Final Report

eGrapher: A Programming Language for Art

Long Long: ll3078@columbia.edu
Xinli Jia: xj2191@columbia.edu
Jiefu Ying: jy2799@columbia.edu
Darren Chen: dsc2155@columbia.edu
Linnan Wang: lw2645@columbia.edu

Outline

1. Introduction

eGrapher offers an innovative way of drawing art with pinpoint accuracy. The language gives the user a wide set of tools to accurately illustrate detailed drawings compared to simple and rigid pre-installed tools such as paint on Windows. The goal of eGrapher is to make drawing on digital systems easier as well as allow for users to draw more complicated paintings through mathematical functions. The project will allow us to also better understand the intersection of mathematics and art. Users will be able to use simple syntax and math expression to draw graphs, diagrams, and more complicated objects through simple objects.

eGrapher is a language for art, for painting freely on board, every graphic shape is treated independently as an instance of basic structures, and the language handles the rest, including shape translation and drawing work. In order to implement this goals, we need to expand greatly from the base MicroC language. MircoC is a good introduction of what a language looks like basically, but it is too simple. We want to have our user coding in free style when using eGrapher, which means they can declare variables and functions wherever they want, and plot things easily using structures.

Graphic coding always includes lots of parameters. To do this in an elegant way, list is a must. List is different from arrays, it provides a flexible method to add, remove, and modify elements. Using list to store information in structure and call build-in functions to draw shapes on board, this is the whole lifecycle of eGrapher programming language.

So our work mainly contains three parts:

1. Reconstruction microC and realize free declaration and calling style.
2. Add basic expansions like printf with multiple function and string support.

3. Linking to c library, add list and struct support.
4. Call opengl functions to implement drawing.

We go through this line and try to build a brand new art programming language.

Functions

2. Language Tutorial

Write a valid file_name.eg file with eGrapher syntax. Inside the eGrapher folder, run

```
$ make
```

This creates the eGrapher to LLVM compiler, “egrapher.native”. Once the compiler is made, run

```
$ ./hellotest.sh file_name.eg
```

to compile a eGrapher program to LLVM and run the file. This script compiles the backend data structure modules in the “src” directory from C to LLVM bitcode, links the bitcode with the output of “egrapher.native,” produces an LLVM IR, and compiles it into an executable using CLANG. After you finish running the program, just run

```
$ make clean
```

to delete all generated file, the IRs and executable after execution.

The following brief eGrapher code demonstrates how to create the mandatory main function, calls the builtin “print” function using a string literal.

```
int main()
{ print("Hello World!\n");
}
```

Result:

```
Hello World!
```

3. Language Manual

1.Introduction

eGrapher is a general purpose, with additional drawing function language. The principal is simplicity, write a c like language that can draw. eGrapher is a high level language that utilizes LLVM IR to abstract away hardware implementation of code. Utilizing the LLVM as a backend allows for automatic garbage collection of variables as well. eGrapher is a strongly typed programming language, meaning that at compile time the language will be typechecked, thus preventing runtime errors of type. eGrapher is static scoping. The following is a reference manual for using eGrapher. It describes in detail the ideas and design thoughts behind lexical conventions, basic types scoping rules, built-in functions, and also gives a sample program with its output.

2. Types and Literals

eGrapher has a set of types and literals which are similar to those of most programming languages, and most of these specifications match those of the compiled language. Types in eGrapher can be largely classified into primitive types and nonprimitive (user-defined) types. List, named tuples and graph nodes fall into the latter. Primitive data types are passed by value. List is passed by reference.

2.1 Primitive types

2.1.1 Number

The num data type in egrapher is a combination of traditional integer and float data type and it can contain either the floating point or not. By default, it is a base 10 digit that are 4 bytes in size and can represent any signed integer in the range [2147483647, +2147483647] when the decimal point is not contained. Literal declaration must be complete. If consists of a decimal point, the integer part must appear before the floating point, while the fraction part cannot be missing. Float type is 8 byte in size which is identical to double type in c++ language.

Standard declaration:
int **a = 10;**
float **b = 1.25;**

Fraction part cannot be omitted:
float **c = 5.;**
This line cannot pass parser.

2.1.2 String

String is a sequence of ASCII symbols in between double quotes, e.g. “hello world”. For variable instantiation, the *string* keyword is used to state a type. In order for double quote (“) or single quote (‘) symbols to be a part of the string, they must be preceded with a backslash (e.g. “he said \“Hi\” ”)

string **a = “Hello!”; (in global only allow string a;)**

2.1.4 Boolean

Boolean represents the most basic unit of logic. It takes either the value of *True* or *False*. Variables are typed using the keyword *bool*. Just use keywords true and false to represent logic concept true and false.

bool **a =** ***true;***

2.2 Lists

List types are kind of python-style. We take this design in order to emphasize that this type is as flexible as like python arrays. Manipulations like add, remove, get, set values are allowed. Scale of a list is automatically resized and we have integer list and float list are supported.

Initialize a list:
list int **a;**

Add integer element into int list
a.add(1);
a.add(2);

Access element at id :
a.get(0);
which returns 1.

Remove value at id :
a.remove(0);
If we call **a.get(0);** again, we will get value 1 returned.

Set value at id position:
a[0] = 3;
Here the access style is a little array-like because instead of list.set(value, id), this style is more simpler to understand.

Get length of a list:
a.length();
Return 1 in this case.

2.3 Comment

The characters /* introduce a comment, which terminates with the characters */.

3. Expressions

3.1 Primary expressions

3.1.1 **identifier**
An identifier is a sequence of letters and digits; the first character must be alphabetic. The underscore ‘‘_’’ counts as alphabetic. Letters are case sensitive.

3.1.2 **literals**

A literal refers to a data type literal. The integer, floating point, boolean, string literals have a range of possible values defined in the previous section.

3.1.3 grouping expressions: (*expression*)

The set of opening and closing parenthesis can be used to group an expression into a higher precedence. The type and value of this primary expression are that of the contained expression.

3.1.4 list index access: **identifier** [*expression*]

An identifier followed by expression surrounded by opening and closing brackets, denotes element index access. The identifier is a list, and the expression must be a nonnegative integer, and the type and value of the primary expression are that of the element at index.

3.2 Unary operators

The operators in this section have right-to-left associativity

3.2.1 - *expression*

The - operator denotes arithmetic negation for integer and floating point types.

3.2.2 !*expression*

The ! operator denotes boolean negation for boolean types.

3.3 Multiplicative Operators

The operators in this section have left-to-right associativity. Both expressions can be either integers or floating point numbers.

3.3.1 *expression* * *expression*

The * denotes multiplication of integers or floating point numbers.

3.3.2 *expression / expression*

The */* denotes division of integers or floating point numbers.

3.3.3 expression ^ expression

The ^ denotes power of integers or floating point numbers.

3.3.4 *expression* % *expression*

The % denotes module operator. The number produced by the evaluation of the operator will have the same sign as the dividend (left) operand.

3.4 Additive Operators

The operators in this section have left-to-right associativity.

3.4.1 *expression* + *expression*

The + operators denote integer and floating point addition.

3.4.2 *expression* - *expression*

The - operators denote integer and floating point subtraction.

3.5 Relational operators

The operators in this section have left-to-right associativity. The result of relational operator evaluation is a boolean type, the value of which corresponds to the truth value of the expression. Relational operators support integer or floating point operands, but not a combination of both.

3.5.1 *expression* < *expression*, *expression* > *expression*

The < and > operators denote the less than and greater than comparison.

3.5.2 *expression* <= *expression*, *expression* >= *expression*

The <= and >= operators denote the less than or equal to and greater than or equal to comparison.

3.6 Equality Operators

The operators in this section have left-to-right associativity. The result of equality operator evaluation is a boolean type, the value of which corresponds to the truth value of the expression.

3.6.1 expression == expression, expression != expression.

The == and != operators denote the equal to and not equal to comparison, which support integer and floating point operands. If an integer is compared to a floating point, the integer is automatically promoted to a floating point.

3.7 Boolean AND
3.7.1 expression && expression

The && operator denotes the boolean AND operation. It has lefttoright associativity and only supports boolean operators; type conversions or demotions are not supported.

3.8 Boolean OR
3.8.1 expression | | expression The | | operator denotes the boolean OR operation.

It has lefttoright associativity and only supports boolean operators; type conversions or demotions are not supported.

3.9 Assignment
3.9.1 **identifier** = *expression*

The = operator denotes assignment. It has right-to-left associativity. The identifier must have the same type as the expression; type conversions, promotions, or demotions, are not supported.

4. Statements

4.1 Conditional

Conditional statements include if and if/else statements and have the following form:

selection-statement:
    if (*expression*) **statement**
    if (expression) **statement** *else* **statement**

Selection statements choose one of a set of statements to execute, based on the evaluation of the expression. The expression is referred to as the controlling expression.

The controlling expression of an if statement must have scalar type. For both forms of the if statement, the first statement is executed if the controlling expression evaluates to nonzero. For the second form, the second statement is executed if the controlling expression evaluates to zero. An else clause that follows multiple sequential else-less if statements is associated with the most recent if statement in the same block (that is, not in an enclosed block).

Example:

```
    if (expression)

        statement
        if (expression) statement

        else statement

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

The first example indicates that later if statement is corresponding to the else statement.

The first example indicates that former if statement is corresponding to the else statement, because the later if statement is inside the scope of the first statement.

4.2 *while*

while statement execute the attached statement (called the body) repeatedly until the controlling expression evaluates to zero. In the for statement, the second expression is the controlling expression. The format is as follows:

iteration-statement:

    while (expression) {**statement**}

The controlling expression of a while statement is evaluated before each execution of the body.

4.3 *for* loop

The for statement has the following form:

    for (expression1; expression2; expression3) {**statement**}

The first expression specifies initialization for the loop. The second expression is the controlling expression, which is evaluated before each iteration. The third expression often specifies incrementation. It is evaluated after each iteration.

This statement is equivalent to the following:

*expression1; while(expression2) {***statement** *expression3;}*

One exception exists, however. If a continue statement is encountered, *expression3* of the for statement is executed prior to the next iteration.

Any or all of the expressions can be omitted. A missing *expression2* makes the implied while clause equivalent to while. Other missing expressions are simply dropped from the previous expansion.

4.6 *return*

A function returns to its caller by means of the return statement. The value of the expression is returned to the caller (after conversion to the declared type of the function), as the value of the function call expression. The return statement cannot have an expression if the type of the current function is void. If the end of a function is reached before the execution of an explicit return, an implicit return (with no expression) is executed. If the value of the function call expression is used when none is returned, the behavior is undefined.

5. Functions

Functions are defined with the *fun* keyword:

fun **Function-name** (*argument-type arg1, argument-type arg2,...*) { **statement** }

They should be defined before being used. Arguments are passed by reference, and are passed by value for other data types, and passing arguments requires the specification of their types. Types include integer, string, list and other build-in types. There can be multiple arguments given to a function. Each argument’s type should be defined prior to its name when defining a function. The type of the argument should not be specified when calling a function. The program checks the types of arguments as the function is being called. The return keyword returns value. The function will terminate when it sees the return keyword, and the program will return to where the function was called. Return type can be any types that are supported in eGrapher, but it needs to match the predefined type of the function. If the return value is type void, the function returns nothing.

6. Built-in Function

6.1 *print*(**identifier**) or *print*(**literals**)

Prints an integer, a floating point number or a string.

```
print(s);
```

6.2 Plotting

6.2.1 *triangle*(**coordinates**)

Show the triangle graph plotted by given 3 coordinates, 6 arguments needed.

Example:

```
triangle(0.5,1.0,-0.5,1.0,1.0,0.5);
```

7. Struct

Struct is a user-defined data type that combines multiple other data types into an aggregate with named fields. Then the fields can be easily accessed using the specified notation.

Declaration

```
struct  struct_name[
        type field_name1;
        type field_name2;
        ...
]
```

Usage:

```
struct struct_name variable_name;
variable_name.field_name;
```

Example:

```
struct person[
int age;
string name;
]
int main(){
        struct person p;
        p.age = 1;
        print(p.age);
}
```

7. Program Structure

Every statement in eGrapher must be defined inside the main function or in external functions outside of main. The external functions are used to write simple programs that can be executed in the main function. As a result, only functions, variables, and objects called in the main function will be executed and stored. In addition, the name main in the global name space is reserved and does not need a return statement.

Any program will consist of a sequence of external definitions. This is designed to give the user more flexibility. External definitions may be given for function, for simple variables, and for lists. They are used to help declare and allocate storage for objects when called upon.

8. Sample Program

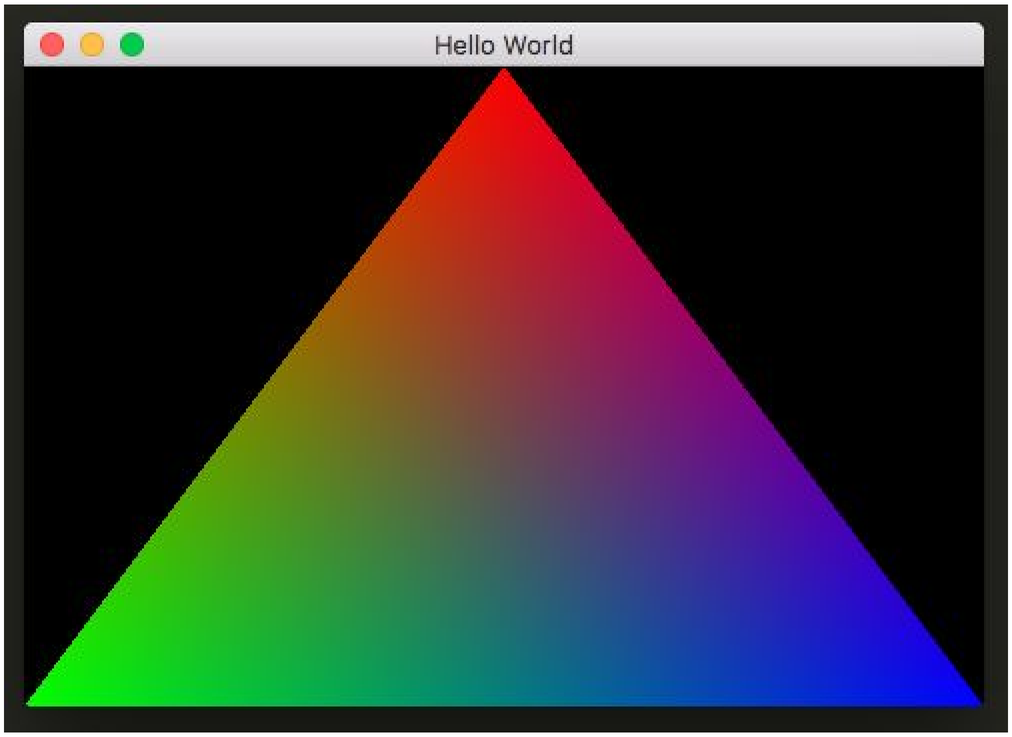
8.1 Bubble sort

```
/* Bubble sort code */
int main(){
    list int l;
    l.add(3);
    l.add(5);
    l.add(2);
    l.add(1);
    l.add(4);
    int c, d, swap;
    int n = l.length();
    for (c = 0 ; c < ( n - 1 ); c=c+1){
        for (d = 0 ; d < n - c - 1; d=d+1){
```

```
                    if (l.get(d) > l.get(d+1)){
                            swap = l.get(d);
                            l[d] = l.get(d+1);
                            l[d+1] = swap;
                    }
                }
            }
        print("Sorted list in ascending order:\n");
        for ( c = 0 ; c < n ; c=c+1 ){
                print(l.get(c));
                print(" ");
        }
    }
```

8.2 Draw a triangle

triangle(0.0,0.5,-1.0,-1.0,1.0,-1.0);

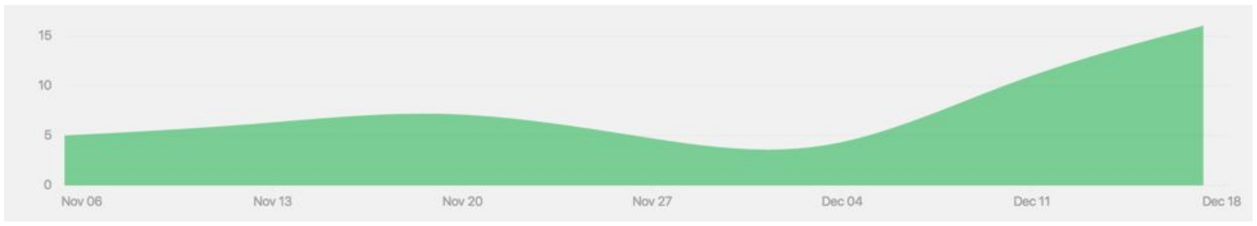


9. Complete Table of Keywords

Keywords	**Description**
int	signed integer value
string	list of characters
float	floating point number
bool	boolean value
list	Sequence of numeric, Boolean, or other types.
fun fun_name (*type* **arg1***, type* **arg2***, …)*	defines an external function
main()	the main script for execution
l*.add(x)*	adds an element to the end of list
l*.remove(x)*	deletes the element in the list
l.length	Return length of a list
*triangle (***coordinates***)*	Plot a triangle graph
return val	returns value
for(type name; conditional; postloop) {}	c-style for loop
while(conditional) {}	while loop
if(condition) {} *else* {}	conditional statement
true	Boolean value is true
false	Boolean value is false

4. Project Plan

We aimed to employ agile development throughout the project. This meant that tasks were equally split among team members and collectively we worked together one step at a time: parser, scanner, analyzer, codegen, and the final report. We took turns pair programming as well as worked in groups to ensure that everyone was on track due to the scale of the project. In addition, there was one master branch where every team member could see the latest additions as well as publish their code on GitHub. Finally, we worked hard with our TA Rachel Gordon who kept us on track throughout the process as well as attended lectures to further understand fundamental concept with Professor Edwards at Columbia University.



Progress on GitHub

Programming Style Guide

Our group aimed to follow some general guidelines regarding programming style. We aimed to comment as much of code as possible throughout the development process. In Ocaml this can be tricky sometimes, we tried to write comments wherever possible in line, and subsequently write some quick points for blocks of entire code. The second main guideline was using spaces instead of tabs, so that there would be no issues across editors.

Project Timeline

Date	Project Milestone
Nov 11	Scanner and Parser
Nov 13	Semantic Checking
Nov 16	Test Suite
Nov 21	Hello World
Dec 4	Struct
Dec 17	List

Dec 19	Draw Triangle

Identify roles and responsibilities of each team member

Our roles were not restricted by our titles and the entire project was a collaborative effort. We made most of our progress during our meetings and sometimes we divided problems into parts so each member had chance to work on different aspects of the project at different times.

Darren Chen	Project planning, Test case creation, Compiler Front End, Documentation
Xinli Jia	Compiler front end, Code generation, Test case creation, Testing automation
Long Long	Semantics, Test case creation, Linking C Libraries, Documentation
Linnan Wang	Compiler front end, Semantics, Code generation, Linking C Libraries
Jiefu Ying	List implementation, Linking C Libraries, Code generation

Software Development Environment

- Operating System: We mainly used our Mac OS systems for the development of our project.
- Unix: All members had a Mac so most of the code development was done in the UNIX environment machines.
- Languages: We used Ocaml extensively to build our compiler while we also include C as part of our bindings. We used shell for our automated test suite.
- Slack: We used slack for communication and planning for our meetings and discussions as well as sharing resources.
- Github: We used Github for version control throughout our project. Since we mostly developed our project during our meetings, we maintained one master branch to publish the latest updates and keep track of the process.
- Sublime: We used Sublime as our text editor for OCaml even though Sublime did not pick up the OCaml syntax very well sometimes.

Project Log

```
commit fe2fbaa75a3159cbee35a85157c8bf1a705920a6
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Tue Dec 20 22:36:33 2016 -0500

    update readme and makefile

commit 9e8e93cf936ced2e92a7fa6747d9815ff3af8082
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Tue Dec 20 22:18:16 2016 -0500

    run .sh without enter

commit ab2b9c5f51a3177f3167f92074085b102f89097e
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Tue Dec 20 22:11:44 2016 -0500

    add list test

commit cf3ba21c0758043566d59425c667c46efcbaeed6
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Tue Dec 20 22:11:27 2016 -0500

    add list test

commit dda613e26fa5e276f191708db996c86a9ab876e2
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Tue Dec 20 20:58:52 2016 -0500

    nothgin

commit 2d4947bc9ab367576410ec657f6a36a2cde87d7e
Merge: d0ac5b6 ba1ffe6
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Tue Dec 20 20:56:59 2016 -0500

    Merge branch 'master' of https://github.com/wanglnxp/PLT-project
    "sdfs"

commit d0ac5b604bcc8bf1fc8cf05e5474ca4f71025079
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Tue Dec 20 20:56:57 2016 -0500

    merger

commit ba1ffe62492ab4d70d2886fade3270f21f635334
Author: Long Long <long.long@columbia.edu>
Date:   Tue Dec 20 19:27:21 2016 -0500

    Update fail-dead1.eg

commit bb19e8de568383848beee469c5157eaf03fe0992
Merge: 0f126c1 d0da73f
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Tue Dec 20 16:59:37 2016 -0500

    merger

commit d0da73f52676d22c38f8cca7868ed5e25d97f40e
Author: longlongCU <long.long@columbia.edu>
Date:   Tue Dec 20 15:08:15 2016 -0500

    fixed quicksort grammer

commit 84be88efb1cb6cc3874026affcf2c2e4fda04f90
Author: longlongCU <long.long@columbia.edu>
Date:   Tue Dec 20 12:40:48 2016 -0500

    added recursion test

commit 0f126c1bb9d5629c410a44e5c908e07bc2c2d683
Merge: 63bbe4f ba1aa67
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Mon Dec 19 23:10:19 2016 -0500
```

```
    delet

commit 63bbe4f057702ecac05ef577f8e50e14ed16fc14
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Mon Dec 19 23:09:43 2016 -0500

    asd

commit ba1aa67b5d1bcf8330ade6f5a31a4959f0fb96e1
Author: longlongCU <long.long@columbia.edu>
Date:   Mon Dec 19 23:06:01 2016 -0500

    added triangle

commit a840930a9a7cf38bed32f36583a31fc72972bbad
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Mon Dec 19 18:57:41 2016 -0500

    fix list bug

commit de02c9631c21dfcb2f2dbc045f42abd1cbb707d4
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Mon Dec 19 14:49:13 2016 -0500

    unfinished list

commit 9ca33ebd33e82fc39e221e48e7ae57ee50ac30df
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Mon Dec 19 11:39:53 2016 -0500

    finish basic 4 for list

commit 4764845e82b4f09af96c6409272ca349f9300d41
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Mon Dec 19 06:40:31 2016 -0500

    try to fix list'

commit 90c74867a8ab9ede08716feb36290a0d318e9636
```

```
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Mon Dec 19 05:44:57 2016 -0500

    semant func formal checking

commit de06641b592e0657bbd31c29942993a4dd82c15a
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Sun Dec 18 22:56:35 2016 -0500

    write auto test for unit test

commit 7263fff11a9e34b54514f0c77389794849627a55
Author: wanglnxp <wanglnxp@gmail.com>
Date:   Sun Dec 18 22:37:09 2016 -0500

    Delete test.ll

commit 2cd17418472d73d9e432db8d22cc30b81a2b9139
Author: wanglnxp <wanglnxp@gmail.com>
Date:   Sun Dec 18 22:36:48 2016 -0500

    Delete test.c

commit f45f9879fbc66a583a1ecdd2581836fa24e326a9
Author: xinlijia <jiaxinli93@gmail.com>
Date:   Sun Dec 18 22:36:46 2016 -0500

    microc->egrapher & testcases updates

    testall.sh and some new testcases

commit 307bbc8bcd02ddb272a5ec90b17cdbf5200d5325
Author: wanglnxp <wanglnxp@gmail.com>
Date:   Sun Dec 18 22:36:37 2016 -0500

    Delete error.txt

commit ea166c8efb6fefecd00f3e4e6a38485ac5ef4ba9
Author: wanglnxp <wanglnxp@gmail.com>
```

```
Date:   Sun Dec 18 22:36:21 2016 -0500

    Delete list.ll

commit e81f1d2fcd309fbdedb6bd8d4fa414068bf68064
Author: wanglnxp <wanglnxp@gmail.com>
Date:   Sun Dec 18 22:36:11 2016 -0500

    Delete list.c

commit 60b76bb9c447fc0b95bf4e62f1a306544d23b560
Author: wanglnxp <wanglnxp@gmail.com>
Date:   Sun Dec 18 22:36:00 2016 -0500

    Delete demo.ll

commit 0fd52d55535b6df663f57d8bb42f8254319b4a7f
Author: wanglnxp <wanglnxp@gmail.com>
Date:   Sun Dec 18 22:35:40 2016 -0500

    Delete a.out

commit 9fc0f84d180e7f7c2124a8969be4fdce7ea057ee
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Sun Dec 18 22:13:12 2016 -0500

    finish print bool in true and false

commit a28b33354b37fec50160b57f9e41a4b73b49dbda
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Sun Dec 18 21:48:53 2016 -0500

    update print

commit d926bc241e0f1cbc1d88081f70eecbad67579608
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Sun Dec 18 05:45:57 2016 -0500

    list partially work
```

```
commit ef571f863bac37bab3ff22db6dc28324587d9350
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Sat Dec 17 23:33:55 2016 -0500

    xxx

commit b69890d46b3b692df06120d15ec397ccea5274f5
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Sat Dec 17 23:33:19 2016 -0500

    finish semant

commit 0e52f51f5077fc77b05a485ef68d21add691cdda
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Sat Dec 17 22:58:05 2016 -0500

    semant check objcall

commit 3694ef9d1b7dddf4c41fb4690741827d7f487a0f
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Sat Dec 17 05:32:14 2016 -0500

    try add list type

commit 33bdffb6b414779d406a94a3bc98626ef09a966d
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Sat Dec 17 02:40:22 2016 -0500

    finish int and float mix computation

commit 402ac0ba778a3c0cf75612cd1c9da7e71e9a4cce
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Fri Dec 16 23:07:27 2016 -0500

    fix binop in codegen

commit 9a55502ccacc8508e1744ebe97c7e0a83a2c32f1
Merge: c7d496d 3e650df
```

```
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Fri Dec 16 04:58:12 2016 -0500

    for merge

commit c7d496dd2abde54769e0a18118764c6e476c2478
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Fri Dec 16 00:00:17 2016 -0500

    ok

commit 3e650df0bb26639e30d48ba073e1e859989e857a
Author: xinlijia <jiaxinli93@gmail.com>
Date:   Thu Dec 15 23:58:22 2016 -0500

    struct works in function

commit 19bd45a1cddb35c808edc9333329063397b42df0
Merge: d5ba855 d9831fc
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Thu Dec 15 20:28:36 2016 -0500

    Merge branch 'master' of https://github.com/wanglnxp/PLT-project

    not sure what i did

commit d5ba855aeab15f6614e3da5064a49ab2d4a89bbf
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Thu Dec 15 20:28:28 2016 -0500

    clean up

commit d9831fc51c9360aa8c1cc91bc69ac62757a7618f
Author: longlongCU <long.long@columbia.edu>
Date:   Thu Dec 15 18:45:06 2016 -0500

    test case created

commit afef5acdfb44e2e013ee8e4156e2d9c15ee7da22
```

```
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Thu Dec 15 18:38:46 2016 -0500

    improve semant

commit 31b7ccbbf1e2b66e1ab1e32bf81e20e4ff65ff2c
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Sun Dec 4 22:27:08 2016 -0500

    struct prototype can use

commit d77565d36069620aad487cfaf0f8672ed3ba189b
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Sun Dec 4 04:12:30 2016 -0500

    All library function can call recursively, print now can print bool in string

commit 39084da2e4dbaad50b5b0cdf7b1903bab2676c22
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Sat Dec 3 20:02:28 2016 -0500

    successfully link self define c lib

commit 80f0b87db4e8c17ad1232bdd4e742a94b8cd484d
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Thu Dec 1 03:12:21 2016 -0500

    variable can be declared at global. string must be null

commit 2634a28856d601bcb64f7297c4a29217bd38208d
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Tue Nov 29 18:42:28 2016 -0500

    delete -= +=

commit ea83e8c6ea6bb4241a4ebb88a31252daa1aca16b
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Tue Nov 29 18:33:41 2016 -0500
```

```
    revert

commit d534c0f52efa4f8a9dbc09f895ee6de1cd3fe7ac
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Mon Nov 28 20:15:30 2016 -0500

    finish test case

commit 6f494a6f92c6994312232b37ddf6b97e8697d038
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Sat Nov 26 17:13:59 2016 -0500

    finish print all type and match operation type

commit 0479aa1f8e14cfea90d31a49e2602d575a7aaea6
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Wed Nov 23 22:13:59 2016 -0500

    check gloabl valid

commit 048e7ecbe4da00204dd33f1a02b4fb365e371bed
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Wed Nov 23 20:28:40 2016 -0500

    hello world done

commit 77adc3758f62d9a29371bbf0a25e781aa1fa3272
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Wed Nov 23 18:58:02 2016 -0500

    group update

commit 684d4d86c71bc9a98379a5219b0961936ef800f6
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Wed Nov 23 00:38:10 2016 -0500

    find bug in codegen but not fix

commit ce97a5df64134cf6e98d1d746b8c330d595cc9a3
```

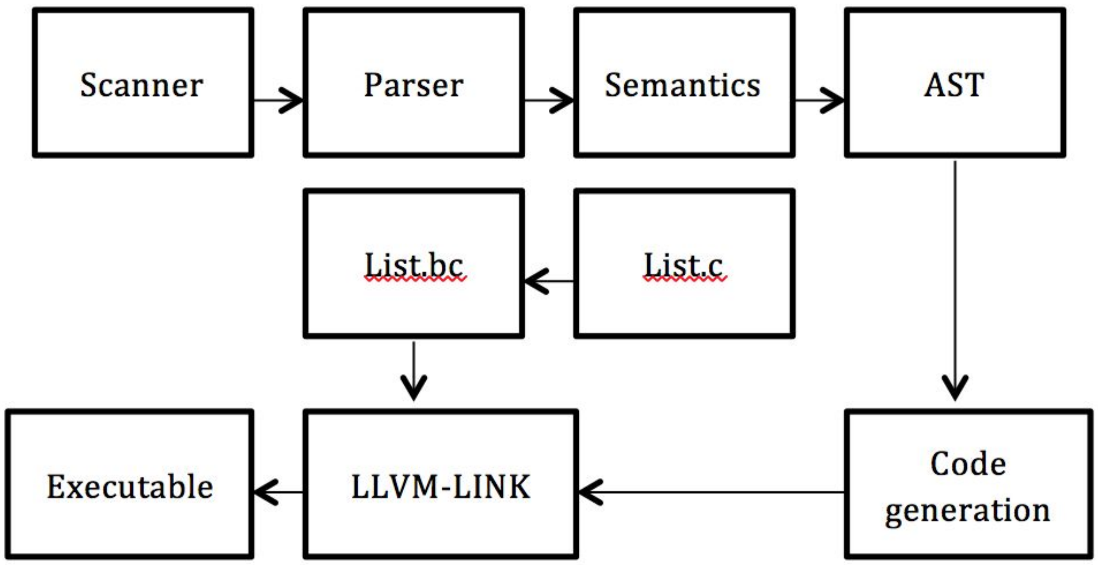
```
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Sun Nov 20 19:34:21 2016 -0500

    add make file

commit 8c3e8a7934f462d9fc0cd2e4959617ba36b2d0ca
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Sun Nov 20 19:33:55 2016 -0500

    remove unwanted

commit b2f6e34cc21abca9e9c14c6d442458b867ab7845
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Sun Nov 20 19:30:19 2016 -0500

    start codegen

commit ca8cf8ac9196d36d468383bafa448281c57bfada
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Fri Nov 18 18:25:11 2016 -0500

    find semant checklist

commit 222b2f2d242869de5d964b5bfa4e9152f537ebee
Merge: 9116206 2db08b5
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Mon Nov 14 00:12:48 2016 -0500

    Merge branch 'master' of https://github.com/wanglnxp/PLT-project

commit 9116206009a1d769e608c29f5c2a6abda3fb4e7d
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Mon Nov 14 00:12:13 2016 -0500

    add test file

commit 2db08b50c3911b6e85f0ada2104f01f937a7b0ce
Author: longlongCU <long.long@columbia.edu>
Date:   Sun Nov 13 17:06:22 2016 -0500
```

```
    changed

commit 8bee89fc01ba472010a1ef1b479648c727dcdf3e
Author: longlongCU <long.long@columbia.edu>
Date:   Sun Nov 13 16:57:48 2016 -0500

    test

commit 99525eeb792e0acdce134050a9b3f4a2d372b1a5
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Sun Nov 13 16:28:47 2016 -0500

    doing semant

commit 499827f1e9cf34965753af3cfcbb531df25133b8
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Sun Nov 13 16:26:14 2016 -0500

    doing semant

commit 1a581010354b00dcae67fd0c6bfa09690ac13196
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Fri Nov 11 22:12:33 2016 -0500

    change program structure and solve shift reduce conflict in else if

commit afea12d2292f358b9ab5ac656f77dd7865b2b532
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Mon Nov 7 20:40:22 2016 -0500

    start write semant

commit 4bda01c86c870a7619641acb35a3893e676e14d1
Author: Linnan Wang <wanglnxp@gmail.com>
Date:   Sun Nov 6 20:31:17 2016 -0500

    temp finish parser
```

```
commit 6e27f65363a06b453b0ea94d0f2427bbf7f70f0b
Author: jiaxinli <xinli.jia@sjtu.edu.cn>
Date:   Sun Nov 6 16:42:11 2016 -0500

    xinli add to repo

commit c134da3ad2f18e43aa8471c3ca6043e93a086183
Author: Linnan Wang <lw2645@columbia.edu>
Date:   Sun Nov 6 16:21:27 2016 -0500

    first few code of project

commit 752fa79c174800808b449e3bbc650eaa6d099403
Author: Linnan Wang <lw2645@columbia.edu>
Date:   Sun Nov 6 16:18:07 2016 -0500

    first commit
```

5. Architectural Design

eGrapher has a standard architecture similar to MicroC. It contains scanner, parser, semantic checker, ast, codegen as the basic components. In addition to the basic parts, eGrapher also utilizes LLVM bitcode compiled from C as external links for list data structure. And llvm-link outputs the LLVM IR needed. Refer to the block diagram below for the basic architectural design.



5.1 Scanner (Darren, Linnan, Xinli)

The Scanner takes the source code as the input and separate it into specific tokens. The tokens includes comments, keywords, identifiers, operators, statements and functions. The detailed rules refer to the reference manual. The tokens are passed to parser to parse further.

5.2 Parser (Linnan, Xinli)

The Parser component takes the token generated by Scanner and construct the AST based on the eGrapher syntax we designed. It will pass the AST to semantic checker.

5.3 Semantic Checker (Darren, Linnan, Long)

The Semantic Checker takes the AST generated by Parser and check the semantic errors in the source codes. The checking items include variable types, assignment types, valid function call and statements, struct check, and scope of variable and keywords.

The Semantic Checker ensures the valid semantic input source code. It will raise the corresponding error when the semantic error is detected.

5.4 Code Generation (Xinli, Linnan, Jiefu)

Codegen takes the checked AST and produced necessary LLVM IR. In the Codegen it uses ocaml-LLVM bindings and builds basic blocks. The ocaml-LLVM bindings translates the AST line by line into LLVM.

5.5 External Module Links (Long, Jiefu)

An external module link is used for “list” data structure. It is implemented in C and compiled into LLVM bitcode (.bc) using “clang”. Then the bitcode file is linked with former output files by “llvm-link” and finally produces the LLVM IR.

6. Test Plan

6.1 Automation and Testing Suite

For our test, we write a shell script to run all the test cases of either detecting failure or function tests. All the tests are placed in a folder. And the script will automatically compile and execute all the tests and compare the results with correct ones. The automation script is written by Xinli.

6.2 Tests Types

We create several test cases to cover nearly all the features our language can do. Mainly the test cases are unit tests because we intend to test only one aspect of the language in one time in order to identify the error easily and correct it. In addition, we have several overall test cases as sample codes to realize certain algorithms to show that our language is suitable for some practical works.

Out test cases mainly contain two types, failure tests and success tests. Failure tests test the cases when there’s an error occurring in the source code. Our failure tests will examine the failure raised by the compiler to see whether the problem is detected. It includes the semantic error. And for success tests, we test functions that our languages can finish. All the codes generated properly and there should be no errors. These two kinds of tests can ensure our language can either deal with errors in codes or finish the given test well when there’s no error.

6.3 Test cases list

Here is a test case list to show what each test does and who is the one create it. The codes of all the test cases will be in appendix.

Test name	Function	Creator
fail-assign1	different type assign	Xinli
fail-assign2	different type assign	Xinli
fail-assign3	void assign	Xinli
fail-dead1	code after return	Xinli
fail-expr1	bool add int	Xinli
fail-func-arg-num	arg num not match	Xinli
fail-func-duplicate -formal-local	formal and local variable duplicate	Xinli
fail-func-duplicate-formal	formal variable duplicate	Xinli
fail-formal-struct	wrong struct type as formal	Xinli
fail-void-local	local variable void	Xinli
fail-global-duplicate	global variable duplicate	Xinli
fail-struct-access	access not exist member	Xinli
fail-struct-return	return wrong type in struct	Xinli
test-comparison	comparison operators	Xinli
test-fac	recursive factorial	Xinli
test-flt-binops	float bin operations	Xinli
test-for1	for style1	Xinli
test-for2	for style 2	Xinli

test-for3	for style 3	Xinli
test-func-loop	function in function	Xinli
test-func-recur	recursive function	Xinli
test-gcd	gcd	Xinli
test-hello	hello world	Xinli
test-if-dagling	if dagling test	Xinli
test-if-emptyblock1	If emptyblock test1	Xinli
test-if-emptyblock2	If emptyblock test2	Xinli
test-if1	if normal test	Xinli
test-if2	if in if	Xinli
test-if3	if if else	Xinli
test-if4	if take boolean	Xinli
test-if5	If mix test	Xinli
test-int-binops	int bin operations	Xinli
test-int-flt-binops	mix bin operations	Xinli
test-intMax	Check max int	Xinli
test-list-obj-call	call list member	Xinli
test-local1	dealing local variable	Xinli
test-local2	dealing local variable	Xinli
test-ne	negation test	Xinli
test-ops1	int operations	Xinli
test-ops2	boolean operations	Xinli
test-print	print mix test	Xinli

test-struct-access	struct access	Xinli
test-while1	while style 1	Xinli
test-while2	while style 2	Xinli

7. Lessons Learned

Darren Chen:

I greatly underestimated the sophistication of OCaml and the time it took to learn the language. However, this was well worth it given the surprising amount of simplicity it can provide for building languages. My greatest lesson is for those with less programming experience, or not that strong, to quickly go online at the very beginning of the semester and learn as much as you can. They have a great set of practice problems you can work through to better understand how functional programming works. In addition, this project will require you to work collectively together as a group. You simply cannot work alone, or even in pairs, the sophistication and overall structure will require ample input and thoughts from together as a team. Also, testing is so critical on this project - make sure to thoroughly test each step and to get bare essentials working (the debugger and error response is not very useful in Ocaml).

Xinli Jia:

I think everyone in the group worked really hard, but sitting together more often would have been beneficial for the overall project. I found out that this project really required people to sit together and talk about their thoughts out loud: it is definitely not one of those projects where people are able to work individually and push on to GitHub. Moreover, OCaml is a tough language and working collectively with team members will surely go along a long way.

Long Long:

The greatest thing I learned was that it is important for every team member to work cohesively as a group. We started the project out working solo thinking as long as we committed and regularly checked in, the project would go smoothly. However, this was a terrible assumption and it cost us a lot of time. If anything, due to the tricky nature of learning OCaml in the very beginning, I strongly encourage people to meet together as much as possible in the beginning to hash out at a high level the code and overall structure. Working together saved us a lot more time and would have made the end of the finals semester much easier. Be warned that this is not an easy project, and take that advice very much from the beginning on.

Linnan Wang:

Detail. Detail. Detail. This project will punish you for it. Be careful of missing characters and comment throughly throughout your code. This will save you ample amounts of time. OCaml is a tough language, but works beautifully if you can get it right. To paraphrase something someone else said in the class: “never i have worked so hard on such little code that does so much.”

In addition, do not underestimate any parts of the project. They are all difficult and it is better to tackle them as early as possible. For example, the semantic check, and even getting hello world to work, turned out much more difficult to get expected. Work from the very beginning and try to fully understand the difficult of concepts ahead of writing the code.

One final comment, I strongly emphasize to think about the overall program you are building. The point of the project is to build a language with simple tools that can produce some pretty sophisticated solutions - do not get burdened down by features and keep thinking about how all the individual parts fit together.

Jiefu Ying:

When working with a teammate on a feature where you will have to make decisions about design that affect the compiler, do not make any high-level assumptions. Take the time and walk through each step thoroughly while also regularly updating each other during the implementation. Communication is critical and you need to make sure both of you have an exact detail of understanding on the design - OCaml will punish you for this if not. In addition, hash out some of the bare minimum code to make sure that the overall code works. This will save you ample time as all the planning in the world is futile unless the program runs.

8. Appendix

egrapher.ml

```
(* Top-level of the compiler: scan & parse the input,
check the resulting AST, generate LLVM IR, and dump the module *)

type action = Ast | LLVM_IR | Compile

let _ =
  let action = if Array.length Sys.argv > 1 then
```

```
        List.assoc Sys.argv.(1) [ ("-a", Ast);  (* Print the AST only *)
                 ("-l", LLVM_IR);  (* Generate LLVM, don't check *)
                 ("-c", Compile) ] (* Generate, check LLVM IR *)
  else Compile in
  let lexbuf = Lexing.from_channel stdin in
  let ast = Parser.program Scanner.token lexbuf in
  Semant.check ast;
  match action with
        (*Ast -> print_string (Ast.string_of_program ast)*)
  | LLVM_IR -> print_string (Llvm.string_of_llmodule (Codegen.translate ast))
  | Compile -> let m = Codegen.translate ast in
        Llvm_analysis.assert_valid_module m;
      print_string (Llvm.string_of_llmodule m)
```

scanner.mll

```
 (* Ocamllex scanner for eGrahper *)

{ open Parser }

let digit = ['0'-'9']
let float = '-'?(digit+) ['.'] digit+
let bool = "true" | "false"

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"    { comment lexbuf }            (* Comments *)
| '('     { LPAREN }
| ')'     { RPAREN }
| '['     { LBRACKET }
| ']'     { RBRACKET }
| '{'     { LBRACE }
| '}'     { RBRACE }
| ';'     { SEMI }
| ','     { COMMA }
| '+'     { PLUS }
| '-'     { MINUS }
| '*'     { TIMES }
| '/'     { DIVIDE }
| '%'    { MOD }
| '='    { ASSIGN }
| '.'    { DOT }
| "=="  { EQ }
| "!="   { NEQ }
| '<'    { LT }
| "<="  { LEQ }
| '>'    { GT }
| ">="  { GEQ }
| "&&" { AND }
| "||"    { OR }
| '!'     { NOT }
| "if"    { IF }
| "else"   { ELSE }
| "elif"   { ELSEIF }
| "for"  { FOR }
| "in"    { IN }
| "while"  { WHILE }
| "return" { RETURN }
| "break"  { BREAK }
| "continue" { CONTINUE }
(*| "endelif"{ ENDELIF }*)

| "int"   { INT }
| "float"  { FLOAT }
| "bool"   { BOOL }
| "string" { STR }
| "void"   { VOID }
| "list"   { LIST }
| "point"  { POINT }
| "line"   { LINE }
| "struct" { STRUCT }
(*| "class"  { CLASS }*)

| "NULL"   { NULL }

| bool as lxm { BOOLEAN_LITERAL (bool_of_string lxm) }
| ['0'-'9']+ as lxm { LITERAL(int_of_string lxm) }
| float as lxm { FLOAT_LITERAL(float_of_string lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
| '"' {let buffer = Buffer.create 1 in STRING_LITERAL (stringl buffer lexbuf) }

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

and stringl buffer = parse
          | '"' { Buffer.contents buffer }
          | "\\t" { Buffer.add_char buffer '\t'; stringl buffer lexbuf }
          | "\\n" { Buffer.add_char buffer '\n'; stringl buffer lexbuf }
          | "\\\"" { Buffer.add_char buffer '"'; stringl buffer lexbuf }
          (*| "\\\\" { Buffer.add_char buffer '\\'; stringl buffer lexbuf }*)
   | _ as char { Buffer.add_char buffer char; stringl buffer lexbuf }

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

parser.mly

```
%{
open Ast
%}

%token SEMI LPAREN RPAREN LBRACKET RBRACKET LBRACE RBRACE COMMA
DOT
%token PLUS MINUS TIMES DIVIDE MOD ASSIGN NOT
%token EQ NEQ LT LEQ GT GEQ TRUE FALSE AND OR
%token RETURN IF ELSE ELSEIF FOR IN WHILE BREAK CONTINUE
%token INT FLOAT STR BOOL VOID POINT LINE
%token LIST NULL STRUCT /*CLASS*/
%token <int> LITERAL
%token <float> FLOAT_LITERAL
%token <bool> BOOLEAN_LITERAL
%token <string> ID
%token <string> STRING_LITERAL

%token EOF

%nonassoc NOELSE /*how to use*/
%nonassoc ELSE
%right ASSIGN
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE
%left MOD
%right NOT NEG /*should we use nonassoc?*/

%start program
%type <Ast.program> program

%%

program:
  decls EOF { let (a, b, c) = $1 in  (a, List.rev b, c) }

/*How to make sure stmt always before fedcl*/
decls:
          /* nothing */   { [], [], []}
  | decls stmt { let (a, b, c) = $1 in ($2 :: a), b,c }
  | decls fdecl { let (a, b, c) = $1 in a, ($2 :: b),c }
  | decls sdecl { let (a, b, c) = $1 in a,b, ($2 :: c) }

vdecl_list:
          /* nothing */   { [] }
  | vdecl_list typ ID SEMI { Vdecl($2, $3) :: $1 }

vdecl:
   typ ID SEMI          { Vdecl($1, $2) }
  |typ ID ASSIGN expr SEMI { Block([Vdecl($1, $2);Expr(Assign($2,$4))]) }

/*sdecl_list:
          { [] }
  | sdecl_list sdecl { $2 :: $1 }*/

sdecl:
  STRUCT ID LBRACKET vdecl_list RBRACKET
          { {
          sname = $2;
          s_stmt_list = List.rev $4;
          } }



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



elseif_list:
  | elseif_list elseif  { $2 :: $1 }

elseif:
  ELSEIF LPAREN expr RPAREN stmt { Elseif($3, $5) }
```

```
stmt:
        expr SEMI                       { Expr $1 }
  | vdecl                               { $1 }
  | LBRACE stmt_list RBRACE             { Block(List.rev $2) }
  | RETURN expr SEMI      { Return($2) }

  | IF LPAREN expr RPAREN stmt %prec NOELSE  { If($3, $5, Block([]), Block([])) }
  | IF LPAREN expr RPAREN stmt ELSE stmt  { If($3, $5, Block([]), $7) }
  | IF LPAREN expr RPAREN stmt elseif_list ELSE stmt  { If($3, $5,  Block($6), $8) }

  | FOR LPAREN expr_opt SEMI expr SEMI expr_opt RPAREN stmt { For($3, $5, $7,
$9) }
  | FOR LPAREN expr IN expr RPAREN stmt { Foreach($3, $5, $7)}
  | WHILE LPAREN expr RPAREN stmt  { While($3, $5) }
  | BREAK SEMI  { Break }
  | CONTINUE SEMI  { Continue }


fdecl:
   typ ID LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE
        { { typ = $1;
        fname = $2;
        formals = $4;
        body = List.rev $7 } }

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

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

typ:
        INT { Int }
  | FLOAT { Float }
  | BOOL { Bool }
  | STR  { Str }
  | VOID { Void }
  | LIST typ { ListTyp($2) }
  | POINT{ Pot }
  | LINE { Lin }
  | STRUCT ID   { Objecttype($2) }

/*point:
        LPAREN LITERAL COMMA LITERAL COMMA STRING COMMA STRING
RPAREN  { {x_ax=$2; y_ax=$4; form=$6 color=$8 } }*/
        /*LPAREN expr COMMA expr COMMA expr COMMA expr RPAREN  {
($2,$4,$6,$8) }*/


/*line:
        LPAREN LPAREN expr COMMA expr RPAREN COMMA LPAREN expr
COMMA expr RPAREN COMMA expr COMMA expr RPAREN  { {star_p=($3,$5);
end_p=($9,$11); form=$14 color=$16 } }*/
        /*LPAREN LPAREN expr COMMA expr RPAREN COMMA LPAREN expr
COMMA expr RPAREN COMMA expr COMMA expr RPAREN  {
($3,$5,$9,$11,$14,$16) }*/

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

expr:
        LITERAL      { Literal($1) }
  | FLOAT_LITERAL{ FloatLit($1) }
  | STRING_LITERAL   { StringLit($1) }
  | BOOLEAN_LITERAL  { BoolLit($1) }
  | ID            { Id($1) }
  /*| point                 { $1 }*/
  /*| line         { Line($1) }*/
  /*| ID DOT ID ASSIGN expr { Dotassign($1, $3, $5) }
  | ID DOT ID ASSIGN LPAREN expr COMMA expr RPAREN { Lineassign($1, $3, $6,
$8) }*/

  | NULL                 { Noexpr }
  /*| LPAREN expr RPAREN { $2 } */
  | expr PLUS   expr { Binop($1, Add,   $3) }
  | expr MINUS  expr { Binop($1, Sub,   $3) }
  | expr TIMES  expr { Binop($1, Mult,  $3) }
  | expr DIVIDE expr { Binop($1, Div,   $3) }
  | expr MOD  expr { Binop($1, Mod,   $3) }
  | expr EQ     expr { Binop($1, Equal, $3) }
  | expr NEQ    expr { Binop($1, Neq,   $3) }
  | expr LT     expr { Binop($1, Less,  $3) }
  | expr LEQ    expr { Binop($1, Leq,   $3) }
  | expr GT     expr { Binop($1, Greater, $3) }
  | expr GEQ    expr { Binop($1, Geq,   $3) }
  | expr AND    expr { Binop($1, And,   $3) }
  | expr OR     expr { Binop($1, Or, $3) }
  | MINUS expr %prec NEG { Unop(Neg, $2) }
  | NOT expr          { Unop(Not, $2) }

  | ID ASSIGN expr   { Assign($1, $3) }

  | LBRACKET list_opt RBRACKET { List($2) }
  | ID DOT ID ASSIGN expr {StructAssign($1, $3, $5)}
  | ID DOT ID           { StructAccess($1, $3) }
  | ID LBRACKET expr RBRACKET ASSIGN expr       { ListAssign($1, $3, $6) }
  | ID LBRACKET expr RBRACKET { Mem($1, $3) }
  | ID DOT ID LPAREN list_opt RPAREN { Objcall($1, $3, $5) }


  | ID LPAREN list_opt RPAREN { Call($1, $3) }
  | LPAREN expr RPAREN { $2 }



list_opt:
        /*nothing*/  { [] }
  |list           { List.rev $1 }

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

ast.ml

```
 (* Abstract Syntax Tree and functions for printing it *)

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

type uop = Neg | Not

type typ = Int | Float | Bool | Str | Void | ListTyp of typ | Pot | Lin | Objecttype of string

type pot = {
  x_ax: float;
  y_ax: float;
  form: string;
  color: string;
}

type line = {
  star_p: float * float;
  end_p: float * float;
  form: string;
  color: string;
}

type expr =
           Literal of int
  | FloatLit of float
  | StringLit of string
  | BoolLit of bool
  | Id of string
  | Point of pot
  | Line of line
  | Binop of expr * op * expr
  | Unop of uop * expr
  | Assign of string * expr
  | Call of string * expr list (*function call*)
  | Objcall of string * string * expr list
  (*| Dotassign of string * string * expr
  | Lineassign of string * string * expr * expr *)
  | List of expr list
  | Mem of string * expr
  | ListAssign of string * expr * expr
  | StructAssign of string * string * expr
  | StructAccess of string * string
  | Noexpr


type stmt =
        Vdecl of typ * string
  | Block of stmt list
  | Expr of expr
  | Return of expr
  | If of expr * stmt * stmt * stmt
  | Elseif of expr * stmt
  | For of expr * expr * expr * stmt
  | Foreach of expr * expr * stmt
  | While of expr * stmt
  | Break
  | Continue

type formal = typ * string

type func_decl = {
        typ : typ;
        fname : string; (* Name of the function *)
        formals : formal list; (* Formal argument names *)
        body : stmt list;
  }

type s_decl = {
        sname : string;
        s_stmt_list : stmt list;
  }

type program = stmt list * func_decl list * s_decl list
```

```
let string_of_op = function
          Add -> "+"
  | Sub -> "-"
  | Mult -> "*"
  | Div -> "/"
  | Mod -> "%"
  | Equal -> "=="
  | Neq -> "!="
  | Less -> "<"
  | Leq -> "<="
  | Greater -> ">"
  | Geq -> ">="
  | And -> "&&"
  | Or -> "||"

let string_of_uop = function
          Neg -> "-"
  | Not -> "!"

let rec string_of_expr = function
          Literal(l) -> string_of_int l
  | FloatLit(f) -> string_of_float f
  | StringLit(s) -> s
  | BoolLit(x) -> (if x then "true" else "false")
  | Id(s) -> s
(*   | Point of pot
  | Line of line *)
  | Binop(e1, o, e2) ->
          string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr e2
  | Unop(o, e) -> string_of_uop o ^ string_of_expr e
  | Assign(v, e) -> v ^ " = " ^ string_of_expr e
  | Call(f, el) ->
          f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | Objcall(id1, id2, e) -> id1 ^ "." ^ id2 ^ "(" ^ String.concat ", " (List.map string_of_expr e)
^ ")"
(*   | Dotassign of string * string * expr
  | Lineassign of string * string * expr * expr *)
  (* | List of expr list *)
  (* | Mem of string * expr
  | ListAssign of string * expr * expr *)
  | StructAssign(v, seg, e) -> v ^ "." ^ seg ^ "=" ^ string_of_expr e
  | StructAccess(v, seg) -> v ^ "." ^ seg
  | Noexpr -> ""

let rec string_of_typ = function
          Int -> "int"
  | Bool -> "bool"
  | Float -> "float"
  | Str -> "string"
  | Void -> "void"
  | ListTyp a -> "list " ^ string_of_typ a
  | Objecttype s -> "struct " ^ s
  | _ -> raise(Failure("no matching type to print this type"))
```

prettyprint.ml

```
(* Pretty-printing functions *)

let string_of_op = function
          Add -> "+"
  | Sub -> "-"
  | Mult -> "*"
  | Div -> "/"
  | Equal -> "=="
  | Neq -> "!="
  | Less -> "<"
  | Leq -> "<="
  | Greater -> ">"
  | Geq -> ">="
  | And -> "&&"
  | Or -> "||"

let string_of_uop = function
          Neg -> "-"
  | Not -> "!"
```

```
let rec string_of_expr = function
        Literal(l) -> string_of_int l
  | BoolLit(true) -> "true"
  | BoolLit(false) -> "false"
  | Id(s) -> s
  | Binop(e1, o, e2) ->
        string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr e2
  | Unop(o, e) -> string_of_uop o ^ string_of_expr e
  | Assign(v, e) -> v ^ " = " ^ string_of_expr e
  | Call(f, el) ->
        f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | Noexpr -> ""

let rec string_of_stmt = function
        Block(stmts) ->
        "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
  | Expr(expr) -> string_of_expr expr ^ ";\n";
  | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n";
  | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s
  | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^
        string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
  | For(e1, e2, e3, s) ->
        "for (" ^ string_of_expr e1  ^ " ; " ^ string_of_expr e2 ^ " ; " ^
        string_of_expr e3  ^ ") " ^ string_of_stmt s
  | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt s

let string_of_typ = function
        Int -> "int"
  | Bool -> "bool"
  | Void -> "void"

let string_of_vdecl (t, id) = string_of_typ t ^ " " ^ id ^ ";\n"

let string_of_fdecl fdecl =
  string_of_typ fdecl.typ ^ " " ^
  fdecl.fname ^ "(" ^ String.concat ", " (List.map snd fdecl.formals) ^
  ")\n{\n" ^
  String.concat "" (List.map string_of_vdecl fdecl.locals) ^
  String.concat "" (List.map string_of_stmt fdecl.body) ^
  "}\n"

let string_of_program (vars, funcs) =
  String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
String.concat "\n" (List.map string_of_fdecl funcs)
```

semant.ml

```
(* Semantic checking for the eGrapher compiler *)

open Ast

module StringMap = Map.Make(String)

let check (statements, functions, structs) =

  (*struct check*)
  let struct_map =
    let test m s =
          let in_map =
        let x_map map a =
          match a with
           Vdecl (t, n) ->
           StringMap.add n t map
           | _ -> raise (Failure ("should not assign value in struct"))
           in
           List.fold_left x_map StringMap.empty s.s_stmt_list

            in
           StringMap.add s.sname in_map m
           in
        List.fold_left test StringMap.empty structs
  in

  (* Raise an exception if the given list has a duplicate *)
  let report_duplicate exceptf list =
        let rec helper = function
        n1 :: n2 :: _ when n1 = n2 -> raise (Failure (exceptf n1))
        | _ :: t -> helper t
        | [] -> ()
        in helper (List.sort compare list)
  in

  (* Raise an exception if a given binding is to a void type *)
  let check_not_void exceptf = function
        (Void, n) -> raise (Failure (exceptf n))
        | _ -> ()
  in

  (* Raise an exception of the given rvalue type cannot be assigned to
        the given lvalue type *)
  let check_assign lvaluet rvaluet err =
        match (lvaluet, rvaluet) with
        (ListTyp a, ListTyp b) -> if a = b then lvaluet else raise err
        | (Objecttype a, Objecttype b) ->  if a = b then lvaluet else raise err
        | (a, _) -> ignore(print_endline("; "^string_of_typ a));if lvaluet == rvaluet then
lvaluet else raise err
  in

  (* Separate global variable from statements *)
  (* Only allow variable declaration and assignment*)
  let globals =
    let rec test pass_list = function
        [] -> pass_list
        | hd :: tl -> let newlist =
      let match_fuc hd pass_list= match hd with
        Vdecl (a, b) -> (a, b)::pass_list
        | Expr (a) -> (fun x -> match x with Assign _-> pass_list | _ -> raise (Failure
("Wrong declare type in Block")) ) a
        | _ -> raise (Failure ("wrong declare in Block"))
```

```
          in match_fuc hd pass_list
          in test newlist tl
          in
          let test_function pass_list head = match head with
          Vdecl (a, b) -> (a, b)::pass_list
          | Expr (a) -> (fun x -> match x with Assign _-> pass_list | _ -> raise (Failure
("wrong declare in Block")) ) a
          | Block (block) ->  test pass_list block
          | _ -> raise (Failure ("Should not declare other than vdecl"))
          in List.fold_left test_function [] statements
  in

  List.iter (check_not_void (fun n -> "illegal void global " ^ n)) globals;

  report_duplicate (fun n -> "duplicate global " ^ n) (List.map snd globals);

  (**** Checking Functions ****)

  (* Check that a function named print is not defined *)
  if List.mem "print" (List.map (fun fd -> fd.fname) functions)
  then raise (Failure ("function print may not be defined")) else ();

  (* Check that there are no duplicate function names *)
  report_duplicate (fun n -> "duplicate function " ^ n)
          (List.map (fun fd -> fd.fname) functions);

  (* Function declaration for a named function *)
  let built_in_decls =  StringMap.add "print"
          { typ = Void; fname = "print"; formals = [(Int, "x")];
          body = [] } (StringMap.singleton "triangle"
          { typ = Void; fname = "triangle"; formals = [(Float, "x1");(Float, "x2");(Float,
"x3");(Float, "x4");(Float, "x5");(Float, "x6")];
          body = [] })
   in
   (* let built_in_decls = StringMap.add "prints"
          { typ = Void; fname = "printf"; formals = [(Float, "x")];
          body = [] } built_in_decls
   in
   let built_in_decls = StringMap.add "prints"
```

```
    { typ = Void; fname = "prints"; formals = [(Str, "x")];
          body = [] } built_in_decls
   in *)

  (* Add all function into function_decls *)
  let function_decls = List.fold_left (fun m fd -> StringMap.add fd.fname fd m)
                    built_in_decls functions
  in

  (* Check if function is declared before *)
  let function_decl s = try StringMap.find s function_decls
          with Not_found -> raise (Failure ("unrecognized function " ^ s))
  in

  let _ = function_decl "main" in (* Ensure "main" is defined *)

  let check_function func =

          (* Get all function locals*)
          let func_locals =
      let rec get_local pass_list head = match head with
          Vdecl (typ, name) -> (typ, name)::pass_list
          | Block (block) -> List.fold_left get_local pass_list block
          | _ -> pass_list
          in List.fold_left get_local [] func.body
          in

          List.iter (check_not_void (fun n -> "illegal void formal " ^ n ^
          " in " ^ func.fname)) func.formals;

          List.iter (check_not_void (fun n -> "illegal void local " ^ n ^
          " in " ^ func.fname)) func_locals;

          report_duplicate (fun n -> "duplicate formal or local " ^ n ^ " in function " ^
func.fname ^ "()")
          (List.map snd (func.formals@func_locals));

          let local_syb = List.fold_left (fun m (t, n) -> StringMap.add n t m)
          StringMap.empty (func.formals @ func_locals )
```

```
        and glob_syb = List.fold_left (fun m (t, n) -> StringMap.add n t m)
        StringMap.empty (globals)
        in

        let type_of_identifier s=
        try StringMap.find s local_syb
        with Not_found -> try StringMap.find s glob_syb
        with Not_found -> raise (Failure ("undeclared identifier " ^ s))
        in

        (* Add all object methods *)
        (* let obj_methods =  StringMap.add "add"
        (ListTyp Void,Int,1) (StringMap.singleton "plot"
        (Int,Int,1))
        in
        let obj_method s = try StringMap.find s obj_methods
        with Not_found -> raise (Failure ("unrecognized object function " ^ s))
        in *)

        (* Return the type of an expression or throw an exception *)
        let rec expr = function
        Literal _ -> Int
        | BoolLit _ -> Bool
        | FloatLit _ -> Float
        | StringLit _ -> Str
        | Id s -> type_of_identifier s
        | Binop(e1, op, e2) as e -> let t1 = expr e1 and t2 = expr e2 in
                (match op with
        Add | Sub | Mult | Div when (t1 = Int || t1 = Float) && (t2 = Int || t2 = Float) -> (if t1
= Float || t2 = Float then Float
                                else Int)
                | Equal | Neq when t1 = t2 -> Bool
                | Less | Leq | Greater | Geq when (t1 = Int || t1 = Float) && (t2 = Int || t2 =
Float) -> Bool
                | And | Or when t1 = Bool && t2 = Bool -> Bool
                | _ -> raise (Failure ("illegal binary operator " ^
                string_of_typ t1 ^ " " ^ string_of_op op ^ " " ^
                string_of_typ t2 ^ " in " ^ string_of_expr e))
        )
        | Unop(op, e) as ex -> let t = expr e in
        (match op with
        Neg when t = Int -> Int
        | Neg when t = Float -> Float
        | Not when t = Bool -> Bool
        | _ -> raise (Failure ("illegal unary operator " ^ string_of_uop op ^
                string_of_typ t ^ " in " ^ string_of_expr ex)))
        | Noexpr -> Void
        | Assign(var, e) as ex -> let lt = type_of_identifier var
                        and rt = expr e in
        check_assign lt rt (Failure ("illegal assignment " ^ string_of_typ lt ^
                " = " ^ string_of_typ rt ^ " in " ^
               string_of_expr ex))
        | Objcall(obj, meth, args) ->

        (match meth with
        | "length" ->
          if List.length args = 0 then
          (let lst = type_of_identifier obj in
          match lst with
                ListTyp(t) -> t
        | _ -> raise (Failure("variable "^obj^" is not matching type of "^meth^" method "))
        )
        else
        raise (Failure("list function "^meth^" takes no argument"))

| "add" ->
      if List.length args != 1 then
        raise (Failure("list function "^meth^" should has one argument"))
        else
        (match meth with
        "add" ->
          (let lst = type_of_identifier obj in
        match lst with
                ListTyp(t) -> let ele = expr (List.hd args) in
                if t <> ele then
                raise (Failure("variable "^obj^" is not matching type of input"))
             else
                ListTyp(t)
```

```
  | _ -> raise (Failure("variable "^obj^" is not matching type of "^meth^" method "))
  )
  | "get" ->
   (let lst = type_of_identifier obj in
  match lst with
           ListTyp(t) -> t
  | _ -> raise (Failure("variable "^obj^" is not matching type of "^meth^" method "))
  )
  | "remove" ->
  (let lst = type_of_identifier obj in
  match lst with
          ListTyp(t) -> t
  | _ -> raise (Failure("variable "^obj^" is not matching type of "^meth^" method "))
  )

  | a -> raise (Failure("have not define obj call " ^ a))
  )
  | Call(fname, actuals) as call ->
 (let fd = function_decl fname in
  if List.length actuals != List.length fd.formals then
  raise (Failure ("expecting " ^ string_of_int
           (List.length fd.formals) ^ " arguments in " ^ string_of_expr call))
  else
  match fname with
     "print" -> ignore(expr (List.hd actuals));Int
     | _ ->
           List.iter2 (fun (ft, _) e -> let et = expr e in
           ignore (check_assign ft et
           (Failure ("illegal actual argument found " ^ string_of_typ et ^
           " expected " ^ string_of_typ ft ^ " in " ^ string_of_expr e))))
           fd.formals actuals;
  fd.typ)

  | ListAssign (id, _, e) ->
(let et = expr e in let el = type_of_identifier id in
  match el with
  ListTyp(t) ->
  (if t != et then
  raise (Failure(string_of_typ et ^"\'s type is differ from list "^id))
```

```
        else
        ListTyp(t) )
        | _ -> raise (Failure("Not valid list type"))

        ignore(check_assign el et (Failure (string_of_typ et ^"\'s type is differ from list
"^id)));
        et
        )

        | StructAccess (id,field) ->
      (let item = type_of_identifier id in
            match item with
        Objecttype st_n -> (let args = StringMap.find st_n struct_map in
          try StringMap.find field args with Not_found -> raise(Failure("struct has no
matched field "^field))
        )
        | _-> raise (Failure ("No matched struct"))
      )

        | StructAssign (id, field, e) ->
      let item = type_of_identifier id in
        match item with
        Objecttype st_n ->
         (let args = StringMap.find st_n struct_map in
          let left = try  StringMap.find field args with Not_found -> raise(Failure("struct
has no matched field")) in
                let et = expr e
        in
                check_assign left et (Failure ("illegal actual argument found in structassign
"^id))
         )
        | _-> raise (Failure ("No matched struct"))
        in

        let check_bool_expr e = if expr e != Bool
        then raise (Failure ("expected Boolean expression in " ^ string_of_expr e))
        else () in

        (* Verify a statement or throw an exception *)
        let rec stmt = function
        Block sl -> let rec check_block = function
        [Return _ as s] -> stmt s
        | Return _ :: _ -> raise (Failure "nothing may follow a return")
        | Block sl :: ss -> check_block (sl @ ss)
        | s :: ss -> stmt s ; check_block ss
        | [] -> ()
        in check_block sl
        | Expr e -> ignore (expr e)
        | Vdecl (_, _) -> ()
        | Return e -> let t = expr e in if t = func.typ then () else
        raise (Failure ("return gives " ^ string_of_typ t ^ " expected " ^
                string_of_typ func.typ ^ " in " ^ string_of_expr e))

        | If(p, b, b1, b2) -> check_bool_expr p; stmt b; stmt b1; stmt b2
        | Elseif(e, s) -> check_bool_expr e; stmt s
        | For(e1, e2, e3, st) -> ignore (expr e1); check_bool_expr e2;
                        ignore (expr e3); stmt st
        | While(p, s) -> check_bool_expr p; stmt s
        | _ -> raise (Failure ("Not a vaid stmt"))
        in

        stmt (Block func.body)

  in
  List.iter check_function functions
```

codegen.ml

```
(* Code generation: translate takes a semantically checked AST and
produces LLVM IR
LLVM tutorial: Make sure to read the OCaml version of the tutorial
http://llvm.org/docs/tutorial/index.html
Detailed documentation on the OCaml LLVM library:
http://llvm.moe/
http://llvm.moe/ocaml/
*)

open Llvm
open Ast

module L = Llvm
module A = Ast

module StringMap = Map.Make(String)
module SymbolsMap = Map.Make(String)
module TypMap = Map.Make(String)


let struct_types:(string, lltype) Hashtbl.t = Hashtbl.create 10
let struct_datatypes:(string, string) Hashtbl.t = Hashtbl.create 10

let struct_field_indexes:(string, int) Hashtbl.t = Hashtbl.create 50
let struct_field_datatypes:(string, typ) Hashtbl.t = Hashtbl.create 50

let translate (statements, functions, structs) =
  let context = L.global_context () in
  let the_module = L.create_module context "eGrapher" in
  let llctx = L.global_context () in
  let llmem = L.MemoryBuffer.of_file "list.bc" in
  let llm = Llvm_bitreader.parse_bitcode llctx llmem in


  let i32_t  = L.i32_type  context
  and i8_t   = L.i8_type   context
  and i1_t   = L.i1_type   context
  and flt_t  = L.double_type context
  and str_t  = L.pointer_type (L.i8_type context)
  and void_t = L.void_type context
  and node_t = L.pointer_type (match L.type_by_name llm "struct.ListNode" with
          None -> raise (Invalid_argument "Option.get ListNode")
  | Some x -> x)
  and list_t = L.pointer_type (match L.type_by_name llm "struct.NodeList" with
          None -> raise (Invalid_argument "Option.get")
  | Some x -> x)
in

  let find_struct name =
          try Hashtbl.find struct_types name
          with | Not_found ->  raise (Failure ("Struct not found")) in


  let ltype_of_typ input = match input with
          A.Int   -> i32_t
          | A.Float -> flt_t
          | A.Bool  -> i1_t
          | A.Void  -> void_t
    | A.Str   -> str_t
          | A.ListTyp _  -> list_t
          | A.Objecttype(struct_name) -> find_struct struct_name
          | _ -> raise(Failure("No matching pattern in ltype_of_typ"))
  in



  (*Define the structs and its fields' datatype, storing in struct_types and
struct_field_datatypes*)
          let struct_decl_stub sdecl =
          let struct_t = L.named_struct_type context sdecl.A.sname in (*make llvm for this
struct type*)
          Hashtbl.add struct_types sdecl.sname struct_t;  (* add to map name vs
llvm_stuct_type *)
          in

          let struct_decl_field_datatypes sdecl =
          let svar_decl_list =
          let rec test_function pass_list head = match head with
```

```
        A.Vdecl (t, n) -> (t, n)::pass_list
        | A.Block (a) -> List.fold_left test_function pass_list a
        |_ -> pass_list
        in List.fold_left test_function [] sdecl.A.s_stmt_list
        in

        let type_list = List.map (fun (t,_) -> t) svar_decl_list in (*map the datatypes*)
        let name_list = List.map (fun (_,n) -> n) svar_decl_list in (*map the names*)
  (* Add key all fields in the struct *)
        ignore(
        List.map2 (fun f t ->
       let n = sdecl.sname ^ "." ^ f in
        Hashtbl.add struct_field_datatypes n t; (*add name, datatype*)
      ) name_list type_list;
        );

        in

        let struct_decl sdecl =
        let svar_decl_list =
        let rec test_function pass_list head = match head with
        A.Vdecl (t, n) -> (t, n)::pass_list
        | A.Block (a) -> List.fold_left test_function pass_list a
        |_ -> pass_list
        in List.fold_left test_function [] sdecl.A.s_stmt_list
        in

        let struct_t = Hashtbl.find struct_types sdecl.sname in (*get llvm struct_t code for
it*)
        let type_list = List.map (fun (t,_) -> ltype_of_typ t) svar_decl_list in (*map the
datatypes*)
        let name_list = List.map (fun (_,n) -> n) svar_decl_list in (*map the names*)
        let type_list = i32_t :: type_list in
        let name_list = ".k" :: name_list in
        let type_array = (Array.of_list type_list) in
        List.iteri (fun i f ->
        let n = sdecl.sname ^ "." ^ f in
        Hashtbl.add struct_field_indexes n i; (*add to name struct_field_indices*)
        ) name_list;
```

```
    L.struct_set_body struct_t type_array true
  in

  (* Add var_types for each struct so we can create it *)
  let _ = List.map (fun s -> struct_decl_stub s) structs in
  let _ = List.map (fun s -> struct_decl s) structs in
  let _ = List.map (fun s -> struct_decl_field_datatypes s) structs in

  (*take out globals*)
  let globals =
        let rec test_function pass_list head = match head with
        A.Vdecl (a, b) -> (a, b)::pass_list
        | A.Block (a) -> List.fold_left test_function pass_list a
        | _ -> pass_list
        in List.fold_left test_function [] statements
  in

   (*record list type*)
  let global_typ =
    let find_list m (t, n) = match t with
        | A.ListTyp a -> TypMap.add n a m
    | _ -> m
        in List.fold_left find_list TypMap.empty globals
  in

  let global_map = List.fold_left (fun map (t, n) -> StringMap.add n t map)
StringMap.empty globals
  in
  (* Declare each global variable; remember its value in a map *)
        let global_vars =
        let global_var m (t, n) =
        let init = L.const_null (ltype_of_typ t)
        in StringMap.add n ((L.define_global n init the_module)) m in
        List.fold_left global_var StringMap.empty globals in

  (* Global assignment *)
  let lookup_global n = try StringMap.find n global_vars
      with Not_found -> raise(Failure("Global value" ^ n ^" not declared"))
        in
```

```
 let rec global_expr = function
         A.Literal i -> L.const_int i32_t i
         | A.FloatLit f  -> L.const_float flt_t f
         | A.StringLit s -> (* str_t *) (* ignore(L.define_global ("test") (L.const_stringz
context s) the_module );  *) L.const_pointer_null str_t (* L.const_stringz context s *)
         | A.BoolLit b -> L.const_int i1_t (if b then 1 else 0)
         (* | A.Id s -> L.build_load (lookup_global s) s *)
         | A.Assign (s, e) ->
      (* match e with
         | A.StringLit cont-> let gl = lookup_global s in
         (ignore (L.delete_global gl);
         L.define_global s (L.const_stringz context cont) the_module)
         | _ -> ( *)
         match e with
         | A.Literal _
      | A.FloatLit _
         | A.BoolLit _ ->
      let e' = global_expr e
         and gl = lookup_global s in
         (* match t with
      | A.Str ->(ignore (L.delete_global gl);
         ignore(L.define_global s (e') the_module ); e')
         | _ -> *)(ignore (L.delete_global gl);
         ignore (L.define_global s e' the_module); e')
         | _ -> raise(Failure("Assign variable type is not primitive type"))

         | _ -> raise(Failure("Expression not allowed in global"))
 in

 let rec global_stmt = function
         A.Block sl -> List.iter global_stmt sl
         | A.Vdecl _ -> ()
         | A.Expr e -> ignore (global_expr e);
         | _ -> raise(Failure("statements not allowed in global"))
 in

 List.iter global_stmt statements;
```

```
let store_str typ name pass_list = match typ with
        A.Str -> name::pass_list
        | _ -> pass_list
in

let rec store_str_var pass_list = function
        A.Block sl -> List.fold_left store_str_var pass_list sl
        | A.Vdecl (typ, name) -> store_str typ name pass_list
        | _ -> pass_list
in
let str_var_list = List.fold_left store_str_var [] statements in


(* let map_str_var name = match typ with
        | A.StringLit s
        | A.Assign (s, e) -> lookup s in list  List.mem let e' = global_expr e
        in let check_empty inp = match value with
        | patt -> expr
        | _ -> " "
        in check_empty e' *)

(* Declare printf(), which the print built-in function will call *)

let printf_t = L.var_arg_function_type i32_t [| L.pointer_type i8_t |] in
let printf_func = L.declare_function "printf" printf_t the_module in
let print_bool_t = L.function_type i32_t [| i32_t |] in
let print_bool_f = L.declare_function "print_bool" print_bool_t the_module in

(* Declare draw triangle *)
let triangle_t = L.function_type i32_t [| L.pointer_type i8_t |] in
let triangle_func = L.declare_function "system" triangle_t the_module in

(* Declare list functions *)
let initIdList_t  = L.function_type list_t [| |] in
let initIdList_f  = L.declare_function "init_List" initIdList_t the_module in

let appendId_t       = L.function_type list_t [| list_t; L.pointer_type i8_t |] in
let appendId_f       = L.declare_function "add_back" appendId_t the_module in
```

```
  let int_to_pointer_t = L.function_type (L.pointer_type i8_t) [| i32_t |] in
  let int_to_pointer_f = L.declare_function "int_to_pointer" int_to_pointer_t the_module in

  let float_to_pointer_t = L.function_type (L.pointer_type i8_t) [| flt_t |] in
  let float_to_pointer_f = L.declare_function "float_to_pointer" float_to_pointer_t
the_module in

  let pointer_to_int_t = L.function_type i32_t [| L.pointer_type i8_t |] in
  let pointer_to_int_f = L.declare_function "pointer_to_int" pointer_to_int_t the_module in

  let pointer_to_float_t = L.function_type flt_t [| L.pointer_type i8_t |] in
  let pointer_to_float_f = L.declare_function "pointer_to_float" pointer_to_float_t
the_module in

  let index_acess_t = L.function_type (L.pointer_type i8_t) [| list_t; i32_t |] in
  let index_acess_f = L.declare_function "index_acess" index_acess_t the_module in

  let list_length_t = L.function_type i32_t [| list_t |] in
  let list_length_f = L.declare_function "length" list_length_t the_module in

  let list_remove_t = L.function_type i32_t [| list_t; i32_t |] in
  let list_remove_f = L.declare_function "remove_node" list_remove_t the_module in

  let list_length_t = L.function_type i32_t [| list_t |] in
  let list_length_f = L.declare_function "length" list_length_t the_module in

  let node_change_t = L.function_type i32_t [| list_t; i32_t; L.pointer_type i8_t |] in
  let node_change_f = L.declare_function "node_change" node_change_t the_module in

(*   let removeIdList_t = L.function_type idlist_t [| idlist_t; L.pointer_type i8_t |] in
  let removeIdList_f = L.declare_function "removeIdList" removeIdList_t the_module in
  let findNodeId_t = L.function_type node_t [| idlist_t; L.pointer_type i8_t |] in
  let findNodeId_f = L.declare_function "findNodeId" findNodeId_t the_module in
  let isEmptyIdList_t = L.function_type i8_t [| idlist_t |] in
  let isEmptyIdList_f = L.declare_function "isEmptyList" isEmptyIdList_t the_module in *)

  (* Define each function (arguments and return type) so we can call it *)
  let function_decls =
        let function_decl m fdecl =
        let name = fdecl.A.fname
        and formal_types =
        Array.of_list (List.map (fun (t,_) -> ltype_of_typ t) fdecl.A.formals)
        in let ftype = L.function_type (ltype_of_typ fdecl.A.typ) formal_types in
        StringMap.add name (L.define_function name ftype the_module, fdecl) m in
        List.fold_left function_decl StringMap.empty functions in

  (* Fill in the body of the given function *)
  let build_function_body fdecl =
   let (the_function, _) = try StringMap.find fdecl.A.fname function_decls
                          with Not_found -> raise(Failure("No matching pattern in
build_function_body"))
        in

        let builder = L.builder_at_end context (L.entry_block the_function) in
        let local_struct_datatypes:(string, string) Hashtbl.t = Hashtbl.create 10 in

        let int_format_str = L.build_global_stringptr "%d\n" "fmt" builder
        and float_format_str = L.build_global_stringptr "%f\n" "fmt" builder
        and bool_format_str = L.build_global_stringptr "%d\n" "fmt" builder
  and string_format_str = L.build_global_stringptr "%s\n" "fmt" builder
        in

        (*if name == main add check string add string value*)


        (* Construct the function's "locals": formal arguments and locally
        declared variables.  Allocate each on the stack, initialize their
        value, if appropriate, and remember their values in the "locals" map *)
        (*let local_vars =
        let add_formal m (t, n) p = L.set_value_name n p;
        let local = L.build_alloca (ltype_of_typ t) n builder in
        ignore (L.build_store p local builder);
        StringMap.add n local m in*)

        let local_vars =
        let add_formal m (t, n) p = L.set_value_name n p;
        let formal = match t with
        |Objecttype(struct_n) ->
```

```
                ignore(Hashtbl.add struct_datatypes n struct_n);
                let local = L.build_alloca (ltype_of_typ t) n builder in
                ignore (L.build_store p local builder); local

              | _ -> let local = L.build_alloca (ltype_of_typ t) n builder in
                ignore (L.build_store p local builder); local
        in
        StringMap.add n formal m
    in
      (* l_val = L.build_load (lookup objs) objs builder in
        let check = L.build_call list_length_f [| l_val |] "tmp" builder in
        let l_val = L.build_call initIdList_f [||] "init" builder in *)
        let add_local m (t, n) =
      match t with
        Objecttype(struct_n) -> ignore(Hashtbl.add struct_datatypes n struct_n);
        StringMap.add n (L.build_alloca (find_struct struct_n) n builder) m
        | ListTyp(ty) ->
          let alloc = L.build_alloca (ltype_of_typ t) n builder in
        let p = L.build_call initIdList_f [||] "init" builder in
        ignore (L.build_store p alloc builder);
        StringMap.add n alloc m
        | _ ->
       StringMap.add n (L.build_alloca (ltype_of_typ t) n builder) m
        in

        let locals =
        let rec test pass_list = function
        [] -> pass_list
        | hd :: tl -> let newlist =
          let match_fuc hd pass_list= match hd with
                A.Vdecl (a, b) -> (a, b)::pass_list
        | A.Block (a) -> test pass_list a
              | _ -> pass_list
        in match_fuc hd pass_list
        in test newlist tl

        in
        let test_function pass_list head = match head with
        A.Vdecl (a, b) -> (a, b)::pass_list
        | A.Block (block) -> test pass_list block
        | _ -> pass_list
        in List.fold_left test_function [] fdecl.A.body in

        let locals =
       if fdecl.A.fname = "main" then
        let add_list old_list (t, n) = match t with
        ListTyp(ty) -> (t, n) :: old_list
        | _ -> old_list
        in List.fold_left add_list locals globals
        else locals
        in

        let formals = List.fold_left2 add_formal StringMap.empty fdecl.A.formals
        (Array.to_list (L.params the_function)) in
        List.fold_left add_local formals locals in

        let locals =
        let rec test pass_list = function
        [] -> pass_list
        | hd :: tl -> let newlist =
          let match_fuc hd pass_list= match hd with
                A.Vdecl (a, b) -> (a, b)::pass_list
        | A.Block (a) -> test pass_list a
        | _ -> pass_list
        in match_fuc hd pass_list
        in test newlist tl

        in

        let test_function pass_list head = match head with
        A.Vdecl (a, b) -> (a, b)::pass_list
        | A.Block (block) -> test pass_list block
        | _ -> pass_list
        in List.fold_left test_function [] fdecl.A.body
    in

        (*record list type*)
        let locals_typ =
        let find_list m (t, n) = match t with
          | A.ListTyp a -> TypMap.add n a m
        | _ -> m
          in List.fold_left find_list global_typ (fdecl.A.formals@locals)
          in
          (* print_endline(string_of_int(StringMap.cardinal local_vars)); *)

          (* Return list type when called *)
          let look_typ n = try TypMap.find n locals_typ
        with Not_found -> raise(Failure("No matching list type in any variable"))
          in

          (* Return the value for a variable or formal argument, for a struct return the
pointer *)
    let lookup n = try StringMap.find n local_vars
          with Not_found -> try StringMap.find n global_vars with Not_found ->
raise(Failure("No matching pattern in Local_vars/Global_vars access in lookup"))
          in

          let symbol_vars =

          let add_to_symbol_table m (t, n) =
          SymbolsMap.add n t m in

          let locals =
          let rec test pass_list = function
          [] -> pass_list
          | hd :: tl -> let newlist =
             let match_fuc hd pass_list=
               match hd with
                    A.Vdecl (a, b) -> (a, b)::pass_list
                    | A.Block (a) -> test pass_list a
                    | _ -> pass_list
                    in match_fuc hd pass_list
          in test newlist tl

          in

          let test_function pass_list head = match head with
          A.Vdecl (a, b) -> (a, b)::pass_list
          | A.Block (block) -> test pass_list block
          | _ -> pass_list
          in List.fold_left test_function [] fdecl.A.body
      in

          let symbolmap = List.fold_left add_to_symbol_table SymbolsMap.empty
fdecl.A.formals in
          List.fold_left add_to_symbol_table symbolmap locals
      in

          let global_vars_2 =
        let add_to_symbol_table m (t, n) =
          SymbolsMap.add n t m in
          List.fold_left add_to_symbol_table SymbolsMap.empty globals
      in

          (* Return the type for a variable or formal argument *)
          let lookup_datatype n = try SymbolsMap.find n symbol_vars
          with Not_found ->
        try SymbolsMap.find n global_vars_2
        with Not_found -> raise(Failure("No matching pattern in globals access in
lookup_datatype"))
          in

          let format_str x_type = match x_type with
          "i32"  -> int_format_str
          | "double"  -> float_format_str
          | "i8*" -> string_format_str
      | "i1" -> int_format_str
          | _ -> (* string_format_str *) raise (Failure "Invalid printf type")
          in

          (* Construct code for an expression; return its value *)
          (*builder type*)
          let int_binops op =  (
          match op with
          A.Add     -> L.build_add
          | A.Sub          -> L.build_sub
          | A.Mult       -> L.build_mul
          | A.Div        -> L.build_sdiv
          | A.Mod        -> L.build_urem
          | A.Equal   -> L.build_icmp L.Icmp.Eq
          | A.Neq        -> L.build_icmp L.Icmp.Ne
          | A.Less    -> L.build_icmp L.Icmp.Slt
          | A.Leq        -> L.build_icmp L.Icmp.Sle
          | A.Greater -> L.build_icmp L.Icmp.Sgt
          | A.Geq        -> L.build_icmp L.Icmp.Sge
          | _ -> raise (Failure "Invalid Int Binop")
          )
          in

          let float_binops op =  (
          match op with
          A.Add          -> L.build_fadd
          | A.Sub        -> L.build_fsub
          | A.Mult       -> L.build_fmul
          | A.Div        -> L.build_fdiv
          | A.Mod        -> L.build_frem
          | A.Equal   -> L.build_fcmp L.Fcmp.Oeq
          | A.Neq        -> L.build_fcmp L.Fcmp.One
          | A.Less       -> L.build_fcmp L.Fcmp.Ult
          | A.Leq        -> L.build_fcmp L.Fcmp.Ole
          | A.Greater -> L.build_fcmp L.Fcmp.Ogt
          | A.Geq        -> L.build_fcmp L.Fcmp.Oge
          | _ -> raise (Failure "Invalid")
          )
          in

          let bool_binops op =
      match op with
          | A.And        -> L.build_and
          | A.Or         -> L.build_or
          | A.Equal   -> L.build_icmp L.Icmp.Eq
          | A.Neq        -> L.build_icmp L.Icmp.Ne
          | _ -> raise (Failure "Unsupported bool binop")
          in
```

```
        (* Return the datatype for a struct *)
        let lookup_struct_datatype(id, field) =
    let struct_name = Hashtbl.find struct_datatypes id in (*gets name of struct*)
        let search_term = ( struct_name ^ "." ^ field) in (*builds struct_name.field*)
        let my_datatype = Hashtbl.find struct_field_datatypes search_term in (*get
datatype*)
        my_datatype
        in

        (*Struct access function*)
        let struct_access struct_id struct_field isAssign builder = (*id field*)
        let struct_name = try Hashtbl.find struct_datatypes struct_id with Not_found -> try
Hashtbl.find local_struct_datatypes struct_id with Not_found -> raise(Failure("111"))
        in
        let search_term = (struct_name ^ "." ^ struct_field) in
        let field_index =try Hashtbl.find struct_field_indexes search_term
        with Not_found ->raise(Failure(search_term^""))
        in
        let value = lookup struct_id in
        (*and t = find_struct struct_name in
        let struct_pt = L.build_pointercast value t "tmp" builder in *)
        let _val = L.build_struct_gep value field_index struct_field builder in
        let _val =
        if isAssign then
        build_load _val struct_field builder
        else
        _val
        in
        _val
        in

        (* let str_float str = try float_of_string(String.concat "." [(String.sub str 4
(String.length str-4));"0"]) with Not_found -> raise(Failure("fucked up"))
        in *)
        let rec expr builder = function
        A.Literal i -> L.const_int i32_t i
         | A.FloatLit f  -> L.const_float flt_t f
        | A.StringLit s -> L.build_global_stringptr s "str" builder
        | A.BoolLit b -> L.const_int i1_t (if b then 1 else 0)
```

```
          | A.Noexpr -> L.const_int i32_t 0
          | A.Id s -> L.build_load (lookup s) s builder
          | A.StructAccess (id,field) ->
      (* ignore(print_endline("; SAccess"^id)); *)(
          struct_access id field true builder
          )
          | A.Binop (e1, op, e2) ->
          let e1' = expr builder e1
          and e2' = expr builder e2 in
          let combine = (L.string_of_lltype(L.type_of e1'), L.string_of_lltype(L.type_of e2'))
          in
          let binop_match combine e1' e2'= match combine with
                    ("i32", "i32") -> (* ignore(print_endline("; binop"^string_of_llvalue(e2')));
*)(int_binops op) e1' e2' "tmp" builder
          | ("double", "i32") -> let e3' = build_sitofp e2' flt_t "x" builder in (float_binops op)
e1' e3' "tmp" builder
          | ("i32", "double") -> let e3' = build_sitofp e1' flt_t "x" builder in (float_binops op)
e3' e2' "tmp" builder
          | ("double", "double") -> (float_binops op) e1' e2' "tmp" builder
          | ("i1", "i1") -> (bool_binops op) e1' e2' "tmp" builder
          | _ -> raise (Failure "Invalid binop type")
          in
          binop_match combine e1' e2'
          | A.Unop(op, e) ->
          let e' = expr builder e in
          (match op with
          A.Neg          ->
            (match e with
                    A.FloatLit f -> L.build_fneg
                     | A.Literal i -> L.build_neg
                    | A.Id s ->
                    let mytyp = lookup_datatype s in
                    (match mytyp with
                    A.Int -> L.build_neg
                    | A.Float -> L.build_fneg
                         | _ -> raise (Failure "Invalid Unop id type")
                    )
                    | A.StructAccess(id,field) ->(
                    let my_datatype = lookup_struct_datatype(id,field) in (*get datatype*)
                    match my_datatype with
                  | A.Int -> L.build_neg
                    | A.Float -> L.build_fneg
                    | _ ->  raise (Failure "Invalid Types of Struct binop")
                    )
                    | _ -> raise (Failure "Invalid Unop type")
           )
          | A.Not          -> L.build_not) e' "tmp" builder
          | A.Assign (s, e) -> let e' = expr builder e in
                    ignore (L.build_store e' (lookup s) builder); e'
          | A.StructAssign (id, field, e) ->
         let e' = expr builder e in
          let des =(struct_access id field false builder) in
          ignore (L.build_store e' des builder);e'

          | A.Objcall (objs, funs, args) ->
          let check_fun objs funs args = match funs with
          | "add" ->
            (let l_val = L.build_load (lookup objs) objs builder in
                let d_val = expr builder (List.hd args) in

                    let void_d_ptr =
                    match look_typ objs with
                    | A.Int ->
                    L.build_call int_to_pointer_f [| d_val |] "tmp" builder
                    | A.Float ->
                    L.build_call float_to_pointer_f [| d_val |] "tmp" builder
                    | _ -> raise(Failure("List contains element other then int or float"))
                in

                    let app = L.build_call appendId_f [| l_val; void_d_ptr |] "tmp" builder
                    in
                    (* ignore (L.build_store app (lookup objs) builder); *)
          app)

          | "get" ->
              (let l_val = L.build_load (lookup objs) objs builder in
                    let d_val = expr builder (List.hd args) in
                    let void_ptr = L.build_call index_acess_f [| l_val;d_val |] "tmp" builder in
```

```
                    match look_typ objs with
                    | A.Int ->
                    L.build_call  pointer_to_int_f [| void_ptr |] "tmp" builder
                    | A.Float ->
                    L.build_call  pointer_to_float_f [| void_ptr |] "tmp" builder
                  | _ -> raise(Failure("List contains element other then int or float"))
                   L.build_call  pointer_to_int_f [| void_ptr |] "tmp" builder)
                    (* let void_res = L.build_call index_acess_f [| void_d_ptr;d_val |] "tmp"
builder in
                    L.build_call pointer_to_int_f [| void_res |] "tmp" builder *)

         | "remove" ->
             (let l_val = L.build_load (lookup objs) objs builder in
                    let d_val = expr builder (List.hd args) in
                   let app = L.build_call list_remove_f [| l_val;d_val |] "rmv" builder in
                    app )

         | _ -> L.const_int i32_t 42
         in
        check_fun objs funs args

         | A.ListAssign (id, pos, e) ->
        (let l_val = L.build_load (lookup id) id builder in
         let position = expr builder pos in
         let data = expr builder e in
         let data2 =
           match TypMap.find id locals_typ with
         | A.Int ->
         L.build_call  int_to_pointer_f [| data |] "tmp" builder
         | A.Float ->
         L.build_call  float_to_pointer_f [| data |] "tmp" builder
         | _ -> raise(Failure("List contains element other then int or float"))
          in
         L.build_call node_change_f [| l_val;position;data2 |] "tmp" builder )

         | A.Call ("triangle", [a1;a2;a3;a4;a5;a6]) ->
        let triangle_args= String.concat " " ["./lib/run ";
string_of_expr(a1);string_of_expr(a2);string_of_expr(a3);string_of_expr(a4);string_of_e
xpr(a5);string_of_expr(a6)] in
```

```
        let e' =  L.build_global_stringptr triangle_args "str" builder in
        L.build_call triangle_func [| e' |] "triangle" builder

        | A.Call ("print", [e]) ->
        (* ignore(print_endline("; print")); *)
        let e' = expr builder e in
        let typ_e' = L.string_of_lltype(L.type_of e') in
        if typ_e' = "i1" then
        (* if (L.string_of_llvalue(e')) = "i1 true" then
        L.build_call printf_func [| format_str typ_e' ; L.build_global_stringptr ("true") "str"
builder |] "printf" builder
        else
        L.build_call printf_func [| format_str typ_e' ; L.build_global_stringptr ("flase") "str"
builder |] "printf" builder *)
            let e1' = build_sitofp e' flt_t "x" builder in
        let e2' = build_fptosi e1' i32_t "x2" builder in
        (* ignore(print_endline("; "^L.string_of_lltype(L.type_of e2'))); *)
        L.build_call print_bool_f [| e2' |] "print_bool" builder
        (* L.build_call printf_func [| format_str typ_e' ; e' |] "printf" builder *)
        else
        L.build_call printf_func [| format_str typ_e' ; e' |] "printf" builder
        (* L.build_call printf_func [| int_format_str ; (expr builder e) |]
        "printf" builder *)

        | A.Call (f, act) ->
        let (fdef, fdecl) = try StringMap.find f function_decls
                    with Not_found -> raise(Failure("Not_found expr in A.Call"))
                  in
        let actuals = List.rev (List.map (expr builder) (List.rev act)) in
        let result = (match fdecl.A.typ with A.Void -> ""
                                  | _ -> f ^ "_result") in
        L.build_call fdef (Array.of_list actuals) result builder

        | _ -> raise(Failure("No matching pattern in expr"))
        in

        (* Invoke "f builder" if the current block doesn't already
        have a terminal (e.g., a branch). *)
        let add_terminal builder f =
        match L.block_terminator (L.insertion_block builder) with
        Some _ -> ()
        | None -> ignore (f builder) in

        (* Build the code for the given statement; return the builder for
        the statement's successor *)
        let rec stmt builder = function
     A.Block sl -> List.fold_left stmt builder sl
        | A.Vdecl (t, n) -> (* ignore(print_endline("; Vdecl "^n)); *)builder
        | A.Elseif _ -> builder
        | A.Foreach _ -> builder
        | A.Break  -> builder
        | A.Continue  -> builder
        | A.Expr e -> ignore (try expr builder e with Not_found -> raise(Failure("In stmt
function Expr error"))); builder
        | A.Return e -> (* ignore(print_endline("; Return")); *) ignore (match fdecl.A.typ
with
        A.Void -> L.build_ret_void builder
           | _ -> L.build_ret (expr builder e) builder); builder

        | A.If (predicate, then_stmt, elif_stmt, else_stmt) ->
        let bool_val = expr builder predicate in
        let merge_bb = L.append_block context "merge" the_function in

        let then_bb = L.append_block context "then" the_function in
        add_terminal (stmt (L.builder_at_end context then_bb) then_stmt)
        (L.build_br merge_bb);

        (* let elif_bb = L.append_block context "elif" the_function in
 add_terminal (stmt (L.builder_at_end context elif_bb) elif_stmt)
        (L.build_br merge_bb); *)

        let else_bb = L.append_block context "else" the_function in
        add_terminal (stmt (L.builder_at_end context else_bb) else_stmt)
        (L.build_br merge_bb);

        ignore (L.build_cond_br bool_val then_bb else_bb builder);
        L.builder_at_end context merge_bb
```

```
        | A.While (predicate, body) ->
        let pred_bb = L.append_block context "while" the_function in
        ignore (L.build_br pred_bb builder);

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

        let pred_builder = L.builder_at_end context pred_bb in
        let bool_val = expr pred_builder predicate in

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

        | A.For (e1, e2, e3, body) -> stmt builder
        ( A.Block [A.Expr e1 ; A.While (e2, A.Block [body ; A.Expr e3]) ] )

        | _ -> raise(Failure("No matching pattern in stmt"))
        in

        (* Build the code for each statement in the function *)
   let builder = try stmt builder (A.Block fdecl.A.body) with Not_found ->
raise(Failure("stmt function error")) in

        (* Add a return if the last block falls off the end *)
        add_terminal builder (match fdecl.A.typ with
        A.Void -> L.build_ret_void
        | t -> L.build_ret (L.const_int (ltype_of_typ t) 0))
  in

        try List.iter build_function_body functions;
  (* ignore(Llvm_linker.link_modules the_module llm); *)
  the_module

     with Not_found -> raise(Failure("No matching pattern in buuilding function"))
```

Makefile

```
# Make sure ocamlbuild can find opam-managed packages: first run

# eval `opam config env`

# Easiest way to build: using ocamlbuild, which in turn uses ocamlfind

.PHONY : egrapher.native

egrapher.native :

ocamlbuild -use-ocamlfind -pkgs
llvm,llvm.analysis,llvm.linker,llvm.bitreader,llvm.irreader -cflags -w,+a-4 egrapher.native

# "make clean" removes all generated files

.PHONY : clean

clean :

ocamlbuild -clean

rm -rf testall.log *.diff egrapher scanner.ml parser.ml parser.mli

rm -rf *.cmx *.cmi *.cmo *.cmx *.o a.out *.bc *.ll
```

```
# More detailed: build using ocamlc/ocamlopt + ocamlfind to locate LLVM

OBJS = ast.cmx codegen.cmx parser.cmx scanner.cmx semant.cmx egrapher.cmx

egrapher : $(OBJS)

ocamlfind ocamlopt -linkpkg -package llvm.linker -package llvm.bitreader -package
llvm.irreader -package llvm -package llvm.analysis $(OBJS) -o egrapher

scanner.ml : scanner.mll

ocamllex scanner.mll

parser.ml parser.mli : parser.mly

ocamlyacc parser.mly

%.cmo : %.ml

ocamlc -c $<

%.cmi : %.mli

ocamlc -c $<
```

```
%.cmx : %.ml

ocamlfind ocamlopt -c -package llvm $<

### Generated by "ocamldep *.ml *.mli" after building scanner.ml and parser.ml

ast.cmo :

ast.cmx :

codegen.cmo : ast.cmo list.bc

codegen.cmx : ast.cmx list.bc

egrapher.cmo : semant.cmo scanner.cmo parser.cmi codegen.cmo ast.cmo

egrapher.cmx : semant.cmx scanner.cmx parser.cmx codegen.cmx ast.cmx

parser.cmo : ast.cmo parser.cmi

parser.cmx : ast.cmx parser.cmi

scanner.cmo : parser.cmi

scanner.cmx : parser.cmx
```

```
semant.cmo : ast.cmo


semant.cmx : ast.cmx


parser.cmi : ast.cmo


egrapher-llvm.tar.gz : $(TARFILES)


cd .. && tar czf egrapher-llvm/egrapher-llvm.tar.gz \

$(TARFILES:%=egrapher-llvm/%)
```

External C Libraries

list.c

```
#include <stdio.h>
#include <stdlib.h>
#include "list.h"

struct NodeList* init_List()
{
  struct NodeList* new = (struct NodeList*) malloc(sizeof(struct NodeList));
  new->head = NULL;
  return new;
}

struct NodeList *add_front(struct NodeList *list, void *data)
{
 struct ListNode *node = (struct ListNode*) malloc(sizeof(struct ListNode));

  /* judge */
  if (node == NULL){
          perror("malloc returned a NULL");
```

```
        exit(1);
  }

  node->data = data;
  node->next = list->head;
  list->head = node;
  return list;
}

struct NodeList *add_back(struct NodeList *list, void *data)
{
  struct ListNode* new;
  if (list->head == NULL){
        list->head = (struct ListNode*) malloc(sizeof(struct ListNode));
        new = list->head;
  }
  else{
        struct ListNode* node = list->head;
        while (node->next != NULL)
        node = node->next;
        node->next = (struct ListNode*) malloc(sizeof(struct ListNode));
        new = node->next;
  }

  new->data = data;
  new->next = NULL;
  return list;
}

void *index_acess(struct NodeList *list, int id)
{
  /*printf("%d\n", id);
  if(list->head)
        printf("%d\n", length(list));*/
  if (id >= length(list)){
        perror("id is longer than list length");
        exit(1);
  }
  struct ListNode* node = list->head;
  while (id > 0){
          node = node->next;
          id--;
  }

  // printf("%d\n", pointer_to_int(node->data));
  return node->data;
}

int node_change(struct NodeList *list, int id, void *data)
{
  if (id >= length(list)){
          perror("id is longer than list length");
          exit(-1);
  }

  struct ListNode* node = list->head;
  while (id > 0){
          node = node->next;
          id--;
  }

  node->data = data;
  return 0;
}

int remove_node(struct NodeList *list, int id)
{
  if (id >= length(list)){
          perror("id is longer than list length");
          exit(-1);
  }

  struct ListNode *dummy = (struct ListNode*) malloc(sizeof(struct ListNode));
  dummy->data = NULL;
  dummy->next = list->head;

  struct ListNode *tmp_pre = dummy;
  struct ListNode *tmp = dummy->next;
  while(id > 0)
  {
          tmp_pre = tmp_pre->next;
          tmp = tmp->next;
  }

  tmp_pre->next = tmp->next;
  free(tmp);

  list->head = dummy->next;
  free(dummy);

  return 0;
}

int length(struct NodeList *list)
{
  if (!list) {
          return -1;
  }
  int l = 0;
  struct ListNode* tmp = list->head;

  while(tmp)
  {
          l++;
          tmp = tmp -> next;
  }

  return l;
}

int isEmptyList(struct NodeList *list)
{
  return (list->head == NULL);
}
```

```
void *int_to_pointer(int i)
{
  int *pi = (int *)malloc(sizeof(int));
  *pi = i;
  return (void*)pi;
}

void *float_to_pointer(double f)
{
  double *pf = (double *)malloc(sizeof(double));
  *pf = f;
  return (void*)pf;
}

int pointer_to_int(void *pi)
{
  return *((int*)pi);
}

double pointer_to_float(void *pf)
{
  return *((double*)pf);
}

int print_bool(int a)
{
        fputs(a ? "true\n" : "false\n", stdout);
        return a;
}
```

list.h

```
#ifndef _LIST_H_
#define _LIST_H_

struct ListNode {
        void *data;
```

```
           struct ListNode *next;
};

struct NodeList {
  struct ListNode *head;
};

struct NodeList *init_List();

struct NodeList *add_front(struct NodeList *list, void *data);

struct NodeList *add_back(struct NodeList *list, void *data);

void *index_acess(struct NodeList *list, int id);

int node_change(struct NodeList *list, int id, void *data);

int remove_node(struct NodeList *list, int id);

int length(struct NodeList *list);

int isEmptyList(struct NodeList *list);

void *int_to_pointer(int i);

void *float_to_pointer(double f);

int pointer_to_int(void *pi);

double pointer_to_float(void *pf);

int print_bool(int a);

#endif /* #ifndef _SOURCE_H_ */
```

ulity.c

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include "ulity.h"

void *int_to_pointer(int i)
{
  int *pi = (int *)malloc(sizeof(int));
  *pi = i;
  return (void*)pi;
}

void *float_to_pointer(float f);
{
  float *pf = (float *)malloc(sizeof(float));
  *pf = f;
  return (void*)pf;
}

int pointer_to_int(void *pi);
{
  return *((int*)pi);
}

float pointer_to_float(void *pf);
{
  return *((float*)pf);
}
```

ulity.h

```
#ifndef _ULITY_H_
#define _ULITY_H_

int *int_to_pointer(int i);

float *float_to_pointer(float f);

int pointer_to_int(int *pi);
```

```
float pointer_to_float(float *pf);

#endif /* #ifndef _ULITY_H_ */
```

Test Suite code

testall.sh

```
#!/bin/sh
#Run testcases under dir /tests
make
# Path to the LLVM interpreter
LLI="/usr/local/opt/llvm/bin/lli"
LLL="/usr/local/opt/llvm/bin/llvm-link"


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

# Set time limit for all operations
ulimit -t 30

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

keep=0

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

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

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

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

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

```
}

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

   echo -n "$basename..."

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

        generatedfiles=""

        generatedfiles="$generatedfiles ${basename}.ll ${basename}.out" &&
        Run "clang -emit-llvm -o list.bc -c src/list.c" &&
        Run "$EGRAPHER" "<" $1 ">" "${basename}.ll" &&
        Run "$LLL" "${basename}.ll list.bc" "-o" "a.out" &&
        Run "$LLI" "a.out" ">" "${basename}.out"&&
        Compare ${basename}.out ${reffile}.out ${basename}.diff

        # Report the status and clean up the generated files

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

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

        echo -n "$basename..."

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

        generatedfiles=""

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

        # Report the status and clean up the generated files

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

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

```
done

shift `expr $OPTIND - 1`

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

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


if [ $# -ge 1 ]
then
        files=$@
else
        files="tests/test-*.eg tests/fail-*.eg"
fi

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

exit $globalerror
```

fail-assign1.eg

```
int main(){
        int a;
        bool b;
        a = 123;
        a = 1234;
        b = true;
        b = false;
    a = true; /* Fail: assigning bool to integer */
```

fail-assign1.err

```
Fatal error: exception Failure("illegal assignment int = bool in a = true")
```

fail-assign2.eg

```
int main(){
        int a;
        bool b;
    b = 10; /* Fail: assigning interger to bool */
    return 0;
}
```

fail-assign2.err

```
Fatal error: exception Failure("illegal assignment bool = int in b = 10")
```

fail-assign3.eg

```
void myvoid(){
}

int main(){
        int i;
        i = myvoid(); /* Fail: assign void to integer */
}
```

fail-assign3.err

```
Fatal error: exception Failure("illegal assignment int = void in i = myvoid()")
```

fail-dead1.eg

```
int main(){
        int i;
        return 1;
i = 123; /* Code after return */
```

fail-dead1.err

```
Fatal error: exception Failure("nothing may follow a return")
```

fail-expr1.eg

```
int main(){
int a = 1;
bool b = true;
a = a + b; /* Error, bool add int */
```

fail-expr1.err

```
Fatal error: exception Failure("illegal binary operator int + bool in a + b")
```

fail-func-arg-num.eg

```
int get(int x, int b){
        return 0;
}

int main(){
        int b = 10 ;
        print(get(b));
        return 0;
}
```

fail-func-arg-num.err

```
Fatal error: exception Failure("expecting 2 arguments in get(b)")
```

fail-func-duplicate-formal-local.eg

```
int change(int a){

        int a = 9;

        a = a+1;

        return a;

}

int main(){

        int a = 9;

        a = change(a);

        print(a);

        return 0;

}
```

fail-func-duplicate-formal-local.err

```
Fatal error: exception Failure("duplicate formal or local a in function change()")
```

fail-func-duplicate-locals.eg

```
int change(int a){

        a = a+1;
        int b;
        int b = 10;
        return a;
}

int main(){

        int a = 9;
        a = change(a);
        print(a);
        return 0;
}
```

fail-func-duplicate-locals.err

```
Fatal error: exception Failure("duplicate formal or local b in function change()")
```

fail-func-formal-struct.eg

```
struct person[

        string name;

        int age;

]

struct xxxx[

        string name;

        int age;

]

int get(struct xxxx a){

        print(a.age);
        return a.age;

}

int main(){

        struct person a;

        a.age = 100;

        print(get(a));

        return 0;

}
```

fail-func-formal-struct.err

```
Fatal error: exception Failure("illegal actual argument found struct person expected
struct xxxx in a")
```

fail-func-void-local.eg

```
int change(int a){

        a = a+1;

        void b;

        return a;

}

int main(){

        int i = 0;

        change(i);
```

```
}

fail-func-void-local.err

Fatal error: exception Failure("illegal void local b in change")

fail-global-duplicate.eg

int a = 1+1;

string b = "asdf";

int a;

int main()  {

        prints("Hello");

        return 0;

}

fail-global-duplicate.err

Fatal error: exception Failure("duplicate global a")

fail-struct-access2.eg

struct person[

        string name;

        int age;

]

int get(struct person a){
        print(a.age);

        return a.p;

}

int main(){

        struct person a;

        print(get(a));

        return 0;

}
```

fail-struct-access2.err

```
Fatal error: exception Failure("struct has no matched field p")
```

fail-struct-retrun1.eg

```
struct person[

        string name;

        int age;

]

int get(struct person a){

        print(a.age);

        return a.name;

}

int main(){
        struct person a;

        print(get(a));

        return 0;

}
```

fail-struct-retrun1.err

```
Fatal error: exception Failure("return gives string expected int in a.name")
```

test-comparison.eg

```
int main(){

        print(12==12);

        print(12==22);

        print(1.2==1.2);

        print(1.2==1.3);

        print(12>12);

        print(12<22);

        print(1.2>1.2);

        print(1.2>1);

        print(1.2<1.3);

        return 0;

}
```

test-comparison.out

```
true

false

true

false

false

true

false

true

true
```

test-flt-binops.eg

```
int main(){

	float x=6.11;

	float y=3.45;

	print(x+y);

	print(x-y);

	print(x*y);

	print(x/y);

	print(x+y);

	print(x==y);
	print(x<=y);

	print(x>=y);

	print(x<y);

	print(x>y);

	print(x!=y);

	return 0;

}
```

test-flt-binops.out

```
9.560000

2.660000

21.079500

1.771014

9.560000
```

test-for1.eg

```
int main(){

	int i;

	i = 0;

	for (;i<10;){

	print(i);
```

```
          i = i + 1;

      }
          return 0;

}
```

test-for1.out

```
0
1
2
3
4
5
6
7
8
9
```

test-for2.eg

```
int main(){

          int i;

          i = 0;

          for (i=1;i<10;){

          print(i);

          i = i + 1;

          }

          return 0;

}
```

test-for2.out

```
1
2
3
4
5
6
7
8
9
```

test-for3.eg

```
int main(){

        int i;

        i = 0;

        for (i=1;i<10;i=i+1){

        print(i);

        }

        return 0;

}
```

test-for3.out

```
1
2
3
4
5
```

```
6
7
8
9
```

test-func-loop.eg

```
int func(int i) {



   if(i == 0) {

           return 0;

   }

   if(i == 1) {

           return 1;

   }

   return i*2;

}


int main() {

   int i = 0;

   for (i; i < 5; i=i+1) {

           print(func(i+2));

   }
   return 0;

}
```

test-func-loop.out

```
4

6

8

10

12
```

test-func-recur.eg

```
int fibonacci(int i) {

   if(i == 0) {
         return 0;
   }

   if(i == 1) {
         return 1;
   }

   return fibonacci(i-1) + fibonacci(i-2);
}

int main() {

   int i;

   for (i = 0; i < 10; i=i+1) {
          print(fibonacci(i));
   }

   return 0;
}
```

test-func-recur.out

```
0
1
1
2
3
5
8
13
21
34
```

test-gcd.eg

```
int gcd(int a, int b) {

          while (a != b){

          if (a > b)

          a = a - b;

          else

                b = b - a; }

          return a;

}
```

```
int main(){

print(gcd(2,14));

print(gcd(3,15));

print(gcd(99,121));

return 0;

}
```

test-gcd.out

```
2

3

11
```

test-hello.eg

test-hello.eg

```
int main(){

	print("Hello, world!");

	return 0;

}
```

test-hello.out

```
Hello, world!
```

test-if-dagling.eg

```
int main()  {

        if (3>2){

        print("true");

        if(3<4){

        print("3<4");

        }

        }

        if (3>6){

        print("second if");

        }

        else{

        print("false");

        }

        return 0;

}
```

test-if-dagling.out

```
true

3<4
```

false

test-if1.eg

```
test-if1.eg
int main(){

	int a = 1;

	if(a>0){

	print("a>0");

	}

	else{

	print("not a>0");

	}

	return 0;

}
```

test-if1.out

```
a>0
```

test-if2.eg

```
int main(){

	int a = 1;

	int b = 2;
```

```
        if(a>0){

        print("a>0");

        if(a<b){

        print("a<b");

        }

        }

        else{

        print("not a>0");

        }

        return 0;

}
```

test-if2.out

```
a>0

a<b
```

test-if3.eg

```
int main(){

        int a = 1;

        int b = 2;

        if(a>0){
```

```
        print("a>0");
        }
        if(a>b){
        print("a>b");
        }
        else{
        print("a<b");
        }
        return 0;
}
```

test-if3.out

```
a>0
a<b
```

test-if4.eg

```
int main(){
        if(false){
        print(123);
        }
        else{
```

```
        print(321);

        }

        return 0;
}
```

test-if4.out

```
321
```

test-int-binops.eg

```
int main(){

        int x=5;

        int y=3;

        print(x+y);

        print(x-y);

        print(x*y);

        print(x/y);

        print(x+y);

        print(x==y);

        print(x<=y);

        print(x>=y);

        print(x<y);

        print(x>y);
        print(x!=y);

        return 0;

}
```

test-int-binops.out

```
8

2

15

1

8

false

false

true

false

true

true
```

test-int-flt-binops.eg

```
int test(int a){

        a = 8;

        float c = 12.03;
        c = 8 + 9.9;

        print(c);

        c = a + 9.9;

        print(c);

        c = 10.9 + 8;

        print(c);

        c = 10.9 + a;

        print(c);

        return a;

}

int main(){

        test(1);

        return 0;

}
```

test-int-flt-binops.out

```
17.900000

17.900000

18.900000

18.900000
```

test-list-obj-call.eg

```
list int l;

int main(){

	l.add(12);

	l.add(2);

	print(l.get(0));

	print(l.get(1));

	return 0;

}
```

test-list-obj-call.out

```
12

2
```

test-local1.eg

```
void foo(bool a){

	int i = 23;

	print(i+1);

}
```

```
int main(){

	int i = 7;

	foo(true);

	print(i);

	return 0;

}
```

test-local1.out

```
24

7
```

test-local2.eg

```
int foo(int a, bool b){

	if(b){

	return 10;

	}

	return a + 10;

}

int main(){

	print(foo(53, false));

	return 0;

}
```

test-local2.out

```
63
```

test-ne.eg

```
int main(){
	int x = 7;
	float y = 3.1;
	print(-x);
	print(-y);
	x = -7;
	y = -3.1;
	print(x);
	print(-x);
	print(y);
	print(-y);
	return 0;
}
```

test-ne.out

```
-7
-3.100000
```

```
-7

7

-3.100000

3.100000
```

test-ops1.eg

```
int main(){

  print("+-*/");

  print(1 + 2);

  print(1 - 2);

  print(1 * 2);

  print(10 / 2);

  print("==");

  print(1 == 2);

  print(1 == 1);

  print("!=");

  print(1 != 2);

  print(1 != 1);

  print("< > <= >=");

  print(1 > 2);

  print(2 > 1);
  print(1 < 2);

  print(2 < 1);

  print(1 <= 2);

  print(1 <= 1);

  print(2 <= 1);

  print(1 >= 2);

  print(1 >= 1);

  print(2 >= 1);

  return 0;

}
```

test-ops1.out

```
+-*/

3

-1

2

5

==

false

true
```

```
!=

true

false

< > <= >=

false

true

true

false

true

true

false

false

true

true
```

test-ops2.eg

```
int main(){

  bool x;

  print(true);

  print(false);

  x = true && false;
  x = x && x;
  print(true && true);
  print(true && false);
  print(false && true);
  print(false && x);
  print(true || true);
  print(true || false);
  print(false || true);
  print(false || false);
  print(!false);
  print(!true);
  return 0;
}
```

test-ops2.out

```
true
false
true
false
false
```

```
false

true

true

true

false

true

false
```

test-print.eg

```
int main()  {

        int a = 1;

        float b = 2.1;

        bool c = true;

        string d = " World!";

        print(0);

        print(a);

        print(0.23);

        print(b);

        print(false);

        print(c);
        print("Hello");

        print(d);

        return 0;

}
```

test-print.out

```
0

1

0.230000

2.100000

false

true

Hello

 World!
```

test-struct-access.eg

```
struct person[

        string name;

        int age;

]

int get(struct person a){
        print(a.age);

        return a.age;

}

int main(){

        struct person a;

        a.age = 100;

        print(get(a));

        return 0;

}
```

test-struct-access.out

```
100

100
```

test-while1.eg

```
int main(){

        int i=5;

        while(i>0){

        print(i);

        i=i-1;

        }

        return 0;
}
```

test-while1.out

```
5

4

3

2

1
```

test-while2.eg

```
int foo(int a){

        int i=1;

        while(a>0){

        i=i+3;

        a=a-1;

        }

        return i;

}
```

test-while2.out

```
16
```

test-fac.eg

```
int factorial(int n){
	int temp;
	if(n<=1){
		return 1;
	}
	temp=factorial(n-1)*n;
	return temp;
}

int main(){
	print(factorial(10));
	return 0;
}
```

test-fac.out

```
3628800
```

test-if-emptyblock1.eg

```
int main(){
	int a = 1;
	if(a>0){
	}
	print("1");
	return 0;
}
```

test-if-emptyblock1.out

```
1
```

test-if-emptyblock2.eg

```
int main(){
	int a = 1;
	if(a>0){
	}
	else{}
	print("1");
	return 0;
}
```

test-if-emptyblock2.out

```
1
```

test-if5.eg

```
int main(){
	if(true){
		if(true){
			print(1);
		}
	print(2);
	}
	else{
		print(3);
	}
	if(false){
		if(true){
			print(4);
		}
		print(5);
	}
	else{
	print(8);
	}
	return 0;
}
```

test.if5.out

```
1
2
8
```

test.intMax.eg

```
int main(){
    int a = 2147483647;
    int b = -2147483648;
    print(a);
    print("\n");
    print(b);
           return 0;
}
```

test.intMax.out

```
2147483647


-2147483648
```

fail-list-call-notdefined.eg

```
int main(){
    list int l;
    l.append(23.2);
    print(l.get(0));
    return 0;
}
```

fail-list-call-notdefined.err

```
Fatal error: exception Failure("have not define obj call append")
```

fail-list-type.eg

```
int main(){
    list int l;
    l.add(23.2);
    print(l.get(0));
    return 0;
}
```

fail-list-type.err

```
Fatal error: exception Failure("variable l is not matching type of input")
```

test-float-list.eg

```
int main(){
   list float l;
   l.add(23.2);
   print(l.get(0));
   return 0;
}
```

test-float-list.out

```
23.200000
```

test-globle-list.eg

```
list int l;

int main(){
        l.add(10);
        print(l.get(0));
        return 0;
}
```

test-globle-list.out

```
10
```

test-list-remove.eg

```
int main(){
    list float l;
    l.add(23.2);
    l.add(34.4);
    l.remove(0);
    print(l.get(0));
    return 0;
}
```

test-list-remove.out

```
34.400000
```

test-local-list.eg

```
int main(){
    list int l;
    l.add(10);
    print(l.get(0));
    return 0;
}
```

test-local-list.out

```
10
```