_string("The weight of the dfs tree is: ");
      print(w);
      print_endline();}
```

2. Language Tutorial

This chapter represents a tutorial for a novice to get started with *BRAWL* . The *BRAWL* compiler runs on Linux and can be obtained by emailing the authors of the language.

The language source file has an extension ***.bwl***.

The basic types defined by the language are : int, float, edge, graph, string.

The description of each type is specified in detail in the Language Reference manual in section 3.

The following is a simple program written in the BRAWL programming language.

```
int main(){

	graph g = $5, [0, 1, 2], [1, 2, 3], [0, 2, 3], [1, 3, 3],
	[2, 4, 7]$;

	foreach (e in g->1){
		print (sizeof e);
	}
}
```

A user can compile the program using the steps shown below:

$> ./brawlc test.bwl

The above command generates a C++ target program file. Users can use the g++ compiler to compile this file and run the program:

$>g++ test.cpp
$>./a.out

3. Language Manual

3.1. Lexical Conventions

A BRAWL program is consists of a single translation unit stored in a file. The file is written using the ASCII character set.

3.1.1. Comments

BRAWL comments begin with a # character at the beginning of the line and are terminated at the end of the line.

3.1.2. White space

BRAWL is a free form language. White space is ignored unless bounded by quotes (“) on either side.

3.1.3. Tokens

Tokens fall into five major categories: identifiers, keywords, constants, operators and separators.

3.1.3.1. Identifiers

Identifiers begin with a letter or an underscore (_) and are followed by any sequence of letters, digits or underscores. Two characters are considered equal if their ASCII values are equal. Two identifiers are considered equal if all their characters match.

3.1.3.2. Keywords

The following identifiers are reserved as keywords and using them in any BRAWL program as a regular identifier will result in an error:

`int`	**`float`**	**`edge`**	**`graph`**
`string`	**`while`**	**`if`**	**`else`**
`foreach`	**`in`**	**`sizeof`**	**`where`**

3.1.3.3. Constants

Constants provide the BRAWL programmer to conveniently initialize each of the supported primitives.

3.1.3.3.1. Integer Constants

An Integer constant consists of an optional plus ('+') sign or minus ('-') sign followed by a string of decimal digits [0-9].

3.1.3.3.2. Floating point Constants

A floating point constant is defined similar to the definition in the 'C' language, that is, it consists of an optional plus ('+') sign or minus ('-') sign followed by an integer part of one or more digits. This has to be followed by a decimal point or an exponent sign. The decimal point might be followed by more digits. The exponent is always followed by a positive or negative integer.

3.1.3.3.3. String Constants

String constants consist of a sequence of characters surrounded by double quotes. The quotes are not considered part of the string and the '\' character is used to generate escape sequences. If the constant has to contain the '\' character literally, one has to use the '\\' sequence. The following escape sequences are recognized by BRAWL:

```
\n        newline

\t        The horizontal tab
```

3.1.3.4. Operators

3.1.3.4.1. Arithmetic operators

The following arithmetic operators are supported by BRAWL: '+', '-', '*', '/' and '%'. Their meanings are respectively those of addition, subtraction, multiplication, division and remainder after division. The '-' operator can also be used as a unary operator to indicate the negativity of a number. The binary operators' operands must have have the same type. '+=', '-=', '*=', '/=' and '%=' are used to mean the same thing as they do in the 'C' language. The associativities of these operators also follow those of the 'C' language.

3.1.3.4.2. Collection operators

BRAWL supports the '<>' operator to define collections. The '<' sign followed by a sequence of comma (',') separated values, terminated by a '>' sign is construed to be a collection. The '[]' operator is also supported to randomly access collection's elements.

Hence, in order to access the i_{th} element of a collection C, one might use C[i]. Boolean operators The following boolean operations are supported in BRAWL: <, <=, >, >=, ==, ! which respectively mean _ less than_ , _ less than or equal to_ , _ grea ter than_ , _ greater than or equal to_ , _ equal to_ and the negation of the boolean operation. In addition BRAWL supports the && and || operators to mean boolean and and boolean or.

3.1.3.4.3. Sizeof

The sizeof operator, when used with strings returns their length and with integers and floats returns the value. Behavior of this operator when applied to Edges, Graphs and Collections is explained later. Precedence The precedence and associativities of the operators in BRAWL are identical to their counterparts in 'C '. To override this, one might use parentheses (()).

3.1.3.5 Separators

, and ; are used as separators in BRAWL. The ; separator indicates end of executable statement.

3.1.3.6 Scope

The scope of an identifier begins at immediately after its definition and ends at the end of the block. Blocks are delimited using the '{' and '}' separators.

3.2 Types

3.2.1 Integers, Floats and Strings

Integers and Double precision floating point numbers are the only two numeric types supported by BRAWL. Strings are sequences of ASCII characters.

3.2.2 Collections

BRAWL supports a basic collection type, not very different from the array type in 'C'. It provides random access of elements and the elements are ordered in the collection according to the order that they were inserted. While creating collections, the size of the collection has to be specified. Collections support the + operator which allows the programmer to add a new element to the collection. The sizeof operator can be used to get the number of elements in the collection.

3.2.3 Edges

In order to describe the connections in a graph, BRAWL supports the edge data type. An edge is composed of three parameters, namely, the 2 nodes that it connects and a weight associated with the two edges. Edges literals are defined as a comma separated list of these 3 elements delimited by parentheses. For example, (1, 2, “red”) defines an edge between nodes 1 and 2 with weight _ red_ . The > operator applied to an edge returns the node that it is incident on and the <returns the node that it is incident from. On applying the sizeof operator to an edge, the weight is returned.

3.2.4 Graphs

Graphs in BRAWL are defined just as a pair, namely that of the number of nodes and a collection of edges. Graph literals are of the form ((num_of_nodes, edge_collection)) .

Graphs also support the binary > operator, which returns a collection of edges going out from the given node. For example, g>3 returns a collection of edges going out from node number 3. Nodes are numbered from 0 to n – 1, where n is the number of nodes in the Graph. They also support the <operator which returns a collection of edges incident on the specified node. Using the sizeof keyword, one can get the number of nodes in the Graph.

3.3 Control Structures

3.3.1 foreach – in where

foreach is a keyword used to iterate over a collection. Used in conjunction with the in operator, it allows for obtaining the next element from the collection being iterated over. The where keyword can be used to further limit the execution of this loop. For example, to find the sum of the elements in an integer collection where the value is greater than or equal to 2, one might write:

```
int num = 0;
int coll[10] = <1, 2, 3>;

foreach ( num in coll where num >= 2)
     sum += num;
```

3.3.2 while

while is a looping construct. It tests a condition at the beginning of each iteration and executes the statements in the loop if the condition evaluates to be true. The statements associated with the loop must be enclosed in braces.

3.3.3 if

if is a keyword used for conditional execution. The statements associated with the if conditional are executed once if the condition evaluates to be true.

3.3.4 else

When used in conjunction with the if conditional, the else block is executed whenever the condition in the if conditional evaluates to false.

3.4 Grammar

The yacc listing of the grammar is as shown below:

```
program:
stat_list
| EOF
;

toplevel_stat:
stat
| LBRACE stat_list RBRACE
| LBRACE RBRACE
```

```
| SEMICOLON
;

stat_list:
stat
| stat stat_list
;

stat:
function_definition
| return_statement
| if_statement
| while_statement
| foreach_statement
| foreachw_statement
| variable_declaration
| expr_statement
;

expr_statement:
expr_list SEMICOLON
;

function_definition:
tid LPAREN tid_list RPAREN LBRACE stat_list RBRACE
| tid LPAREN RPAREN LBRACE stat_list RBRACE
;

return_statement:
RETURN exp SEMICOLON
;

if_statement:
IF LPAREN exp RPAREN toplevel_stat %prec LOWER_THAN_ELSE
| IF LPAREN exp RPAREN toplevel_stat ELSE toplevel_stat
;

while_statement:
WHILE LPAREN exp RPAREN toplevel_stat
;

foreach_statement:
FOREACH LPAREN ID IN exp RPAREN toplevel_stat
;

foreachw_statement:
FOREACH LPAREN ID IN exp WHERE exp RPAREN toplevel_stat
;

variable_declaration:
tid SEMICOLON
| tid ASSIGNMENT exp SEMICOLON
| tid LBRACKET exp RBRACKET SEMICOLON
```

```
| tid LBRACKET exp RBRACKET ASSIGNMENT exp SEMICOLON
;

tid_list:
tid
| tid COMMA tid_list
;

tid:
type_name ID
;

type_name:
INTK
| FLOATK
| STRINGK
| EDGEK
| GRAPHK
;

expr_list:
exp
| exp COMMA expr_list
;

exp:
INT
| FLOAT
| STRING
| lvalue
| ID LPAREN expr_list RPAREN
| ID LPAREN RPAREN
| edge_expression
| graph_expression
| exp PLUS exp
| exp MINUS exp
| exp MULTIPLY exp
| exp DIVIDE exp
| exp MODULO exp
| exp EQ exp
| exp GT exp
| exp LT exp
| exp GE exp
| exp LE exp
| exp LOGICALAND exp
| exp LOGICALOR exp
| MINUS exp %prec UMINUS
| NEGATE exp %prec UNOT
| LPAREN exp RPAREN
| LARROW exp %prec ULARROW
| RARROW exp %prec URARROW
| exp LARROW exp
```

```
| exp RARROW exp
| SIZEOF exp
| assignment_expression
| collection_expression
;

lvalue:
ID
| ID LBRACKET exp RBRACKET
;

edge_expression:
LBRACKET exp COMMA exp COMMA exp RBRACKET
;

graph_expression:
DOLLAR exp COMMA expr_list DOLLAR
;

collection_expression:
LBRACE expr_list RBRACE
;

assignment_expression:
lvalue ASSIGNMENT exp
| lvalue ADDAS exp
;
```

4. Project Plan

In this chapter we will describe the project plan and implementation

4.1 Process

The team used to meet twice a week to discuss project goals for the coming week and discuss problems faced by team members while working on the project. The first meeting of a week was solely focussed on brainstorming work which was completed and finding solutions for any issues which came up over the weeks work. The second meeting was focussed on planning new work and setting strict deadlines for each team member.

The primary goal was to get a basic working compiler program, test it thoroughly and incrementally build the remaining features on top of it.

4.2 Programming Style

The time spent writing a program is negligible compared to the time spent reading/understanding it. The main intention of the programming style used to write the translator is to keep the code concise, simple and readable.

The following is an example of the coding style used in the project:

```
(** String table: uses a global hash function to give each
string a unique integer identifier *)

module StringHash = Hashtbl.Make(struct
      type t = string
      let equal x y = x = y
      let hash = Hashtbl.hash
end)

(* The types of the symbol in the system *)
      type data_type = Undefined | SymInt | SymFloat |
      SymString | SymEdge | SymGraph | SymFunction
      | SymIntArray | SymFloatArray | SymStringArray |
      SymEdgeArray | SymGraphArray

      (* Damned global variable for the symbol table *)
      let symboltable = ref [StringHash.create 128] ;;

      (* Returns type of the symbol, Undefined if symbol is
      not found *)
```

```
let typeof name =
let rec rec_typeof tbl =
match tbl with
[] -> Undefined
| hd :: tl ->
      try StringHash.find hd name
      with Not_found -> rec_typeof tl
in rec_typeof !symboltable ;;


(* Add a new symbol to symbol table. Returns false if
symbol exists *)
let addSymbol name dType =
let head = List.hd !symboltable in
if StringHash.mem head name then
      false
else
      (StringHash.add head name dType; true);;

(* Adds a new hash, representing the local scope *)
let rec start_scope dummy =
symboltable := (StringHash.create 128) :: !symboltable
;;

(* Deletes the local scope *)
let rec end_scope dummy =
      match !symboltable with
      [] -> false
      | hd :: tl -> symboltable := tl; true ;;

let sym_to_array sym =
      match sym with
      SymInt -> SymIntArray
      | SymFloat -> SymFloatArray
      | SymString -> SymStringArray
      | SymEdge -> SymEdgeArray
      | SymGraph -> SymGraphArray
      | _ -> Undefined
      ;;

let array_to_sym a =
      match a with
      SymIntArray -> SymInt
      | SymFloatArray -> SymFloat
      | SymStringArray -> SymString
      | SymEdgeArray -> SymEdge
      | SymGraphArray -> SymGraph
      | _ -> Undefined
;;

(* vim: set ts=4 sw=4 noet: *)
```

4.3 Project Timeline

09/25/2007	Language Proposal
10/18/2007	Language Reference Manual
10/30/2007	SVN Running
11/10/2007	Lexer completed
11/30/2007	Parser Completed
12/02/2007	Simple test suites created
12/9/2007	Static Semantic Analysis and Code gen completed
12/12/2007	Added test suites
12/15/2007	Project Report created
12/16/07	All tests passed and demo prepared

4.4 Roles and Responsibilities

Grammar, Parser, Symbol Table, Code Generation, Demo Programs	Rajesh Venkataraman, Harish JP
Lexer, Symbol Table, Code Generation, Test Suites	Amoghavarsha Ramappa
C++ Library	Rajesh Venkataraman, Amoghavarsha Ramappa
Project Report	Amoghavarsha Ramappa, Rajesh Venkataraman

4.5 Development Environment

BRAWL was developed on Linux. The code is implemented using the functional programming language OCAML. The lexer has been implemented using OCAML Lex and the parser using the OCAML Yacc tools. The Unix Makefile is used to build the source code.

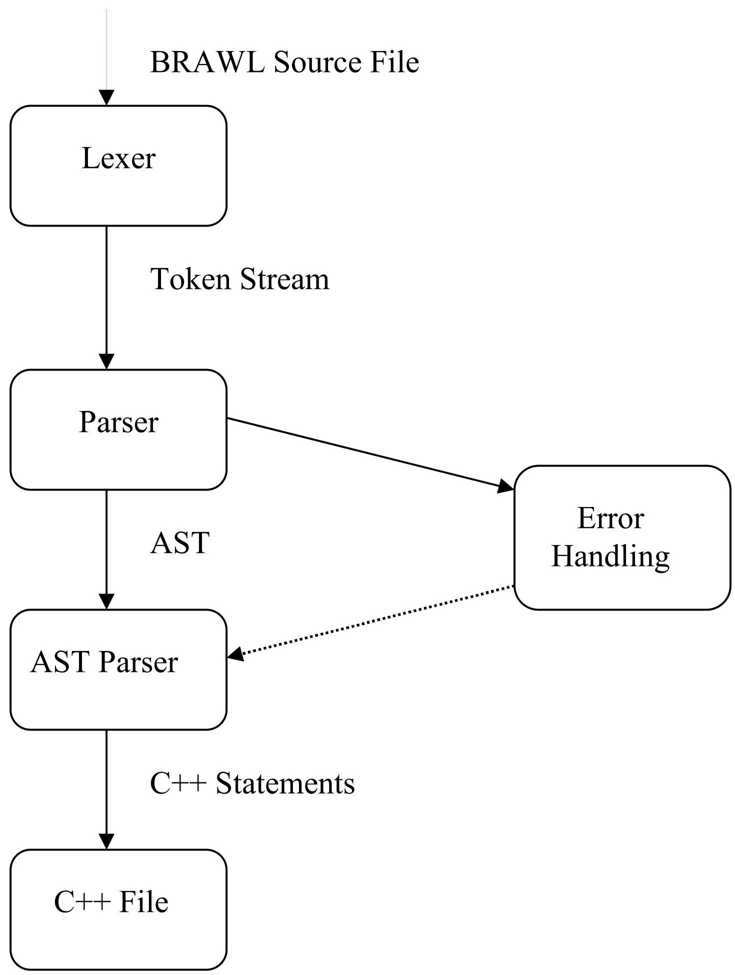
The source code management was accomplished using Assembla. Assembla provides tools and services for building software in a global development environment.

4.6 Project Log

Language Proposal initial draft	9/19/07
Language Proposal final draft	9/25/07
Language Reference Manual initial draft	10/14/07
Language Reference Manual final draft	10/18/07
Lexer	11/10/07
Parser for BRAWL specific statements	11/15/07
Parser for control flow statements	11/25/07
Completed Parser	11/28/07
Tree Walker and Code Generation (version1)	11/30/07
Tree Walker and Code Generation (version2)	12/05/07
Tree Walker and Code Generation (version3)	12/05/07
Static-Semantic Analysis	12/12/06
Comprehensive test-suites written	12/13/07
Comprehensive Testing	12/15/07
Freeze Source Code and Present	12/18/07

5. Architectural Design

5.1 Block Diagram of Translator



The *BRAWL* translator – Flow of information from one component to the next

5.2 Description of Architecture

BRAWL has mainly four components : *Lexer, Parser, Symbol Table, and AST Walker* (performs *Static Semantic Analysis* and the *Code generation*). The flow of information between the components is shown in the block diagram above. The lexical analyzer or the scanner reads the input characters and produces an output a sequence of token streams that the parser uses for syntax analysis. The parser performs syntax analysis by analyzing the sequence of input token streams to determine the grammatical structure with respect to the BRAWL grammar. The parser transforms the input token stream into an abstract syntax tree data structure. Each interior node of the syntax tree represent a BRAWL programming language construct and each of the nodes represents a component of the construct . The interior nodes in the syntax tree are operators supported by the BRAWL language and the leaf nodes represent the operands of these operators. The AST written for BRAWL also implements the static semantic analyzer and the code generator.

6. Testing Plan

6.1 Source and target language programs

The following section shows three source language programs along with the generated C++ target code:

Source Language Listing #1:

Simple BRAWL program for ***conditional checking***:

```
int a = 0;
int b = 1;

int main(){
      if((a < b))
      {
            a = a + b;
      }
      else
      {
            b = b - a;
      }
}
```

Target Language Listing #1:

The following listing shows the C++ code generated for the program shown above:

```
#include "graph.hpp"

int a = 0;
int b = 1;

int main(){
      if(((a < b) != 0))
      {
            (a = (a + b));
      }
      else
      {
            (b = (b - a));
      }
}
```

Source Language Listing #2:

The following listing shows a more complex program which uses the ***foreach*** and the ***foreach where*** looping constructs:

```
int i = 0;

int A(int a){
      return a;
}

int main(){

      while(i <= 10){
            i = i + 1;
      }

      edge eA[0];

      eA + [1, 2, 3];
      eA + [0, 1, 2];
      int w = 0;

      foreach (e in eA) {
            w = w + sizeof e;
      }

      graph g = $3, [1, 2, 3], [1, 0, 2]$;
      w = 0;

      foreach (e in g->2 where sizeof e > 2){
            w = w + sizeof e;
            A(w);
      }
}
```

Target Language Listing #2:

The following listing shows the C++ code generated for the program shown above:

```
#include "graph.hpp"

int i = 0;

int A(int a){
return a;
}

int main(){
      while(((i <= 10) != 0)){
```

```
            (i = (i + 1));
      }

      vector<Edge> eA(0);
      (eA.push_back((Edge(1,2,3))));
      (eA.push_back((Edge(0,1,2))));
      int w = 0;

      for(int __x = 0; __x < eA.size(); ++__x){
            Edge e = eA[__x];
            (w = (w + e._weight));
      }

            Graph                            g                             =
            Graph(3).addEdge((Edge(1,2,3))).addEdge((Edge(1,0,2)))
            ;(w = 0);

            for(int  __x  =  0;  __x  <  g.outIncidence(2).size();
            ++__x){
            Edge e = g.outIncidence(2)[__x];

            if(((e._weight > 2) != 0)){
                  (w = (w + e._weight));
                  A(w);
            }
      }
}
```

The foreach and foreach where looping constructs are allow users to write concise looping constructs. A separate test suite is added for the two constructs since they are an important feature that *BRAWL* provides.

6.2 Test Suites

Test suite #*1* (negative test case)

The following test suite tests some of the basic types in the BRAWL language. As shown below this is a **negative test case**, and should print errors because of type mismatches and variable redeclarations. These test suite regress the basic types commonly used by users.

```
int a = 0;
edge a = [0, 0, 0];
int a = 0.0;
```

Test suite #*2* (negative test case)

The following test suite tests a function ***foo*** which accepts a string but is passed an integer value. The test suite was added to test function calls and also test the type checking of arguments passed to functions.

```
int foo(string a){
      return 0;
}
foo(10);
```

Test suite #*3*

The following test suite will throw an error because of the undefined function call to ***undefined***. The test suite is added as a negative test and expects an error to be thrown. The test is required to effectively test calls to functions which have not yet been defined.

```
int print(int a){
      return 0;
}
undefined("abcd");
```

Test suite #*4*

The following test suite tests the ***foreach*** construct of the language. The foreach is an important construct both in terms of its usage and also the elegant way it is implemented by the language. The construct has to be tested because loops are important contructs of any programming language.

```
int main(){

      graph g = $5, [0, 1, 2], [1, 2, 3], [0, 2, 3], [1, 3, 3],
[2, 4, 7]$;

      foreach (e in g->1){
            print (sizeof e);
      }
}
```

6.3 Test Automation

The following shell script invokes the brawl compiler on all the test programs and diffs the outputof the executed programs with the expected logs. The shell script redirects the errors to an error file with the error message and the file name where the error was detected.

```
#!/bin/sh
for f in `ls *.bwl`
do
  ../brawlc.sh "$f";
  fname=`basename "$f" .bwl`.cpp;
  expected="expected/"$fname;
  diff $fname $expected | grep -i "<"  && echo "Difference found
in $fname"
  rm $fname -f
done
echo "All passed"
```

7. Lessons Learned

Rajesh Venkataraman

Design the type system well. Understand the semantics of the target language before deciding on one! (We wanted to generate OCaml code, but, being novices, ran into problems).

Amoghavarsha Ramappa

Learn language of implementation in the first month, not in the last month! Every team needs a *dictator* in it to delegate work. Make sure you have a *dictator* in your team. The project effectively meets deadlines and implements most of the features initially proposed.

8. Appendix

8.1 Source Code

```
(* file: main.ml
     Author: Rajesh Venkataraman
*)

let main () =
  try
    let lexbuf = Lexing.from_channel stdin in
    while true do
      let prog = BRAWLparse.program BRAWLlex.token lexbuf
        in BRAWLcodegen.cg_prog prog
    done
  with End_of_file -> exit 0
  | Parsing.Parse_error -> exit 1
  | BRAWLcodegen.SemanticError -> (
           print_endline !BRAWLcodegen.s_error;
           exit 1
      )

let _ = Printexc.print main ()


(** ****************************************************** ***)

(* Author: Harish JP *)

type location = BRAWLlocation.t
type data_type = BRAWLsymbol.data_type
type id = string
type t_id = data_type * id
type arg_list =
       Arglist0
      | Arglist1 of t_id
      | Arglist2 of t_id * arg_list

type expression = expression_detail * location

and expression_detail =
       NullExpr
      | Int of int
      | Float of float
      | String of string
      | Variable of id
      | Edge of expression * expression * expression
```

```
      | Graph of expression * exp_list
      | Add of expression * expression
      | Sub of expression * expression
      | Mul of expression * expression
      | Div of expression * expression
      | Mod of expression * expression
      | Or of expression * expression
      | And of expression * expression
      | Equal of expression * expression
      | Less of expression * expression
      | Greater of expression * expression
      | Neq of expression * expression
      | Leq of expression * expression
      | Geq of expression * expression
      | Assignment of expression * expression
      | AddAssign of expression * expression
      | SubAssign of expression * expression
      | MulAssign of expression * expression
      | DivAssign of expression * expression
      | ModAssign of expression * expression
      | FunCall of id * exp_list
      | Array of id * expression
      | Collection of exp_list
      | Uminus of expression
      | Not of expression
      | InIncidence of expression
      | OutIncidence of expression
      | GInIncidence of expression * expression
      | GOutIncidence of expression * expression
      | Sizeof of expression

and exp_list =
        Explist0
      | Explist1 of expression
      | Explist2 of expression * exp_list

type statement =
        Function of data_type * id * arg_list * statements
      | Return of expression
      | If of expression * statements * statements
      | While of expression * statements
      | Foreach of id * expression * statements
      | Foreachw of id * expression * expression * statements
      | ExpressionList of exp_list
      | Declaration of data_type * id * expression
      | ArrDeclaration of data_type * id * expression *
      expression

and statements =
        Statement0
      | Statement1 of statement
      | Statement2 of statement * statements
```

```
type prog = Program of statements

(** ************************************************* ***)

(*code generator
     Authors : Harish JP, Rajesh Venkataraman, Amoghavarsha
     Ramappa
*)

open BRAWLast
open BRAWLsymbol
open Random

exception SemanticError
let mainStatements = ref ""

let s_error = ref ""

let fn_table = Hashtbl.create 64

let raise_error msg =
     s_error := msg;
     raise SemanticError

let expr_to_bool (estr, etype) =
     match etype with
     SymInt -> "(" ^ estr ^ " != 0)"
     | SymFloat -> "(" ^ estr ^ " != 0.)"
     | _ -> raise_error "Cannot convert to boolean"

let string_of_type a =
     match a with
     SymInt -> "int"
     | SymFloat -> "float"
     | SymString -> "string"
     | SymEdge -> "Edge"
     | SymGraph -> "Graph"
     | _ -> raise_error "Invalid type"

let list_reduce lst fxn startValue =
     let rec do_reduce lst currValue =
          match lst with
          [] -> currValue
          | hd :: tl -> do_reduce tl (fxn hd currValue)
     in do_reduce lst startValue

let unify_exprlist exprl =
     let elist = List.map (fun (estr, etype) -> estr) exprl
     in let rec reduce_type (nstr, newType) currType =
          if currType = newType && currType != Undefined then
          currType
          else match (currType, newType) with
```

```
                (SymInt, SymFloat) -> SymFloat
                | (SymFloat, SymInt) -> SymFloat
                | (Undefined, Undefined) -> raise_error "Expr
                cannot have undefined type"
                | (Undefined, _) -> newType
                | (_, _) -> raise_error "Expr is not of uniform
                type"
     in let etype = list_reduce exprl (reduce_type) Undefined
     in (elist, etype)

type parser_state = int

(* utility functions *)

let output_string str = (
     print_string str
)

let output_line str = (
     print_endline str
)

let begin_block (blockCount) =
     start_scope blockCount;
     output_line "{";
     blockCount + 1

let end_block (blockCount) =
     if blockCount != 0 then
          output_line "}";
          if not (end_scope blockCount) then
               raise_error "Unknown error"

(* Code generation routines *)
let rec cg_prog ast =
     match ast with Program(statements) -> print_string
     "#include \"graph.hpp\"\n"; cg_statements 0 statements

and cg_statements state ast =
     match ast with
       Statement0 -> ()
     | Statement1(a) -> cg_statement state a
     | Statement2(a,b) -> (cg_statement state a; cg_statements
     state b)

and cg_statement state ast =
     match ast with
     Function(a, b, c, d) -> (
          let arglist = cg_arg_list c in (
               let rec gen_argnames lst =
                    match lst with
                    [] -> ("", [], [])
```

```
                    | (atype, astr) :: [] -> ((string_of_type
                    atype) ^ " " ^ astr, [atype], [(astr,
                    atype)])
                    | (atype, astr) :: tl ->
                         match gen_argnames tl with
                         (m, n, o) -> ((string_of_type atype) ^
                    " " ^ astr ^ ", " ^ m, atype :: n, (astr,
                    atype) :: o)
                    in let (estr, etypes, slist) = gen_argnames
                    arglist in
                    (if (addSymbol b SymFunction) = false then
                    raise_error (b ^ " already defined"));
                    output_string ((string_of_type a) ^ " " ^ b
                    ^ "(");
                    Hashtbl.add fn_table b (a, etypes);
                    output_string estr;
                    output_string ")";
                    let state = begin_block state
                    in let rec add_list lst =
                         match lst with [] -> ()
                         | (astr, atype) :: tl ->
                              if false = (addSymbol astr
                         atype) then
                                        raise_error "argument
                                        repeated"
                              else add_list tl
                    in add_list slist;
                         cg_statements state d;
                         end_block state
          )
     )
     | Return(a) ->
          let (estr, etype) = cg_expression a in
               output_line ("return " ^ estr ^ ";") (* TODO:
               type check with function type required *)
     | While(a, b) -> (
               output_string  ("while(" ^ (expr_to_bool
               (cg_expression a)) ^ ")");
               let state = begin_block state in
               cg_statements state b;
               end_block state
     )
     | Foreach(a, b, c) -> (
          let (estr, etype) = cg_expression b in (
               output_string ("for(int __x = 0; __x < " ^ estr ^
               ".size(); ++__x)");
               let state = begin_block state in
                              output_line ((string_of_type
               (array_to_sym etype)) ^ " " ^ a ^ " = " ^ estr ^
               "[__x];");
               (if (addSymbol a (array_to_sym etype)) = true
               then cg_statements state c);
```

```
            end_block state
        )
    )
    | Foreachw(a, b, c, d) -> (
        let (estr, etype) = cg_expression b in (
            output_string ("for(int __x = 0; __x < " ^ estr ^
            ".size(); ++__x)");
            let state = begin_block state in
            output_line ((string_of_type (array_to_sym
            etype)) ^ " " ^ a ^ " = " ^ estr ^ "[__x];");
            (if (addSymbol a (array_to_sym etype)) = true
            then
                output_line ("if(" ^ (expr_to_bool
                (cg_expression c)) ^ "){");
                cg_statements state d;
                output_line "}");
                end_block state
        )
    )
    | If(a, b, c) -> (
                output_line ("if(" ^ (expr_to_bool
                (cg_expression a)) ^ ")");
                (let state = begin_block state in
                cg_statements state b;
                end_block state
        );
                if c != Statement0 then (
                    output_string "else ";
                    let state = begin_block state in
                    cg_statements state c;
                    end_block state
        )
    )
    | ExpressionList(a) -> (
        let elist = cg_exp_list a
        in let rec p_expl lst =
            match lst with
            [] -> ()
            | (x, y) :: [] -> output_string x
            | (x, y) :: tl ->
                output_string (x ^ ", ");
                p_expl tl
        in p_expl elist;
        output_line ";"
    )
    | Declaration(a, b, c) ->(
        if (addSymbol b a) == true then (
            output_string ((string_of_type a) ^ " " ^ b ^ " =
            ");
            (match c with
            (NullExpr, _) ->
                let def_value = match a with
```

```
                                SymInt -> "0"
                                | SymFloat -> "0.0"
                                | SymString -> "\"\""
                                | SymEdge -> "Edge()"
                                | SymGraph -> "Graph()"
                                | _ -> ""
                      in output_string def_value
                 | (expr, _) ->
                      let (estr, etype) = cg_expression_detail
                      expr in
                           if a == etype then
                                output_string estr
                           else
                                match (a, etype) with
                           (SymFloat, SymInt) -> output_string
                           ("(float)( " ^ estr ^ ")")
                                | (_, _) -> raise_error "Type
                           conversion error"
                           );
                           output_line ";"
                           ) else raise_error ("Variable already
                           defined: " ^ b)
     )
     | ArrDeclaration(a, b, c, d) -> (
          let typestr = string_of_type a
          in let (indexStr, indexType) = cg_expression c
          in if indexType != SymInt then raise_error ("Index
          type has to be int, got: "  ^ indexStr)
          else
          (if false = addSymbol b (sym_to_array a) then
          raise_error ("symbol redefined " ^ b));
               output_line ("vector<" ^ typestr ^ "> " ^ b ^ "("
               ^ indexStr ^ ");")
               (* cg_expression d; *)
     )

and cg_exp_list ast =
     match ast with
       Explist0 -> []
     | Explist1(a) -> (cg_expression a) :: []
     | Explist2(a, b) -> (cg_expression a) :: (cg_exp_list b)

and cg_id ast = (* string *)
     ()

and cg_arg_list ast =
     match ast with
       Arglist0 -> []
     | Arglist1(a) -> [a]
     | Arglist2(a, b) -> a :: cg_arg_list b

and cg_expression ast =
```

```
      match ast with
      (a, b) ->  cg_expression_detail a

and cg_expression_detail ast =
      match ast with
        NullExpr -> ("", Undefined)
      | Int(a) -> ((string_of_int a), SymInt)
      | Float(a) -> ((string_of_float a), SymFloat)
      | String(a) -> ("\"" ^ a ^ "\"", SymString)
      | Variable(a) -> (
            let atype = typeof a in
            if atype = Undefined then
                  raise_error ("Undefined variable: " ^ a)
            else
                  (a, atype)
      )
      (*
      | Edge(a, b, c) -> ("", Undefined)
      | Graph(a, b) -> ("", Undefined)
      *)
      | Edge(a, b, c) -> (
            let (astring, atype) = cg_expression a
            and (bstring, btype) = cg_expression b
            and (cstring, ctype) = cg_expression c
            in match(atype, btype, ctype) with
                  (SymInt, SymInt, SymInt) -> ("(Edge(" ^ astring ^
                  "," ^ bstring ^ "," ^ cstring ^ "))", SymEdge)
      | (_, _, _) -> raise_error "All edge properties must be
      integers";
      )
      | Graph(a, b) -> (
            let (astring, atype) = cg_expression a
            and graphString = ref ""
            and elist = cg_exp_list b
            in match atype with
                  SymInt ->
                        let rec p_expl lst =
                              match lst with
                              [] -> ()
                                          | (x, y) :: [] ->
            graphString := !graphString ^ ".addEdge(" ^ x ^ ")";
                              | (x, y) :: tl ->
                                    graphString := !graphString ^
                                    ".addEdge(" ^ x ^ ")";
                                    p_expl tl
                        in p_expl elist;
                        ("Graph(" ^ astring ^ ")" ^ !graphString,
                        SymGraph)
                  | _ -> raise_error "The number of vertices must
                  be integral"
      )
      | Add(a, b) -> (
```

```
        let (astring, atype) = cg_expression a
        and (bstring, btype) = cg_expression b
        in match (atype, btype) with
             (SymInt, SymInt) -> ("(" ^ astring ^ " + " ^
             bstring ^ ")", SymInt)
             | (SymInt, SymFloat) -> ("((float " ^ astring ^
             ") + " ^ bstring ^ ")", SymFloat)
             | (SymFloat, SymInt) -> ("(" ^ astring ^ " +
             (float " ^ bstring ^ "))", SymFloat)
             | (SymFloat, SymFloat) -> ("(" ^ astring ^ " + "
             ^ bstring ^ ")", SymFloat)
             | (SymGraph, SymEdge) -> ("(" ^ astring ^
             ".addEdge(" ^ bstring ^ "))", SymGraph)
             | (SymIntArray, SymInt) -> ("(" ^ astring ^
             ".push_back(" ^ bstring ^ "))",SymIntArray)
             | (SymFloatArray, SymFloat) -> ("(" ^ astring ^
             ".push_back(" ^ bstring ^ "))",SymFloatArray)
             | (SymStringArray, SymString) -> ("(" ^ astring ^
             ".push_back(" ^ bstring ^ "))",SymStringArray)
             | (SymEdgeArray, SymEdge) -> ("(" ^ astring ^
             ".push_back(" ^ bstring ^ "))",SymEdgeArray)
             | (SymGraphArray, SymGraph) -> ("(" ^ astring ^
             ".push_back(" ^ bstring ^ "))",SymGraphArray)
             | (_, _) -> raise_error "Invalid type for
             addition"
)
             | Sub(a, b) -> (
                  let (astring, atype) = cg_expression a
                  and (bstring, btype) = cg_expression b
                  in match (atype, btype) with
                  (SymInt, SymInt) -> ("(" ^ astring ^ " - "
                  ^ bstring ^ ")", SymInt)
             | (SymInt, SymFloat) -> ("((float " ^ astring ^
        ") - " ^ bstring ^ ")", SymFloat)
             | (SymFloat, SymInt) -> ("(" ^ astring ^ " -
             (float " ^ bstring ^ "))", SymFloat)
             | (SymFloat, SymFloat) -> ("(" ^ astring ^ " - "
             ^ bstring ^ ")", SymFloat)
             | (_, _) -> raise_error "Invalid type for
             subtraction"
        )

| Mul(a, b) -> (
        let (astring, atype) = cg_expression a
        and (bstring, btype) = cg_expression b
        in match (atype, btype) with
        (SymInt, SymInt) -> ("(" ^ astring ^ " * " ^ bstring ^
        ")", SymInt)
        | (SymInt, SymFloat) -> ("((float " ^ astring ^ ") * "
        ^ bstring ^ ")", SymFloat)
        | (SymFloat, SymInt) -> ("(" ^ astring ^ " * (float "
        ^ bstring ^ "))", SymFloat)
```

```
            | (SymFloat, SymFloat) -> ("(" ^ astring ^ " * " ^
            bstring ^ ")", SymFloat)
            | (_, _) -> raise_error "Invalid type for
            multiplication"
        )

        | Div(a, b) -> (
            let (astring, atype) = cg_expression a
            and (bstring, btype) = cg_expression b
            in match (atype, btype) with
            (SymInt, SymInt) -> ("(" ^ astring ^ " / " ^ bstring ^
            ")", SymInt)
            | (SymInt, SymFloat) -> ("((float " ^ astring ^ ") / "
            ^ bstring ^ ")", SymFloat)
            | (SymFloat, SymInt) -> ("(" ^ astring ^ " / (float "
            ^ bstring ^ "))", SymFloat)
            | (SymFloat, SymFloat) -> ("(" ^ astring ^ " / " ^
            bstring ^ ")", SymFloat)
            | (_, _) -> raise_error "Invalid type for division"
        )

        | Mod(a, b) -> (
            let (astring, atype) = cg_expression a
            and (bstring, btype) = cg_expression b
            in match (atype, btype) with
            (SymInt, SymInt) -> ("(" ^ astring ^ " % " ^ bstring ^
            ")", SymInt)
            | (_, _) -> raise_error "Invalid type for modulo"
        )

        | Or(a, b) -> (
            let (astring, atype) = cg_expression a
            and (bstring, btype) = cg_expression b
            in match (atype, btype) with
            (SymInt, SymInt) -> ("(" ^ astring ^ " || " ^ bstring
            ^ ")", SymInt)
            | (SymFloat, SymInt) -> ("(" ^ astring ^ " || " ^
            bstring ^ ")", SymInt)
            | (SymInt, SymFloat) -> ("(" ^ astring ^ " || " ^
            bstring ^ ")", SymInt)
            | (SymFloat, SymFloat) -> ("(" ^ astring ^ " || " ^
            bstring ^ ")", SymInt)
            | (_, _) -> raise_error "Invalid type for logical or"
        )

        | And(a, b) -> (
            let (astring, atype) = cg_expression a
            and (bstring, btype) = cg_expression b
            in match (atype, btype) with
            (SymInt, SymInt) -> ("(" ^ astring ^ " && " ^ bstring
            ^ ")", SymInt)
```

```
        | (SymFloat, SymInt) -> ("(" ^ astring ^ " && " ^
        bstring ^ ")", SymInt)
        | (SymInt, SymFloat) -> ("(" ^ astring ^ " && " ^
        bstring ^ ")", SymInt)
        | (SymFloat, SymFloat) -> ("(" ^ astring ^ " && " ^
        bstring ^ ")", SymInt)
        | (_, _) -> raise_error "Invalid type for logical and"
    )

    | Equal(a, b) -> (
        let (astring, atype) = cg_expression a
        and (bstring, btype) = cg_expression b
        in match (atype, btype) with
        (SymInt, SymInt) -> ("(" ^ astring ^ " == " ^ bstring
        ^ ")", SymInt)
        | (SymFloat, SymFloat) -> ("(" ^ astring ^ " == " ^
        bstring ^ ")", SymInt)
        | (SymFloat, SymInt) -> ("(" ^ astring ^ " ==
        ((float)(" ^ bstring ^ ")))", SymInt)
        | (SymInt, SymFloat) -> ("(((float)(" ^ astring ^ "))
        == " ^ bstring ^ ")", SymInt)
        | (_, _) -> raise_error "Invalid type for equals"
    )

    | Neq(a, b) -> (
        let (astring, atype) = cg_expression a
        and (bstring, btype) = cg_expression b
        in match (atype, btype) with
        | (SymInt, SymInt) -> ("(" ^ astring ^ " != " ^
        bstring ^ ")", SymInt)
        | (SymFloat, SymFloat) -> ("(" ^ astring ^ " != " ^
        bstring ^ ")", SymInt)
        | (SymFloat, SymInt) -> ("(" ^ astring ^ " !=
        (float)(" ^ bstring ^ "))", SymInt)
        | (SymInt, SymFloat) -> ("((float)(" ^ astring ^ ") !=
        " ^ bstring ^ ")", SymInt)
        | (_, _) -> raise_error "Invalid type for not equals"
    )

    | Less(a, b) -> (
        let (astring, atype) = cg_expression a
        and (bstring, btype) = cg_expression b
        in match (atype, btype) with
                   | (SymInt, SymInt) -> ("(" ^ astring ^ " <
        " ^ bstring ^ ")", SymInt)
        | (SymFloat, SymFloat) -> ("(" ^ astring ^ " < " ^
        bstring ^ ")", SymInt)
        | (SymFloat, SymInt) -> ("(" ^ astring ^ " < (float)("
        ^ bstring ^ "))", SymInt)
        | (SymInt, SymFloat) -> ("((float)(" ^ astring ^ ") <
        " ^ bstring ^ ")", SymInt)
        | (_, _) -> raise_error "Invalid type for Less"
    )
```

```
    | Greater(a, b) -> (
         let (astring, atype) = cg_expression a
         and (bstring, btype) = cg_expression b
         in match (atype, btype) with
                   | (SymInt, SymInt) -> ("(" ^ astring ^ " >
         " ^ bstring ^ ")", SymInt)
         | (SymFloat, SymFloat) -> ("(" ^ astring ^ " > " ^
         bstring ^ ")", SymInt)
         | (SymFloat, SymInt) -> ("(" ^ astring ^ " > (float)("
         ^ bstring ^ "))", SymInt)
         | (SymInt, SymFloat) -> ("((float)(" ^ astring ^ ") >
         " ^ bstring ^ ")", SymInt)
         | (_, _) -> raise_error "Invalid type for Greater"
    )

    | Leq(a, b) -> (
         let (astring, atype) = cg_expression a
         and (bstring, btype) = cg_expression b
         in match (atype, btype) with
         | (SymInt, SymInt) -> ("(" ^ astring ^ " <= " ^
         bstring ^ ")", SymInt)
         | (SymFloat, SymFloat) -> ("(" ^ astring ^ " <= " ^
         bstring ^ ")", SymInt)
         | (SymFloat, SymInt) -> ("(" ^ astring ^ " <=
         (float)(" ^ bstring ^ "))", SymInt)
         | (SymInt, SymFloat) -> ("((float)(" ^ astring ^ ") <=
         " ^ bstring ^ ")", SymInt)
         | (_, _) -> raise_error "Invalid type for Less or
         equal"
    )

    | Geq(a, b) -> (
         let (astring, atype) = cg_expression a
         and (bstring, btype) = cg_expression b
         in match (atype, btype) with
         | (SymInt, SymInt) -> ("(" ^ astring ^ " >= " ^
         bstring ^ ")", SymInt)
         | (SymFloat, SymFloat) -> ("(" ^ astring ^ " >= " ^
         bstring ^ ")", SymInt)
         | (SymFloat, SymInt) -> ("(" ^ astring ^ " >=
         (float)(" ^ bstring ^ "))", SymInt)
         | (SymInt, SymFloat) -> ("((float)(" ^ astring ^ ") >=
         " ^ bstring ^ ")", SymInt)
         | (_, _) -> raise_error "Invalid type for Greater or
         equal"
    )
    | Assignment(a, b) -> (
         let (astring, atype) = cg_expression a
         and (bstring, btype) = cg_expression b
         in if atype = btype then
              ("(" ^ astring ^ " = " ^ bstring ^ ")", atype)
         else match (atype, btype) with
```

```
            | (SymFloat, SymInt) -> ("(" ^ astring ^ " =
        (float)(" ^ bstring ^ "))", SymFloat)
        | (_, _) -> raise_error "type mismatch for assignment"
    )
    | AddAssign(a, b) -> (
        let (astring, atype) = cg_expression a
        and (bstring, btype) = cg_expression b
        in match (atype, btype) with
        | (SymInt, SymInt) -> ("(" ^ astring ^ " += " ^
        bstring ^ ")", SymInt)
        | (SymFloat, SymFloat) -> ("(" ^ astring ^ " += " ^
        bstring ^ ")", SymInt)
        | (SymFloat, SymInt) -> ("(" ^ astring ^ " +=
        (float)(" ^ bstring ^ "))", SymInt)
        | (_, _) -> raise_error "type mismatch for +="
    )
    | SubAssign(a, b) -> (
        let (astring, atype) = cg_expression a
        and (bstring, btype) = cg_expression b
        in match (atype, btype) with
        | (SymInt, SymInt) -> ("(" ^ astring ^ " += " ^
        bstring ^ ")", SymInt)
        | (SymFloat, SymFloat) -> ("(" ^ astring ^ " += " ^
        bstring ^ ")", SymInt)
        | (SymFloat, SymInt) -> ("(" ^ astring ^ " +=
        (float)(" ^ bstring ^ "))", SymInt)
        | (_, _) -> raise_error "type mismatch for +="
    )
    | MulAssign(a, b) -> (
        let (astring, atype) = cg_expression a
        and (bstring, btype) = cg_expression b
        in match (atype, btype) with
        | (SymInt, SymInt) -> ("(" ^ astring ^ " *= " ^
        bstring ^ ")", SymInt)
        | (SymFloat, SymFloat) -> ("(" ^ astring ^ " *= " ^
        bstring ^ ")", SymInt)
        | (SymFloat, SymInt) -> ("(" ^ astring ^ " *=
        (float)(" ^ bstring ^ "))", SymInt)
        | (_, _) -> raise_error "type mismatch for *="
    )
    | DivAssign(a, b) -> (
        let (astring, atype) = cg_expression a
        and (bstring, btype) = cg_expression b
        in match (atype, btype) with
            | (SymInt, SymInt) -> ("(" ^ astring ^ " /= " ^
        bstring ^ ")", SymInt)
        | (SymFloat, SymFloat) -> ("(" ^ astring ^ " /= " ^
        bstring ^ ")", SymInt)
        | (SymFloat, SymInt) -> ("(" ^ astring ^ " /=
        (float)(" ^ bstring ^ "))", SymInt)
        | (_, _) -> raise_error "type mismatch for /="
    )
    | ModAssign(a, b) -> (
```

```
        let (astring, atype) = cg_expression a
        and (bstring, btype) = cg_expression b
        in match (atype, btype) with
        | (SymInt, SymInt) -> ("(" ^ astring ^ " %= " ^
        bstring ^ ")", SymInt)
        | (_, _) -> raise_error "type mismatch for %="
    )
    | Uminus(a) -> (
        let (astring, atype) = cg_expression a
        in match atype with
        SymInt -> ("(-" ^ astring ^ ")", atype)
        | SymFloat -> ("(-" ^ astring ^ ")", atype)
        | _ -> raise_error "Cannot apply unary minus"
    )
    | Not(a) -> (
        let (astring, atype) = cg_expression a
        in match atype with
        SymInt -> ("(!" ^ astring ^ ")", atype)
        | SymFloat -> ("(!" ^ astring ^ ")", atype)
        | _ -> raise_error "Cannot apply not"
    )
    | FunCall(a, b) -> (
        (if (typeof a) != SymFunction then raise_error (a ^ "
        is not a function"));
        let exps = cg_exp_list b
        in let (atype, aparams) = Hashtbl.find fn_table a
        in let check_compat str type1 type2 =
            if type1 = type2 then str
            else match (type1, type2) with
                (SymInt, SymFloat) -> str
                |(_,_) -> raise_error "Type mismatch in
                arguments"
                in let rec param_string params argtypes =
                match (params, argtypes) with
                ((pstr, ptype) :: [], argtype :: []) ->
                    check_compat pstr ptype argtype
                | ((pstr, ptype) :: ptail, argtype ::
                argtail) ->
                    (check_compat pstr ptype argtype) ^ ",
                    " ^ (param_string ptail argtail)
                | ([], []) -> ""
                | (_::_, []) -> raise_error ("Too many
                arguments passed to function: " ^ a)
                | ([], _::_) -> raise_error ("Too less
                arguments passed to function: " ^ a)
                in (a ^ "(" ^ (param_string exps aparams) ^
                ")", atype)
    )
    | Array(a, b) -> (
        let (index, indexType) = cg_expression b
        in if indexType != SymInt then raise_error "Array
        index has to be an integer"
        else let retType = array_to_sym (typeof a)
```

```
            in (a ^ "[" ^ index ^ "]", retType)
      )
      (*
      | Collection(a) -> unify_exprlist (cg_exp_list a)
      *)
      | InIncidence(a) -> (
            let (astring, atype) = cg_expression a
            in match atype with
                  SymEdge -> (astring ^ "._target", SymInt)
                  | _ -> raise_error (astring ^ " is not an edge")
      )
      | OutIncidence(a) -> (
            let (astring, atype) = cg_expression a
            in match atype with
                  SymEdge -> (astring ^ "._source", SymInt)
                  | _ -> raise_error (astring ^ " is not an edge")
      )
      | GInIncidence(a, b) -> (
            let (astring, atype) = cg_expression a
            and (bstring, btype) = cg_expression b
            in match (atype, btype) with
                  (SymGraph, SymInt) -> (astring ^ ".inIncidence("
                  ^ bstring ^ ")", SymEdgeArray)
                  | (_, _) -> raise_error "Invalid types for in-
                  incidence on this graph"
      )
      | GOutIncidence(a, b) -> (
            let (astring, atype) = cg_expression a
            and (bstring, btype) = cg_expression b
            in match (atype, btype) with
                  (SymGraph, SymInt) -> (astring ^ ".outIncidence("
                  ^ bstring ^ ")", SymEdgeArray)
                  | (_, _) -> raise_error "Invalid types for out-
                  incidence on this graph"
      )
      | Sizeof(a) -> (
            let (astring, atype) = cg_expression a
            in match(atype) with
                  SymEdge -> (astring ^ "._weight", SymInt)
                  | SymGraph -> (astring ^ "._nVertices", SymInt)
                  | SymIntArray -> (astring ^ ".size()", SymInt)
                  | SymFloatArray -> (astring ^ ".size()", SymInt)
                  | SymStringArray -> (astring ^ ".size()", SymInt)
                  | SymEdgeArray -> (astring ^ ".size()", SymInt)
                  | SymGraphArray -> (astring ^ ".size()", SymInt)
                  | _   -> raise_error "Invalid type for sizeof"
      )
;;


(** *************************************************** ***)
```

```
(* Lexer
     Authors : Rajesh Venkataraman, Amoghavarsha Ramappa
 *)
{
  let strConstant = ref ""
  open BRAWLparse

}

let digit = ['0'-'9']
let id = ['a'-'z' 'A' - 'Z' '_'] ['a'-'z' 'A' - 'Z' '0'-'9' '_']*

rule token = parse
  | digit+ as inum
      {
        INT (int_of_string inum)
      }
  | digit+ '.' digit* as fnum
      {
        FLOAT (float_of_string fnum)
      }
  | '"'
      {
        strConstant := "";
        stringToken lexbuf
      }
  | [' ' '\t']  { token lexbuf } (* eat up whitespace *)
  | '\n' {
        let pos = lexbuf.Lexing.lex_curr_p in
              lexbuf.Lexing.lex_curr_p <- { pos with
                    Lexing.pos_lnum = pos.Lexing.pos_lnum + 1;
                    Lexing.pos_bol = pos.Lexing.pos_cnum;
              };
        token lexbuf
    }
  | "if" { IF }
  | "else" { ELSE }
  | "while" { WHILE }
  | "foreach" { FOREACH }
  | "in" { IN }
  | "sizeof" { SIZEOF }
  | "where" { WHERE }
  | "return" { RETURN }
  | "int" { INTK }
  | "float" { FLOATK }
  | "string" { STRINGK }
  | "edge" { EDGEK }
  | "graph" { GRAPHK }
  | id as text  { ID text }
  | '+' { PLUS }
  | '-' { MINUS }
  | '*' { MULTIPLY }
```

```
  | '/' { DIVIDE }
  | '%' { MODULO }
  | '(' { LPAREN }
  | ')' { RPAREN }
  | '[' { LBRACKET }
  | ']' { RBRACKET }
  | '{' { LBRACE }
  | '}' { RBRACE }
  | '=' { ASSIGNMENT }
  | "+="    { ADDAS }
  | "-="    { SUBAS }
  | "*="    { MULAS }
  | "/="    { DIVAS }
  | "%="    { MODAS }
  | '>'         { GT }
  | '<'         { LT }
  | "=="   { EQ }
  | ">="   { GE }
  | "<="   { LE }
  | "!="   { NE }
  | '!'         { NEGATE }
  | "&&"   { LOGICALAND }
  | "||"   { LOGICALOR }
  | ';'         { SEMICOLON }
  | ','         { COMMA }
  | '$'         { DOLLAR }
  | "->"   { RARROW }
  | "<-"   { LARROW }
  | eof         { EOF }

and stringToken = parse
  | '"'
     {
          STRING     !strConstant
     }
  | '\\'
     {
       escape lexbuf
     }
  | _ as ch
     {
       strConstant := !strConstant ^ (String.make 1 ch);
       stringToken lexbuf
     }
     and escape = parse
  | 'n'
     {
       strConstant := !strConstant ^ "\n";
       stringToken lexbuf
     }
  | 't'
     {
       strConstant := !strConstant ^ "\t";
```

```
        stringToken lexbuf
      }
  | '"'
      {
        strConstant := !strConstant ^ "\"";
        stringToken lexbuf
      }
```

```
(** ************************************************** ***)

(** Location in a source file -- an annotation for AST nodes

Authors : Harish JP, Rajesh Venkataraman

*)

type t = {
    loc_start : Lexing.position;
    loc_end : Lexing.position
  }

let string_of {loc_start = p1; loc_end = p2 } =
  "File \"" ^ p1.Lexing.pos_fname ^ "\" line " ^
  string_of_int p1.Lexing.pos_lnum ^
  " characters " ^ string_of_int (p1.Lexing.pos_cnum -
p1.Lexing.pos_bol) ^
  "-" ^ string_of_int (p2.Lexing.pos_cnum - p1.Lexing.pos_bol)

let of_symbol () = {
  loc_start = Parsing.symbol_start_pos ();
  loc_end = Parsing.symbol_end_pos ()
}

let current lexbuf = {
  loc_start = Lexing.lexeme_start_p lexbuf;
  loc_end =  Lexing.lexeme_end_p lexbuf
}

let nowhere = {
  loc_start = Lexing.dummy_pos;
  loc_end = Lexing.dummy_pos
}


(** ************************************************** ***)
(* Parser

     Authors : Harish JP, Rajesh Venkataraman

*)

%{
```

```
      open BRAWLast

      let parse_error s = (* Called by the parser function on
error *)
            print_endline (s ^ ": " ^ (BRAWLlocation.string_of
(BRAWLlocation.of_symbol () )));
      flush stdout

      let addloc detail = (detail, BRAWLlocation.of_symbol ())

      let debug str = () (*print_endline str*)

%}

%token IF
%token ELSE
%token WHILE
%token FOREACH
%token IN
%token WHERE
%token RETURN

%token <int> INT
%token <float> FLOAT
%token <string> STRING
%token <char> CHAR
%token <string> ID

%token INTK FLOATK STRINGK EDGEK GRAPHK

%token PLUS MINUS MULTIPLY DIVIDE MODULO
%token GT LT EQ GE LE NE
%token NEGATE LOGICALAND LOGICALOR

%token ASSIGNMENT
%token ADDAS SUBAS MULAS DIVAS MODAS

%token LBRACKET RBRACKET
%token LBRACE RBRACE
%token LPAREN RPAREN
%token SEMICOLON
%token COMMA
%token DOLLAR
%token LARROW
%token RARROW

%token EOF
%token SIZEOF

%left ASSIGNMENT
%left ADDAS SUBAS MULAS DIVAS MODAS
%left LOGICALOR
%left LOGICALAND
```

```
%left EQ
%left LT GT LE GE
%left PLUS MINUS
%left MULTIPLY DIVIDE MODULO
%left LARROW RARROW
%left SIZEOF

%nonassoc UMINUS
%nonassoc UNOT
%nonassoc ULARROW
%nonassoc URARROW

%nonassoc LOWER_THAN_ELSE
%nonassoc ELSE

%start program
%type <BRAWLast.prog> program

%%

program:
      stat_list {debug "parsed program"; Program($1)}
      | EOF { raise End_of_file }
;

toplevel_stat:
      stat {Statement1($1)}
      | LBRACE stat_list RBRACE {$2}
      | LBRACE RBRACE { Statement0 }
      | SEMICOLON { Statement0 }
;

stat_list:
      stat {Statement1($1)}
      | stat stat_list {Statement2($1, $2) }
;

stat:
      function_definition { debug "parsed function"; $1}
      | return_statement { debug "parsed return statement"; $1}
      | if_statement { debug "parsed if statement"; $1}
      | while_statement { debug "parsed while statment"; $1}
      | foreach_statement { debug "parsed foreach statment"; $1}
      | foreachw_statement { debug "parsed foreachw statment";
      $1}
      | variable_declaration { debug "parsed variable
      declaration"; $1}
      | expr_statement { debug "parsed expression statement"; $1
}
;

expr_statement:
      expr_list SEMICOLON { ExpressionList($1) }
```

```
;

function_definition:
     tid LPAREN tid_list RPAREN LBRACE stat_list RBRACE {
                match $1 with
                (a, b) -> Function(a, b, $3, $6)
     }
     | tid LPAREN RPAREN LBRACE stat_list RBRACE {
                match $1 with
                (a, b) -> Function(a, b, Arglist0, $5)
     }
;

return_statement:
     RETURN exp SEMICOLON { Return($2) }
;

if_statement:
     IF LPAREN exp RPAREN toplevel_stat %prec LOWER_THAN_ELSE {
If($3, $5, Statement0) }
     | IF LPAREN exp RPAREN toplevel_stat ELSE toplevel_stat {
If($3, $5, $7) }
;

while_statement:
     WHILE LPAREN exp RPAREN toplevel_stat { While($3, $5) }
;

foreach_statement:
     FOREACH LPAREN ID IN exp RPAREN toplevel_stat { Foreach($3,
$5, $7) }
;

foreachw_statement:
     FOREACH LPAREN ID IN exp WHERE exp RPAREN toplevel_stat {
Foreachw($3, $5, $7, $9) }
;

variable_declaration:
     tid SEMICOLON { match $1 with (a, b) -> Declaration(a, b,
     (NullExpr, BRAWLlocation.nowhere)) }
     | tid ASSIGNMENT exp SEMICOLON { match $1 with (a, b) ->
     Declaration(a, b, $3) }
     | tid LBRACKET exp RBRACKET SEMICOLON { match $1 with (a,
     b) -> ArrDeclaration(a, b, $3, (NullExpr,
     BRAWLlocation.nowhere)) }
     | tid LBRACKET exp RBRACKET ASSIGNMENT exp SEMICOLON {
     match $1 with (a, b) -> ArrDeclaration(a, b, $3, $6) }
;

tid_list:
     tid { Arglist1($1) }
     | tid COMMA tid_list { Arglist2($1, $3) }
```

```
;

tid:
     type_name ID { ($1, $2) }
;

type_name:
     INTK { BRAWLsymbol.SymInt }
     | FLOATK { BRAWLsymbol.SymFloat }
     | STRINGK { BRAWLsymbol.SymString }
     | EDGEK { BRAWLsymbol.SymEdge }
     | GRAPHK { BRAWLsymbol.SymGraph }
;

expr_list:
     exp {Explist1($1)}
     | exp COMMA expr_list {Explist2($1,$3)}
;

exp:
            INT { addloc(Int($1)) }
           | FLOAT { addloc(Float($1)) }
           | STRING { addloc(String($1)) }
           | lvalue { $1 }
           | ID LPAREN expr_list RPAREN { addloc(FunCall($1, $3))
}
           | ID LPAREN RPAREN { addloc(FunCall($1, Explist0)) }
           | edge_expression { $1 }
           | graph_expression { $1 }
           | exp PLUS exp { addloc(Add($1, $3)) }
           | exp MINUS exp { addloc(Sub($1, $3)) }
           | exp MULTIPLY exp { addloc(Mul($1, $3)) }
           | exp DIVIDE exp { addloc(Div($1, $3)) }
           | exp MODULO exp { addloc(Mod($1, $3)) }
           | exp EQ exp { addloc(Equal($1, $3)) }
           | exp GT exp { addloc(Greater($1, $3)) }
           | exp LT exp { addloc(Less($1, $3)) }
           | exp GE exp { addloc(Geq($1, $3)) }
           | exp LE exp { addloc(Leq($1, $3)) }
           | exp LOGICALAND exp { addloc(And($1, $3)) }
           | exp LOGICALOR exp { addloc(Or($1, $3)) }
           | MINUS exp %prec UMINUS { addloc(Uminus($2)) }
           | NEGATE exp %prec UNOT { addloc(Not($2)) }
           | LPAREN exp RPAREN { $2 }
           | LARROW exp %prec ULARROW { debug "parsed in
           incidence"; addloc(InIncidence($2)) }
           | RARROW exp %prec URARROW { addloc(OutIncidence($2))
}
           | exp LARROW exp { addloc(GInIncidence($1, $3)) }
           | exp RARROW exp { addloc(GOutIncidence($1, $3)) }
           | SIZEOF exp    { debug "parsed sizeof";
           addloc(Sizeof($2)) }
           | assignment_expression { $1 }
```

```
           | collection_expression { $1 }
;

lvalue:
     ID { addloc(Variable($1)) }
     | ID LBRACKET exp RBRACKET { addloc(Array($1, $3)) }
;

edge_expression:
     LBRACKET exp COMMA exp COMMA exp RBRACKET { addloc(Edge($2,
$4, $6)) }
;

graph_expression:
     DOLLAR exp COMMA expr_list DOLLAR { addloc(Graph($2, $4)) }
;

collection_expression:
     LBRACE expr_list RBRACE { debug "parsed collection
expression"; addloc(Collection($2)) }
;

assignment_expression:
     lvalue ASSIGNMENT exp { addloc(Assignment($1, $3)) }
     | lvalue ADDAS exp { addloc(AddAssign($1, $3)) }
;

%%

(* vim:set ts=4 sw=4 noet: *)
```

```
(** ************************************************** ***)

(** Symbol table: uses a global hash function to give each string
a unique integer identifier

Authors : Harish JP, Amoghavarsha Ramappa

*)

module StringHash = Hashtbl.Make(struct
  type t = string
  let equal x y = x = y
  let hash = Hashtbl.hash
end)

(* The types of the symbol in the system *)
type data_type = Undefined | SymInt | SymFloat | SymString |
SymEdge | SymGraph | SymFunction| SymIntArray | SymFloatArray |
SymStringArray | SymEdgeArray | SymGraphArray

(* Damned global variable for the symbol table *)
let symboltable = ref [StringHash.create 128] ;;
```

```
(* Returns type of the symbol, Undefined if symbol is not found
*)
let typeof name =
	let rec rec_typeof tbl =
		match tbl with
		[] -> Undefined
		| hd :: tl ->
			try StringHash.find hd name
			with Not_found -> rec_typeof tl
	in rec_typeof !symboltable ;;


(* Add a new symbol to symbol table. Returns false if symbol
exists *)
let addSymbol name dType =
	let head = List.hd !symboltable in
		if StringHash.mem head name then
			false
		else
			(StringHash.add head name dType; true);;

(* Adds a new hash, representing the local scope *)
let rec start_scope dummy =
	symboltable := (StringHash.create 128) :: !symboltable ;;

(* Deletes the local scope *)
let rec end_scope dummy =
	match !symboltable with
	[] -> false
	| hd :: tl -> symboltable := tl; true ;;

let sym_to_array sym =
	match sym with
	SymInt -> SymIntArray
	| SymFloat -> SymFloatArray
	| SymString -> SymStringArray
	| SymEdge -> SymEdgeArray
	| SymGraph -> SymGraphArray
	| _ -> Undefined
;;

let array_to_sym a =
	match a with
	SymIntArray -> SymInt
	| SymFloatArray -> SymFloat
	| SymStringArray -> SymString
	| SymEdgeArray -> SymEdge
	| SymGraphArray -> SymGraph
	| _ -> Undefined
;;

(* vim: set ts=4 sw=4 noet: *)
```

```
(** ************************************************** ***)
/* C++ Library
      Authors : Rajesh Venkataraman, Amoghavarsha Ramappa
 */

#ifndef __GRAPH_HPP__
#define __GRAPH_HPP__
#include <vector>
#include <string>
#include <algorithm>

using std::vector;
using std::string;

class Exception {
      string _message;
public:
      Exception(const string& message = ""):_message(message){
      }
      string getMsg(){
            return _message;
      }
      string setMsg(const string& message){
            _message = message;
      }
};

struct Edge {
      unsigned _source, _target;
      int _weight;
      bool operator == (const Edge& e){
            return _source == e._source && _target == e._target;
      }
      Edge(const unsigned& source = 0, const unsigned& target =
      0, const unsigned& weight = 0):_source(source),
      _target(target), _weight(weight) { }

      Edge(const Edge& e):_source(e._source), _target(e._target),
      _weight(e._weight){
      }
};

struct Graph {
      unsigned _nVertices;
      vector<Edge> _edges;

      Graph(const unsigned& nVertices = 0, const vector<Edge>&
      edges = vector<Edge>()):_nVertices(nVertices),
      _edges(edges){}

      Graph& addEdge(Edge e){
```

```
            if(e._source < 0 || e._source > (_nVertices - 1) ||
        e._target < 0 || e._target > (_nVertices - 1) )
                    throw new Exception("Invalid source / target");
            vector<Edge>::iterator iter;
                    if((iter = find(_edges.begin(), _edges.end(), e))
        == _edges.end()) _edges.push_back(e);
            else (*iter)._weight = e._weight;
            return *this;
        }

        vector<Edge> outIncidence(const unsigned vertex) const{
            if(vertex < 0 || vertex > (_nVertices - 1))
                    throw new Exception("Vertex not in graph");
            vector<Edge> retVal;
            int sz = _edges.size();
            for(int i = 0; i < sz; ++i)
                    if(_edges[i]._source == vertex)
            retVal.push_back(Edge(_edges[i]));
            return retVal;
        }

        vector<Edge> inIncidence(const unsigned vertex) const{
            if(vertex < 0 || vertex > (_nVertices - 1))
                    throw new Exception("Vertex not in graph");
            vector<Edge> retVal;
            int sz = _edges.size();
            for(int i = 0; i < sz; ++i)
                    if(_edges[i]._target == vertex)
            retVal.push_back(Edge(_edges[i]));
            return retVal;
        }
};
#endif
```

(** ***)**