Concise Animation Language (CAL) Final Report

Authors
Tianliang Sun, ts2825
Xinan Xu, xx2153
Jingyi Guo, jg3421

Supervisor
Prof. Stephen A. Edwards

Teaching Assistant
Mengqi Zhang

Summary
The final project report for Concise Animation Language (CAL).

Revision:
Last updated: 12/20/2013 11:19:00 PM

1. Table of Contents

2. Introduction 1
3. Language Tutorial 2
   3.1. Simple Examples 2
   3.2. Compiling and Running 2
4. CAL Language Reference Manual 4
   4.1. Lexical Convention 4
   4.2. Syntax Notation 5
   4.3. What's in a Name 5
   4.4. Objects and lvalues 6
   4.5. Conversions 6
   4.6. Expressions 6
   4.7. Declarations 10
   4.8. Statements 12
   4.9. Function Definitions 13
   4.10. Program Structure 14
   4.11. Scope Rules 14
   4.12. Types Revisited 14
   4.13. Syntax Summary 15
5. Project Plan 19
   5.1. Development Process 19
   5.2. Programming Style Guide 19
   5.3. Project Timeline 19
   5.4. Work Assignment 20
   5.5. Software Development Environment 20
   5.6. Project Log 20
6. Architectural Design 25
   6.1. Overview 25
   6.2. Interface design 26
7. Test Plan 27
   7.1. Overview 27
   7.2. Examples 27
   7.3. Unit Test (Scanner, parser) 33

7.4. Unit Test (bytecode) 34
8. Lessons Learned 36
9. Appendix 37
9.1. Concise Animation Language 37
9.2. Abstract Syntax Tree Test Suite 69
9.3. Bytecode Test Suite 74
9.4. Demo Program 80

2. Introduction

CAL is a compact programming language used to facilitate the process of making simple and common 2D animations. For users who just want to create a simple animation with basic elements such as points, lines, and shapes, current graphics programming interfaces like OpenGL often seems too complicated. To create animations with OpenGL, one has to set up the environment, write the customized display() function, compute the vertex coordinates, and do lots of other things, to just create a moving square. However, making animations will be much simpler with the help of CAL. To draw an element to screen, one just has to declare it, set a smaller number of properties, then pass it to the add() function. CAL will automatically handle all the preparation and animation calculations for you. Like other graphical interfaces, CAL gives users the ability to precisely draw elements on a window by positioning using 2D coordinates and coloring using RGB values. It also keeps users away from repetitive tasks, such as declaring or reformatting tens of rectangular shapes, with the help of the built in if/else and for loop control structures.

The syntax of CAL is C like, but it removes a lot of redundancies in C which will not be used for generating animations. CAL contains primitive types int, double, and string. It also includes built in data structures such as Point and Shape, all of which are frequently used in making animation. To define the contents of a particular animation, the user creates built-in data structures such as Shape and then sets their properties accordingly. Those properties typically include size, position, color, and if it is a shape, the number of polygon sides will be also included. The user defines animations and transitions by assign 2D velocities to those objects.

3. Language Tutorial

3.1. Simple Examples

A CAL program consists of statements and function definitions. A statement can be an assignments, an if statement, a for statement, a single break, or a return expression. A function definition begins with a type_specifier, such as int, a function name and a list of arguments and is followed by statements.

CAL supports int, string, list, point, shape and struct data types. The element of the list can be of any type. The following is an example showing some general features.

```
Struct foo{
  list<double> D;
  string S;
};
struct foos{
  foo F;
  list<double> D;
};
int main(){
  int i = 0;
  struct foos f;
  for(i = 0; i < 100; i++){
    if(i % 2)
      f.D[i]=f.F.D[i];
    else
      break;
  }
  return 0;
}
```

```
struct foo{
  list<point> points;
  string S;
};

int main(){
  int i = 0;
  struct foo f;
  for(i = 0; i < 100; i++){
    foo.shapes[i].x = i;
    foo.shapes[i].y = i;
    add_shape(foo.shapes[i]);
    wait(0.1);// sleep for 0.1
  }
  wait(5.0);// sleep for 5.0s
  byebye(); // Done!
  return 0;
}
```

```
int Fib(int n) {
  if(n == 0 || n == 1) {
    return n;
  }
  else {
    return fib(n - 1) + fib(n - 2);
  }
}
int main(){
  point x;
  int i;
  for(i = 0; i < 10; i++){
    x.x=fib(i);
    x.y=fib(i+1);
    add_point(x);
  }
  wait(10.0);
  return 0;
}
```

3.2. Compiling and Running

The compilation process is actually not simple, however, we have made a Makefile to simplify the compiling process. A quick compilation through Makefile is simply:

foo.c: make foo-cal

The compilation process is to completely compile and link the CAL language into executable file. To run it, use:

foo.c: ./foo-cal

A manual compilation includes preprocessing, compilation, postprocessing, assembling and linking:

Preprocessing:

```
cat gltypes.cal > foo.tmp0
cat foo.c >> foo.tmp0
```

Compilation:

```
./cal < foo.tmp0 > foo.tmp
```

Postprocessing:

```
grep "private constant" foo.tmp > foo.ll
grep -v "private constant" foo.tmp >> foo.ll
rm -f foo.tmp foo.tmp0
```

Assembling and Linking:

```
llvm-as foo.ll -o foo.bc
llc foo.bc -o foo.s
gcc -c glsupport.c -o gl.o
gcc foo.s gl.o -o foo -lglut -lpthread
```

4. CAL Language Reference Manual

4.1. Lexical Convention

A CAL program consists of a number of categories of tokens, including Comments, Identifiers, Reserved Keywords, Constants, Strings, Expression Operators, and other separators like some special punctuation. A whitespace of any form (tab, newline, single or multiple spaces) is meaningless but to separate tokens.
When parsing a CAL program, if the input stream has been parsed up to a given character c, the next token will be the longest string of characters starting from c that could be matched as a legitimate token.

4.1.1. Line Comment

A line of comment starts with // and ends with the \n character. All the text between // and the end of line will be ignored, including block comment tokens. The escape character does not have any effect in the comment line and will not make the line comment splitting into multiple lines.

4.1.2. Block Comment

A comment block starts with one /* and ends with the next following */. A comment block could be either a single-lined or a multiple-lined one. However, CAL does not support nested comments. With that being said, a /* will be paired with the first */ in the following input. So the following example will not be a correct comment:

```
/* This is a comment. /* Nested Comment? */ Not a comment anymore! */
```

Only "/* This is a comment. /* Nested Comment? */" will be considered as a comment. The rest part will possibly cause a parsing error. The escape character does not have any effect in the comment block.

4.1.3. Identifiers

An identifier is to represent a data in the program. An valid Identifier has the form of ['a-z','A-Z']['0-9','a-z','A-Z']{0,19}, which means it consists of a string of alphabetic letters and digits of length 1 to 20. It must starts with an alphabetic letter. It is also case-sensitive.

4.1.4. Reserved Keywords

CAL inherits the minimum set of reserved keywords from C to support graphics programming, which includes the keywords for primitive data types:

```
int
double
string
list
point
shape
struct
```

and keywords for control flow:

break
return
if
else
for

Note that we use string instead of char and char* in CAL. All those keywords are reserved and no other tokens should be identical to them.

4.1.5. Constants

There are several kinds of constants of follows:

Integer Constants:
An integer constant is a continuous sequence of numerical digits [‘0’-’9’]+. All numbers in CSL are in decimal representation.

String Constants:
An string constant is a sequence of characters enclosed by a pair of “ and ”. The characters in a string constant could be anything (e.g. letters, digits, punctuations) but certain characters have to be escaped by a preceding \. Those characters are \, “, and ”. When parsing a string and a \ is encountered, it will be used to escape the next following character. If the escaped character is \, “, or ”, then that character will appear in the string. If the next character is n, t, or r (i.e. a \n, \t, \r in the string), they will be considered as a newline, tab, or return, respectively.

Double Constants:
Those are floating point values. A double consists of a signed integer part, a decimal point, a fraction part, and an optional e and signed exponent. The first three parts must present. A regular expression for a double constant is [‘0’-‘9’]+‘.’[‘0’-‘9’]+(‘e’ ‘-’{0,1}[‘0’-‘9’]+){0,1}.

4.2. Syntax Notation

In syntax notations throughout this manual, syntax categories are distinguished by *Italic* type. Each alternative expansion is listed in a separate line. All the optional symbols are labeled with a subscript “$_{opt}$”.

4.3. What’s in a Name

Being different from C, an identifier has the form identifier_type identifier_name.
The type determines the meaning of the values found in the identifier’s storage. CAL does not have storage classes. All variables are visible to their containing functions and anything within the scopes of containing functions, and all variables are allocated on the stack.
There are six primitive types in CAL: int, double, string, point, shape, and image.
From the primitive types, one can construct more complex types like:

list: a list of variable length that contains variables of a same type
function: which will return a value of the given type
struct: a container for variables of different types

CAL supports recursive construction of list and struct, but not function.

4.4. Objects and lvalues

An object is a manipulative region of memory. An lvalue is an expression referring to an object. “lvalue” is shorthand for left hand side values. In an assignment expression “E1 = E2”, E1 must be a lvalue expression. Later in section “Expressions”, for every operator we discuss whether it expects operands of lvalue type, and whether it yields a result of lvalue type.

4.5. Conversions

CAL does not provide automatic conversions between types. The operands of an operator or the arguments to a function must match the expected types.

4.6. Expressions

Following the convention of traditional programming languages, expression operators are categorized into different subsections, each associated with precedence. Within each subsection, the operators have the same precedence. Left or right associativity is specified in each subsection for the operators discussed therein.

Precedence	Operator	Description	Associativity
1	<>	Specify list type	Left-to-right
	()	Function call	
	[]	Array subscripting	
	.	Element selection by reference	
2	++ --	Prefix increment and decrement	Right-to-left
3	−	Unary minus	

<table>
<tr><td></td><td>!</td><td>Logical NOT</td><td></td></tr>
<tr><td>4</td><td>* / %</td><td>Multiplication, division, and remainder</td><td>Left-to-right</td></tr>
<tr><td>5</td><td>+ −</td><td>Addition and subtraction</td><td></td></tr>
<tr><td>6</td><td>< <=</td><td>For relational operators < and ≤ respectively</td><td></td></tr>
<tr><td></td><td>> >=</td><td>For relational operators > and ≥ respectively</td><td></td></tr>
<tr><td>7</td><td>== !=</td><td>For relational = and ≠ respectively</td><td></td></tr>
<tr><td>8</td><td>&&</td><td>Logical AND</td><td></td></tr>
<tr><td>9</td><td>||</td><td>Logical OR</td><td></td></tr>
<tr><td>10</td><td>=</td><td>Direct assignment</td><td>Right-to-left</td></tr>
<tr><td></td><td>+= −=</td><td>Assignment by sum and difference</td><td></td></tr>
<tr><td></td><td>*= /= %=</td><td>Assignment by product, quotient, and remainder</td><td></td></tr>
</table>

4.6.1. *identifier* (*expression-list$_{opt}$*)

The identifier followed by a pair of parenthesis and an optional expression_list represents a function call to function identifier.

4.6.2. *constant*

An integer, string, double with a fixed value is a primary expression. The regular expression of those three types has been defined in the previous sections.

4.6.3. *lvalue*

As discussed before, an lvalue is a special type of expression that can be assigned with some values.

4.6.4. *identifier*
An identifier is the most common form of an lvalue. It is used to associate a constant value with a textual symbol.

4.6.5. *lvalue* [*expression*]
An lvalue followed by a pair of square brackets and an expression is used to address a specific element in a list. This expression is also an lvalue.

4.6.6. *lvalue . identifie*r
An *lvalue* followed by an identifier represents a member named identifier in struct lvalue. This expression itself is also an lvalue.

4.6.7. Unary operations

− *expression*

The result is the negation of the expression, and has the same type. The type of the expression must be int or double.

! expression

The result of the logical negation operator ! is 1 if the value of the expression is 0, and 0 otherwise. The type of the expression must be int or double, and the return type will be the same type.

++ lvalue

The object referred to by the lvalue expression is incremented. The value is the new value of the lvalue expression and the type is the type of the lvalue. The type of the expression is int, and it increments by 1.

-- *lvalue*

The object referred to by the lvalue expression is decreased. The value is the new value of the lvalue expression and the type is the type of the lvalue. The type of the expression is int, and it decreases by 1.

4.6.8. Multiplicative operators
The multiplicative operators *, /, and % group left-to-right.

*expression * expression*

The binary * operator indicates multiplication. If both operands are int, the result is int. If both are double, the result will be double. Type exception will be thrown if none of the two conditions are met.

expression / expression

The binary / operator indicates division. The same type considerations as for multiplication apply.

expression % expression

The binary % operator yields the remainder from the division of the first expression by the second. Both operands must be int and the result will be int. The remainder has the same sign as the dividend.

4.6.9. Additive operators

expression + expression

The result is the sum of the expressions. If both operands are int, the result is int. If both are double, the result will be double. If both operands are string, the result, which returns another copy, is the concatenation of both strings.

expression – expression

The binary - indicates minus. If both operands are int, the result is int. If both are double, the result will be double.

4.6.10. Relational operators

expression < expression
expression > expression
expression <= expression
expression >= expression

The operators < (less than), > (greater than), <= (less than or equal to) and >= (greater than or equal to) all yield 0 if the specified relation is false and 1 if it is true. Both operands must be the same type and they will be either int or double.

4.6.11. Equality operators

expression == expression
expression != expression

The == (equal to) and the != (not equal to) operators are exactly analogous to the relational operators.

expression && expression

The && operator returns 1 if both its operands are non-zero, 0 otherwise. Moreover, the second operand is not evaluated if the first operand is 0. Either side accepts int only.

expression || expression

The || operator returns 1 if either of its operands is non-zero, and 0 otherwise. Moreover, the second operand is not evaluated if the value of the first operand is non-zero. Either sides accept int only.

4.6.12. Assignment operators

lvalue = expression

The value of the expression replaces that of the object referred to by the lvalue. The operands need to have the same type.

lvalue += expression
lvalue -= expression
*lvalue *= expression*
lvalue /= expression
lvalue %= expression

The behavior of an expression of the form ‘E1 op=E2’ is equivalent to ‘E1=E1 op E2’

4.6.13. (*expression*)
A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression.

4.6.14. *expression-list*
An expression-list takes the form:

expression-list:
expression
expression-list , *expression*

An expression-list itself is generally not considered as an expression, but a list of expressions. It is used in function calls as function arguments.

4.7. Declarations

Declarations in CAL appear at the beginning of the program as global declarations and at the beginning of a function body as function local variables. They are to tell CAL how it should allocate and possibly initialize the identifiers associated with them. The declaration discussed here are only data declarations. Function declarations are specified later in this document.

Declarations have the form:

declaration:
    type-specifier declarator-list$_{opt}$;
    struct *identifier* { *struct-declaration-list* } ;

struct-declaration-list:
    struct-declaration
    struct-declaration-list struct-declaration

struct-declaration:
    type-specifier identifier ;

4.7.1. Type specifiers

Type specifiers in CAL have the form:

type-specifier:
    int
    double
    string
    list<type-specifier>
    point
    shape
    image
    struct identifier

4.7.2. Declarators

Slightly different from the syntax in raw C, CAL allows declaring and initializing an identifier at the same time, and it is strongly suggested to initialize a variable upon declaration.

declarator-list:
    declarator
    declarator-list, declarator

declarator:
    identifier initializer$_{opt}$

initializer:
    = constant-expression
    = { constant-expression-list }

constant-expression:
constant
- constant

constant-expression-list:
constant
constant-expression-list , *constant*

4.7.3. Structure Declarations

Structure declarations have two forms as noted previously:

struct identifier { struct-declaration-list }
struct identifier

When declaring a structure for the first time, always use the first form to define the data layout in this structure. Then the subsequent declarations of this structure must use the second form.

4.8. Statements

4.8.1. Expression Statement

The basic form of a statement is an expression followed by a semicolon:

statement:
expression ;

Assignments and function calls are most commonly used expression statement.

4.8.2. Statement group

A group of statements can appear wherever a single one is allowed and it has the form:

statement-group:
{ statement-list }

statement-list:
statement
statement statement-list

4.8.3. Conditional statement

The two forms of conditional statement are:

if (expression) statement
if (expression) statement else statement

First the expression is evaluated, if it is non-zero, the first statement is executed. In the second situation, if the expression is zero, then the statement after else is executed. In addition, the “else” ambiguity is resolved by connecting an “else” with the nearest elseless “if”.

4.8.4. For statement

For statement has the form:

for ($expression_{opt}$; $expression_{opt}$; $expression_{opt}$) *statement*

The first expression initializes the for loop; it's executed once when the loop begins; the second expression makes a test before each iteration and the loop is exited when the expression evaluates to zero; the last expression is invoked after each iteration through the loop.

4.8.5. Break statement

The statement

break ;

terminate the current for loop and control passes to the statement following the terminated statement.

4.8.6. Return statement

A function returns to its caller by means of return statement as follows:

return expression ;

when it is used, it will return to its caller with one or multiple values which could have different types.

4.9. Function Definitions

Function definitions must take the form:

function-definition:
type-specifier function-declarator function-body

However, unlike C which does not allow functions to return structures, CAL does allow functions returning structures.

The function declarator takes the form:

function-declarator:
identifier ($parameter\text{-}list_{opt}$)

parameter-list:

function-parameter
function-parameter , *parameter-list*

function-parameter:
type-specifier identifier

CAL does not allow passing functions as parameters to a function.
And the body of a function takes the form:

function-body:
{ *declaration-list*$_{opt}$ *statement-list* }

4.10. Program Structure

A CAL program always has this form:

program:
declaration-list$_{opt}$ *function-definition-list*

declaration-list:
declaration
declaration-list declaration

function-definition-list:
function-definition
function-definition-list function-definition

4.11. Scope Rules

4.11.1. Variables

A variable is visible and accessible within the body of the function it is declared in (a.k.a the containing function), and only accessible by the code after its declaration.

4.11.2. Functions

Functions must be defined at the outermost scope in a program, and therefore nested function declaration is not supported in CAL. Functions can be accessed globally within the program it is declared in, but only after its declaration.

4.11.3. Structures

Structures must be defined at the outermost scope in a program. Although a structure may recursively contains a structure, but the definition of the contained structure must be again at the outermost scope in a program.

4.12. Types Revisited

This section summarizes the operations that can be done on a given type.

4.12.1. Structs

Use . to select a member in the structure. Structures can also be passed as function parameters. Structures cannot be directly assigned since it is not an lvalue type.

4.12.2. Point, Shape

These types are actually predefined structures in CAL. So the only difference between these two and other structures is that there is no need to use the first form of structure declaration

struct identifier { *struct-declaration-list* }

when declaring them, because their data layout have been specified already. Even the data members in those structures are initialized already, but those initial values are usually useless. Use the predefined initialization function to adjust those values.

4.12.3. Lists

List is also similar to a vector in C. One operation that can be performed on a list is to use [*expression*] to access the element at the offset specified by *expression*.

4.12.4. Functions

The only operation that is available to a function is to call it. A function can neither be passed as a parameter, nor directly assigned.

4.13. Syntax Summary

expression:

identifier (*expression-list*$_{opt}$)
constant
lvalue
− *expression*
! *expression*
++ *lvalue*
-- *lvalue*
expression binop expression
lvalue asgop expression
(*expression*)

lvalue:

identifier
lvalue [*expression*]
lvalue . *identifier*

expression-list:
expression
expression-list , *expression*

binop:
*
/
%
+
-
<
>
<=
>=
==
!=
&&
||

asgop:
=
+=
-=
*=
/=
%=

statement:
expression ;
{ *statement-list* }
if (*expression*) *statement*
if (*expression*) *statement* else *statement*
for (*expression*$_{opt}$; *expression*$_{opt}$; *expression*$_{opt}$) *statement*
break ;
return *expression* ;

statement-list:
statement
statement-list statement

declaration-list:
declaration
declaration-list declaration

declaration:
type-specifier declarator-list ;

struct *identifier* { *struct-declaration-list* }

struct-declaration-list:
struct-declaration
struct-declaration-list struct-declaration

struct-declaration:
type-specifier identifier ;

type-specifier:
int
double
string
list < *type-specifier* >
point
shape
struct *identifier*

declarator-list:
declarator
declarator-list , *declarator*

declarator:
identifier initializer$_{opt}$

initializer:
= *constant-expression*
= { *constant-expression-list* }

constant_expression:
constant
- *constant*

constant-expression-list:
constant-expression
constant-expression-list , *constant-expression*

constant:
const_int
const_dbl
const_str

function-definition:
type-specifier function-declarator function-body

function-declarator:

identifier (*parameter-list$_{opt}$*)

parameter-list:
function-parameter
function-parameter , *parameter-list*

function-parameter:
type-specifier identifier

function-body:
{ *declaration-list$_{opt}$ statement-list* }

program:
declaration-list$_{opt}$ function-definition-list

declaration-list:
declaration
declaration-list declaration

function-definition-list:
function-definition
function-definition-list

5. Project Plan

5.1. Development Process

Various methods were used to facilitate the communication and discussion between group members. The group members hold regular meetings before and after class to discuss the progress and the problems encountered by each team member. Emails were sent out to keep the other members with the most updated information. We use Google Docs to share and edit project documents, as well as the schedule and milestones. Our Git repository is hosted on Google Code. Through this way, we guaranteed every group member could share code and check each other's progress now and then.

While developing the parser, scanner, and abstract syntax tree, we used our CAL Language Reference Manual as the sole source of specification. For LLVM byte code development, we referred to LLVM's Language Reference Manual as the specification. And for implementing the OpenGL support library, we consulted OpenGL library's API as the specification.

Our project development was separated into three major phases: parsing and syntax tree generation, semantics checks, and byte code generation. We test our code consistently throughout the development. A regression test suite including both unit test cases and integration test cases were built, and the unit tests are automated by shell script.

5.2. Programming Style Guide

We followed the standard coding style as follows:

i. Use 4 spaces as one level of indentation.
ii. Indent each level of nested statements and definitions.
iii. Truncate long source lines into multiple lines.
iv. Name variables/functions with something meaningful, instead of a,a1, or i.
v. Break long functions into smaller ones.
vi. Break a large module into smaller ones.
vii. Follow the C naming convention (lowercase letters and underscores) to name variables and functions, instead of the Java convention (camel cases).
viii. For readability purpose, use parentheses to group statements and expressions even if when they are necessary.

5.3. Project Timeline

10.18.2013: Project repo created
10.20.2013: Scanner draft
10.28.2013: LRM finished
11.11.2013: Fixes in scanner and update the operator precedence
11.20.2013: Parser draft for statements and expressions
12.04.2013: Print_debug for testing scanner and parser

12.07.2013: Print_debug passed all parsing tests
12.13.2013: Semantic checks implemented
12.15.2013: Bytecode generation implemented
12.17.2013: OpenGL interface implemented

5.4. Work Assignment

Tianliang Sun: Implementation of parser, abstract syntax tree, print debugger, semantics check and OpenGL interface.

Xinan Xu: Implementation of scanner, bytecode generation and bytecode testing.

Jingyi Guo: Implementation of scanner, parser, ast and testing.

5.5. Software Development Environment

The software is developed using several different languages and tools.

Languages we used include:

1. Ocaml: for the core part of the compiler, used for parsing and bytecode generation
2. C: for the OpenGL interface and wrapper of external function calls

Tools we used include:

1. llvm-as: LLVM assembler which converts the bytecode output to LLVM bitcode
2. llc: LLVM native compiler to compile the LLVM bitcode to native assembly code
3. gcc: linker for linking object files and assembly codes
4. grep: preprocessor and postprocessor for bytecode generation

5.6. Project Log

```
* 8821cf1 - shane -2013-12-20  warnings suppression
* e20215b - shane -2013-12-20  fixs
* 20db754 - Xinan Xu -2013-12-20  fixs
* 00096e0 - Xinan Xu -2013-12-20  more fancy
* 209b50e - Tianliang -2013-12-20  now can automatically add setup() and run(); updated test.c
* 61e36e2 - Tianliang -2013-12-20  now can automatically add setup() and run()
* f3d9833 - Xinan Xu -2013-12-20  fixs to type conversion
* a565b98 - Tianliang -2013-12-20  updated test.c
* 0c2a89f - Xinan Xu -2013-12-20  Implicit type conversion
*   b2a98e6 - Tianliang -2013-12-20  Merge branch 'master' of https://code.google.com/p/w4115-project
|\
| * 8e4f269 - Xinan Xu -2013-12-20  remove intermediate files
* | 24e8f86 - Tianliang -2013-12-20  weird recursive behavior
|/
* 20a0386 - Xinan Xu -2013-12-20  animation
*   8b34cb3 - Xinan Xu -2013-12-20  Merge branch 'master' of https://code.google.com/p/w4115-project
|\
| * 8913242 - Tianliang -2013-12-20  binop type check logic reworked
```

```
| *   4f15f1f - Tianliang -2013-12-20  Merge branch 'master' of https://code.google.com/p/w4115-project
| |\
| * | ae33d02 - Tianliang -2013-12-20  binop type check logic reworked
* | | b10fc20 - Xinan Xu -2013-12-20  animation...
| |/
|/|
* | 72f2da5 - Xinan Xu -2013-12-20  color added
|/
* b1bf243 - Tianliang -2013-12-20  now can use point/shape instead of struct point/shape, also updated test.c and gltypes.h
* 2498b48 - Xinan Xu -2013-12-20  remove machine dependency
* f436da2 - Xinan Xu -2013-12-19  bug fix
* 061a5fa - Xinan Xu -2013-12-19  gltypes.cal
* 170c6e0 - Xinan Xu -2013-12-19  fixs
* 49376b7 - Xinan Xu -2013-12-19  llvm 2.9 also works
*   7bab912 - Xinan Xu -2013-12-19  Merge branch 'master' of https://code.google.com/p/w4115-project
|\
| * 29405a6 - Tianliang -2013-12-19  more detailed error messages in interpreter
* | ed84365 - Xinan Xu -2013-12-19  Initial rungit status! make test-cal
|/
* 20c80b0 - Tianliang -2013-12-19  fix: for statement type check
* 2f5aa18 - Tianliang -2013-12-19  scanner: definition of double constant fixed
* 58d7c34 - Xinan Xu -2013-12-19  parsing error
* 0d4304b - Xinan Xu -2013-12-19  fixs
* c5ae509 - Xinan Xu -2013-12-19  support for struct init
* 55eb96c - Tianliang -2013-12-19  updated gl files, create sample.c for final test
* 5143a9a - Tianliang -2013-12-19  now shape is defined as a square, can declare it with shape s = {double,double,double}
* f63d8aa - Tianliang -2013-12-19  extern functions now in global symbol table
* 78b0498 - Tianliang -2013-12-19  return types in glsupport.c changed to int
* 495343a - Xinan Xu -2013-12-19  parsing error
* 74fdf0e - Xinan Xu -2013-12-19  fac rec functions good
* 59d44f0 - Xinan Xu -2013-12-19  basic10,11.c: list passed
* e4f90b9 - Xinan Xu -2013-12-19  || operator support
* bdd79c2 - Xinan Xu -2013-12-19  basic9:if,for passed
* f0d30d5 - Tianliang -2013-12-19  now can initialize point with = { double, double }
* fb5ef05 - Tianliang -2013-12-19  now can initialize point with = { double, double }
*   61776ec - Tianliang -2013-12-19  Merge branch 'master' of https://code.google.com/p/w4115-project
|\
| *   42612fb - Xinan Xu -2013-12-19  Merge branch 'master' of https://code.google.com/p/w4115-project
| |\
| * | c8a4e27 - Xinan Xu -2013-12-19  basic6.c:struct probably right
* | | 11d40e0 - Tianliang -2013-12-19  now can initialize struct members with = { , , , }
| |/
|/|
* |   bcb418d - Tianliang -2013-12-18  Merge branch 'master' of https://code.google.com/p/w4115-project
|\ \
| |/
| * 2d9c3c3 - Xinan Xu -2013-12-18  basic6.c:string =, += passed
| * d45fe0c - Xinan Xu -2013-12-18  basic5.c : string test passed
| * 3986dc3 - Xinan Xu -2013-12-18  basic4.c : local variable intialized
| * c4364b7 - Xinan Xu -2013-12-18  basic3.c passed, non-initialized global variable
| * 6709cf0 - Xinan Xu -2013-12-18  basic2.c passed
| * a405695 - Xinan Xu -2013-12-18  basic.c passed; Modification to Makefile
| * 93ea06a - Xinan Xu -2013-12-18  struct +=
* | 3f110bc - Tianliang -2013-12-18  new gl interface
```

```
|/
* 181a03a - Tianliang -2013-12-18  opengl testcode
* d722ece - Xinan Xu -2013-12-18  Struct half done
* 3e596ec - Xinan Xu -2013-12-17  struct example by llvm
* 96cdf5e - Xinan Xu -2013-12-17  struct example
* 76b5dae - Tianliang -2013-12-16  binop/asgop precedence fix
* 18f2856 - Xinan Xu -2013-12-16  Still not right precedence
* f5d6deb - Xinan Xu -2013-12-16  Remove temporary files
* a495510 - Tianliang -2013-12-16  binop/asgop precedence fix
* 81238ae - Tianliang -2013-12-16  dependency fix
* 3a2b37a - Tianliang -2013-12-16  added new function sequency
* 68f2008 - Tianliang -2013-12-16  all util functions moved to utils.ml; added new function sizeof
* f6eb7b4 - Tianliang -2013-12-16  fix: struct decl ;
*   ccc984a - Xinan Xu -2013-12-16  Merge branch 'master' of https://code.google.com/p/w4115-project
|\
| * 0437dbb - Xinan Xu -2013-12-16  Nits
| * 6c4b7fd - Xinan Xu -2013-12-16  String added
| * 838bc7a - Tianliang -2013-12-16  += supports string type
| * 12f12e8 - Xinan Xu -2013-12-16  Nits.
| * a63fc9e - Xinan Xu -2013-12-16  udpate to Makefile
| * 8447884 - Xinan Xu -2013-12-15  function call added
| * bf9d132 - Xinan Xu -2013-12-15  break statement; example update
| * 49c4126 - Xinan Xu -2013-12-15  update the example
| * e7bb2ba - Xinan Xu -2013-12-15  add ifthen/ifthenelse
* | e00cb30 - Xinan Xu -2013-12-16  Nits.
|/
* d1a9288 - Xinan Xu -2013-12-15  sequential statements and expression finish
* c5d1a0f - Xinan Xu -2013-12-15  update example
* 64c879c - Xinan Xu -2013-12-15  half done
* b273b35 - Xinan Xu -2013-12-15  Nits
* 5a06045 - Xinan Xu -2013-12-15  update the example bytecode
* 808962f - Xinan Xu -2013-12-15  update the example
*   aeea880 - Xinan Xu -2013-12-15  Merge branch 'master' of https://code.google.com/p/w4115-project
|\
| *   02dfe5a - Tianliang -2013-12-15  Merge branch 'master' of https://code.google.com/p/w4115-project
| |\
| * | cb85596 - Tianliang -2013-12-15  semantics check: main defined?
* | | 3d89ffb - Xinan Xu -2013-12-15  Modification to Makefile to support multiple scanner; Add lexer string2string
to convert a C string into llvm type string
| |/
|/|
* |   bce5069 - Xinan Xu -2013-12-15  Merge branch 'master' of https://code.google.com/p/w4115-project
|\ \
| |/
| * 500c4a1 - Tianliang -2013-12-14  fix: const_expr_list
* | aa0fb14 - Xinan Xu -2013-12-15  Fixes to Makefile, probably solving the building problem; Stack allocation
complete
|/
* 3121d86 - Tianliang -2013-12-14  several order fixes: declaration list, declarator list, stmt list ...
*   89b92d6 - Tianliang -2013-12-14  function param list order fixed
|\
| * 998d193 - Xinan Xu -2013-12-14  All right, so far so good
* | 1c08b32 - Tianliang -2013-12-14  function param list order fixed
* |   b16f54a - Tianliang -2013-12-14  Merge branch 'master' of https://code.google.com/p/w4115-project
|\ \
| |/
```

```
| * fdecc2b - Xinan Xu -2013-12-14  function definition implementing. Parsing error for basic.c
| * 1c69414 - Xinan Xu -2013-12-14  update the example
| * 1fe15c4 - jingyiguo -2013-12-14  add
| *   fde1d50 - Xinan Xu -2013-12-14  Merge branch 'master' of https://code.google.com/p/w4115-project
| |\
| * | f2c6a7c - Xinan Xu -2013-12-14  finish type declaration
* | |   72bde8b - Tianliang -2013-12-14  Merge branch 'master' of https://code.google.com/p/w4115-project
|\ \ \
| | |/
| |/|
| * | 62f2c82 - jingyiguo -2013-12-14  add more example
* | | 8bd04db - Tianliang -2013-12-14  modified debug_print
|/ /
* | ea4c3a3 - Tianliang -2013-12-14  more semantics checks
|/
* e4fbc8c - Tianliang -2013-12-14  change: constant_expr: constant | -constant
* 2274090 - Tianliang -2013-12-14  parser test files moved to /ast_test
* 6bbb5c0 - Tianliang -2013-12-14  sast completed and merged to root folder
* 22bbe8f - Tianliang -2013-12-14  sast completed and merged to root folder
* c31df10 - Xinan Xu -2013-12-14  buggy case
* f318d42 - Tianliang -2013-12-14  fix: < and > problem
* 3e43fe6 - Tianliang -2013-12-14  fix: < and > problem
*   6d2236d - Tianliang -2013-12-14  Merge branch 'master' of https://code.google.com/p/w4115-project
|\
| * 0ef4abf - Xinan Xu -2013-12-13  global declaration, not complete, waiting for Tianliang
| * 6218445 - Xinan Xu -2013-12-13  adding basic bytecode
* | da8a352 - Tianliang -2013-12-14  semantics checks implemented
|/
* 8a7105d - Tianliang -2013-12-13  files moved to project root dir
* 3cd48bd - Tianliang -2013-12-13  files moved to project root dir
* 16b93e0 - Tianliang -2013-12-13  files moved to project root dir
* ceb59a3 - Tianliang -2013-12-13  files moved to project root dir
*   211c41d - Tianliang -2013-12-13  Merge branch 'master' of https://code.google.com/p/w4115-project
|\
| * 7b65050 - Xinan Xu -2013-12-13  Added Prolog
| * 2f54e17 - Xinan Xu -2013-12-13  buggy case--basic.c
| * 054f526 - Xinan Xu -2013-12-13  Adding llvm-2.7 building script, Makefiles, and 1 test case
| * fd1b217 - Xinan Xu -2013-12-13  remove previous test case
| * a33a5e4 - Xinan Xu -2013-12-13  Copying test program to root folder, again
* | e725619 - Tianliang -2013-12-13  fix: initiailizer: constant_expr | constant_expr_list
|/
* 8538d5f - Tianliang -2013-12-12  fixes: expr(...) added, (expr) added, (lval) removed, MINUS expr assoc adjusted
* 843b1ee - jingyiguo -2013-12-11  add error case
* 7e2c4ff - jingyiguo -2013-12-11  add testcase
* 0900fbb - Xinan Xu -2013-12-11  Copying the test project into root folder
* 380e403 - Xinan Xu -2013-12-10  string constant scanner
* d68bfb2 - Tianliang -2013-12-07  sample test cases by tianliang
* 2027b28 - Tianliang -2013-12-07  added a simple script for ast testing
* 99e8586 - Tianliang -2013-12-07  ast_test completed for testing
* f509711 - Tianliang -2013-12-05  ast_test: parser with no conflicts
* 73c478e - Tianliang -2013-12-04  parser rules all implemented
* 81f83a8 - Student -2013-11-20  tianliang: parser for stmt and expr
* d3b2281 - Xinan Xu -2013-11-19  remove expr_struct, adding lvalue
* fbd291e - Xinan Xu -2013-11-19  updates for expr and stmt; declaration and definition unmodified
* e6d2b81 - jingyiguo -2013-11-18  Add in parser and modify few in scanner
* d3b6b49 - Xinan Xu -2013-11-11  Fix in scanner, update parser's sequency
```

* d48c24b - Xinan Xu -2013-11-11 scanner complete, not sure escape character will work or not
* e7fe302 - Xinan Xu -2013-10-20 some fixes to Makefile
* c780b25 - Xinan Xu -2013-10-20 simple scanner
* a256b3c - Xinan Xu -2013-10-20 initial paser definition, not complete
* 5a25b29 - Xinan Xu -2013-10-20 helloworld test file, the result should be like helloworld.png
* 7d3fb6d - Xinan Xu -2013-10-20 Makefile works for microc...
* 01e37f2 - Xinan Xu -2013-10-18 Nits
* 65027c8 - Xinan Xu -2013-10-18 makefile almost finish
* 046dbba - Xinan Xu -2013-10-18 Makefile half done
* 28851df - Student -2013-10-10 added readme
* 66c044b - Student -2013-10-09 initial commit

6. Architectural Design

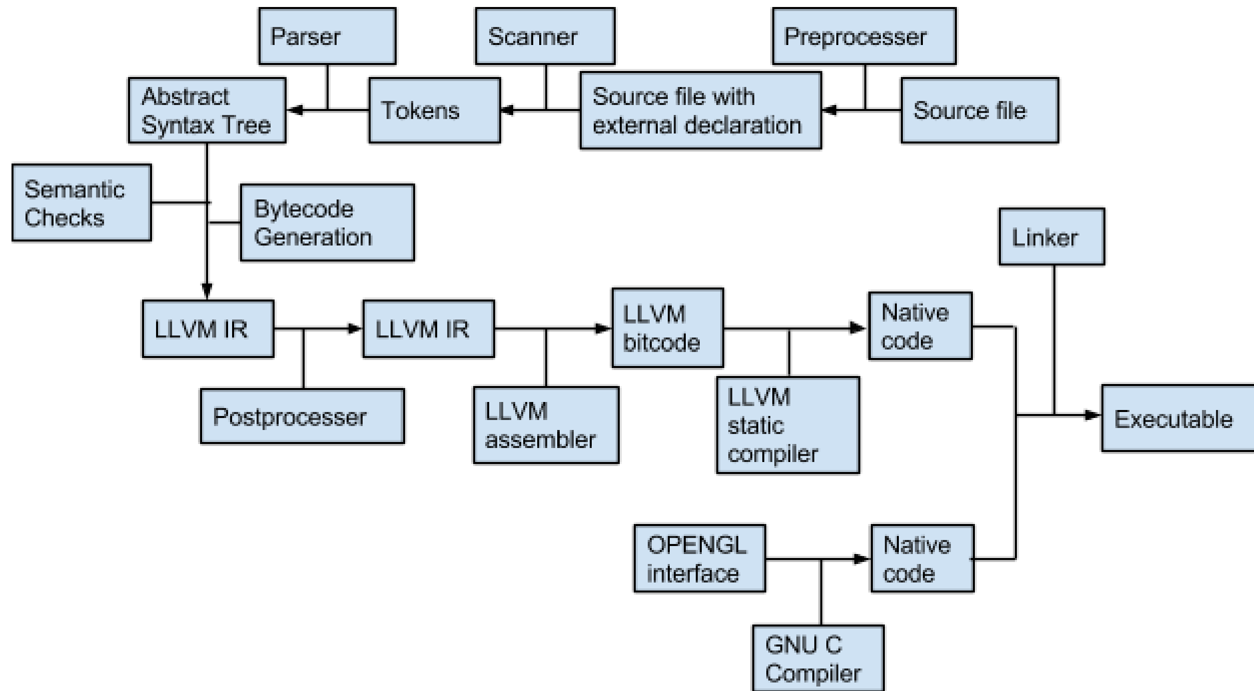

Figure 1. Architectural Design.

6.1. Overview

The compilation process is a little bit complicated, however, very simple to explain. The main components in our compiler are:

i. Preprocessor: to include the external declaration of Opengl interfaces and Built-in structures, this is a cheap and simple way to realize
ii. Scanner: to scan the source files to generate token lists, using Ocaml lexer(Xinan Xu, Jingyi Guo)
iii. Parser: to parse the tokens into abstract syntax tree, using Ocaml yacc (Tianliang Sun, Jingyi Guo)
iv. Semantic checks: Semantic checks for syntax errors, including type mismatches, undeclared variables (Tianliang Sun)
v. Bytecode Generation: to generate LLVM IR from the abstract syntax tree and a symbol table (Xinan Xu)
vi. Postprocessor: to move some constant value (to be more specifically, String constant) to the beginning of the LLVM IR (Xinan Xu)
vii. LLVM assembler: to convert the human-readable IR to LLVM bitcode
viii. LLVM static compiler: the backend of our compiler, which compile the bitcode into native code

ix. OPENGL interface: to define built-in functions to be called for Opengl rendering (Xinan Xu, Tianliang Sun)
x. Linker: gcc

6.2. Interface design

6.2.1. Scanner -> Parser

The scanner defines the set of terminal tokens that will be used by the parser. Those terminals include language reserved keywords, punctuations, arithmetic and logical operators, and constants. At the end, the parser will generate an abstract syntax tree.

6.2.2. Parser -> Semantic checks -> Program data

After the parser has finished parsing the input CAL source, it will pass the generated abstract syntax tree to the semantics checker. The checker will then perform a series of checks, including type consistency checks and symbol declaration checks on the syntax tree. Once it has finished, the checker will generate a data structure named “program data”. Program data includes a global symbol table and a list of declared functions, and pass it to the bytecode generator.

6.2.3. Program data -> bytecode

Bytecode generation relies on the structure of the abstract syntax tree and the symbol tables for both global and functional scales.

6.2.4. Declaration sharing

The external OpenGL rendering implementation is sharing the same head file with the preprocessor of CAL compiler. point and shape structure is predefined and will be used by both languages. Functions for the interface are defined in OpenGL rendering source file and declared in the bytecode.

7. Test Plan

7.1. Overview

We test our source programs consistently throughout the development process. We used both white box and black box testing methods and built up a regression test suite consists of many unit test cases and integration test cases. Please refer to Section 8 for a complete listing with source code and test cases.

7.2. Examples

7.2.1. String Utilities

```
int a=200;
double b;
int c;
double d=10.01;
int main(){
        string s="\n\t\\";
        s+="aaaa";
        s="bbbb";
        return 0;
}
```

```
@.str.main.s = private constant [100 x i8]
c"\0A\09\5C\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\0
0\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\
00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00", align 1
@.str.main.1 = private constant [100 x i8]
c"aaaa\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\0
0\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\
00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00", align 1
@.str.main.3 = private constant [100 x i8]
c"bbbb\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\0
0\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\
00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00", align 1
@a = global i32 200
@b = common global double 0.0e+00
@c = common global i32 0
@d = global double 10.01
%struct.point = type {  double, double, double, double, double, double, double}
%struct.shape = type {  double, double, double, double, double, double, double, double, double, double}

define i32 @main() nounwind {
entry:
        %retval = alloca i32
        %s0 = alloca [100 x i8]
        %s = bitcast [100 x i8]* %s0 to i8*
        call void @llvm.memcpy.i64(i8* %s, i8* getelementptr inbounds ([100 x i8]* @.str.main.s, i64 0, i64
0), i64 100, i32 1)
        %reg1 = call i32 @setup(  ) nounwind
        %reg2 = bitcast [100 x i8]* %s0 to i8*
```

```
        %reg3 = call i8* @strcat(i8* noalias %reg2, i8* noalias getelementptr inbounds ([100 x i8]*
@.str.main.1, i64 0, i64 0)) nounwind
        %reg4 = bitcast [100 x i8]* %s0 to i8*
        %reg5 = call i8* @strcpy(i8* noalias %reg4, i8* noalias getelementptr inbounds ([100 x i8]*
@.str.main.3, i64 0, i64 0)) nounwind
        %reg6 = call i32 @byebye( ) nounwind
        store i32 0, i32* %retval, align 1
        br label %return

return:
        %retval4 = load i32* %retval
        ret i32 %retval4
}
```

7.2.2. Control Flow

```
int main(){
        int i = 10;
        int sum = 0;
        for(i = 1; i < 10; i++) {
                if(sum > 10)
                        return sum;
                sum+=i;
        }
        return sum;
}
```

```
%struct.point = type {  double, double, double, double, double, double, double}
%struct.shape = type {  double, double, double, double, double, double, double, double, double, double}

define i32 @main() nounwind {
entry:
        %retval = alloca i32
        %i = alloca i32
        store i32 10, i32* %i, align 1
        %sum = alloca i32
        store i32 0, i32* %sum, align 1
        %reg1 = call i32 @setup( ) nounwind
        store i32 1, i32* %i, align 1

        br label %v2bb0forentry
v2bb0forentry:
        %reg2 = load i32* %i, align 1
        %reg3 = icmp slt i32 %reg2, 10
        %reg4 = zext i1 %reg3 to i32
        %reg5 = trunc i32 %reg4 to i1
        br i1 %reg5, label %v5bb0, label %v5bb1

v5bb0:
        %reg6 = load i32* %sum, align 1
        %reg7 = icmp sgt i32 %reg6, 10
```

```
            %reg8 = zext i1 %reg7 to i32
            %reg9 = trunc i32 %reg8 to i1
            br i1 %reg9, label %v9bb0, label %v9bb1

v9bb0:
            %reg10 = load i32* %sum, align 1
            store i32 %reg10, i32* %retval, align 1
            br label %return

v9bb1:
            %reg11 = load i32* %sum, align 1
            %reg12 = load i32* %i, align 1
            %reg13 = add nsw i32 %reg11, %reg12
            store i32 %reg13, i32* %sum, align 1
            %reg14 = load i32* %i, align 1
            %reg15 = add nsw i32 %reg14, 1
            store i32 %reg15, i32* %i, align 1
            br label %v2bb0forentry

v5bb1:
            %reg16 = call i32 @byebye( ) nounwind
            %reg17 = load i32* %sum, align 1
            store i32 %reg17, i32* %retval, align 1
            br label %return

return:
            %retval4 = load i32* %retval
            ret i32 %retval4
}
```

7.2.3. Struct and List dereference

```
struct foo {
            int a;
            double b;
            list<int> c;
};

struct foofoo{
            int a;
            double b;
            struct foo c;
};
int main(){
            struct foo x;
            int c = 10;
            list<int> y;
            x.b=1.0;
            x.c[10] = 10;
            y[c] = 100;
            return c;
}
```

```
%struct.point = type {  double, double, double, double, double, double, double}
%struct.shape = type {  double, double, double, double, double, double, double, double, double, double}
%struct.foo = type {  i32, double, [100 x i32]}
%struct.foofoo = type {  i32, double, %struct.foo}

define i32 @main() nounwind {
entry:
        %retval = alloca i32
        %x = alloca %struct.foo
        %c = alloca i32
        store i32 10, i32* %c, align 1
        %y = alloca [100 x i32]
        %reg1 = call i32 @setup(  ) nounwind
        %reg2 = getelementptr inbounds %struct.foo* %x, i32 0, i32 1
        store double 1., double* %reg2, align 1
        %reg3 = getelementptr inbounds %struct.foo* %x, i32 0, i32 2, i32 10
        store i32 10, i32* %reg3, align 1
        %reg4 = load i32* %c, align 1
        %reg5 = getelementptr inbounds [100 x i32]* %y, i32 0, i32 %reg4
        store i32 100, i32* %reg5, align 1
        %reg6 = call i32 @byebye(  ) nounwind
        %reg7 = load i32* %c, align 1
        store i32 %reg7, i32* %retval, align 1
        br label %return

return:
        %retval4 = load i32* %retval
        ret i32 %retval4
}
```

7.2.4. Struct function return and Struct function args

```
struct foo{
        int a;
        double b;
        string c;
};
struct foo func(struct foo x){
        return x;
}
int main(){
   struct foo foo2;
   struct foo foo1;
   foo1.b=foo2.b;
   return 0;
}
```

```
%struct.point = type {  double, double, double, double, double, double, double}
%struct.shape = type {  double, double, double, double, double, double, double, double, double, double}
```

```
%struct.foo = type {  i32, double, [100 x i8]}

define void @func(%struct.foo* noalias sret %agg.result, %struct.foo* byval %x) nounwind {
entry:
        %agg.result1 = bitcast %struct.foo* %agg.result to i8*
        %reg1 = bitcast %struct.foo* %x to i8*
        call  void @llvm.memcpy.i64(i8* %agg.result1, i8* %reg1, i64 116, i32 8)
        br label %return

return:
        ret void
}

define i32 @main() nounwind {
entry:
        %retval = alloca i32
        %foo2 = alloca %struct.foo
        %foo1 = alloca %struct.foo
        %reg1 = call i32 @setup(  ) nounwind
        %reg2 = getelementptr inbounds %struct.foo* %foo2, i32 0, i32 1
        %reg3 = load double* %reg2, align 1
        %reg4 = getelementptr inbounds %struct.foo* %foo1, i32 0, i32 1
        store double %reg3, double* %reg4, align 1
        %reg5 = call i32 @byebye(  ) nounwind
        store i32 0, i32* %retval, align 1
        br label %return

return:
        %retval4 = load i32* %retval
        ret i32 %retval4
}
```

7.2.5. Implicit type conversion and external function call

```
int main() {
        int i;
        point pt;
        for(i = 0; i < 10; i++){
                pt.x = i + 0.1;
                pt.y = i - 0.1;
                add_point(pt);
        }
        return 0;
}
```

```
%struct.point = type {  double, double, double, double, double, double, double}
%struct.shape = type {  double, double, double, double, double, double, double, double, double, double}

define i32 @main() nounwind {
entry:
        %retval = alloca i32
```

```
            %i = alloca i32
            %pt = alloca %struct.point
            %reg1 = call i32 @setup( ) nounwind
            store i32 0, i32* %i, align 1

            br label %v2bb0forentry
v2bb0forentry:
            %reg2 = load i32* %i, align 1
            %reg3 = icmp slt i32 %reg2, 10
            %reg4 = zext i1 %reg3 to i32
            %reg5 = trunc i32 %reg4 to i1
            br i1 %reg5, label %v5bb0, label %v5bb1

v5bb0:
            %reg6 = load i32* %i, align 1
            %reg7 = sitofp i32 %reg6 to double
            %reg8 = fadd double 0.1, %reg7
            %reg9 = getelementptr inbounds %struct.point* %pt, i32 0, i32 0
            store double %reg8, double* %reg9, align 1
            %reg10 = load i32* %i, align 1
            %reg11 = sitofp i32 %reg10 to double
            %reg12 = fsub double 0.1, %reg11
            %reg13 = getelementptr inbounds %struct.point* %pt, i32 0, i32 1
            store double %reg12, double* %reg13, align 1
            %reg14 = bitcast %struct.point* %pt to i8*
            %reg15 = bitcast i8* %reg14 to %struct.point*
            %reg16 = call i32 @add_point(%struct.point* byval %reg15) nounwind
            %reg17 = load i32* %i, align 1
            %reg18 = add nsw i32 %reg17, 1
            store i32 %reg18, i32* %i, align 1
            br label %v2bb0forentry

v5bb1:
            %reg19 = call i32 @byebye( ) nounwind
            store i32 0, i32* %retval, align 1
            br label %return

return:
            %retval4 = load i32* %retval
            ret i32 %retval4
}


declare void @llvm.memcpy.i64(i8* nocapture, i8* nocapture, i64, i32) nounwind
declare i8* @strcpy(i8* noalias, i8* noalias) nounwind
declare i8* @strcat(i8* noalias, i8* noalias) nounwind
declare i32 @add_shape(%struct.shape* byval)
declare i32 @add_point(%struct.point* byval)
declare i32 @setup()
declare i32 @run()
declare i32 @pop_point()
declare i32 @pop_shape()
declare i32 @wait(double)
declare i32 @byebye()
```

[

7.3. Unit Test (Scanner, parser), automated, by Tianliang and Jingyi

To test the correctness of parser and the syntax tree generator, we let the parser prints out the parsed symbols while traversing through the source code, so that we can use that output for debugging purposes.

More specifically, we used white box testing approach to create a unit test case to test each of the reduction rules in the parser. For each test case i, we create test_case_i.txt as the input to the parser, and test_result_i.txt as the expected result. We then used a shell script to automatically run those tests and use diff to compare the parser output against the expected outputs.

Table 3. shows the test case names and content.

File name	Test Content
test_case_0.txt	return
test_case_1.txt	assignment
test_case_2.txt	constant, string, int
test_case_3.txt	reserved key data type
test_case_4.txt	arithmetic
tes_case_5.txt	precedence and binop mathematical operation
test_case_6.txt	logic operator
test_case_7.txt	if and relational ==
test_case_8.txt	comparison
test_case_9.txt	urinary minus
test_case_10.txt	struct
test_case_11.txt	for
test_case_12.txt	function call
test_case_13.txt	passing data types to functions
test_case_14.txt	postfix increment and decrement
test_case_15.txt	recursion
test_case_16.txt	element selection by reference
test_case_17.txt	relational !=

test_case_18.txt	assignment by sum, difference, product, quotation and remainder
test_case_19.txt	comma
test_case_20.txt	[]

Table 3. Unit test cases for scanner and parser.

7.4. Unit Test (bytecode), manually, by Xinan Xu
When initially the bytecodes generated is not linked with OpenGL libraries, it is hard to print them on the screen. However, since the main function will be linked into __libc_start_main and print the result into $? in bash. We used this information to perform the unit test. The following components have been tested:

basic1.c	Function definition, return	The basic test for checking the function definition and return value allocation
basic2.c	Global variable definition without initializer	Check the syntax of global variable definition
basic3.c	Global variable definition with initializer	Check the syntax of global variable definition with initialized value
basic4.c	Local variable definition	Check the local allocation of local variables, memory store instruction for local variable defined with initial value
basic5.c	String definition	Check the string parsing of an constant string
basic6.c	String definition with escape characters String assignment and concatenation	Check the external call of strcpy and strcat
basic7.c	Struct definition and function with struct args and struct return	Check the type definition of struct and the syntax of functions whose arg or return type are structs
basic8.c	Nested struct definition	Check the complicated case for struct definition
basic9.c	For statement and if statement	Check the control flow basic blocks
basic10.c	List definition	Check the list type definition

basic11.c	List and struct dereference and assignment	Check whether getelementptr has successfully get the member(of member…) of one lvalue
basic12.c	Recursive function definition	Check whether the program runs with recursive function call
basic13.c	Struct and list Nested definition	Check the complicated case for struct definition
basic14.c	Struct definition with initializer	Check whether the initialization of a struct has been correctly executed
basic15.c	External function call	Check the interface of external function

8. Lessons Learned

Tianliang Sun:
I have never programed in a functional language so learning OCaml was a challenging experience to me. But it is interesting to see how OCaml combines functional programming and object-oriented programming together. I was responsible for writing the parser and syntax tree generator, and I really learned a lot about programming language designs by resolving the conflicts/bugs in the parser.
For a project of this size, we really should follow the standard software engineering approach (strict schedule, milestones, Gantt charts, etc/), but we didn't. As a result, there was really a rush during the week before the presentation, and some features we intended to build were not implemented.

Jingyi Guo:
I am getting familiar with Ocaml and impressed on its recursion function and pattern matching. I am also marvelous on the way of how parsing and Lambda. Through this course, I am getting to know the story behind every day coding. Teamwork is also essential to me and I learned a lot from the discussion among team members.

Xinan Xu:
It is a really amazing experience coding with a functional language. It made me more familiar with recursion function design and overall object-oriented design.
LLVM IR is another interesting language to study from. Clang is a good frontend to learn from. I learned how a real-world compiler do for typecasting, member pointer acquiring and basic block creation for control flow statements.

Advice for future group:
It is strongly recommended to learn Ocaml as early as possible. You need to follow a standard software engineering timeline to produce best results. And most importantly, Think before act!

9. Appendix

9.1. Concise Animation Language Source

9.1.1. Compiler Makefile (Xinan Xu)

```
OCAMLC = ocamlc
OCAMLLEX = ocamllex
OCAMLYACC = ocamlyacc
OCAMLOPT = ocamlopt
OCAMLDEP = ocamldep
OCAMLMKLIB = ocamlmklib
OCAMLMKTOP = ocamlmktop
OCAMLDOC = ocamldoc

#------------------------------------------
# Project name Concise animation language
#------------------------------------------

PROJECT = cal


LEXFILES:=$(wildcard *.mll)
MLFROMLEX:=$(LEXFILES:.mll=.ml)
YACCFILES:=$(wildcard *.mly)
MLFROMYACC:=$(YACCFILES:.mly=.ml)
MLIFROMYACC:=$(YACCFILES:.mly=.mli)
MLFILES:=$(MLFROMYACC) $(MLFROMLEX) $(filter-out $(MLFROMYACC),$(filter-out $(MLFROMLEX),$(wildcard *.ml)))
OBJS:= parser.cmo scanner.cmo string2string.cmo errors.cmo utils.cmo llvm_def.cmo debug_print.cmo extern_funcs.cmo interpreter.cmo
OCAMLFLAGS = -w -a

all: .depend $(MLFROMLEX) $(MLFROMYACC) $(PROJECT)

$(MLFROMLEX) : $(LEXFILES)
	result="$(foreach file, $(LEXFILES), $(shell $(OCAMLLEX) $(file)))"
$(MLFROMYACC) $(MLIFROMYACC): $(YACCFILES)
	$(OCAMLYACC) $^
%.cmi : %.mli
	$(OCAMLC) $(OCAMLFLAGS) -c $<
%.cmo : %.ml
	$(OCAMLC) $(OCAMLFLAGS) -c $<

.depend: $(MLFILES) $(MLIFROMYACC)
	$(OCAMLDEP) $^ > .depend

clean:
	rm -f *.cmi *.cmo .depend .objs $(MLFROMLEX) $(MLFROMYACC) $(MLIFROMYACC) $(PROJECT)

.PHONY: clean all test

-include .depend

$(PROJECT): $(OBJS)
	$(OCAMLC) $(OCAMLFLAGS) -o $(PROJECT) $^

test: $(PROJECT)
	make -C test
```

9.1.2. scanner.mll (Xinan Xu, Jingyi Guo, Tianliang Sun)

```
{ open Parser }
```

```
rule token = parse
   [' ' '\t' '\r' '\n']          { token lexbuf } (* Whitespace *)
  | "/*"                         { block_comment lexbuf }         (* Comments *)
  | "//"                         { line_comment lexbuf }
   (* Tokens *)
  | '('                                  { LPAREN }
  | ')'                          { RPAREN }
  | '{'                          { LBRACE }
  | '}'                          { RBRACE }
  | '['                          { LSBRACE }
  | ']'                          { RSBRACE }
  | ';'                          { SEMI }
  | ','                          { COMMA }
  | "++"                         { PLUSPLUS}
  | "--"                         { MINUSMINUS}
  | "=="                         { EQ }
  | "!="                         { NE }
  | "<="                         { LE }
  | ">="                         { GE }
  | "&&"                                 { AND }
  | "||"                         { OR }
  | '+'                          { PLUS }
  | '-'                          { MINUS }
  | '*'                          { TIMES }
  | '/'                          { DIVIDE }
  | '%'                          { MOD }
  | "+="                         { PLUS_ASSIGN }
  | "-="                         { MINUS_ASSIGN }
  | "*="                         { TIMES_ASSIGN }
  | "/="                         { DIVIDE_ASSIGN }
  | "%="                                 { MOD_ASSIGN }
  | '='                          { ASSIGN }
  | '<'                          { LT }
  | '>'                          { GT }
  | '!'                          { NOT }
  | '.'                          { DOT }
   (* reserved keywords *)
   (* primitive data types *)
  | "int"                        { INT }
  | "double"                             { DOUBLE }
  | "string"                     { STRING } (* XINAN: to use C++ string type *)
  | "list"                       { LIST }   (* to use C++ STL vector *)
  | "struct"                     { STRUCT } (* Same as C *)
  | "point"                      { POINT }
  | "shape"                      { SHAPE }
   (* keywords for control flow *)
  | "if"                         { IF }
  | "else"                       { ELSE }
  | "for"                        { FOR }
  | "return"                     { RETURN }
  | "break"                      { BREAK }
   (* constant *)
  | ['0'-'9']+ as lxm            { CONST_INT(int_of_string lxm) } (*integers*)
  | ['0'-'9']+ '.' ['0'-'9']+ ('e' ['-']? ['0'-'9']+)? as lxm { CONST_DBL(float_of_string lxm) }
  | ['a'-'z' 'A'-'Z' ]['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
  | eof                      { EOF }
  | ['.']['0'-'9']* as float  { raise (Failure("No support for float type " ^ float)) }
  | '"' ( [^ '\\' '"' '\n']|'\\' _ )* '"' as lxm { CONST_STR(lxm) }
  | _ as char                            { raise (Failure("illegal character " ^ Char.escaped char)) }

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

and line_comment = parse
   '\n'                          { token lexbuf }
  | _                                    { line_comment lexbuf }
```

9.1.3. string2string.ml (Xinan Xu)

```
{}
rule string2string str length = parse
|"\\\\" { string2string (str ^ "\\5C") (length+1) lexbuf }
|"\\n"  { string2string (str ^ "\\0A") (length+1) lexbuf }
|"\\t"  { string2string (str ^ "\\09") (length+1) lexbuf }
|"\\\""  { string2string  (str ^ "\\22") (length+1) lexbuf }
|"\""  { string2string  str length lexbuf }
|eof { (str, length) }
|_ as s { string2string  (str ^ Char.escaped s) (length+1) lexbuf }
```

9.1.4. parser.mly (Tianliang Sun, Jingyi Guo)

```
%{ open Ast %}

//    (     )     {     }       [     ]     ;     ,
%token LPAREN RPAREN LBRACE RBRACE LSBRACE RSBRACE  SEMI COMMA
//       ++         --
%token PLUSPLUS MINUSMINUS
//    == != <  <= > >=
%token EQ NE LT LE GT GE
//     && ||  !  .
%token AND OR NOT DOT
//     +    -     *     /    %
%token PLUS MINUS TIMES DIVIDE MOD
//      =     +=             -=          *=         %=
%token ASSIGN PLUS_ASSIGN MINUS_ASSIGN TIMES_ASSIGN DIVIDE_ASSIGN MOD_ASSIGN
//reserved keywords for control flow
%token RETURN IF ELSE FOR BREAK
//reserved keywords for types
%token INT DOUBLE STRING LIST STRUCT POINT SHAPE
%token <string> ID
//constants
%token <int> CONST_INT
%token <float> CONST_DBL
%token <string> CONST_STR

%token EOF

%nonassoc NOELSE
%nonassoc IF ELSE RETURN FOR BREAK

%right ASSIGN PLUS_ASSIGN MINUS_ASSIGN TIMES_ASSIGN DIVIDE_ASSIGN MOD_ASSIGN COMMA
%left OR
%left AND
%left EQ NE
%left LT GT LE GE
%left PLUS MINUS
%left TIMES DIVIDE MOD
%right NOT
%left PLUSPLUS MINUSMINUS
%left DOT LPAREN RPAREN LSBRACE RSBRACE

%start program
%type <Ast.program> program

%%
```

```
program:
        function_definition_list                          { { decl_list = [];
                                                              func_def_list = List.rev $1; } }
      | declaration_list function_definition_list         { { decl_list = List.rev $1;
                                                              func_def_list = List.rev $2; } }

expr:
        ID LPAREN expr_list_opt RPAREN                    { Expr_func($1, $3) }
      | constant                                          { Expr_const($1) }
      | lvalue                                            { Expr_lval($1) }
      | MINUS expr %prec NOT                              { Expr_op("-", $2) }
      | NOT expr                                          { Expr_op("!", $2) }
      | lvalue PLUSPLUS                                   { Expr_lval_op($1, "++") }
      | lvalue MINUSMINUS                                 { Expr_lval_op($1, "--") }
      | expr TIMES expr                                   { Expr_binop($1, "*", $3) }
      | expr DIVIDE expr                                  { Expr_binop($1, "/", $3) }
      | expr MOD expr                                     { Expr_binop($1, "%", $3) }
      | expr PLUS expr                                    { Expr_binop($1, "+", $3) }
      | expr MINUS expr                                   { Expr_binop($1, "-", $3) }
      | expr LT expr                                      { Expr_binop($1, "<", $3) }
      | expr GT expr                                      { Expr_binop($1, ">", $3) }
      | expr LE expr                                      { Expr_binop($1, "<=", $3) }
      | expr GE expr                                      { Expr_binop($1, ">=", $3) }
      | expr EQ expr                                      { Expr_binop($1, "==", $3) }
      | expr NE expr                                      { Expr_binop($1, "!=", $3) }
      | expr AND expr                                     { Expr_binop($1, "&&", $3) }
      | expr OR expr                                      { Expr_binop($1, "||", $3) }
      | lvalue ASSIGN        expr                         { Expr_asgop($1, "=", $3) }
      | lvalue PLUS_ASSIGN expr                           { Expr_asgop($1, "+=", $3) }
      | lvalue MINUS_ASSIGN expr                          { Expr_asgop($1, "-=", $3) }
      | lvalue TIMES_ASSIGN expr                          { Expr_asgop($1, "*=", $3) }
      | lvalue DIVIDE_ASSIGN expr                         { Expr_asgop($1, "/=", $3) }
      | lvalue MOD_ASSIGN expr                            { Expr_asgop($1, "%=", $3) }
      | LPAREN expr RPAREN                                { Expr_paren($2) }

lvalue:
        ID                                                { Lval_id($1) }
      | lvalue LSBRACE expr RSBRACE                       { Lval_arr($1, $3) }
      | lvalue DOT ID                                     { Lval_struct($1, $3) }

expr_list_opt:
                                                          { [] }
    | expr_list                                           { List.rev $1 }

expr_list:
    expr                                                  { [$1] }
    | expr_list COMMA expr                                { $3 :: $1 }

stmt_list:
        stmt                                              { [$1] }
      | stmt_list stmt                                    { $2 :: $1 }

stmt:
        expr SEMI                                         { Stmt($1) }
      | LBRACE stmt_list RBRACE                           { Stmt_block(List.rev $2) }
      | IF LPAREN expr RPAREN stmt %prec NOELSE           { Stmt_if($3, $5) }
      | IF LPAREN expr RPAREN stmt ELSE stmt              { Stmt_ifelse($3, $5, $7) }
      | FOR LPAREN expr_opt SEMI expr_opt SEMI expr_opt RPAREN stmt  { Stmt_for($3, $5, $7, $9) }
      | BREAK SEMI                                        { Stmt_break }
      | RETURN expr SEMI                                  { Stmt_ret($2) }
      | SEMI                                              { Stmt_noop }

expr_opt:
        /* nothing */                                     { Expr_opt(None) }
      | expr                                              { Expr_opt(Some($1)) }

declaration_list:
        declaration                                       { [$1] }
      | declaration_list declaration                      { $2 :: $1 }
```

```
declaration:
        type_specifier declarator_list SEMI                       { Decl($1, List.rev $2) }
    | STRUCT ID LBRACE struct_decl_list RBRACE SEMI               { Decl_struct($2, List.rev $4) }
    | STRUCT POINT LBRACE struct_decl_list RBRACE SEMI            { Decl_struct("point", List.rev $4) }
    | STRUCT SHAPE LBRACE struct_decl_list RBRACE SEMI            { Decl_struct("shape", List.rev $4) }

struct_decl_list:
        struct_decl                                               { [$1] }
        | struct_decl_list struct_decl                            { $2 :: $1 }

struct_decl:
        type_specifier ID SEMI                                    { ($2, $1) }

type_specifier:
        INT                                                       { Int }
        | DOUBLE                                                  { Double }
        | STRING                                                  { String }
        | LIST LT type_specifier GT %prec DOT                     { List($3) }
        | STRUCT ID                                               { Struct($2) }
        | POINT                                                   { Struct("point") }
        | SHAPE                                                   { Struct("shape") }

declarator_list:
        declarator                                                { [$1] }
        | declarator_list COMMA declarator                        { $3 :: $1 }

declarator:
        ID                                                        { Declarator($1, None) }
        | ID initializer_                                         { Declarator($1, Some($2)) }

initializer_:
        ASSIGN constant_expr                                      { Init_const_expr($2) }
        | ASSIGN LBRACE constant_expr_list RBRACE                 { Init_const_expr_list(List.rev $3) }

constant:
        CONST_INT                                                 { Const_int($1) }
        | CONST_DBL                                               { Const_dbl($1) }
        | CONST_STR                                               { Const_str($1) }

constant_expr:
        constant                                                  { Const_expr($1) }
        | MINUS constant %prec NOT                                { Const_expr_neg($2) }

constant_expr_list:
        constant_expr                                             { [$1] }
        | constant_expr_list COMMA constant_expr                  { $3 :: $1 }

function_definition:
        type_specifier function_declarator function_body      { { type_spec = $1;
                                                                  func_decl = $2;
                                                                  func_body = $3; } }
function_definition_list:
        function_definition                                       { [$1] }
    | function_definition_list function_definition            { $2 :: $1 }

function_declarator:
        ID LPAREN param_list_opt RPAREN                           { $1, List.rev $3 }

param_list_opt:
        /* nothing */                                             { [] }
        | parameter_list                                          { $1 }

parameter_list:
        function_parameter                                        { [$1] }
        | parameter_list COMMA function_parameter                 { $3 :: $1 }

function_parameter:
        type_specifier ID                                         { $2, $1 }
```

```
function_body:
        LBRACE declaration_list_opt stmt_list RBRACE                    { { func_decl_list = List.rev $2;
                                                                            stmt_list = List.rev $3; } }

declaration_list_opt:
        /* nothing */                                                           { [] }
        | declaration_list                                              { $1 }
```

9.1.5. ast.mli (Tianliang Sun)

```
type symbol_table = {
  parent : symbol_table option;
  mutable variables : (string * type_specifier * string list) list;
  mutable functions : ((string * type_specifier) * ((string * type_specifier) list)) list;
  mutable structs: (string * (string * type_specifier) list) list;
  ret_type : type_specifier;
}

and lvalue =
    Lval_id of string
  | Lval_arr of lvalue * expr
  | Lval_struct of lvalue * string

and expr =
    Expr_func of string * expr list
  | Expr_const of constant
  | Expr_op of string * expr
  | Expr_lval of lvalue
  | Expr_lval_op of lvalue * string
  | Expr_binop of expr * string * expr
  | Expr_asgop of lvalue * string * expr
  | Expr_paren of expr

and expr_opt =  Expr_opt of expr option

and stmt =
    Stmt of expr
  | Stmt_block of stmt list
  | Stmt_if of expr * stmt
  | Stmt_ifelse of expr * stmt * stmt
  | Stmt_for of expr_opt * expr_opt * expr_opt * stmt
  | Stmt_break
  | Stmt_ret of expr
  | Stmt_noop

and constant =
    Const_int of int
  | Const_dbl of float
  | Const_str of string

and constant_expr =
    Const_expr of constant
  | Const_expr_neg of constant

and type_specifier =
      Int
    | Double
    | List of type_specifier
    | String
    | Struct of string

and initializer_ =
      Init_const_expr of constant_expr
    | Init_const_expr_list of constant_expr list
```

```
and declarator = Declarator of string * initializer_ option

and declaration =
                Decl of type_specifier * declarator list
              | Decl_struct of string * struct_decl_list

and struct_decl_list = (string * type_specifier) list

and function_body = { func_decl_list : declaration list;
                                mutable stmt_list : stmt list; }

and function_parameter = string * type_specifier

and function_declarator = string * function_parameter list

and function_definition = { type_spec : type_specifier;
                                        func_decl : function_declarator;
                                        func_body : function_body; }

and program = { decl_list : declaration list;
            func_def_list : function_definition list; }

and program_data = { prog_env : symbol_table;
                prog_functions : function_data list; }

and function_data = { func_env : symbol_table;
                                func_definition : function_definition; }
```

9.1.6. errors.ml (Tianliang Sun)

```
open Ast

exception Error of string

let rec func_args_to_string typs =
  List.fold_left (fun str typ -> str ^ (type_to_string typ) ^ ",") "" typs

and type_to_string = function
    Int -> "int"
  | Double -> "double"
  | List(typ) -> "list<" ^ (type_to_string typ) ^ "> "
  | String -> "string"
  | Struct(str) -> "struct " ^ str
```

9.1.7. extern_funcs.ml (Tianliang Sun)

```
open Ast
open Utils

let rec make_extern_func_list =
  let extern_func_list = [] in
  let add_point = ("add_point", Int), [("pt", Struct("point"))] in
    let extern_func_list = List.rev (add_point :: List.rev extern_func_list) in
  let add_shape = ("add_shape", Int), [("shp", Struct("shape"))] in
    let extern_func_list = List.rev (add_shape :: List.rev extern_func_list) in
  let display_points = ("display_points", Int), [] in
    let extern_func_list = List.rev (display_points :: List.rev extern_func_list) in
  let display_shapes = ("display_shapes", Int), [] in
```

```
    let extern_func_list = List.rev (display_shapes :: List.rev extern_func_list) in
  let display = ("display", Int), [] in
    let extern_func_list = List.rev (display :: List.rev extern_func_list) in
  let setup = ("setup", Int), [] in
    let extern_func_list = List.rev (setup :: List.rev extern_func_list) in
  let pop_point = ("pop_point", Int), [] in
    let extern_func_list = List.rev (pop_point :: List.rev extern_func_list) in
  let pop_shape = ("pop_shape", Int), [] in
    let extern_func_list = List.rev (pop_shape :: List.rev extern_func_list) in
  let wait = ("wait", Int), [("seconds", Double)] in
    let extern_func_list = List.rev (wait :: List.rev extern_func_list) in
  let byebye = ("byebye", Int), [] in
    let extern_func_list = List.rev (byebye :: List.rev extern_func_list) in
  let run =  ("run", Int), [] in
    List.rev (run :: List.rev extern_func_list)

and load_extern_funcs (env : symbol_table) =
  env.functions <- (merge_list make_extern_func_list env.functions); env
```

9.1.8. utils.ml (Tianliang Sun)

```
open Ast
open Errors

let rec sizeof env = function
     Int -> 4
  | Double -> 8
  | List(list_typ) -> 100 * (sizeof env list_typ)
  | String -> 100
  | Struct(str) -> let ( _ , size, _ ) =  (sizeof_struct env (Struct(str))) in size

and sizeof_struct env typ =
  let typ_list = (get_struct_type_list env typ) in
    List.fold_left add_struct_size (env, 0, 0) typ_list

and add_struct_size (env, total_size, max_single_size) typ =
  begin match typ with
       List(list_typ) -> get_new_size_list env (sizeof env list_typ) total_size max_single_size
     | String -> (env, total_size + 100, max_single_size)
     | Struct(str) -> get_new_size env (sizeof env typ) total_size max_single_size
     | _ -> get_new_size env (sizeof env typ) total_size max_single_size
  end

and get_new_size env member_size total_size max_single_size =
  let new_max_single_size =
    if (member_size > max_single_size) then member_size else max_single_size in
    let rem = total_size mod member_size in
      let new_total_size = total_size + ((member_size - rem) mod member_size) + member_size in
        let total_rem = new_total_size mod new_max_single_size in
          (env, new_total_size + ((new_max_single_size - total_rem) mod new_max_single_size), new_max_single_size)

and get_new_size_list env type_size total_size max_single_size =
  let new_max_single_size =
    if (type_size > max_single_size) then type_size else max_single_size in
    let rem = total_size mod type_size in
      (env, total_size + rem + type_size * 100, new_max_single_size)

and check_const_type = function
       Const_int(int) -> Int
            | Const_dbl(dbl) -> Double
     | Const_str(str) -> String

and check_const_expr_type = function
     Const_expr(const) -> check_const_type const
```

```
  | Const_expr_neg(const) -> check_const_type const

and get_struct_type_list (scope : symbol_table) typ =
  try let ( _ , member_list ) = find_struct_by_type scope typ in
    List.fold_left (fun ret_list member -> List.rev ((snd member) :: List.rev ret_list)) [] member_list
  with Not_found -> raise(Error("undefined struct"))

and find_variable (scope : symbol_table) name =
     try
       List.find (fun (s, _, _ ) -> s = name) scope.variables
     with Not_found ->
       match scope.parent with
          Some(parent) -> find_variable parent name
        | _ -> raise Not_found

and check_variable (scope : symbol_table) name =
  let result = fst (List.fold_left (fun (result, name) var ->
    let (var_name, _ , _ ) = var in if(name = var_name) then (true, name)
      else (result, name)) (false, name) scope.variables)
  in if(result) then result
  else match scope.parent with
     Some(parent) -> check_variable parent name
   | _ -> result

and get_func_arg_typs args_list =
  List.fold_left (fun ret_list arg -> List.rev ((snd arg) :: List.rev ret_list)) [] args_list

and find_function (scope : symbol_table) name typ_list =
  try
    List.find (fun ((s, _ ), args ) ->
      let arg_typs = get_func_arg_typs args in (s = name) && (Pervasives.compare arg_typs typ_list = 0)) scope.functions
     with Not_found ->
       match scope.parent with
          Some(parent) -> find_function parent name typ_list
        | _ -> raise Not_found

and check_function (scope : symbol_table) name typ_list =
  let result = fst (List.fold_left (fun (result, name) func ->
    let ((func_name, _ ) , args ) = func in
      let arg_typs = get_func_arg_typs args in
        if((func_name = name) && (Pervasives.compare arg_typs typ_list = 0)) then (true, name)
        else (result, name)) (false, name) scope.functions)
  in if(result) then result
  else match scope.parent with
     Some(parent) -> check_function parent name typ_list
   | _ -> result

and find_struct_by_type (scope : symbol_table) typ =
     try
       List.find (fun (s, _ ) -> Pervasives.compare (Struct s) typ = 0) scope.structs
     with Not_found ->
       match scope.parent with
          Some(parent) -> find_struct_by_type parent typ
        | _ -> raise Not_found

and check_struct (scope : symbol_table) name =
  let result = fst (List.fold_left (fun (result, name) struc ->
    let (struct_name, _ ) = struc in if(name = struct_name) then (true, name)
      else (result, name)) (false, name) scope.structs)
  in if(result) then result
  else match scope.parent with
     Some(parent) -> check_struct parent name
   | _ -> result

and find_struct_member struct_decl_list name =
     try
       List.find (fun (s, _ ) -> s = name) struct_decl_list
     with Not_found -> raise Not_found
```

```
and match_struct_members member_list typ_list index =
   if((List.length member_list) <> (List.length typ_list)) then false
   else (
      let ( _ , member_typ) = (List.nth member_list index) in
      (Pervasives.compare member_typ (List.nth typ_list index) = 0) &&
      (if(index = (List.length member_list) - 1) then true
       else (match_struct_members member_list typ_list (index+1))))

and sequency env struc member_name =
   let ( _ , member_list) = find_struct_by_type env struc in
      let ( index, _, _) = (List.fold_left check_member_sequency (0, false, member_name) member_list)
      in index

and check_member_sequency (index, found, member_name) member =
   if(found) then (index, found, member_name)
   else
      let (name, _ ) = member in
         if(name = member_name) then (index + 1, true, member_name)
         else (index + 1, false, member_name)

and member2type struc member_name env =
   let ( _ , member_list) = find_struct_by_type env struc in
                  let(s,t) = List.find (fun (str, typ) -> str = member_name) member_list
                           in t

and merge_list snd_list fst_list =
   List.fold_left (fun list list_element -> List.rev (list_element :: List.rev list)) snd_list fst_list

and struct_decl_to_stmts id member_list const_expr_list =
   let (stmt_list, _, _, _) = (List.fold_left struct_decl_to_stmt ([], id, member_list, 0) const_expr_list)
   in stmt_list

and struct_decl_to_stmt (stmt_list, id, member_list, index) const_expr =
   let member = (List.nth member_list index) in
      let (member_name, _ ) = member in
         let lval = Lval_id(id) in
            let lval_struct = Lval_struct(lval, member_name) in
               let expr_const = const_expr_to_expr const_expr in
                  let expr = Expr_asgop(lval_struct, "=", expr_const) in
                     let stmt = Stmt(expr) in
                        ((List.rev (stmt :: List.rev stmt_list)), id, member_list, (index+1))

and point_decl_to_stmts str const_expr_list =
   let lval = Lval_id(str) in
      let lval_x = Lval_struct(lval, "x") in
         let const_expr_x = (List.nth const_expr_list 0) in
            let expr_const_x = const_expr_to_expr const_expr_x in
               let expr_x = Expr_asgop(lval_x, "=", expr_const_x) in
                  let lval_y = Lval_struct(lval, "y") in
                     let const_expr_y = (List.nth const_expr_list 1) in
                        let expr_const_y = const_expr_to_expr const_expr_y in
                           let expr_y = Expr_asgop(lval_y, "=", expr_const_y) in
                              let stmt_x = Stmt(expr_x) in
                                 let stmt_y = Stmt(expr_y) in
                                    let tmp_list = [stmt_y] in
                                       stmt_x :: tmp_list

and shape_decl_to_stmts str const_expr_list =
   let lval = Lval_id(str) in
      let lval_size = Lval_struct(lval, "size") in
         let const_expr_size = (List.nth const_expr_list 0) in
            let expr_const_size = const_expr_to_expr const_expr_size in
               let expr_size = Expr_asgop(lval_size, "=", expr_const_size) in
                  let lval_x = Lval_struct(lval, "center_x") in
                     let const_expr_x = (List.nth const_expr_list 1) in
                        let expr_const_x = const_expr_to_expr const_expr_x in
                           let expr_x = Expr_asgop(lval_x, "=", expr_const_x) in
                              let lval_y = Lval_struct(lval, "center_y") in
                                 let const_expr_y = (List.nth const_expr_list 2) in
```

```
                                    let expr_const_y = const_expr_to_expr const_expr_y in
                                      let expr_y = Expr_asgop(lval_y, "=", expr_const_y) in
                                        let stmt_size = Stmt(expr_size) in
                                          let stmt_x = Stmt(expr_x) in
                                            let stmt_y = Stmt(expr_y) in
                                              let tmp_list = [stmt_y] in
                                                let tmp_list = stmt_x :: tmp_list in
                                                  stmt_size :: tmp_list

and const_expr_to_expr = function
    Const_expr(const) -> Expr_const(const)
  | Const_expr_neg(const) ->
    let expr = Expr_const(const) in
      Expr_op("-", expr)

and get_lval_id = function
    Lval_id(str) -> str
  | Lval_arr(lval, expr) -> get_lval_id lval
  | Lval_struct(lval, str) -> get_lval_id lval

and type_to_string = function
    Int -> "int"
  | Double -> "double"
  | List(type_spec) -> "list<" ^ (type_to_string type_spec) ^ "> "
  | String -> "string"
  | Struct(str) -> "struct " ^ str

and add_gl_calls stmt_list =
  let ret_list = [] in
    let expr_setup = Expr_func("setup", []) in
    let stmt_setup = Stmt(expr_setup) in
    let ret_list = (stmt_setup :: ret_list)in
      List.fold_left (fun list stmt ->
        begin match stmt with
            Stmt_ret(expr) ->
              let expr_run = Expr_func("byebye", []) in
              let stmt_run = Stmt(expr_run) in
              let list = List.rev (stmt_run :: List.rev list) in
              List.rev (stmt :: List.rev list)
          | _ -> List.rev (stmt :: List.rev list)
        end) ret_list stmt_list
```

9.1.9. debug_print.ml (Tianliang Sun)

```
open Ast
open Utils

let rec print_prog prog = print_endline("GLOBAL SCOPE:"); print_env prog.prog_env;print_endline("function definitions:");
  print_function_data_list prog.prog_functions

and print_env env = print_endline("variables:"); print_variables env.variables; print_endline("structs:");
  print_structs env.structs; print_endline("return type:"); print_type env.ret_type; print_endline("\nfunction signatures:");
  print_functions env.functions

and print_variables variables = List.iter print_variable variables;

and print_variable variable =
  let (name, typ, value) = variable in
    print_type(typ); print_string( name ^ " = ");
    begin match value with
        [] -> print_endline("")
      | _ ->  print_endline(List.hd value)
    end
```

```
and print_type = function
    Int -> print_string("int ")
  | Double -> print_string("double ")
  | List(type_spec) -> print_string("list<"); print_type(type_spec); print_string("> ")
  | String -> print_string("string ")
  | Struct(str) -> print_string("struct " ^ str ^ " ")

and print_structs structs = List.iter print_struct structs

and print_struct struc =
  let (name, member_list) = struc in print_endline("struct " ^ name ^ " = {");
    List.iter print_struct_member member_list; print_endline("}")

and print_struct_member member = print_string("\t"); print_type (snd member); print_endline(fst member)

and print_functions functions = List.iter print_func_sig functions

and print_func_sig func = print_string((fst (fst func)) ^ "("); print_func_sig_params (snd func);
  print_endline(")")

and print_func_sig_params params = List.iter print_func_sig_param params

and print_func_sig_param param = print_string((fst param) ^ ", ")

and print_function_data_list func_data_list = List.iter print_function_data func_data_list

and print_function_data func = print_endline("\nSCOPE OF " ^ (fst func.func_definition.func_decl) ^ ":");
  print_env func.func_env;
  print_endline("function body:");
  print_type func.func_definition.type_spec;
  print_string((fst func.func_definition.func_decl) ^ "(");
  print_function_params (snd func.func_definition.func_decl); print_endline("){");
  print_function_stmts func.func_definition.func_body.stmt_list; print_endline("}")

and print_function_params params = List.iter print_function_param params

and print_function_param param = print_type (snd param); print_string((fst param) ^ ",")

and print_function_stmts stmt_list = List.iter print_stmt stmt_list

and print_stmt = function
    Stmt(expr) -> print_expr expr; print_endline(";")
  | Stmt_block(stmt_list) -> print_endline("{"); List.iter print_stmt stmt_list; print_endline("}")
  | Stmt_if(expr, stmt) -> print_string("if("); print_expr expr; print_endline(") then"); print_stmt stmt
  | Stmt_ifelse(expr, stmt1, stmt2) -> print_string("if("); print_expr expr; print_endline(") then");
    print_stmt stmt1; print_endline("else"); print_stmt stmt2
  | Stmt_for(expr_opt1, expr_opt2, expr_opt3, stmt) -> print_string("for("); print_expr_opt expr_opt1;
    print_string("; "); print_expr_opt expr_opt2; print_string("; "); print_expr_opt expr_opt3;
    print_endline(")"); print_stmt stmt
  | Stmt_break -> print_endline("break;")
  | Stmt_ret(expr) -> print_string("return "); print_expr expr; print_endline(";")
  | Stmt_noop -> print_endline(";")

and print_expr = function
    Expr_func(str, expr_list_opt) -> print_string(str ^ "("); List.iter print_expr expr_list_opt;
    print_string(")")
  | Expr_const(constant) -> print_constant constant
  | Expr_op(str, expr) -> print_string(str); print_expr expr
  | Expr_lval(lvalue) -> print_lval lvalue
  | Expr_lval_op(lvalue, str) -> print_lval lvalue; print_string(str)
  | Expr_binop(expr1, str, expr2) -> print_string("[");print_expr expr1; print_string(" " ^ str ^ " "); print_expr expr2;print_string("]")
  | Expr_asgop(lvalue, str, expr) -> print_lval lvalue; print_string(" " ^ str ^ " "); print_expr expr
  | Expr_paren(expr) ->  print_string("("); print_expr expr; print_string(")")

and print_expr_opt = function
    Expr_opt(None) -> ()
  | Expr_opt(Some(expr)) -> print_expr expr

and print_constant = function
```

```
    Const_int(int) -> print_string(string_of_int(int))
  | Const_dbl(dbl) -> print_string(string_of_float(dbl))
  | Const_str(str) -> print_string(str)

and print_lval = function
    Lval_id(str) -> print_string(str)
  | Lval_arr(lval, expr) -> print_lval lval; print_string("["); print_expr expr; print_string("]")
  | Lval_struct(lval, str) -> print_lval lval; print_string("." ^ str)
```

9.1.10. interpreter.ml (Tianliang Sun)

```
open Ast
open Errors
open Llvm_def
open Utils
open Extern_funcs
open Debug_print

let rec eval_lval env = function
      Lval_id(str) ->
        let result = try
          find_variable env str
          with Not_found -> raise (Error("undeclared identifier " ^ str))
        in let (_, typ, _) = result in (typ, env)
    | Lval_arr(lval, expr) ->
        let typ = fst (eval_lval env lval) in
          begin match typ with
             List( list_typ ) -> (list_typ, env)
           | _ -> raise (Error("referenced variable " ^ (get_lval_id lval) ^" is not a list"))
          end
    | Lval_struct(lval, str) ->
        let typ = fst (eval_lval env lval) in try
          let ( struct_name , struct_decl_list) = find_struct_by_type env typ in try
            let ( _ , member_typ) = find_struct_member struct_decl_list str in (member_typ, env)
            with Not_found -> raise (Error( str ^ " is not a member of struct " ^ struct_name))
          with Not_found ->
            raise (Error("referenced variable " ^ (get_lval_id lval) ^ " is not a struct "))


and eval_expr env = function
      Expr_func(str, expr_list_opt) ->
        let typ_list = fst (eval_expr_list_opt env expr_list_opt) in
          let result = try
            find_function env str typ_list
            with Not_found -> raise (Error("undefined function: " ^ str ^ "(" ^
              (func_args_to_string typ_list) ^ ")"))
          in (snd (fst result), env)
    | Expr_const(constant) -> (check_const_type constant, env)
    | Expr_op(str, expr) ->
        let expr_type = fst (eval_expr env expr) in
          if (Pervasives.compare expr_type Int = 0 || Pervasives.compare expr_type Double = 0)
            then (expr_type, env)
          else raise(Error("operator " ^ str ^ " not supported for " ^ (type_to_string expr_type)))
    | Expr_lval(lvalue) -> eval_lval env lvalue
    | Expr_lval_op(lvalue, str) ->
        let lval_type = fst (eval_lval env lvalue) in
          if (Pervasives.compare lval_type Int = 0) then (lval_type, env)
          else raise(Error("operator " ^ str ^ " not supported for " ^ (type_to_string lval_type)))
    | Expr_binop(expr1, str, expr2) ->
        let expr1_type = fst (eval_expr env expr1) in
          let expr2_type = fst (eval_expr env expr2) in
            if (str = "*" || str = "/" || str = "%" || str = "+" || str = "-") then
              if ((Pervasives.compare expr1_type Int = 0 || Pervasives.compare expr1_type Double = 0) &&
                (Pervasives.compare expr2_type Int = 0 || Pervasives.compare expr2_type Double = 0)) then
                if(Pervasives.compare expr1_type Double = 0 || Pervasives.compare expr2_type Double = 0) then
```

```
              (Double, env)
            else (Int, env)
          else raise(Error("binary operator " ^ str ^ " not supported for " ^ (type_to_string expr1_type)
            ^ " and " ^ (type_to_string expr2_type)))
        else if (str = "&&" || str = "||") then
          if (Pervasives.compare expr1_type expr2_type = 0) then
            if (Pervasives.compare expr1_type Int = 0) then (Int, env)
            else raise(Error("binary operator " ^ str ^ " not supported for " ^ (type_to_string expr1_type)))
          else raise(Error("types of operands does not match: " ^ (type_to_string expr1_type)
            ^ " " ^ str ^ " " ^ (type_to_string expr2_type)))
        else
          if ((Pervasives.compare expr1_type Int = 0 || Pervasives.compare expr1_type Double = 0) &&
            (Pervasives.compare expr2_type Int = 0 || Pervasives.compare expr2_type Double = 0)) then
            (Int, env)
          else raise(Error("binary operator " ^ str ^ " not supported for " ^ (type_to_string expr1_type)
            ^ " and " ^ (type_to_string expr2_type)))
    | Expr_asgop(lvalue, str, expr) ->
      let lval_type = fst (eval_lval env lvalue) in
        let expr_type = fst (eval_expr env expr) in
          if (Pervasives.compare expr_type lval_type = 0) then
            if (str = "=") then (lval_type, env)
            else if (Pervasives.compare lval_type Int = 0 || Pervasives.compare lval_type Double = 0)
              then (lval_type, env)
              else if (str = "+=" && Pervasives.compare lval_type String = 0) then
                (lval_type, env)
                else raise(Error("operator " ^ str ^ " not supported for " ^ (type_to_string lval_type)))
          else if ((Pervasives.compare lval_type Int = 0 || Pervasives.compare lval_type Double = 0) &&
              (Pervasives.compare expr_type Int = 0 || Pervasives.compare expr_type Double = 0)) then
            (lval_type, env)
          else raise(Error("types of operands does not match: " ^
            (type_to_string lval_type) ^ " " ^ str ^ " " ^ (type_to_string expr_type)))
    | Expr_paren(expr) ->  eval_expr env expr

and eval_expr_opt env = function
      Expr_opt(None) -> (Int, env)
    | Expr_opt(Some(expr)) ->
      let result = eval_expr env expr in result

and eval_expr_list_opt env  = function expr_list_opt ->
  let (type_list, env) =
    List.fold_left (fun (typ_list, env) expr ->
      let (typ, env1) = eval_expr env expr in ((typ :: typ_list), env1)) ([], env) expr_list_opt
  in (List.rev type_list, env)

and eval_stmt (has_return, env) = function
      Stmt(expr) -> (has_return, (snd (eval_expr env expr)))
    | Stmt_block(stmt_list) -> List.fold_left eval_stmt (has_return, env) stmt_list
    | Stmt_if(expr, stmt) ->
      let typ = (fst (eval_expr env expr)) in
        if (Pervasives.compare typ Int = 0) then eval_stmt (has_return, env) stmt
        else raise(Error("type of if statement's predicate is " ^ (type_to_string typ)
          ^ ", but must be evaluated to int"))
        | Stmt_ifelse(expr, stmt1, stmt2) ->
      let typ = (fst (eval_expr env expr)) in
        if (Pervasives.compare typ Int = 0) then
          let (has_return1, env1) = eval_stmt (has_return, env) stmt1 in
            eval_stmt (has_return1, env1) stmt2
        else raise(Error("type of if statement's predicate is " ^ (type_to_string typ)
          ^ ", but must be evaluated to int"))
    | Stmt_for(expr_opt1, expr_opt2, expr_opt3, stmt) ->
      let env1 = snd (eval_expr_opt env expr_opt1) in
        let (typ, env2) = eval_expr_opt env1 expr_opt2 in
          if (Pervasives.compare Int typ = 0) then
            let env3 = snd (eval_expr_opt env2 expr_opt3) in
              eval_stmt (has_return, env3) stmt
          else raise(Error("type of for statement's condition is " ^ (type_to_string typ)
          ^ ", but must be evaluated to int"))
    | Stmt_break -> has_return, env
    | Stmt_ret(expr) ->
```

```
          let result = eval_expr env expr in
            if (Pervasives.compare (fst result) env.ret_type = 0) then
              true, snd result
            else raise(Error("function returns type " ^ (type_to_string (fst result)) ^
              ", but should return type " ^ (type_to_string env.ret_type)))
      | Stmt_noop -> has_return, env

and eval_stmt_list (has_return, env) = function stmt_list -> List.fold_left eval_stmt (has_return, env) stmt_list

and eval_const type_spec = function
        Const_int(int) -> if (Pervasives.compare type_spec Int = 0) then (string_of_int(int), Int)
                    else raise(Error("type mismatch: " ^ (type_to_string type_spec) ^ " -> int"))
            | Const_dbl(dbl) -> if (Pervasives.compare type_spec Double = 0) then (string_of_float(dbl), Double)
                    else raise(Error("type mismatch: " ^ (type_to_string type_spec) ^ " -> double"))
      | Const_str(str) -> if (Pervasives.compare type_spec String = 0) then (str, String)
                    else raise(Error("type mismatch: " ^ (type_to_string type_spec) ^ " -> string"))

and eval_const_expr type_spec = function
              Const_expr(const) -> eval_const type_spec const
            | Const_expr_neg(const) ->
         let (value, typ) = eval_const type_spec const in
           if(Pervasives.compare typ Int = 0 || Pervasives.compare typ Double = 0) then
             ("-" ^ value, typ)
           else raise(Error("- operator not supported for type string"))

and eval_const_expr_list env type_spec extra_stmts str = function const_expr_list ->
   match type_spec with
    List(typ) ->
      ((List.fold_left (fun list const_expr ->
        List.rev ((fst (eval_const_expr typ const_expr)) :: List.rev list)) [] const_expr_list), type_spec, extra_stmts)
   | Struct(struct_name) ->
     (try let ( _ , member_list ) = find_struct_by_type env (Struct(struct_name)) in
       let typ_list = List.fold_left (fun typ_list const_expr ->
         List.rev ((check_const_expr_type const_expr) :: List.rev typ_list)) [] const_expr_list in
         let result = (match_struct_members member_list typ_list 0) in
           if (result) then
                     let new_stmts = struct_decl_to_stmts str member_list const_expr_list in
             ([""], type_spec, (merge_list extra_stmts new_stmts))
           else raise(Error("types of the list elements do not match the member types of struct " ^ struct_name))
     with Not_found -> raise(Error("cannot initialize an undefined struct " ^ struct_name)))
   | _ -> raise(Error("cannot assign a list of values to a non-list/non-struct variable"))

and eval_init env type_spec extra_stmts str = function
        Init_const_expr(const_expr) ->
          let (value, typ) = eval_const_expr type_spec const_expr in ([value], typ, extra_stmts)
      | Init_const_expr_list(const_expr_list) ->  eval_const_expr_list env type_spec extra_stmts str const_expr_list

and eval_declarator (type_spec, env, extra_stmts) = function
        Declarator(str, None) ->
          let result = check_variable env str in
            if (not result) then (env.variables <- List.rev ((str, type_spec, []) :: List.rev env.variables);
              (type_spec, env, extra_stmts))
            else raise (Error("variable " ^ str ^ " has already been defined"))
      | Declarator(str, Some(init)) ->
         let (value, typ, stmts) = eval_init env type_spec extra_stmts str init in
           env.variables <- List.rev ((str, type_spec, value) :: List.rev env.variables);
           (type_spec, env, stmts)

and eval_decl (env, extra_stmts) = function
        Decl(typ, declarator_list) ->
          let ( _, new_env, stmts) = List.fold_left eval_declarator (typ, env, extra_stmts) declarator_list in
            (new_env, stmts)
      | Decl_struct(str, struct_decl_list) ->
         let result = check_struct env str in
           if(not result) then
             (env.structs <- List.rev ((str, struct_decl_list) :: List.rev env.structs);
             (env, []))
           else raise (Error("struct " ^ str ^ " has already been defined"))
```

```
and eval_decl_list env = function decl_list -> List.fold_left eval_decl (env, []) decl_list

and eval_decl_list_opt env = function decl_list -> eval_decl_list env decl_list

and eval_func_body typ env = function func_body ->
   let (env1, extra_stmts) = eval_decl_list_opt env func_body.func_decl_list in
      func_body.stmt_list <- (merge_list extra_stmts func_body.stmt_list);
      let (has_return, env2) = eval_stmt_list (false, env1) func_body.stmt_list in
         if(has_return) then env2
         else raise(Error("function must return a type of " ^ (type_to_string typ)))

and eval_func_def env = function func_def ->
   let new_env =
   { parent = Some(env);
     variables = [];
     functions = [];
     structs = [];
     ret_type = func_def.type_spec;
   } in
      let (name, param_list) = func_def.func_decl in
         let typ_list = get_func_arg_typs param_list in
            let result = check_function env name typ_list in
               if(not result) then (
                  env.functions <- List.rev (((name, func_def.type_spec), param_list) :: List.rev env.functions);
                  let new_env1 = eval_function_params new_env (snd func_def.func_decl) in
                  let new_env2 = eval_func_body func_def.type_spec new_env1 func_def.func_body in
                  let (func_name, _ ) = func_def.func_decl in
                     if(func_name = "main") then (func_def.func_body.stmt_list <- add_gl_calls func_def.func_body.stmt_list);
                  ({ func_env = new_env2; func_definition = func_def }, env))
               else raise (Error("function " ^ name ^ "(" ^ (func_args_to_string typ_list) ^ ") has already been defined"))

and eval_function_params env param_list =
   List.fold_left (fun env param ->
      (env.variables <- List.rev ((fst param, snd param, []) :: List.rev env.variables)); env) env param_list

and eval_func_def_list env = function func_def_list -> List.fold_left add_func_data ([], env) func_def_list

and add_func_data (func_data_list, env) = function func_def ->
   let (func_data, new_env) = eval_func_def env func_def in
      (List.rev (func_data :: List.rev func_data_list), new_env)

and eval_prog env = function prog ->
   let env1 = fst (eval_decl_list env prog.decl_list) in
      let env2 = load_extern_funcs env1 in
         let (func_data_list, env3) =  eval_func_def_list env2 prog.func_def_list in
            check_main env.functions;
            { prog_env = env3;
            prog_functions = func_data_list; }

and check_main func_list =
   try List.find (fun ((s, _ ), _ ) -> s = "main") func_list; ()
   with Not_found -> raise (Error("function main is not defined"))

let _ =
   let env =
   { parent = None;
     variables = [];
     functions = [];
     structs = [];
     ret_type = Int;
   } in
      let lexbuf = Lexing.from_channel stdin in
         let prog = Parser.program Scanner.token lexbuf in
            let prog_data = eval_prog env prog in llvm_prog prog_data
```

9.1.11. llvm_def.ml (Xinan Xu)

```
open Ast

let rec llvm_prog prog =
        llvm_global_variables prog.prog_env.variables;
        llvm_struct_def prog.prog_env.structs prog.prog_env;
        llvm_functions prog.prog_functions;
        print_endline("\ndeclare void @llvm.memcpy.i64(i8* nocapture, i8* nocapture, i64, i32) nounwind");
        print_endline("declare i8* @strcpy(i8* noalias, i8* noalias) nounwind");
        print_endline("declare i8* @strcat(i8* noalias, i8* noalias) nounwind");
        print_endline("declare i32 @add_shape(%struct.shape* byval)");
        print_endline("declare i32 @add_point(%struct.point* byval)");
        print_endline("declare i32 @setup()");
        print_endline("declare i32 @run()");
        print_endline("declare i32 @pop_point()");
        print_endline("declare i32 @pop_shape()");
        print_endline("declare i32 @wait(double)");
        print_endline("declare i32 @byebye()");

and llvm_functions prog_functions = List.iter llvm_func prog_functions

and llvm_struct_def structs env =
        match structs with
                [] -> print_endline("");
                |head::tail -> llvm_struct_def_single head env;
                                                llvm_struct_def tail env;
and llvm_struct_def_single (name, type_list) env =
        print_string("%struct." ^ name ^ " = type {  ");
        llvm_print_types type_list env;
        print_endline("}");
and llvm_print_types type_list env =
        match type_list with
         []-> print_string("")
        |[(str, typ)] -> print_string(llvm_print_type typ);
        |(str, typ)::tail -> print_string((llvm_print_type typ)^", ");
                llvm_print_types tail env;
and llvm_print_type typ=
        match typ with
                        Int -> "i32"
                        |Double -> "double"
                        |String -> "[100 x i8]"
                        |Struct(s) -> "%struct." ^ s
                        |List(t)->"[100 x "^llvm_print_type t^"]"


and llvm_func func =
        llvm_func_def func.func_definition;
        let (func_decls, statements, env, (func_name,_)) = (func.func_definition.func_body.func_decl_list,
func.func_definition.func_body.stmt_list, func.func_env,func.func_definition.func_decl) in
                llvm_func_stack_decls func_decls func_name;
                let str_index_typ = ("%reg0", 0, Int) in
                        llvm_stmts str_index_typ statements (func.func_definition, env, "", 0);
                        print_endline("");
                        print_endline("return:");
                        (
                        match func.func_definition.type_spec with
                        Struct(s)-> print_endline("\tret void");
                        |_->
                        print_endline("\t%retval4 = load " ^ type2string func.func_definition.type_spec ^ "* %retval");
                        print_endline("\tret " ^ type2string func.func_definition.type_spec ^" %retval4");
                        );
                        print_endline("}\n");


and llvm_stmts str_index_typ statements env=
        match statements with
         [] -> str_index_typ
```

```
        |head :: tail -> llvm_stmts (llvm_stmt str_index_typ head env) tail env

and llvm_stmt str_index_typ statement env=
        match statement with
         Stmt(expr) -> llvm_expr str_index_typ expr env
        |Stmt_block(stmts) -> llvm_stmts str_index_typ stmts env
        |Stmt_ret(expr) ->
                let (s1,i1,t1) = llvm_expr str_index_typ expr env
                and (def, e,_,_) = env in
                        ( match t1 with
                        |Struct(st) -> print_endline("\t call  void @llvm.memcpy.i64(i8* %agg.result1, i8* " ^ s1 ^ ", i64 " ^
string_of_int(Utils.sizeof e t1) ^", i32 8)");
                        |_->llvm_store def.type_spec s1 "%retval";
                        );
                        llvm_br "return"; (s1,i1,t1)
        |Stmt_if(expr, stmt1) ->
                let (s0,i0,t0) = llvm_expr str_index_typ expr env in
                let (s1,i1,t1) = llvm_trunc (s0,i0,t0) in
                        llvm_bri1 s1 (label i1 0) (label i1 1);
                        print_endline("");
                        print_endline((label i1 0) ^ ":");
                        let (s2,i2,t2) = llvm_stmt (s1,i1,t1) stmt1 env in
                                print_endline("");
                                print_endline((label i1 1) ^ ":");
                                (s2,i2,t2)
        |Stmt_ifelse(expr, stmt1,stmt2) ->
                let (s0,i0,t0) = llvm_expr str_index_typ expr env in
                let (s1,i1,t1) = llvm_trunc (s0,i0,t0) in
                        llvm_bri1 s1 (label i1 0) (label i1 1);
                        print_endline("");
                        print_endline((label i1 0) ^ ":");
                        let (s2,i2,t2) = llvm_stmt (s1,i1,t1) stmt1 env in
                                llvm_br (label i1 2);
                                print_endline("");
                                print_endline((label i1 1) ^ ":");
                                let (s3,i3,t3) = llvm_stmt (s2,i2,t2) stmt2 env in
                                        llvm_br (label i1 2);
                                        print_endline("");
                                        print_endline((label i1 2) ^ ":");
                                        (s3,i3,t3)
        |Stmt_for(expr1_opt, expr2_opt, expr3_opt, stmt) ->
                let (s1,i1,t1) = llvm_expr_opt str_index_typ expr1_opt env in
                        print_endline("");
                        llvm_br((label (i1+1) 0) ^ "forentry");
                        print_endline((label (i1+1) 0) ^ "forentry:");
                        let (s15,i15,t15) = llvm_expr_opt (s1,i1,t1) expr2_opt env in
                        let (s2,i2,t2) = llvm_trunc (s15,i15,t15) in
                                llvm_bri1 s2 (label i2 0) (label i2 1);
                                print_endline("");
                                print_endline((label i2 0) ^ ":");
                                let (def,env0,for_stack,n) = env in
                                let (s3,i3,t3) = llvm_stmt (s2,i2,t2) stmt (def,env0,(label i2 1),n) in
                                        let (s4,i4,t4) = llvm_expr_opt (s3,i3,t3) expr3_opt env in
                                                llvm_br((label (i1+1) 0) ^ "forentry");
                                                print_endline("");
                                                print_endline((label i2 1)^":");
                                                (s4,i4,t4)
        |Stmt_break ->
                let (def,env0,for_stack,_) = env in
                        llvm_br(for_stack); str_index_typ
        | _ -> str_index_typ

and llvm_expr_opt (s,i,t) expr_opt env =
        match expr_opt with
         Expr_opt(Some(expr)) -> llvm_expr (s,i,t) expr env
        |Expr_opt(None) -> (s,i,t)
and llvm_args (s,i,t,arglist) exprs env =
        match exprs with
         [] -> (s,i,t,arglist)
```

```
		|[head] -> llvm_arg_nc (s,i,t,arglist) head env
		|head::tail -> llvm_args (llvm_arg (s,i,t,arglist) head env) tail env
and llvm_arg (s,i,t,arglist) expr env =
		let (str,index,typ) = llvm_expr (s,i,t) expr env
			in
			match typ with
				|Struct(s)->
					let (s2,i2,t2) = llvm_bitcast_i8 (str,index,typ) (index2str (index)) typ env
						in (s2,i2,t2,arglist^(type2string3 typ) ^" "^ s2^", ")
				|_->(str,index,typ,arglist ^ (type2string3 typ) ^ " " ^ str^", ")
and llvm_arg_nc (s,i,t,arglist) expr env =
		let (str,index,typ) = llvm_expr (s,i,t) expr env
			in
			match typ with
				|Struct(s)->
					let (s2,i2,t2) = llvm_bitcast_i8 (str,index,typ) (index2str (index)) typ env
						in (s2,i2,t2,arglist^(type2string3 typ) ^" "^ s2)
				|_->(str,index,typ,arglist ^ (type2string3 typ) ^ " " ^ str)
and llvm_expr (str, index, typ) expression env =
		match expression with
		 Expr_const(konstant) ->
			begin match konstant with
				Const_str(s)->
					let (def,_,_,_) = env in
					let (func_name,_) = def.func_decl in
						print_endline("@.str." ^ func_name ^ "." ^ string_of_int(index) ^ " = private constant
[100 x i8] c" ^ (str2str s) ^ ", align 1");
						("getelementptr inbounds ([100 x i8]* @.str." ^ func_name ^ "." ^string_of_int(index),
index, String)
				|_->(konstant2string_helper konstant, index, konstant2type konstant)
			end
		|Expr_func(identifier, exprs) ->
			let (s,i,t,arglist) = llvm_args (str,index,typ,
					(
					 match exprs with
					 [] -> "  "
					|_ -> ""
					)
				) exprs env in
					llvm_call (s,i,t) (function2type identifier env) identifier arglist
		|Expr_op(unary_op, expr) ->
			let new_sit = llvm_expr (str,index,typ) expr env in
			let (new_s, new_i, new_t) = new_sit in
				begin
					match unary_op with
					 "-" ->
						begin
							match new_t with
							 Int -> llvm_sub new_sit "0" new_s
							|Double -> llvm_fsub new_sit "0.0" new_s
							|_ -> undef "Unary operator - only support Int and Double"; new_sit
						end
					|"!" ->
						begin
							match typ with
							 Int ->	llvm_zext (llvm_icmp new_sit "eq" new_s "0")
							|_ -> undef "Unary operator - only support Int"; new_sit
						end
				end
		|Expr_binop(expr1, binop, expr2) ->
			let (s1,i1,t1) = llvm_expr (str,index,typ) expr1 env in
				begin
					match binop with
					 "*" ->
						let (s2,i2,t2) = llvm_expr (s1,i1,t1) expr2 env in
						begin
							match (t1,t2) with
							 (Int, Int)-> llvm_mul (s2,i2,t2) s1 s2
							|(Int, Double) ->
```

```
                                        let (s3,i3,t3) = llvm_sitofp (s2,i2,t2) s1 in llvm_fmul (s3,i3,t3)
s2 s3
                                |(Double, Int) ->
                                        let (s3,i3,t3) = llvm_sitofp (s2,i2,t2) s2 in llvm_fmul (s3,i3,t3)
s1 s3
                                |(Double,Double) -> llvm_fmul (s2,i2,t2) s1 s2
                        end
                |"/" ->
                        let (s2,i2,t2) = llvm_expr (s1,i1,t1) expr2 env in
                        begin
                                match (t1,t2) with
                                 (Int, Int)-> llvm_sdiv (s2,i2,t2) s1 s2
                                |(Int, Double) ->
                                        let (s3,i3,t3) = llvm_sitofp (s2,i2,t2) s1 in llvm_fdiv (s3,i3,t3)
s2 s3
                                |(Double, Int) ->
                                        let (s3,i3,t3) = llvm_sitofp (s2,i2,t2) s2 in llvm_fdiv (s3,i3,t3)
s1 s3
                                |(Double,Double) -> llvm_fdiv (s2,i2,t2) s1 s2
                        end
                |"%" ->
                        let (s2,i2,t2) = llvm_expr (s1,i1,t1) expr2 env in
                        begin
                                match (t1,t2) with
                                 (Int, Int)-> llvm_srem (s2,i2,t2) s1 s2
                                |(Int, Double) ->
                                        let (s3,i3,t3) = llvm_sitofp (s2,i2,t2) s1 in llvm_frem (s3,i3,t3)
s2 s3
                                |(Double, Int) ->
                                        let (s3,i3,t3) = llvm_sitofp (s2,i2,t2) s2 in llvm_frem (s3,i3,t3)
s1 s3
                                |(Double,Double) -> llvm_frem (s2,i2,t2) s1 s2
                        end
                |"+" ->
                        let (s2,i2,t2) = llvm_expr (s1,i1,t1) expr2 env in
                        begin
                                match (t1,t2) with
                                 (Int, Int)-> llvm_add (s2,i2,t2) s1 s2
                                |(Int, Double) ->
                                        let (s3,i3,t3) = llvm_sitofp (s2,i2,t2) s1 in llvm_fadd (s3,i3,t3)
s2 s3
                                |(Double, Int) ->
                                        let (s3,i3,t3) = llvm_sitofp (s2,i2,t2) s2 in llvm_fadd (s3,i3,t3)
s1 s3
                                |(Double,Double) -> llvm_fadd (s2,i2,t2) s1 s2
                        end
                |"-" ->
                        let (s2,i2,t2) = llvm_expr (s1,i1,t1) expr2 env in
                        begin
                                match (t1,t2) with
                                 (Int, Int)-> llvm_sub (s2,i2,t2) s1 s2
                                |(Int, Double) ->
                                        let (s3,i3,t3) = llvm_sitofp (s2,i2,t2) s1 in llvm_fsub (s3,i3,t3)
s2 s3
                                |(Double, Int) ->
                                        let (s3,i3,t3) = llvm_sitofp (s2,i2,t2) s2 in llvm_fsub (s3,i3,t3)
s1 s3
                                |(Double,Double) -> llvm_fsub (s2,i2,t2) s1 s2
                        end
                |"<" ->
                        let (s2,i2,t2) = llvm_expr (s1,i1,t1) expr2 env in
                        begin
                                match (t1,t2) with
                                 (Int,Int) -> (llvm_icmp (s2,i2,t2) "slt" s1 s2)
                                |(Int,Double) ->
                                        let (s3,i3,t3) = llvm_sitofp (s2,i2,t2) s1 in llvm_fcmp (s3,i3,t3)
"olt" s2 s3
                                |(Double,Int) ->
                                        let (s3,i3,t3) = llvm_sitofp (s2,i2,t2) s2 in llvm_fcmp (s3,i3,t3)
```

```
"olt" s1 s3
                |(Double,Double) -> llvm_fcmp (s2,i2,t2) "olt" s1 s2
        end
        ;llvm_zext (index2str (i2+1),i2+1,Int)
    |">" ->
        let (s2,i2,t2) = llvm_expr (s1,i1,t1) expr2 env in
        begin
                match (t1,t2) with
                 (Int,Int) -> (llvm_icmp (s2,i2,t2) "sgt" s1 s2)
                |(Int,Double) ->
                        let (s3,i3,t3) = llvm_sitofp (s2,i2,t2) s1 in llvm_fcmp (s3,i3,t3)
"ogt" s2 s3
                |(Double,Int) ->
                        let (s3,i3,t3) = llvm_sitofp (s2,i2,t2) s2 in llvm_fcmp (s3,i3,t3)
"ogt" s1 s3
                |(Double,Double) -> llvm_fcmp (s2,i2,t2) "ogt" s1 s2
        end
        ;llvm_zext (index2str (i2+1),i2+1,Int)
    |"<=" ->
        let (s2,i2,t2) = llvm_expr (s1,i1,t1) expr2 env in
        begin
                match (t1,t2) with
                 (Int,Int) -> (llvm_icmp (s2,i2,t2) "sle" s1 s2)
                |(Int,Double) ->
                        let (s3,i3,t3) = llvm_sitofp (s2,i2,t2) s1 in llvm_fcmp (s3,i3,t3)
"ole" s2 s3
                |(Double,Int) ->
                        let (s3,i3,t3) = llvm_sitofp (s2,i2,t2) s2 in llvm_fcmp (s3,i3,t3)
"ole" s1 s3
                |(Double,Double) -> llvm_fcmp (s2,i2,t2) "ole" s1 s2
        end
        ;llvm_zext (index2str (i2+1),i2+1,Int)
    |">=" ->
        let (s2,i2,t2) = llvm_expr (s1,i1,t1) expr2 env in
        begin
                match (t1,t2) with
                 (Int,Int) -> (llvm_icmp (s2,i2,t2) "sge" s1 s2)
                |(Int,Double) ->
                        let (s3,i3,t3) = llvm_sitofp (s2,i2,t2) s1 in llvm_fcmp (s3,i3,t3)
"oge" s2 s3
                |(Double,Int) ->
                        let (s3,i3,t3) = llvm_sitofp (s2,i2,t2) s2 in llvm_fcmp (s3,i3,t3)
"oge" s1 s3
                |(Double,Double) -> llvm_fcmp (s2,i2,t2) "oge" s1 s2
        end
        ;llvm_zext (index2str (i2+1),i2+1,Int)
    |"==" ->
        let (s2,i2,t2) = llvm_expr (s1,i1,t1) expr2 env in
        begin
                match (t1,t2) with
                 (Int,Int) -> (llvm_icmp (s2,i2,t2) "eq" s1 s2)
                |(Int,Double) ->
                        let (s3,i3,t3) = llvm_sitofp (s2,i2,t2) s1 in llvm_fcmp (s3,i3,t3)
"oeq" s2 s3
                |(Double,Int) ->
                        let (s3,i3,t3) = llvm_sitofp (s2,i2,t2) s2 in llvm_fcmp (s3,i3,t3)
"oeq" s1 s3
                |(Double,Double) -> llvm_fcmp (s2,i2,t2) "oeq" s1 s2
        end
        ;llvm_zext (index2str (i2+1),i2+1,Int)
    |"!=" ->
        let (s2,i2,t2) = llvm_expr (s1,i1,t1) expr2 env in
        begin
                match (t1,t2) with
                 (Int,Int) -> (llvm_icmp (s2,i2,t2) "nq" s1 s2)
                |(Int,Double) ->
                        let (s3,i3,t3) = llvm_sitofp (s2,i2,t2) s1 in llvm_fcmp (s3,i3,t3)
"onq" s2 s3
                |(Double,Int) ->
```

```
                                        let (s3,i3,t3) = llvm_sitofp (s2,i2,t2) s2 in llvm_fcmp (s3,i3,t3)
"onq" s1 s3
                                    |(Double,Double) -> llvm_fcmp (s2,i2,t2) "onq" s1 s2
                                end
                                ;llvm_zext (index2str (i2+1),i2+1,Int)
                            |"&&" ->
                                begin
                                    match t1 with
                                     Int -> let (s2,i2,t2) = llvm_alloca (s1,i1,t1) Int in
                                            let (s3,i3,t3) = llvm_icmp (s2,i2,t1) "eq" s1 "0" in
                                            llvm_bri1 s3 (label i3 2) (label i3 0);
                                            print_endline("");
                                            print_endline((label i3 0) ^ ":");
                                            let (s4,i4,t4) = llvm_expr (s3,i3,t3) expr2 env in
                                            let (s5,i5,t5) = llvm_icmp (s4,i4,t4) "eq" s4 "0" in
                                            llvm_bri1 s5 (label i3 2) (label i3 1);
                                            print_endline("");
                                            print_endline((label i3 1) ^ ":");
                                            llvm_store Int "1" (index2str i2);
                                            llvm_br (label i3 3);
                                            print_endline("");
                                            print_endline((label i3 2) ^ ":");
                                            llvm_store Int "0" (index2str i2);
                                            llvm_br (label i3 3);
                                            print_endline("");
                                            print_endline((label i3 3) ^ ":");
                                            llvm_load (s5,i5,t5) Int s2
                                end
                            |"||" ->
                                begin
                                    match t1 with
                                     Int -> let (s2,i2,t2) = llvm_alloca (s1,i1,t1) Int in
                                            let (s3,i3,t3) = llvm_icmp (s2,i2,t1) "eq" s1 "1" in
                                            llvm_bri1 s3 (label i3 2) (label i3 0);
                                            print_endline("");
                                            print_endline((label i3 0) ^ ":");
                                            let (s4,i4,t4) = llvm_expr (s3,i3,t3) expr2 env in
                                            let (s5,i5,t5) = llvm_icmp (s4,i4,t4) "eq" s4 "1" in
                                            llvm_bri1 s5 (label i3 2) (label i3 1);
                                            print_endline("");
                                            print_endline((label i3 1) ^ ":");
                                            llvm_store Int "0" (index2str i2);
                                            llvm_br (label i3 3);
                                            print_endline("");
                                            print_endline((label i3 2) ^ ":");
                                            llvm_store Int "1" (index2str i2);
                                            llvm_br (label i3 3);
                                            print_endline("");
                                            print_endline((label i3 3) ^ ":");
                                            llvm_load (s5,i5,t5) Int s2
                            end
                        end
                |Expr_lval(lval) ->
                        begin
                            match lval with
                             Lval_id(identifier) ->
                                begin match (identifier2type identifier env) with
                                    |Struct(s)-> llvm_bitcast_struct (str,index,typ) ("%"^identifier)
(identifier2type identifier env) env
                                    |_->llvm_load (str,index,typ) (identifier2type identifier env)
(identifier2addr identifier env)
                                end
                            |Lval_struct(identifier, member) ->
                                let (s1,i1,final_typ) = llvm_getelementptr lval index env in
                                    print_endline("");
                                    begin match final_typ with
                                    Struct(s)-> llvm_bitcast_struct (s1,i1,final_typ) s1 final_typ env
                                    |_->llvm_load (s1, i1,final_typ) final_typ s1
                                    end
```

```
|Lval_arr(identifier, expr) ->
        let (s1,i1,t1) = llvm_getelementptr lval index env in
        print_endline("");
        llvm_load (s1, i1,t1) t1 s1

    end
|Expr_lval_op(lval,op) ->
    begin
        match lval with
         Lval_id(identifier) ->
            let (s1,i1,t1) = llvm_load (str,index,typ) (identifier2type identifier env)
(identifier2addr identifier env) in
                begin match op with
                 "++" -> let (s2,i2,t2)= llvm_add (s1,i1,t1) s1 "1" in
                     llvm_store t2 s2 (identifier2addr identifier env);
                     (s2,i2,t2)
                |"--" -> let (s2,i2,t2)= llvm_sub (s1,i1,t1) s1 "1" in
                     llvm_store t2 s2 (identifier2addr identifier env);
                     (s2,i2,t2)
                end
    end
|Expr_paren(expr) -> llvm_expr (str,index,typ) expr env
|Expr_asgop(lval, op, expr) ->
    begin match op with
        "=" ->
            begin
                match lval with
                 Lval_id(identifier) ->
                    begin match (identifier2type identifier env) with
                        String ->
                            let (s2,i2,t2) = llvm_expr (str,index,typ)
expr env in
                            let (s3,i3,t3) = llvm_bitcast_str (s2,i2,t2)
identifier in
                            print_endline("\t"^(index2str (i3+1))^" =
call i8* @strcpy(i8* noalias " ^ s3 ^", i8* noalias " ^ s2^", i64 0, i64 0)) nounwind");
                            ((index2str (i3+1)),(i3+1),String)
                        |_->
                            let (s2,i2,t2) = llvm_expr (str,index,typ)
expr env in
                            begin match ((identifier2type identifier
env),t2) with
                             (Int, Double) -> let
(s3,i3,t3)=llvm_fptosi (s2,i2,t2) s2 in
                                llvm_store (identifier2type
identifier env) s3 (identifier2addr identifier env); (s3,i3,t3)
                            |(Double, Int) -> let
(s3,i3,t3)=llvm_sitofp (s2,i2,t2) s2 in
                                llvm_store (identifier2type
identifier env) s3 (identifier2addr identifier env); (s3,i3,t3)
                            |_ ->
                                llvm_store (identifier2type
identifier env) s2 (identifier2addr identifier env); (s2,i2,t2)
                            end
                    end
                |Lval_struct(identifier, member) ->
                    let (s2,i2,t2) = llvm_expr (str,index,typ) expr env in
                    let (s3,i3,final_type) = llvm_getelementptr lval i2 env in
                        print_endline("");
                        begin
                            match final_type with
                            Struct(s)-> let (_,e,_,_) =env in
                            let (s4,i4,t4) = llvm_bitcast_struct
(s3,i3,final_type) s3 final_type env in
                            print_endline("\t call  void
@llvm.memcpy.i64(i8* "^s4^", i8* "^s2^", i64 " ^ string_of_int(Utils.sizeof e final_type) ^", i32 1)");(s4,i4,t4)
                            |_->
                                begin match (final_type,t2)
with
```

```
                                                                                          (Int, Double)-> let
(s4,i4,t4) = llvm_fptosi (s3,i3,final_type) s2 in
          llvm_store final_type s4 s3;  (s4, i4, t4)
                                                                                          |(Double, Int)->
let (s4,i4,t4) = llvm_sitofp (s3,i3,final_type) s2 in
          llvm_store final_type s4 s3;  (s4, i4, t4)
                                                                                          |_-> llvm_store
final_type s2 s3;  (s3, i3, final_type)
                                                                                  end
                                                              end
                                                    |Lval_arr(identifier, expr2) ->
                                                              let (s2,i2,t2) = llvm_expr (str,index,typ) expr env in
                                                              let (s3,i3,final_type) = llvm_getelementptr lval i2 env in
                                                                        print_endline("");
                                                                        llvm_store final_type s2 s3;
                                                                        (s3, i3, final_type)
                                          end
                                |"+=" ->
                                          begin
                                                    match lval with
                                                     Lval_id(identifier) ->
                                                              begin match (identifier2type identifier env) with
                                                                        String->
                                                                                  let (s2,i2,t2) = llvm_expr (str,index,typ)
expr env in
                                                                                  let (s3,i3,t3) = llvm_bitcast_str (s2,i2,t2)
identifier in
                                                                                  print_endline("\t"^(index2str (i3+1))^" =
call i8* @strcat(i8* noalias " ^ s3 ^", i8* noalias " ^ s2^", i64 0, i64 0)) nounwind");
                                                                                  ((index2str (i3+1)),(i3+1),String)
                                                                        |_->
                                                                                  let (s2,i2,t2) = llvm_load (str,index,typ)
(identifier2type identifier env) (identifier2addr identifier env) in
                                                                                  let (s3,i3,t3) = llvm_expr (s2,i2,t2) expr
env in
                                                                                  let (s4,i4,t4) =
                                                                                  begin
                                                                                            match (identifier2type
identifier env) with
                                                                                             Int -> llvm_add (s3,i3,t3) s2
s3
                                                                                            |Double -> llvm_fadd
(s3,i3,t3) s2 s3
                                                                                  end
                                                                                  in llvm_store (identifier2type identifier
env) s4 (identifier2addr identifier env); (s4,i4,t4)
                                                              end
                                                    |Lval_struct(identifier,member)->
                                                              let (s2,i2,t2) = llvm_expr (str,index,typ) expr env in
                                                              let (s3,i3,t3) = llvm_getelementptr lval i2 env in
                                                                        let (s4,i4,t4) =
                                                                        begin
                                                                                  match t3 with
                                                                                   Int -> llvm_add (s2,i2,t2) s2 s3
                                                                                  |Double -> llvm_fadd (s2,i2,t2) s2 s3
                                                                        end
                                                                        in
                                                                        print_endline("");
                                                                        llvm_store t4 s4 s3;
                                                                        (s4,i4,t4)
                                          end
                                |"-=" ->
                                          begin
                                                    match lval with
                                                     Lval_id(identifier) ->
                                                              let (s2,i2,t2) = llvm_load (str,index,typ) (identifier2type
```

```
identifier env) (identifier2addr identifier env) in
                                        let (s3,i3,t3) = llvm_expr (s2,i2,t2) expr env in
                                        let (s4,i4,t4) =
                                        begin
                                              match (identifier2type identifier env) with
                                               Int -> llvm_sub (s3,i3,t3) s2 s3
                                              |Double -> llvm_fsub (s3,i3,t3) s2 s3
                                        end
                                        in llvm_store (identifier2type identifier env) s4 (identifier2addr
identifier env); (s4,i4,t4)
                              end
                        |"*=" ->
                              begin
                                   match lval with
                                    Lval_id(identifier) ->
                                        let (s2,i2,t2) = llvm_load (str,index,typ) (identifier2type
identifier env) (identifier2addr identifier env) in
                                        let (s3,i3,t3) = llvm_expr (s2,i2,t2) expr env in
                                        let (s4,i4,t4) =
                                        begin
                                              match (identifier2type identifier env) with
                                               Int -> llvm_mul (s3,i3,t3) s2 s3
                                              |Double -> llvm_fmul (s3,i3,t3) s2 s3
                                        end
                                        in llvm_store (identifier2type identifier env) s4 (identifier2addr
identifier env); (s4,i4,t4)
                              end
                        |"/=" ->
                              begin
                                   match lval with
                                    Lval_id(identifier) ->
                                        let (s2,i2,t2) = llvm_load (str,index,typ) (identifier2type
identifier env) (identifier2addr identifier env) in
                                        let (s3,i3,t3) = llvm_expr (s2,i2,t2) expr env in
                                        let (s4,i4,t4) =
                                        begin
                                              match (identifier2type identifier env) with
                                               Int -> llvm_sdiv (s3,i3,t3) s2 s3
                                              |Double -> llvm_fdiv (s3,i3,t3) s2 s3
                                        end
                                        in llvm_store (identifier2type identifier env) s4 (identifier2addr
identifier env); (s4,i4,t4)
                              end
                        |"%=" ->
                              begin
                                   match lval with
                                    Lval_id(identifier) ->
                                        let (s2,i2,t2) = llvm_load (str,index,typ) (identifier2type
identifier env) (identifier2addr identifier env) in
                                        let (s3,i3,t3) = llvm_expr (s2,i2,t2) expr env in
                                        let (s4,i4,t4) =
                                        begin
                                              match (identifier2type identifier env) with
                                               Int -> llvm_srem (s3,i3,t3) s2 s3
                                              |Double -> llvm_frem (s3,i3,t3) s2 s3
                                        end
                                        in llvm_store (identifier2type identifier env) s4 (identifier2addr
identifier env); (s4,i4,t4)
                              end
                    end


and llvm_func_stack_decls func_decls func_name=
      match func_decls with
       []->()
      |head::tail -> llvm_alloc_func_decl head func_name;
                              llvm_func_stack_decls tail func_name;

and llvm_alloc_func_decl func_decl func_name=
```

```
        match func_decl with
         Decl(typ, decls) ->
                        let llvm_alloc_func_decl_one declarator =
                            begin match declarator with
                                Declarator(identifier, None) ->
                                begin match typ with
                                    String -> print_endline("\t%" ^ identifier ^ " = alloca [100 x i8]")
                                    |Struct(s) -> print_endline("\t%" ^ identifier ^ " = alloca %struct."^s)
                                    |_ -> print_endline("\t%" ^ identifier ^ " = alloca " ^ type2string typ)
                                end
                                | Declarator(identifier, Some(init_val)) ->
                                begin match typ with
                                    String ->
                                        print_endline("@.str." ^ func_name ^ "." ^ identifier ^ " =
private constant [100 x i8] c" ^ (match init_val with Init_const_expr(kStrExp) -> konstant2string kStrExp) ^ ", align 1");
                                        print_endline("\t%" ^ identifier ^ "0 = alloca [100 x i8]");
                                        print_endline("\t%" ^ identifier ^ " = bitcast [100 x i8]* %" ^
identifier ^ "0 to i8*");

                                        print_endline("\tcall void @llvm.memcpy.i64(i8* %" ^
identifier ^ ", i8* getelementptr inbounds ([100 x i8]* @.str." ^ func_name ^"."^ identifier ^ ", i64 0, i64 0), i64 100, i32 1)");
                                    |Struct(s) ->
                                        print_endline("\t%" ^ identifier ^ " = alloca %struct." ^ s);
                                    |_ ->
                                        print_endline("\t%" ^ identifier ^ " = alloca " ^ type2string typ);
                                        begin match init_val with
                                            Init_const_expr(konstant_expr) ->
                                                llvm_store typ (konstant2string
konstant_expr) ("%"^ identifier);

                                            |Init_const_expr_list(_) -> print_string("")
                                        end
                                    end
                                end
                        in List.iter llvm_alloc_func_decl_one decls
        |Decl_struct(_,_) -> print_string("")


and konstant2string konstant_expr =
        match konstant_expr with
            Const_expr(konstant) -> konstant2string_helper konstant
            |Const_expr_neg(konstant) -> "-" ^ (konstant2string_helper konstant)
and konstant2string_helper konstant =
        match konstant with
         Const_int(i) -> string_of_int(i)
        |Const_dbl(d) -> string_of_float(d)
        |Const_str(s) -> str2str(s)
and konstant2type konstant =
        match konstant with
         Const_int(i) -> Int
        |Const_dbl(d) -> Double
        |Const_str(s) -> String
(* store type value, type* addr, align alignment*)
and llvm_store typ value addr =
        print_endline("\tstore " ^ type2string typ ^ " " ^ value ^ ", " ^ type2string typ ^ "* " ^ addr ^ ", align " ^ type2align typ);
and llvm_bitcast_str (str, index, typ) identifier =
        print_endline("\t%reg" ^ string_of_int(index+1) ^ " = bitcast [100 x i8]* %" ^ identifier ^ "0 to i8*"); (index2str (index+1),
index+1, String)
and llvm_bitcast_struct (str, index, typ) identifier str_type env=
        print_endline("\t%reg" ^ string_of_int(index+1) ^ " = bitcast "^(type2string str_type) ^" "^ identifier ^ " to i8*"); (index2str
(index+1), index+1, str_type)
and llvm_bitcast_i8 (str, index, typ) identifier str_type env=
        print_endline("\t%reg" ^ string_of_int(index+1) ^ " = bitcast i8* "^ identifier ^ " to "^(type2string str_type)); (index2str (index+1),
index+1, str_type)
and llvm_load (str, index, typ) load_typ addr=
        print_endline("\t%reg" ^ string_of_int(index+1) ^ " = load " ^ type2string load_typ ^ "* " ^ addr ^ ", align " ^ type2align
load_typ);(index2str (index+1), index+1, load_typ)
and llvm_add (str, index, typ) op1 op2 =
        print_endline("\t%reg" ^ string_of_int(index+1) ^ " = add nsw " ^ type2string typ ^ " " ^ op1 ^ ", " ^ op2); (index2str (index+1),
index+1, Int)
and llvm_fadd (str, index, typ) op1 op2 =
```

```
        print_endline("\t%reg" ^ string_of_int(index+1) ^ " = fadd " ^ type2string typ ^ " " ^ op1 ^ ", " ^ op2); (index2str (index+1),
index+1, Double)
and llvm_sub (str, index, typ) op1 op2 =
        print_endline("\t%reg" ^ string_of_int(index+1) ^ " = sub nsw " ^ type2string typ ^ " " ^ op1 ^ ", " ^ op2); (index2str (index+1),
index+1, Int)
and llvm_fsub (str, index, typ) op1 op2 =
        print_endline("\t%reg" ^ string_of_int(index+1) ^ " = fsub " ^ type2string typ ^ " " ^ op1 ^ ", " ^ op2); (index2str (index+1),
index+1, Double)
and llvm_icmp (str, index, typ) cond op1 op2 =
        print_endline("\t%reg" ^ string_of_int(index+1) ^ " = icmp " ^ cond ^ " "^type2string typ ^ " " ^ op1 ^ ", " ^ op2); (index2str
(index+1), index+1, Int)
and llvm_fcmp (str, index, typ) cond op1 op2 =
        print_endline("\t%reg" ^ string_of_int(index+1) ^ " = fcmp " ^ cond ^ " "^type2string typ ^ " " ^ op1 ^ ", " ^ op2); (index2str
(index+1), index+1, Int)
and llvm_mul (str, index, typ) op1 op2 =
        print_endline("\t%reg" ^ string_of_int(index+1) ^ " = mul nsw " ^ type2string typ ^ " " ^ op1 ^ ", " ^ op2); (index2str (index+1),
index+1, Int)
and llvm_fmul (str, index, typ) op1 op2 =
        print_endline("\t%reg" ^ string_of_int(index+1) ^ " = fmul " ^ type2string typ ^ " " ^ op1 ^ ", " ^ op2); (index2str (index+1),
index+1, Double)
and llvm_sdiv (str, index, typ) op1 op2 =
        print_endline("\t%reg" ^ string_of_int(index+1) ^ " = sdiv " ^ type2string typ ^ " " ^ op1 ^ ", " ^ op2); (index2str (index+1),
index+1, Int)
and llvm_fdiv (str, index, typ) op1 op2 =
        print_endline("\t%reg" ^ string_of_int(index+1) ^ " = fdiv " ^ type2string typ ^ " " ^ op1 ^ ", " ^ op2); (index2str (index+1),
index+1, Double)
and llvm_srem (str, index, typ) op1 op2 =
        print_endline("\t%reg" ^ string_of_int(index+1) ^ " = srem " ^ type2string typ ^ " " ^ op1 ^ ", " ^ op2); (index2str (index+1),
index+1, Int)
and llvm_frem (str, index, typ) op1 op2 =
        print_endline("\t%reg" ^ string_of_int(index+1) ^ " = frem " ^ type2string typ ^ " " ^ op1 ^ ", " ^ op2); (index2str (index+1),
index+1, Double)
and llvm_zext (str, index, typ) =
        print_endline("\t%reg" ^ string_of_int(index+1) ^ " = zext i1 " ^ str ^ " to i32"); (index2str (index+1), index+1, Int)
and llvm_trunc (str, index, typ) =
        print_endline("\t%reg" ^ string_of_int(index+1) ^ " = trunc i32 " ^ str ^ " to i1"); (index2str (index+1), index+1, Int)
and llvm_fptosi (str, index, typ) op =
        print_endline("\t%reg" ^ string_of_int(index+1) ^ " = fptosi double " ^ op ^ " to i32" ); (index2str (index+1), index+1, Int)
and llvm_sitofp (str, index, typ) op =
        print_endline("\t%reg" ^ string_of_int(index+1) ^ " = sitofp i32 " ^ op ^ " to double" ); (index2str (index+1), index+1, Double)
and llvm_alloca (str, index, typ) alloca_typ =
        print_endline("\t%reg" ^ string_of_int(index+1) ^ " = alloca " ^ type2string alloca_typ); (index2str (index+1), index+1, alloca_typ)
and llvm_call (str, index, typ) call_typ identifier arglist =
        print_endline("\t%reg" ^ string_of_int(index+1) ^ " = call " ^ (type2string call_typ) ^ " @" ^ identifier ^"(" ^ arglist ^ ")
nounwind"); (index2str (index+1), index+1, call_typ)
and llvm_bri1 str label1 label0 =
        print_endline("\tbr i1 " ^ str ^ ", label %" ^ label1 ^ ", label %" ^ label0);
and llvm_br label =
        print_endline("\tbr label %" ^ label);
and type2align typ =
        match typ with
         Int -> "1"
        |Double -> "1"
        |String -> "1"
        |_ ->"1"
(*
 define type @identifier(type %identifier, type %identifier) nounwind {
 *)
and llvm_func_def func_def =
        let (typ, func_decl) = (func_def.type_spec, func_def.func_decl) in
                print_string("define ");
                (match typ with
                Struct(_) -> print_string("void");
                |_ -> print_string(type2string typ););
                print_string(" @");
                let (func_name, parameters) = func_decl in
                        print_string(func_name);
                        llvm_func_parameters parameters typ;
                        print_endline(" nounwind {");
```

```
                              llvm_func_parameters_stack parameters typ;
                              begin match typ with
                                    Struct(s) -> print_endline("\t %agg.result1 = bitcast %struct."^s^"* %agg.result to i8*");
                                    |_-> print_endline("\t%retval = alloca " ^ type2string typ);
                              end

and llvm_func_parameters_stack parameters typ= print_endline("entry:");
      match parameters with
             [] -> print_string("")
            | _ -> List.iter llvm_func_parameter_stack parameters

and type2string typ =
      match typ with
             Int -> "i32"
            |Double -> "double"
            |String -> "i8*"
            |Struct(name) -> "%struct." ^ name ^ "*"
            |List(t) -> "[100 x " ^ type2string t ^ "]"
and type2string3 typ =
      match typ with
             Int -> "i32"
            |Double -> "double"
            |String -> "i8*"
            |Struct(name) -> "%struct." ^ name ^ "* byval"
and type2string2 typ =
      match typ with
             Int -> "i32"
            |Double -> "double"
            |String -> "i8*"
            |Struct(name) -> "%struct." ^ name ^ "*"
            |List(t) -> "[100 x " ^ type2string t ^ "]*"

and llvm_func_parameter_stack parameter =
      let (identifier, typ) = parameter in
            match typ with
            Struct(_) -> ();
            |_->print_endline("\t%" ^ identifier ^ " = alloca " ^ type2string typ);
            llvm_store typ ("%" ^ identifier^"_val") ("%" ^ identifier);

and llvm_func_parameters parameters typ= print_string("(");
      (match typ with
            Struct(_) -> print_string((type2string typ)^" noalias sret %agg.result, ")
            |_ -> ()
      );
      llvm_func_paras parameters;
      print_string(")");
and llvm_func_paras parameters =
      match parameters with
            [] -> ();
            |[head]->llvm_func_para_nc head;
            |head::tail -> llvm_func_para head; llvm_func_paras tail;

and llvm_func_para para =
      let (identifier, typ) = para in
            (match typ with
            Struct(s) -> print_string("%struct."^s^"* byval %"^identifier^", ");
            |_->print_string(type2string typ);
            print_string(" %" ^ identifier^"_val, "););
and llvm_func_para_nc para =
      let (identifier, typ) = para in
            (match typ with
            Struct(s) -> print_string("%struct."^s^"* byval %"^identifier);
            |_->print_string(type2string typ);
            print_string(" %" ^ identifier^"_val"););



(*
 @(identifier) = (init_or_not) (type_str_pre)init_val(type_str_post)
```

```
*)
and llvm_global_variables variables =
        List.iter llvm_variable_decl variables;

and llvm_variable_decl variable =
        let (name, typ, value) = variable in
        let (identifier, init_or_not, type_str_pre, type_str_post, init_val) =
                (
                        name,
                        (match value with
                                [] -> "common global"
                                | _ -> "global"
                        ),
                        (match typ with
                                Int -> "i32 "
                                |Double -> "double "
                                |String -> "[100 x i8] c"
                                | _ -> ""
                        ),
                        (match typ with
                                String -> ", align 1"
                                | _ -> ""
                        ),
                        (match value with
                        (* Default value *)
                                [] -> (match typ with
                                                Int -> "0"
                                                |Double -> "0.0e+00"
                                                |String -> "\"" ^ (char_00 100) ^ "\""
                                                |_ -> ""
                                        )
                                | _ -> (match typ with
                                                String -> str2str (List.hd value)
                                                |_ -> (List.hd value)
                                        )
                        )
                ) in print_endline("@" ^ identifier ^ " = " ^ init_or_not ^ " " ^ type_str_pre ^ init_val ^ type_str_post)

(* "\00"*n *)
and char_00 n = match n with
         0 -> ""
        | _ -> "\\00" ^ char_00 (n-1)
and index2str index = "%reg" ^ string_of_int(index)
and str2str str =
        let (final_str, length) =
        let lexbuf = Lexing.from_string str in
                String2string.string2string "" 0 lexbuf
                in
        "\"" ^ final_str
        ^ (char_00 (100 - length)) ^ "\""
and undef str=
        print_endline("undef: "^str);
and label vindex bindex = "v" ^ string_of_int(vindex) ^ "bb" ^ string_of_int(bindex)
and function2type identifier (def,env,stk,n) =
        match env.parent with
        |Some(parent) ->
                let ((str,typ),_) = List.find (fun ((s,_), args) -> s = identifier) parent.functions in typ

and identifier2type identifier (def,env,stk,n) =
        try
                let (str,typ,_) =List.find (fun (s,_,_) -> s =identifier) env.variables in
                        typ
                with Not_found ->
                        match env.parent with
                                Some(parent) -> identifier2type identifier (def,parent,stk,n)
and identifier2addr identifier (def,env,stk,n) =
        try
                let (str,typ,_) =List.find (fun (s,_,_) -> s =identifier) env.variables in
                        "%" ^ identifier
```

```
                    with Not_found ->
                              match env.parent with
                                        Some(parent) -> let (str2,typ2,_) = List.find (fun (s,_,_) -> s=identifier) parent.variables in "@"
^ identifier

and llvm_getelementptr lv i2 env=
          begin match lv with
          Lval_id(struct_id) ->
                    print_string("\t" ^ (index2str (i2+1)) ^ " = getelementptr inbounds " ^ (type2string2 (identifier2type struct_id env)) ^
" %" ^ struct_id ^ ", i32 0"); (index2str (i2+1), i2+1, identifier2type struct_id env)
          |Lval_struct(id,me) ->
                    let (ss,ii,tt)=llvm_getelementptr id i2 env in
                    let (def,sym,s,i)=env in
                              print_string(", i32 "^string_of_int((Utils.sequency sym tt me)-1)); (ss, ii, Utils.member2type tt me sym)
          |Lval_arr(id,expr) ->
                    let (s1,i1,t1)=llvm_expr (index2str i2, i2, Int) expr env in
                    let (s3,i3,t3)=llvm_getelementptr id i1 env in
                              begin
                              match t3 with
                              List(t4)->
                              print_string(", i32 "^s1);(s3,i3,t4)
                              end
          end
```

9.1.12. gltypes.h

```
struct point {
  double x;
  double y;
  double r;
  double g;
  double b;
  double vx;
  double vy;
};

struct shape {
  double size;
  double x;
  double y;
  double r;
  double g;
  double b;
  double vx;
  double vy;
  double theta;
  double omega;
};
```

9.1.13. glsupport.c (Tianliang Sun, Xinan Xu)

```
#include <GL/glut.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <assert.h>
#include <string.h>
#include <pthread.h>
#include "gltypes.h"
```

```
#include <math.h>
#include <sched.h>
#include <unistd.h>

#define MAX 100
#define dt 0.003
int pt_cnt = 0, shape_cnt = 0;

struct point pt_arr[MAX];
struct shape shape_arr[MAX];

int pop_point(){
		pt_cnt --;
		if(pt_cnt<0) pt_cnt=0;
}
int pop_shape(){
		shape_cnt --;
		if(shape_cnt<0) shape_cnt=0;
}
int add_point(struct point pt){
  pt_arr[pt_cnt] = pt;
  pt_cnt ++;
  assert(pt_cnt < MAX);
  return 0;
}

int add_shape(struct shape shp){
  shape_arr[shape_cnt] = shp;
  shape_cnt ++;
  assert(shape_cnt < MAX);
  return 0;
}

int display_points(){
  int i;
  glPointSize(5);
  glBegin(GL_POINTS);
  for(i = 0; i < pt_cnt; i++){
		glColor3d(pt_arr[i].r, pt_arr[i].g, pt_arr[i].b);
    glVertex2f(pt_arr[i].x, pt_arr[i].y);
  }
  glEnd();
  for(i = 0; i < pt_cnt; i++){
		pt_arr[i].x+=dt*pt_arr[i].vx;
		pt_arr[i].y+=dt*pt_arr[i].vy;
		if(pt_arr[i].x>1||pt_arr[i].x<-1)pt_arr[i].vx=-pt_arr[i].vx;
		if(pt_arr[i].y>1||pt_arr[i].y<-1)pt_arr[i].vy=-pt_arr[i].vy;
  }
  return 0;
}

#define c(x) cos((x)/180.*M_PI+shp.theta)
#define s(x) sin((x)/180.*M_PI+shp.theta)
int display_shapes(){
  int i;
  glBegin(GL_QUADS);
  for(i = 0; i < shape_cnt; i++){
    struct shape shp = shape_arr[i];
		glColor3d(shp.r, shp.g, shp.b);
    glVertex2f(shp.x + shp.size/2 * c(45.), shp.y + shp.size/2*s(45.));
    glVertex2f(shp.x + shp.size/2 * c(135.), shp.y + shp.size/2*s(135.));
    glVertex2f(shp.x + shp.size/2 * c(225.) , shp.y + shp.size/2*s(225.));
    glVertex2f(shp.x + shp.size/2 * c(315.), shp.y + shp.size/2*s(315.));
  }
  for(i = 0; i < shape_cnt; i++){
    struct shape *shp = &shape_arr[i];
		shp->x+=dt*shp->vx;
		shp->y+=dt*shp->vy;
		shp->theta+=dt*shp->omega;
```

```
                if(shp->x>1||shp->x<-1)shp->vx=-shp->vx;
                if(shp->y>1||shp->y<-1)shp->vy=-shp->vy;
  }
  glEnd();
  return 0;
}

int display(){
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glColor3f(0.0f, 0.0f, 0.0f);
  display_points();
  display_shapes();
  glutSwapBuffers();
  return 0;
}

int keyboard (unsigned char key, int x, int y){
                switch(key){
                                case 27:
                                                exit(0);
                                                break;
                                case 'q':
                                                exit(0);
                                                break;
                }
}
void* run_func(void* args){
  glutDisplayFunc(display);
  glutIdleFunc(display);
  glutKeyboardFunc(keyboard);
  glutMainLoop();
  return NULL;
}

int setup() {
  char *prog_name = "draw";
  int argc = 1;

  memset((void*)pt_arr, 0, sizeof(struct point)*MAX);
  memset((void*)shape_arr, 0, sizeof(struct point)*MAX);

  glutInit(&argc, &prog_name);
  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
  glutInitWindowSize(500, 500);
  glutInitWindowPosition(200, 200);
  glutCreateWindow("sample draw");
  glClearColor(1.0f, 1.0f, 1.0f, 0.0f);
  pthread_t th;
  pthread_create(&th,NULL,run_func,NULL);
  printf("Entering opengl mainloop...\n");
  printf("Press ESC to exit...\n");
  return 0;
}
int run(){
                return 0;
}
int wait(double seconds){
                unsigned int t=0;
                if(seconds<=0.)
                                t=10000000000;
                else
                                t=seconds*1000000.;
                usleep(t);
                return 0;
}
int byebye(){
                printf("Exiting in 3...\n");
                wait(1.0);
                printf("Exiting in 2...\n");
```

```
            wait(1.0);
            printf("Exiting in 1...\n");
            wait(1.0);
            exit(0);
}
```

9.2. Abstract Syntax Tree Test Suite

9.2.1. Test cases (Jingyi Guo)

Test -0-input

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

Test -0-output

intmain(){return-1;}

Test -1-input

```
int a = 1;
int b = 22;
int main(){
a = b;
}
```

Test -1-output

inta=1;intb=22;intmain(){a=b;}

Test -2-input

```
int a = 1;
int b = 22;
string c = "\n\"aaa";
int main(){
a = b;
}
```

Test -2-output

inta=1;intb=22;stringc="\n\"aaa";intmain(){a=b;}

Test -3-input

```
//test the data types and comment
int a = 1;// this is to test comment
double b=-0.000002;
double x=0.000002e-3;
string c="test";
list <double> d;
//struct hoo;
point e;
shape f;
image g;
int main(){
return -1;
}
```

```
/* This is to test the block
comment*/
```

Test -3-output

```
inta=1;doubleb=-2e-006;doublex=2e-
009;stringc="test";list<double>d;pointe;shapef;imageg;intmain(){return-1;}
```

Test -4-input

```
int main(){
39 + 3;
30 - 2;
30 * 2;
30 / 2;
31 % 3;
}
```

Test -4-output

```
intmain(){39+3;30-2;30*2;30/2;31%3;}
```

Test -5-input

Test -5-input

```
int main(){
int a;
int b;
int c;
int d;
a = (8 - 2) / 4;
b=1 + 2 * 3 - 4;
c=1 + 2 - 8 / 4;
return -1;
}
```

Test -5-output

```
intmain(){inta;intb;intc;intd;a=(8-2)/4;b=1+2*3-4;c=1+2-8/4;return-1;}
```

Test -6-input

```
int main(){
1 && 0;
1 || 0;
!0;
}
```

Test -6-output

```
intmain(){1&&0;1||0;!0;}
```

Test -7-input

```
int a;
int i;
int main(){
if ( i == 3){
break;
}
return a;
}
```

Test -7-output

```
inta;inti;intmain(){if(i==3){break;}returna;}
```

Test -8-input

```
int main(){
1 == 2;
1!= 2;
1 < 2;
1 <= 2
1 > 2;
1 >= 2;
}
```

Test -8-output

```
intmain(){1==2;1!=2;1<2;1<=2;1>2;1>=2;}
```

Test -9-input

```
int main(){
int a;
int b;
a = - 8 + - 3 * 2 + - (6 + 1 );
b = (8 - 2) / 4;
return -1;
}
```

Test -9-output

```
intmain(){inta;intb;a=-8+-3*2+-(6+1);b=(8-2)/4;return-1;}
```

Test -10-input

```
struct ho {
int x;
int y;
}
int main(){
struct ho ahoo;
return -1;
}
```

Test -10-output

```
structho{intx;inty;}intmain(){structhoahoo;return-1;}
```

Test -11-input

```
list<int> l;
int i, j;
int  main()
{
int a=3;
i > j;
for( i =1 ;i <= 5 ; i = i+1)
i = 1;
i = i + 1;
return  -1;
}
```

Test -11-output

```
list<int>l;inti,j;intmain(){inta=3;i>j;for(i=1;i<=5;i=i+1)i=1;i=i+1;return-1;}
```

Test -12-input

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

```
return a+b;
}
int main(){
c = add(3, 4);
return c;
}
```

Test -12-output

intadd(inta,intb){returna+b;}intmain(){c=add(3,4);returnc;}

Test -13-input

```
int x;
int printMultiple(double a, int b, string c, string d){
return b;
}
int main(){
        x=printMultiple(3.16, 17, "concise","x");
        return -1;
}
```

Test -13-output

intx;intprintMultiple(doublea,intb,stringc,stringd){returnb;}intmain(){x=printMultiple(3.14,17,"Pizza toppings","x");return-1;}

Test -14-input

```
int a;
int b;
int c=5;
int main(){
b=b++;
c=c--;
return -1;
}
```

Test -14-output

inta;intb;intc=5;intmain(){b=b++;c=c--;return-1;}

Test -15-input

```
int fac(int n){
if(n==0) return 1;
return fac(n-1);
}

int main(){
fac(1);
return -100;
}
```

Test -15-output

intfac(intn){if(n==0)return1;returnfac(n-1);}intmain(){fac(1);return-100;}

Test -16-input

```
struct ho {
int x;
int y;
}
```

```
int main(){
struct ho ahoo;
ahoo.x;
return -1;
}
```

Test -16-output

structho{intx;inty;}intmain(){structhoahoo;ahoo.x;return-1;}

Test -17-input

```
int a;
int i;
int main(){
if ( i != 3){
break;
}
return a;
}
```

Test -17-output

inta;inti;intmain(){if(i!=3){break;}returna;}

Test -18-input

```
int a;
int i;
int b;
int c;
int d;
int e;
int main(){
a+=5;
i-=5;
b*=5;
c/=5;
d%=5;
return -1;
}
```

Test -18-output

inta;inti;intb;intc;intd;inte;intmain(){a+=5;i-=5;b*=5;c/=5;d%=5;return-1;}

Test -19-input

```
int a,b;
int main(){
a=9;
b=10;
return -1;
}
```

Test -19-output

inta,b;intmain(){a=9;b=10;return-1;}

Test -20-input

```
list<int> a;
```

```
int main(){
            a[3]=2;
            return -1;
}
```
Test -20-output

list<int>a;intmain(){a[3]=2;return-1;}

9.2.2. auto_test.sh (Tianliang Sun, Jingyi)

```
#!/bin/bash
# automated test script for testing AST and parser structure

TEST_PROG="./ast_test"
PARSER_RESULT_FILE="tmp_result.txt"
TEST_CASE_DIR="../test_case"
NUM_OF_CASES=16

case_number=0
test_case_file=
test_result_file=
diff_str=
retval=

echo "start running test cases now ..."

while [ $case_number -lt $NUM_OF_CASES ]; do
   test_case_file="$TEST_CASE_DIR/test_case_$case_number.txt"
   test_result_file="$TEST_CASE_DIR/test_result_$case_number.txt"
   $TEST_PROG < $test_case_file > $PARSER_RESULT_FILE
   diff_str=$(diff $PARSER_RESULT_FILE $test_result_file)
   if [ $? -eq 0 ]; then
            echo "test case $case_number passed"
   else
            echo "test case $case_number failed, diff = $diff_str"
   fi
   case_number=$((case_number + 1))
done
```

9.3. Bytecode Test Suite (Xinan Xu)

9.3.1. Compiling Makefile

```
# Since we are transforming C code into bytecode using CAL,
# it will be compared to the result generated by llvm
# llvm version used is llvm-2.7
# platform is `uname -srmo`:
# Linux 3.10.17-gentoo x86_64 GNU/Linux
```

```
# Please use the script to build llvm-2.7
# !!!!!!you have to use gcc-4.5 to build llvm-2.7!!!!!

CAL=../cal

CLANG=./llvm/install/bin/llvm-gcc
LLVMDIS=llvm-dis
LLVMC=llc
LLVMAS=llvm-as

CFLAGS=-emit-llvm -O0

SRCS=$(wildcard *.c)
EXES=$(patsubst %.c, %-cal, $(SRCS))
LLEXES=$(patsubst %.c, %-ll, $(SRCS)) $(patsubst %.c, %-cal.ll, $(SRCS))

gl.o: ../glsupport.c
	gcc -c $< -o $@

%-cal: %.c $(CAL) gl.o
	rm -f $@.ll
	cat ../gltypes.h > $@.tmp0
	cat $< >> $@.tmp0
	$(CAL) < $@.tmp0 > $@.tmp
	rm -f $@.tmp0
	-grep "private constant" $@.tmp > $@.ll
	grep -v "private constant" $@.tmp  >> $@.ll
	rm -f $@.tmp
	$(LLVMAS) $@.ll -o $@.bc
	$(LLVMC) $@.bc -o $@.s
	rm -f $@.bc
	gcc $@.s gl.o -o $@ -lglut -lpthread
	rm -f $@.s



%-ll : %.c gl.o
	rm -f $@.ll
	cat ../gltypes.h > $@_.c
	cat $< >> $@_.c
	$(CLANG) $(CFLAGS) -c $@_.c -o $@.bc
	$(LLVMDIS) -f $@.bc
	$(LLVMC) $@.bc -o $@.s
	gcc $@.s gl.o -o $@ -lglut
	rm -f $@.bc $@.s


$(CAL):
	make -C ..

.PHONY: clean $(CAL) gl

clean:
	rm -rf *.s $(EXES) $(LLEXES) gl.o
	make -C .. clean
```

9.3.2. basic1.c

```
int main(){
		return 100;
}
```

9.3.3. basic2.c

```
int a=100;
int main(){
		return a;
}
```

9.3.4. basic3.c

```
int a=100;
double b;
int c;
double d=10.01;
int main(){
		return c;
}
```

9.3.5. basic4.c

```
int a=200;
double b;
int c;
double d=10.01;
int main(){
		int e=100;
		return e;
}
```

9.3.6. basic5.c

```
int a=200;
double b;
int c;
double d=10.01;
string s="abcdefg";
```

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

9.3.7. basic6.c

```
int a=200;
double b;
int c;
double d=10.01;
int main(){
		string s="\n\t\\";
		s+="aaaa";
		s="bbbb";
		return 0;
}
```

9.3.8. basic7.c

```
struct foo{
		int a;
		double b;
		string c;
};
struct foo func(struct foo x){
		return x;
}
int main(){
   struct foo foo2;
   struct foo foo1;
   foo1.b=foo2.b;
   return 0;
}
```

9.3.9. basic8.c

```
struct foo {
		int a;
		double b;
};

struct foofoo{
		int a;
		double b;
		struct foo c;
```

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

9.3.10. basic9.c

```
int main(){
	int i = 10;
	int sum = 0;
	for(i = 1; i < 10; i++) {
		if(sum > 10)
			return sum;
		sum+=i;
	}
	return sum;
}
```

9.3.11. basic10.c

```
struct foo {
	int a;
	double b;
	list<int> c;
};

struct foofoo{
	int a;
	double b;
	struct foo c;
};
int main(){
	return 0;
}
```

9.3.12. basic11.c

```
struct foo {
	int a;
	double b;
	list<int> c;
};

struct foofoo{
	int a;
```

```
	double b;
	struct foo c;
};
int main(){
	struct foo x;
	int c = 10;
	list<int> y;
	x.b=1.0;
	x.c[10] = 10;
	y[c] = 100;
	return c;
}
```

9.3.13. basic12.c

```
int fac(int n){
	if(n==1) return 1;
	return n*fac(n-1);
}

int main(){
	int i;
	i=fac(3);
	return i;
}
```

9.3.14. basic13.c

```
struct foo {
	int a;
	list<int> b;
};

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

9.3.15. basic14.c

```
struct foo{
	int a;
	int b;
};
int main(){
	struct foo x={1,2};
```

```
        return x.a;
}
```

9.3.16. basic15.c

```
int main() {
        int i;
        point pt;
        for(i = 0; i < 10; i++){
                pt.x = i + 0.1;
                pt.y = i - 0.1;
                add_point(pt);
        }
        return 0;
}
```

9.4. Demo Program

```
int i = 0, j = 0, size = 10;

struct point_or_shape {
  point pt;
  shape shp;
};

int add_point_or_shape(int  x, int y, struct point_or_shape pos){
  if(x == y || x == size - y - 1)
    add_shape(pos.shp);
  else
    add_point(pos.pt);
  return 0;
}

int main(){
  struct point_or_shape pos;
  point pt;
  shape shp;

  for(i = 0; i < size; i=i++){
    for(j = 0; j < size; j=j++){
      pt.x=0.2*j+0.1-1.0;
      pt.y=-0.2*i-0.1+1.0;
      pt.vx=pt.y+pt.x;
      pt.vy=pt.x-pt.y;
      pt.r=pt.x/2.0+0.5;
      pt.g=pt.y/2.0+0.5;
      pt.b=0.0;
      shp.size=0.2;
      shp.x=0.2*j+0.1-1.0;
```

```
    shp.y=-0.2*i-0.1+1.0;
    shp.vy=shp.x/2.0+shp.y;
    shp.vx=shp.y/2.0-shp.x;
    shp.r=shp.x/2.0+0.5;
    shp.g=shp.y/2.0+0.5;
    shp.b=1.0;
    shp.omega=1.0;
    pos.pt = pt;
    pos.shp = shp;
            wait(0.05);
    add_point_or_shape(j, i, pos);
   }
  }
  for(i=0;i<size*size;i=i++){
            wait(0.05);
          pop_shape();
          pop_point();
  }

  return 0;
}
```