CGC

C with Garbage Collector

Lieyang Chen, Tianze Huang, Zhuoxuan Li, Fanhao Zeng

COMS W4115 S21
Columbia University
04.25.2021

Contents

1 Introduction

The CGC programming language is a language built upon Micro C language, but comes with many handy syntax and features, and most importantly added class feature. In addition to the primitive data types similar to Micro C, CGC supports heap allocation and class object creation.

2 CGC Tutorial

2.1 Environment Setup

The environment we used is the docker provided by Professor.

Installation and testing Steps:

```
$ sudo apt -y install ocaml llvm llvm-runtime m4 opam cmake pkg-config
    ocamlbuild
$ opam init -y
$ opam switch -y create 4.11.1
$ eval $(opam env)
$ opam install -y llvm.10.0.0 utop
$ eval $(opam env)
$ cd CGC
$ make
```

2.2 Writing Programs

2.2.1 Memory Allocation

Array examples:

```
    int main() {
    int[] array;
    array = new int[5];
    print(array[0]);
    print(array[4]);
    return 0;
}
```

From the code above, the ”new” keyword in CGC help users allocated memory on the heap with a given type and size. Once the memory is allocated on the heap, a pointer to that block of memory will be returned. Note that a pointer is a expressed as ”type[]” in CGC, which is same as C. However, in CGC, any variable has to be declared first so that it could be initialized later. Namely, declaration and initialization cannot be in the same statement.

The code above also shows a built-in function ”print” in CGC. Unlike ”printf” in C, CGC does not require a format string.

2.2.2 Control Flow

While Loops

```
int main()
{
  while(true) {
  if (false) { break; }
  else { continue;} }
  for(i = 0; i < 1 ; ){ }
}
```

CGC supports most of control flow implementations in C, such as ”while”, ”for” and ”if” ”else” statements. We add ”continue” and ”break” as additional features to Micro C, therefore a more flexible control flow can be achieved.

Note the condition passed in while loop or any conditional statements has to be either ”true” or ”false”. Unlike C, CGC cannot cast integers to boolean values.

2.2.3 Functions

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

int main()
{
  int d;
  d = add(52, 10);
  print(d);
  return 0;
}
```

2.2.4 Comments

Single-line comments use double backslashes (//). Multi-line comments are denoted with a /* */ notation. For example:

```
int x = 3 // This is a single-line comment
/*
And this is a multi-line comment
```

```
*/
```

2.2.5 Classes

```
class Line {
    int x;
   constructor(int a){
        x = a;
    }
    int getX() {
      int a;
      a = this.x;
      return a;
    }
}


int main() {
    class Line e;
    int i;
    e = new Line(5);
   i = e.getX();
    print(i);
    return 0;
}
```

The ”class” in CGC is very similar to the ”class” in Java. It allows user to wrap as many members as possible inside a class object. Moreover, it can contain member functions so that class object can recall it. CGC also supports constructor so that class members can be initialized when they gets created. However, unlike Java, a constructor has to be explicitly written. A class object is created using ”new” statement because it is actually allocated on the heap. However, one drawback of CGC is that even if a constructor is provided, a class object identifier has to be declared first so that it can later be assigned with the pointer to the memory block of the class object on the heap.

3 Language Reference Manual

3.1 Introduction

The CGC programming language is a language built upon Micro C language,but comes with many handy syntax and features, and most importantly added class feature. In addition to the primitive data types similar to Micro C, CGC supports heap allocation and class object creation.

3.2 Lexical Conventions

3.2.1 Identifiers

- Valid identifiers maybe include ASCII letters, decimal digits and underscore.
- An valid identifier must begin with a letter and can not be a CGC keyword.

Examples:

```
// valid identifier
int studentId
void get_bar2() {}

// invalid identifier
int static = 3
void 2getId() {}
```

3.2.2 Operators

- int: +, -, *, /, ==, <, >, <=, >=, =, &&, ||, !
- char: +, -, *, /, ==, <, >, <=, >=, =, &&, ||, !
- float: +, -, *, /, <, >, <=, >=, =, ==
- Array: =, ==, [ ]

3.2.3 Keywords

- `for, if, else, elseif, return, int, float, char, void, const, class`
- Keywords are reserved identifiers that can not be used as variable names.

3.2.4 Literals

Literals represents strings or one of CGC's primitive types: int, float, char.

- int: any sequence of integers between 0 and 9
- float: a number in decimal form, a fraction, and an exponent
- char: a single character within single quotes
- string: a sequence of characters enclosed in single or double quotation marks

Examples:

```
// int literals
-5
0
130

// float literals
2.5
1e23

// char literals
'a'
'3'

// string literals
"Hi there!"
"When is iphone 13 coming out?"
```

3.2.5 Separators

- Parentheses are used to override the default precedence of expression evaluation.
- Semicolons are used to indicate the end of an expression.

Examples:

```
int x = (a + b) * c; // Using () to override default operator precedence

x = 3; y = x + 1; // Using ; to indicate the end of an expression

/*
Technically you can write all code in one line.
*/
```

3.2.6 Comments

- Single-line comments use double backslashes (//). Multi-line comments are denoted with a /* */ notation.

Examples:

```
int x = 3 // This is a single-line comment
/*
And this is a multi-line comment
```

```
*/
```

3.3 Data Types

CGC is a statically typed language, so all of the variables must be declared first before they can be used and all of them have a data type at compile time. CGC is also a strongly typed language, it does not support variable type coercion and implicit casts. These features enable CGC detects program errors easily in the compile time.

There are two kinds of data types in CGC language, primitive type and object type. The primitive data types of CGC are `int`, `char`, `float`. The object types are array and class instances.

3.3.1 Primitive Data Types

Primitive data types in CGC are predefined and reserved as keywords. Specifically, there are 3 primitive types:

`int`: the `int` data type is a 32-bit signed two's complement integer. `int` is able to represents integer ranges from -2147483648 to 2147483647. `int` variables are considered as signed by default, unsigned is not a keyword in CGC.`int` is the only one type in CGC used to stores integer, CGC does not support keywords like `short` or `long` to specify different variable length.

`float`: The float data type is a 64-bit IEEE 754 floating point.

`char`: The char data type is a single unsigned 8-bit ASCII character.
The operators of `int`, `float` and `char` are listed as below, detailed specification of these operators are listed at part 5.
– The numerical comparison operators <, >, <=, and >=
– The numerical equality operators ==
– The multiplicative operators * and /
– The additive operators + and -
– The unary minus operator –
– The unary logical negation operator !
– The logical AND operator &&
– The logical OR operator ||
– The assignment operator =
– The comma operator ;

3.3.2 Array

Array is an object which holds a fixed number of values of a single type. The length of an array is defined when the array is created and can not be changed.

```
int[] array;
array = new int[10];
```

3.3.3 User-Defined Data Types

In CGC, users can define their own data types by using the keyword `class`. The `class` keyword is used to define new data types and describe how they are implemented. In the body of a `class`, users are able to define its members (variables, methods). Notice that one must explicitly declare a constructor in the class definition, as constructor will be called when an instance is created and no default constructor is implemented. The CGC `class` does not support complex Object-Oriented programming features such as inheritance.

```
    //class
    class foo {
        int bar;
        constructor(int a) {
            bar = a;
        }
        void get_bar() {
            return bar;
        }

    }
    int main{
        foo foo_example;
        foo_example.bar = 10;
        int result = foo_example.get_bar();
        print(result);// result has a value of 10 now
    }
```

3.4 Statements and Expressions

3.4.1 Statements

A CGC program is made up of a list of statements. A statement is one of the following:

```
Expression
Class declaration
Function definition
```

```
Return statement
If statement
If-else statement
For loop
```

Blocks of statements can be enclosed using curly braces.

If-Else Statements If statements consist of a condition (an expression) and a series of statements. The series of state- ments is evaluated if the condition evaluates to True. If the condition evaluates to False, either the program continues or an optional else clause is executed.

Examples:

```
if condition{
    //series of statements
}
else{
    //other series of statements
}
```

For Statements There are two ways to write a `For` Loop in CGC. The first kind of `For` Loop iterate over an `Array`. It consists of a looping variable, an instance of a `Array`. The series of statements is evaluated for each item in the `Array` in order, where the looping variable is assigned to the first element of the list for the first iteration, the second for the second iteration, etc. to be looped over, and a series of statements.

```
Array<int> x = {1,2,3}
//second format
for (int i = 0; i < x.len(); i = i + 1) {
    x[i] = x[i] + 1;
}
```

3.4.2 Expressions and Operators

Expressions are part of statements that are evaluated into expressions and variables using operators. These can be arithmetic expressions or a function call. Unary Operators CGC has two types of operators: unary operators and binary operators.
In CGC, the unary operators are NEG and NOT, i.e "-" and "!".

```
int x = -1 // represents the negative number 1
```

NOT represents negation of a boolean expression. It can also be applied to integers and floats, where any non-zero number is considered to be True, and False otherwise.

```
int a = 1;
int b = 1;
int c = 0;
if (a == b){
   //series of statements that will be evaluated
}
if (!(a == c)){
   //series of statements that will be evaluated
}
if (!a) {
  //series of statements that will not be evaluated
}
```

Binary Operators The following list describes the binary operators in CGC. All of the operators act on two expressions. Unless otherwise stated, they act only on primitives.

1. Assignment Operator
   The assignment operator stores values into variables. This is done with the "=" symbol. The value of the right side is stored in the variable on the left side.

   ```
   int x = 1; // x = 1
   int y = 0; // y = 0
   y = x; // y = 1
   ```

2. Arithmetic Operator

   a. Addition is performed on two values of the same type. Addition between different types is not permitted.

   ```
   int x = 1;
   int y = 2;
   int z = x + y; // z = 3;
   float f = 1.2 + 2.3; // f = 2.5
   ```

   b. Subtraction is performed on two values of the same type. Subtraction between different types is not permitted.

   ```
   int x = 1;
   int y = 2;
   int z = x - y; // z = -1;
   float f = 2.5 - 2.3; // f = 0.2
   ```

   c. Multiplication is performed on two values of the same type. Multiplication between different types is not permitted.

```
int x = 2;
int y = 3;
int z = x * y; // z = 6;
float f = 2.0 * 2.5; // f = 5.0
```

d. Division is performed on two values of the same type. Division between different types is not permitted.

```
int x = 4;
int y = 3;
int z = y / x; // z = 1;
float f = 5.0 / 2.0; // f = 2.5
```

3. Relational Operators
   Relational operators determine how the operands relate to another. There are two values as inputs and outputs either true or false. The operators includes: == >, <, >=, <=, &&, ||

```
int x = 1;
int y = 2;
int z = 3;
if (x > y){
//Evaluate to be false
}
 if (x < y){
//Evaluate to be true
}
 if (x == y){
//Evaluate to be false
}
if (x == y || x < z){
//Evaluate to be true
}
```

3.4.3 Operator Precedence

The following is an operator precedence table for all the operators, from lowest to highest precedence.

3.4.4 Functions

A function is a type of statement. It takes in a list of arguments and return one value. The body of a function is delimited by curly braces. The return type is specified before to the function name. An example of function declaration is as follows:

Operator	Meaning	Associativity
;	Sequencing	Left
=	Assignment	Right
||	Or	Left
&&	And	Left
==	Equality	Left
>, <, >=, <=	Comparison	Left
+ -	Addition/ Subtraction	Left
* /	Multiplication/ Division	Left
!	Not	Right

```
int add_one(int x)
{
    int y = x + 1;
    return y;
}
```

3.4.5 Function Calls

Functions are called using their identifier and arguments inside parentheses. For example:

```
int x = 1;
add_one(x);
```

Variable Assignment from Functions A function may be called as the right value of a variable assignment. The variable would be assigned to the return value of the function.

```
int x = 1;
int z = add_one(x); // z would be 2
```

3.5 Standard Library

3.5.1 print()

The print function acts like printf() in C, but will only take one argument in int, float, bool and string.

Examples:

```
    int main()
    {
        print(1);// prints 1
        print('c');// prints c
        print("string");// prints string
    }
```

4 Project Plan

4.1 Development Process

4.1.1 Planning

In order to complete CGC in a timely way, we had team meetings approximately once per week on Saturday. Since our team members are in different time zone, the meeting time each week is subject to change. During these meetings we discussed the goals and requited steps to implement CGC. We also had some individual meetings between two team members when needed.
Our day to day communication happened over Wechat. This allowed us to find each other whenever needed and discuss an implementation idea.

4.1.2 Specifications

The goal of CGC was to develop a C-like language that has a interior garbage collector. Our plan was to implement basic C syntax and some object-oriented programming features like classes. The reset of our time was spent on researching on and implementing different garbage collection algorithms using LLVM's APIs.

4.1.3 Development

At the beginning of development, we had several large group meetings where we decided on the design and wrote out the language reference manuals. Then, we started to implement different features. Then, as we approached the Hello World deadline, we have a basic working compiler and conduct our first integration test. We then started to work on the garbage collection part and spent most of the time left on that.

4.1.4 Testing

Each feature was tested before being merged with other features. All the test cases are added to the testing suite that would automatically tested by our test script.

4.2 Software Development Tools

We used the following programming and development environments when implementing CGC:

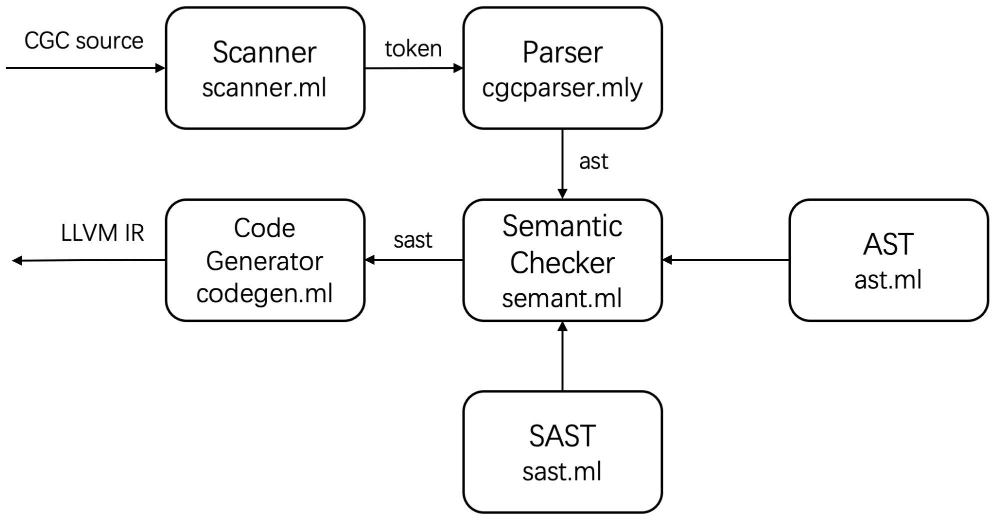
1. Libraries and Languages: Ocaml Version 4.11.1 and LLVM Version 10.0.0 was used.
2. Software: Visual Studio Code with the LiveShare extension
3. OS: Development was done on macOS Big Sur 11.2.3
4. Version Control: GitHub-Hosted Git Repository

4.3 Roles and Responsibilities

Team Member	Responsibility
Lieyang Chen	Scanner, Parser, Semant, Codegen - Arrays
Tianze Huang	Codegen - Classes, Functions
Zhuoxuan Li	Test Suite, Final Report, Codegen - Arrays
Fanhao Zeng	Codegen - Garbage Collection, Primitive Types

5 Architectural Design

The block diagram of CGC compiler front-end is:



5.1 Scanner

The scanner.mll takes in a CGC source program of ASCII characters as input, identify the pattern of lexemes and convert the source program as token stream. The pattern of lexeme is defined by regular expressions. In Ocamllex, we use pattern matching to match the string keywords or regular expression with corresponding tokens. In this step, if any characters in the code are detected to be illegal, lexing errors will be thrown. The comments style also defined in scanner, these comments and whitespaces will be ignored by the scanner. The lexer produces tokens that will then be used by the parser in the next step of compilation.

5.2 Parser

The parser (parser.mly) takes the token stream as the input, and construct an abstract syntax tree (AST). The structure of AST is defined by ast.ml, the AST is constructed based on CGC context-free grammar rules. The parser is implemented by Ocamlyacc, it parses the token stream into a list which can then be used to extract indentation/tabs. This uses a special parser directive, tokenize, which simply returns the input stream as a list of tokens. The top level of the abstract syntax tree is a structure called CGC Program which constructed by three parts, global variables, functions and classes. Classes contain fields and methods. In addition, the method declarations record is the creation of an AST of functions from groups of statements, statements evaluating the results of expressions, and expressions formed from operations and assignments of variables and constants. The context-free grammar regulates the source code pattern. If any violations are detected, misused operator, comma, or semicolon, the parser will throw corresponding error message.

5.3 Semantic Checker

The semantic checker (semant.ml) takes AST constructed in the syntactic analysis phase and generates SAST. SAST in general, has the same structure with AST, but each node of SAST has a type associated with it. The semantic checker first checks the name of binding of global variable and local variable, by checking whether there are two variables in the same name. It builds a series of Ocaml map here, including global variable map, function map and class map. The class map also including two maps, namely class field map and class method map, which stores the field variable and method declaration of a given class. Global variables with a type Void will also be detected. Built-in functions and user-defined functions will also be defined here, with the information of its return type, number and type of formals etc. It checks whether the source code is semantically correct from various aspects including whether there are duplicated fields or methods in one class. It also adds the names of standard library functions and the source code names to the Symbol Table and verifies there are no duplicate names for functions and global variables, or for local variables in a specific block. It checks for their function, class and variable declarations, by looking at the left value and right value of assign operator. It also verifies there is a main function in the program. It checks there are no void variables or function arguments. It checks that statement blocks are well-formed. When checking a function, the semantic checker will call the check statement and check expression helper function defined in the semant.ml. For example, check fucntion call will check number of actuals equals to number of formals for a given funtion or class method or class constructor when calling it. It also check type of formals match with actuals. Check Assign is simply make sure the lvalue type = rvalue type, by pattern matching the array type, make sure Int/Bool/Float array is assigned to Int/Bool/Float value. Check Unop is make sure Neg is used on Int/Float value, Not is used on Bool value. Check Binop is make sure the type of two operands is the same. Check object access is implemented by checking type of ID is an Object() type. Check class field/method is to make sure class name exists in class map, make sure class method/field in corresponding method map and field map.For methods, it checks number of actuals = number of formals, check type of formals match with actuals.

5.4 Code Generator

The code generator (codegen.ml) takes SAST as input and generates the LLVM IR. It traverses the tree and translates each node into LLVM code to build an LLVM module. This code generation is written using the OCaml LLVM library. The program of CGC, in top level, contains three components: List of global variable declaration, list of function declaration and list of class declaration. When code generator generates the LLVM IR, it calls corresponding helper function defined in the codegen.ml to generate each part of the program.

6 Testing

Individual features of CGC were developed in individual github branches. Before being merged into other branches, the one who work on the feature will come up with some test programs and their corresponding desired outputs. The test cases were discussed in our weekly meeting. All team members were responsible for contributing to testing. In total, there are 73 test cases.

6.1 Testing Suite and Automation

All test cases are stored in /test/ folder.
Testing Procedure Examples: (replace test-xxx.mc with test filename)

```
$ ./CGC.native < tests/test-xxx.mc > test-xxx.ll
$ /usr/local/opt/llvm/bin/llc -relocation-model=pic test-xxx.ll >
    test-xxx.s
$ cc -o test-xxx.exe test-xxx.s printbig.o
$ ./test-xxx.exe
```

After the test commands are executed, a .out or .err file will be generated. The .out file means the input program is compiled and executed correctly. The .out file will contain the output printed. In the opposite, the the .err file means the input program has some errors and the error message will be contained in this file.

7 Lessons Learned

7.1 Tianze Huang

Throughout this project, I realize how important that making a good project plan in advance is. I mean both time management and detailed plan of every feature that we implemented on our little language. Writing a little language can be really annoying, but it is also rewarding. This project contains nearly everything we have been taught in the class. From the lexical and syntactical analysis, to semantic analysis and code generation. Doing this project seems reviewing lecture material in a new way, but it is still painful. I realized that it is important to thinking clearly what I am going to implement and how to coding it when adding a feature to our little language. When I was implementing the class feature of our language, it took me a huge amount of time struggling on the sematic checker. It seems confusing in the first place that I have no idea what I needed to check, obviously, a detailed planning and discussing with team members and our TA beforehand is extremely important

7.2 Fanhao Zeng

It can be much harder than one may think to implement an algorithm in a real project. I have learned about many garbage collection algorithms back in my undergraduate studies. I was asked about garbage collection algorithms a couple of times in many different job interviews. While I think I am familiar enough with the garbage collection algorithm, I still meet with a lot difficulties when I actually tried to implement it. For example, the garbage collection function first worked on some simple cases with some programs with very simple syntax. Then after the integration with some other more complicated features, the garbage collector starts to break. I have to roll back once and once again and find it really frustrating. If I would be able to do this project once again, I will definitely design better architecture in the first place and make sure each module does not depends on each other.

7.3 Zhuoxuan Li

One big lesson I learnt throughout this project is how important communication is. We made a simple plan at first and we thought the entire project will go smoothly as expected. It turned out to be much harder to put everything together. Especially, I don't think it's a great idea to divide work into different features, especially when we don't have sufficient communication. Feature code in compilers are not as modularized as web front end code that I'm used to writing. We were kind of doing our own parts, and expect all the code can be put together perfectly in the end, but that is not the case. It was close to the deadline when we were done with our parts and start to put everything together, but code started to break after we merge the code. We ended up putting lots of effort to merge the features written by different teammates and some code had to be discarded and rewritten. This could have been avoided if we had tracked our progress more frequently.

7.4 Lieyang Chen

After this class, I feel that I have gained a lot of new and exciting knowledge. The compiler, which was once mysterious and powerful to me, became less remote by studying compilation principles in depth and in detail. In addition, I learned how to divide a large project into many small modules. With this modular approach, many puzzling problems can be solved. But the most important thing I learned from this class was teamwork. Through this project experience, I realized that a good project is not determined by whether there are good people in the team, but often by the tacit teamwork. After three months of teamwork experience, I found that effective communication is often more useful than individual ability. Therefore, I am very glad that our project can go so smoothly. I also hope I can pay more attention to the ability of teamwork and communication in the future.

8 Appendix

8.1 Source files

CGC.ml

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

type action = Ast | Sast | LLVM_IR | Compile

let _ =
  let action = if Array.length Sys.argv > 1 then
    List.assoc Sys.argv.(1) [ ("-a", Ast); (* Print the AST only *)
                  ("-s", Sast); (* Print the SAST only *)
      ("-l", LLVM_IR);  (* Generate LLVM, don't check *)
      ("-c", Compile) ] (* Generate, check LLVM IR *)
  else Compile in
  let lexbuf = Lexing.from_channel stdin in
  let ast = Parser.program Scanner.token lexbuf in
  let sast = Semant.analyze ast in
  let (globals, function_decls, class_decls, main_decl, builtin_decls) = sast in
  let program = {
global_vars = globals;
functions = function_decls;
    classes = class_decls;
    main = main_decl;
    builtins = builtin_decls;
}
  in
  match action with
    Ast -> print_string (Ast.string_of_program ast)
  | Sast -> print_string (Sast.string_of_s_program sast)
  | LLVM_IR -> print_string (Llvm.string_of_llmodule (Codegen.codegen_sprogram program))
  | Compile -> let m = Codegen.codegen_sprogram program in
    Llvm_analysis.assert_valid_module m;
    print_string (Llvm.string_of_llmodule m)
```

scanner.mll

```
{ open Parser }

let whitespace = [' ' '\t' '\r' '\n']
let ascii = ([' '-'!' '#'-'[' ']'-'~'])
let escape = '\\' ['\\' ''' '"' 'n' 'r' 't']
let escape_char = ''' (escape) '''
let alpha = ['a'-'z' 'A'-'Z']
let digit = ['0'-'9']
let int = digit+
let float = (digit+) '.' (digit+)

let char = ''' (ascii) '''

let string_lit = '"'((ascii|escape)* as lxm)'"'

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"     { comment lexbuf }           (* Comments *)
| '('      { LPAREN }
| ')'      { RPAREN }
| '{'      { LBRACE }
| '}'      { RBRACE }
| '['      { LBRACKET }
| ']'      { RBRACKET }
| ';'      { SEMI }
| ','      { COMMA }
| '+'      { PLUS }
| '-'      { MINUS }
| '*'      { TIMES }
| '/'      { DIVIDE }
| '='      { ASSIGN }
| "=="     { EQ }
| "!="     { NEQ }
| '<'      { LT }
| "<="     { LEQ }
| ">"      { GT }
| ">="     { GEQ }
| "&&"     { AND }
| "||"     { OR }
| "!"      { NOT }
| '.'   { DOT }

(*Flow control*)
| "if"     { IF }
| "else"   { ELSE }
| "for"    { FOR }
| "while"  { WHILE }
| "return" { RETURN }

(*Data types*)
| "int"    { INT }
| "bool"   { BOOL }
| "void"   { VOID }
| "float"  { FLOAT }
| "string" { STRING }
| "float"  { FLOAT }
| "true"   { TRUE }
| "false"  { FALSE }

(*Class*)
| "class"  { CLASS }
| "new"    { NEW }
| "delete" { DELETE }
| "constructor" { CONSTRUCTOR }

| ['0'-'9']+ as lxm { LITERAL(int_of_string lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
| string_lit { STRING_LITERAL(lxm)}
| float as lxm  { FLOAT_LITERAL(float_of_string lxm) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }
```

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

parser.mly

```
%{ open Ast %}

%token CLASS CONSTRUCTOR NEW DELETE DOT
%token SEMI LPAREN RPAREN LBRACE RBRACE LBRACKET RBRACKET COMMA
%token PLUS MINUS TIMES DIVIDE ASSIGN NOT
%token EQ NEQ LT LEQ GT GEQ TRUE FALSE AND OR
%token RETURN IF ELSE FOR WHILE INT BOOL VOID FLOAT STRING
%token <int> LITERAL
%token <string> STRING_LITERAL
%token <float> FLOAT_LITERAL
%token <string> ID
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%right RPAREN
%right ASSIGN
%left LBRACKET
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE
%right NOT NEG DOT DELETE

%start program
%type <Ast.program> program

%%

program:
    decls EOF { $1 }

decls:
   /* nothing */ { [], [], [] }
 | decls vdecl { let (vdecl,fdecl,cdecl) = $1 in ($2::vdecl,fdecl,cdecl) }
 | decls fdecl { let (vdecl,fdecl,cdecl) = $1 in (vdecl,$2::fdecl,cdecl) }
 | decls cdecl { let (vdecl,fdecl,cdecl) = $1 in (vdecl,fdecl,$2::cdecl) }
```

```
cdecl:
        CLASS ID LBRACE cbody RBRACE { {
            cname =  $2;
            cbody =  $4;
        } }

cbody:
        /* nothing */ { {
            fields = [];
            methods = [];
            constructors = [];
        } }
    |   cbody vdecl { {
            fields = $2 :: $1.fields;
            methods = $1.methods;
            constructors = $1.constructors
        } }
    |   cbody fdecl { {
            fields = $1.fields;
            methods = $2 :: $1.methods;
            constructors = $1.constructors
        } }
    |   cbody constructor { {
            fields = $1.fields;
            methods = $1.methods;
            constructors = $2 :: $1.constructors
        } }

constructor:
        CONSTRUCTOR LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE { {
            typ = Constructortyp;
            fname = "constructor";
            formals = $3;
            locals = List.rev $6;
            body = List.rev $7;
        } }

fdecl:
    typ ID LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE { {
        typ = $1;
        fname = $2;
        formals = $4;
        locals = List.rev $7;
        body = List.rev $8
    } }
```

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

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

obj:
    CLASS ID { Object($2) }

primitive:
    INT { Int }
  | BOOL { Bool }
  | VOID { Void }
  | STRING { String }
  | FLOAT { Float }

arraytype:
    primitive LBRACKET RBRACKET { Arraytype($1) }

typ:
    primitive { $1 }
  | obj { $1 }
  | arraytype { $1 }

vdecl_list:
    /* nothing */    { [] }
  | vdecl_list vdecl { $2 :: $1 }

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

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

stmt:
    expr SEMI { Expr $1 }
  | RETURN SEMI { Return Noexpr }
  | RETURN expr SEMI { Return $2 }
  | LBRACE stmt_list RBRACE { Block(List.rev $2) }
  | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
  | IF LPAREN expr RPAREN stmt ELSE stmt    { If($3, $5, $7) }
  | FOR LPAREN expr_opt SEMI expr SEMI expr_opt RPAREN stmt { For($3, $5, $7, $9) }
  | WHILE LPAREN expr RPAREN stmt { While($3, $5) }

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

expr:
    LITERAL          { Literal($1) }
  | STRING_LITERAL   { StringLit($1) }
  | FLOAT_LITERAL    { FloatLit($1) }
  | TRUE             { BoolLit(true) }
  | FALSE            { BoolLit(false) }
  | ID               { Id($1) }
  | LPAREN typ RPAREN expr { Cast( $2,$4) }
  | expr PLUS   expr { Binop($1, Add,   $3) }
  | expr MINUS  expr { Binop($1, Sub,   $3) }
  | expr TIMES  expr { Binop($1, Mult,  $3) }
  | expr DIVIDE expr { Binop($1, Div,   $3) }
  | expr EQ     expr { Binop($1, Equal, $3) }
  | expr NEQ    expr { Binop($1, Neq,   $3) }
  | expr LT     expr { Binop($1, Less,  $3) }
  | expr LEQ    expr { Binop($1, Leq,   $3) }
  | expr GT     expr { Binop($1, Greater, $3) }
  | expr GEQ    expr { Binop($1, Geq,   $3) }
  | expr AND    expr { Binop($1, And,   $3) }
  | expr OR     expr { Binop($1, Or,    $3) }
  | expr DOT    expr { ObjAccess($1, $3) }
  | MINUS expr %prec NEG { Unop(Neg, $2) }
  | NOT expr         { Unop(Not, $2) }
  | expr ASSIGN expr   { Assign($1, $3) }
  | ID LPAREN actuals_opt RPAREN { Call($1, $3) }
  | LPAREN expr RPAREN { $2 }
  | NEW ID LPAREN actuals_opt RPAREN { ObjCreate($2,$4) }
  | NEW primitive LBRACKET LITERAL RBRACKET { ArrayCreate($2, $4) }
  | expr LBRACKET LITERAL RBRACKET { ArrayAccess($1, $3) }
  | DELETE expr      { Delete($2) }

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

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

ast.ml

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

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

type uop = Neg | Not

type typ = Int | Bool | Void | Float | Null | String | Object of string | Arraytype of typ

type bind = typ * string

type expr =
    Literal of int
  | BoolLit of bool
  | StringLit of string
  | FloatLit of float
  | Id of string
  | Binop of expr * op * expr
  | Unop of uop * expr
  | Assign of expr * expr
  | Call of string * expr list
  | ObjAccess of expr * expr
  | ObjCreate of string * expr list
  | ArrayCreate of typ * int
  | ArrayAccess of expr * int
  | Delete of expr
  | Noexpr
  | Nullexpr
  | Cast of typ * expr

type stmt =
    Block of stmt list
  | Expr of expr
  | Return of expr
  | If of expr * stmt * stmt
  | For of expr * expr * expr * stmt
  | While of expr * stmt

type func_decl = {
    typ : typ;
    fname : string;
    formals : bind list;
    locals : bind list;
    body : stmt list;
  }

type cbody = {
    fields: bind list;
    methods: func_decl list;
    constructors: func_decl list;
}

type class_decl = {
    cname: string;
    cbody: cbody;
}

type program = bind list * func_decl list * class_decl list

(* Pretty-printing functions *)
let rec string_of_typ = function
    Int -> "int"
  | Bool -> "bool"
  | Void -> "void"
  | String -> "string"
  | Float -> "float"
  | Null -> "null"
  | Object(cname) -> cname
  | Arraytype(t) -> "array of " ^ string_of_typ t
  | Constructortyp -> "constructortyp"
  | Any -> "any"

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

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

```
let rec string_of_expr = function
    Literal(l) -> string_of_int l
  | BoolLit(true) -> "true"
  | BoolLit(false) -> "false"
  | StringLit(s) -> s
  | FloatLit(f) -> string_of_float f
  | Id(s) -> s
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr e2
  | Unop(o, e) -> string_of_uop o ^ string_of_expr e
  | Assign(e1, e2) -> string_of_expr e1 ^ " = " ^ string_of_expr e2
  | Call(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | ObjCreate(s,el) -> "new " ^ s ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^
  | ObjAccess(e1,e2) -> string_of_expr e1 ^ "." ^ string_of_expr e2
  | ArrayCreate(typ, len) -> "(array of type " ^ string_of_typ typ ^ " with length " ^ strin
  | ArrayAccess(arrayName, index) -> "(array name: " ^ string_of_expr arrayName ^ " index: "
  | Delete(e) -> "delete" ^ string_of_expr e
  | Cast (t,e) -> "(" ^ (string_of_typ t) ^ ") " ^ (string_of_expr e)
  | Noexpr -> ""
  | Nullexpr -> "null"

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

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

let string_of_formal (t, id) = string_of_typ t ^ " " ^ id

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

```
let string_of_cdecl cdecl =
    "class" ^ " " ^ cdecl.cname ^ " { \n" ^
    String.concat "" (List.map string_of_vdecl cdecl.cbody.fields) ^
    String.concat "" (List.map string_of_fdecl cdecl.cbody.constructors) ^
    String.concat "" (List.map string_of_fdecl cdecl.cbody.methods) ^
    "}\n"


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

sast.ml

```
open Ast
type s_expr =
    S_Literal of int
  | S_BoolLit of bool
  | S_StringLit of string
  | S_FloatLit of float
  | S_Id of string * typ
  | S_Binop of s_expr * op * s_expr * typ
  | S_Unop of uop * s_expr * typ
  | S_Assign of s_expr * s_expr * typ
  | S_Call of string * s_expr list * typ
  | S_Noexpr
  | S_Null
  | S_Cast of typ * s_expr * typ
  | S_ObjCreate of string * s_expr list * typ
  | S_ObjAccess of s_expr * s_expr * typ
  | S_ArrayCreate of typ * int
  | S_ArrayAccess of s_expr * int * typ
  | S_Delete of s_expr

type s_stmt =
    S_Block of s_stmt list
  | S_Expr of s_expr * typ
  | S_Return of s_expr * typ
  | S_If of s_expr * s_stmt * s_stmt
  | S_For of s_expr * s_expr * s_expr * s_stmt
  | S_While of s_expr * s_stmt

type s_func_decl = {
    s_typ : typ;
    s_fname : string;
    s_formals : bind list;
    s_locals : bind list;
    s_body : s_stmt list;
}

type s_class_decl = {
    s_cname: string;
    s_fields: bind list;
    s_constructors: s_func_decl list;
    s_methods: s_func_decl list;
}

type s_program = {
    global_vars: bind list;
    functions: s_func_decl list;
    classes: s_class_decl list;
    main: s_func_decl;
    builtins: s_func_decl list;
}

(* Pretty-printing functions *)
let rec string_of_ast_typ = function
    Int -> "int"
    | Bool -> "bool"
    | Void -> "void"
    | Null -> "null"
    | String -> "string"
    | Object(name) -> name
    | Arraytype(t) -> "array of " ^ string_of_ast_typ t
    | Constructortyp -> "constructor"
    | Any -> "any"

let rec string_of_expr_type = function
    S_Literal(_) -> "int"
  | S_BoolLit(_) -> "bool"
  | S_StringLit(_) -> "string"
  | S_Id(name,data_type) -> string_of_ast_typ data_type
  | S_Binop(lhr,op,rhs,data_type) ->  string_of_ast_typ data_type
  | S_Unop(op,exp,data_type) -> string_of_ast_typ data_type
  | S_Assign(lhs,rhs,data_type) -> string_of_ast_typ data_type
  | S_Call(fname,expr_list,data_type) -> string_of_ast_typ data_type
  | S_Noexpr -> "Noexpr"
  | S_Null -> "null"
  | S_ObjCreate(obj_name,expr_list,data_type) -> string_of_ast_typ data_type
  | S_ObjAccess(obj_name,field,data_type) -> string_of_ast_typ data_type
  | S_ArrayCreate(typ, len) -> "(array of type " ^ string_of_ast_typ typ ^ " with length " ^
  | S_ArrayAccess(arrayName, index, data_type) ->  "yuanlaishitype..type: " ^ string_of_ast_
  | S_Delete(expr) -> string_of_expr_type expr
  | S_Cast(oldt,expr,newt) -> string_of_ast_typ newt

let rec string_of_s_expr = function
    S_Literal(l) -> string_of_int l
  | S_BoolLit(true) -> "true"
  | S_BoolLit(false) -> "false"
  | S_StringLit(s) -> s
  | S_FloatLit(f) -> string_of_float f
  | S_Id(s,t) -> "$" ^ s ^ "[" ^ string_of_typ t ^ "]"
  | S_Binop(e1, o, e2, t) ->
      "{" ^ string_of_s_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_s_expr e2 ^ "}[" ^
  | S_Unop(o, e,t) -> string_of_uop o ^ string_of_s_expr e ^ "[" ^ string_of_typ t ^ "]"
  | S_Assign(e1,e2,t) -> string_of_s_expr e1 ^ " = " ^ string_of_s_expr e2 ^ "[" ^ string_of
  | S_Call(f, el,t) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_s_expr el) ^ ")" ^ "[" ^ string_of_ty
  | S_Noexpr -> ""
  | S_Null -> "Null"
  | S_Cast(oldt,expr,newt) ->
     "{" ^ "(" ^ string_of_s_expr(expr) ^ ")" ^ "[" ^ string_of_typ(oldt) ^ "->" ^ string_of
  | S_ObjAccess(s1,s2,t) -> "{Object Access: " ^ string_of_s_expr s1 ^ "." ^ string_of_s_exp
  | S_ObjCreate(s,sel,t) -> "{Object Create: new " ^ s ^ "(" ^ String.concat "," (List.map s
  | S_ArrayCreate(typ, len) -> "(array of type " ^ string_of_typ typ ^ " with length " ^ str
  | S_ArrayAccess(arrayName, index, t) ->
        (* match arrayExpr with S_Id(arr,t) -> "(array_name: " ^ arr ^ "type:" ^ string_of_t
        "(array_name: " ^ string_of_s_expr arrayName ^ " type: " ^ string_of_typ t ^ " index
  | S_Delete(se) -> "delete " ^ string_of_s_expr se

let rec string_of_s_stmt = function
    S_Block(s_stmts) ->
      "{\n" ^ String.concat "" (List.map string_of_s_stmt s_stmts) ^ "}\n"
  | S_Expr(s_expr,t) -> "{" ^ string_of_s_expr s_expr ^ "} [" ^ string_of_typ t ^ "]" ^ ";\n
  | S_Return(s_expr,t) -> "return " ^ string_of_s_expr s_expr ^ ";\n" ^ "[" ^ string_of_typ
  | S_If(e, s, S_Block([])) -> "if (" ^ string_of_s_expr e ^ ")\n" ^ string_of_s_stmt s
  | S_If(e, s1, s2) ->  "if (" ^ string_of_s_expr e ^ ")\n" ^
      string_of_s_stmt s1 ^ "else\n" ^ string_of_s_stmt s2
  | S_For(e1, e2, e3, s) ->
      "for (" ^ string_of_s_expr e1  ^ " ; " ^ string_of_s_expr e2 ^ " ; " ^
      string_of_s_expr e3  ^ ") " ^ string_of_s_stmt s
  | S_While(e, s) -> "while (" ^ string_of_s_expr e ^ ") " ^ string_of_s_stmt s

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

let string_of_s_formal (t, id) = string_of_typ t ^ " " ^ id
let string_of_s_fdecl fdecl =
  string_of_typ fdecl.s_typ ^ " " ^
  fdecl.s_fname ^ "(" ^ String.concat ", " (List.map string_of_s_formal fdecl.s_formals) ^
  ")\n{\n" ^
  String.concat "" (List.map string_of_vdecl fdecl.s_locals)^
  String.concat "" (List.map string_of_s_stmt fdecl.s_body) ^
  "\n}\n"

let string_of_s_cdecl cdecl =
    "class" ^ " " ^ cdecl.s_cname ^ " { \n" ^
    String.concat "" (List.map string_of_s_vdecl cdecl.s_fields) ^
    String.concat "" (List.map string_of_s_fdecl cdecl.s_constructors) ^
    String.concat "" (List.map string_of_s_fdecl cdecl.s_methods) ^
    "}\n"

let string_of_s_program (vars, funcs, classes, main, builtins) =
  String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
  String.concat "\n" (List.map string_of_s_fdecl funcs) ^ "\n" ^
  String.concat "\n" (List.map string_of_s_cdecl classes) ^ "\n" ^
  String.concat "\n" (List.map string_of_s_fdecl builtins) ^ "\n" ^
  (string_of_s_fdecl main) ^ "\n"
```

semant.ml

```
open Ast
open Sast

module StringMap = Map.Make(String)

type class_property_map = {
field_map : Ast.typ StringMap.t;
method_map : Ast.func_decl StringMap.t;
}

type env = {
env_global_map : Ast.typ StringMap.t;
env_local_map : Ast.typ StringMap.t;
env_param_map : Ast.typ StringMap.t;
env_function_map : Ast.func_decl StringMap.t;
env_class_map : class_property_map StringMap.t;
env_return_type : Ast.typ;
}


let define_builtin_functions =
[
   {typ = Void; fname = "cast"; formals = [(String, "x")];
       locals = []; body = []};
       {typ = Void; fname = "malloc"; formals = [(String, "x")];
       locals = []; body = []};
       {typ = Void; fname = "sizeof"; formals = [(String, "x")];
       locals = []; body = []};
{typ = Void; fname = "print"; formals = [(Any, "x")];
       locals = []; body = [] };
   {typ = String; fname = "toString"; formals = [(Any,"x")];
   locals = []; body = [] };

]

let build_maps(functions,builtins,globals) =
(*TODO: check duplicated function declarations here*)
let function_map = List.fold_left (fun m fd ->
if (StringMap.mem fd.fname m) then
raise (Failure("duplicated function declaration " ^ fd.fname))
else
StringMap.add fd.fname fd m)
                         StringMap.empty (functions @ builtins)  in
let global_map = List.fold_left (fun m (t,n) ->
if (StringMap.mem n m) then
raise (Failure("duplicated global declaration " ^ n))
else if (t = Void) then
raise(Failure("global " ^ n ^ " shouldn't be void type"))
else
StringMap.add n t m)
                         StringMap.empty (globals)  in
 (function_map,global_map)

let build_class_field_map(fields,globals) =
List.fold_left (fun m (t,n) ->
if (StringMap.mem n m) then
raise (Failure("duplicated declaration " ^ n ^ ",field " ^ string_of_typ(t) ^ " " ^ n ^ " al

else if (t = Void) then
raise(Failure(n ^ " shouldn't be void type "))
else
StringMap.add n t m)
                         StringMap.empty (fields)

let build_class_method_map(cname,methods,functions) =
let find_constructor = (fun f -> match f.typ with Constructortyp -> true  | _ -> false) in
let get_constructor  =
let constructors = (List.find_all find_constructor methods) in
let count = List.length constructors in
if List.length constructors < 1 then
raise (Failure("Missing constructor for class " ^ cname))
else List.hd constructors
in
ignore(get_constructor);
List.fold_left (fun m fdecl ->
if (StringMap.mem fdecl.fname m) then
raise (Failure("duplicated method declaration " ^ fdecl.fname))
else
StringMap.add fdecl.fname fdecl m)
                         StringMap.empty (methods)

let build_class_map(globals,functions,classes) =
let class_map = List.fold_left (fun m cdecl ->
     if (StringMap.mem cdecl.cname m) then
     raise (Failure("duplicated class declaration" ^ cdecl.cname))
     else
     StringMap.add cdecl.cname
     {
     field_map = build_class_field_map(cdecl.cbody.fields,globals);
     method_map = build_class_method_map(cdecl.cname,cdecl.cbody.constructors@cdecl.cbody.me
     }
     m)
     StringMap.empty (classes) in
      class_map

(* locals could overshadow params, params could overshadow globals *)
let get_id_type(env,s) =
try StringMap.find s env.env_local_map
with | Not_found ->
try StringMap.find s env.env_param_map
with |Not_found ->
try StringMap.find s env.env_global_map
with |Not_found -> raise (Failure("undefined_id " ^ s))

let get_function_decl(env,fname) =
try StringMap.find fname env.env_function_map
with | Not_found -> raise (Failure("UndefinedFunction " ^ fname))

let get_sexpr_type(sexpr)= match (sexpr) with
    S_Literal(_) -> Int
  | S_BoolLit(_) -> Bool
  | S_StringLit(_) -> String
  | S_FloatLit(_) -> Float
  | S_Id(_,t) -> t
  | S_Binop(_,_,_,t) -> t
  | S_Unop(_,_,t) -> t
  | S_Assign(_,_,t) -> t
  | S_Call(_,_,t) -> t
  | S_Noexpr -> Void
  | S_Null -> Null
  | S_Cast(_,_,t) -> t
  | S_ObjCreate(_,_,t) -> t
  | S_ObjAccess(_,_,t) -> t
  | S_ArrayCreate(t,_) -> Arraytype(t)
  | S_ArrayAccess(_,_,t) -> t
  | S_Delete(_) -> Void

let report_duplicate exceptf list =
let rec helper = function
n1 :: n2 :: _ when n1 = n2 -> raise (Failure (exceptf n1))
| _ :: t -> helper t
| [] -> ()
in helper (List.sort compare list)

let get_variable_list lst =
List.fold_left (fun lst (_,n) -> n::lst)
                        [] lst

let get_function_list lst =
List.fold_left (fun lst fdecl -> fdecl.s_fname::lst)
                        [] lst

let rec exprl_to_sexprl (env,el) =
  let env_ref = ref(env) in
  let rec helper = function
  head::tail ->
let a_head, env = generate_sexpr(!env_ref,head) in
env_ref := env;
a_head::(helper tail)
| [] -> []
  in (helper el), !env_ref


and check_call_type (env,fname,exprs) =
let actuals,_ = exprl_to_sexprl(env,exprs) in
let fd = get_function_decl(env,fname) in
let match_params = fun (ft,_) actual ->
if (get_sexpr_type(actual) = ft || ft = Any ) then
()
```

```
else raise(Failure ("illegal actual argument found " ^ string_of_typ(get_sexpr_type(actual))
" expected " ^ string_of_typ ft ^ " in " ^ string_of_s_expr actual))
  in
let _ = if List.length(exprs) != List.length(fd.formals) then
raise (Failure ("expecting " ^ string_of_int
(List.length fd.formals) ^ " argument(s) in " ^ string_of_expr(Call(fname,exprs))))
else
List.iter2 match_params fd.formals actuals
  in
S_Call(fname,actuals,fd.typ)

and check_assign (env,e1,e2) =
let sexpr1,_ = generate_sexpr(env,e1) in
let sexpr2,_ = generate_sexpr(env,e2) in
let type1 = get_sexpr_type (sexpr1) in
let type2 = get_sexpr_type (sexpr2) in
if (type1 = type2) then   (* assignarray[] *)
S_Assign(sexpr1,sexpr2,type1)
else if (type1 = Float && type2 = Int) then
S_Assign(sexpr1,S_Cast(Int,sexpr2,Float) ,type1)
else if (type1 = Arraytype(Int) && type2 = Int) then
S_Assign(sexpr1,sexpr2,Int)
else if (type1 = Arraytype(Bool) && type2 = Bool) then
S_Assign(sexpr1,sexpr2,Bool)
else if (type1 = Arraytype(Float) && type2 = Float) then
S_Assign(sexpr1,sexpr2,Float)
else if (type1 = Arraytype(String) && type2 = String) then
S_Assign(sexpr1,sexpr2,String)
else
raise(Failure ("illegal assignment " ^ string_of_typ type1 ^
" = " ^ string_of_typ type2 ^ " in " ^
string_of_s_expr sexpr1 ^ " = " ^ string_of_s_expr sexpr2))

and check_unop (env,op,e) =
let sexpr,_ = generate_sexpr(env,e) in
let type1 = get_sexpr_type(sexpr) in
match(op) with
Neg -> if (type1 = Int || type1 = Float) then S_Unop(op,sexpr,type1)
else raise (Failure ("illegal unary operator " ^ string_of_uop op ^
 string_of_typ(type1) ^ " in " ^ string_of_expr(Unop(op,e)) ))
| Not -> if (type1 = Bool) then S_Unop(op,sexpr,type1)
else raise (Failure ("illegal unary operator " ^ string_of_uop op ^
 string_of_typ(type1) ^ " in " ^ string_of_expr(Unop(op,e)) ))
and check_binop (env,e1,op,e2) =
let origin_sexpr1,_ = generate_sexpr(env,e1) in
let origin_sexpr2,_ = generate_sexpr(env,e2) in
let origin_type1 = get_sexpr_type (origin_sexpr1) in
let origin_type2 = get_sexpr_type (origin_sexpr2) in
let (sexpr1,type1) =
if (origin_type1 = Int && origin_type2 = Float) then
(S_Cast(Int,origin_sexpr1,Float),Float)
else
(origin_sexpr1,origin_type1) in
let (sexpr2,type2) =
if (origin_type1 = Float && origin_type2 = Int) then
(S_Cast(Int,origin_sexpr2,Float),Float)
else
(origin_sexpr2,origin_type2) in
match(op) with
Add  when ((type1 = Int && type2 = Int) || (type1 = String && type2 = String) || (type1 = Fl
  -> S_Binop(sexpr1,op,sexpr2,type1)
| Sub | Mult | Div when ((type1 = Int && type2 = Int) || (type1 = Float && type2 = Float))
-> S_Binop(sexpr1,op,sexpr2,type1)
    | Equal when (type1 = type2 && type1 = String) ->
S_Binop(S_Literal(0),Equal,S_Call("strcmp",[sexpr1;sexpr2;],Int),Bool)
| Neq when (type1 = type2 && type1 = String) ->
S_Unop(Not,S_Binop(S_Literal(0) ,Equal,S_Call("strcmp",[sexpr1;sexpr2;],Int),Bool),Bool)
| Equal | Neq when (type1 = type2) ->
 S_Binop(sexpr1,op,sexpr2,Bool)
    | Less | Leq | Greater | Geq when ((type1 = Int && type2 = Int) || (type1 = Float && typ
-> S_Binop(sexpr1,op,sexpr2,Bool)
    | And | Or when (type1 = Bool && type2 = Bool) -> S_Binop(sexpr1,op,sexpr2,Bool)
| _ -> raise (Failure ("illegal binary operator " ^
string_of_typ type1 ^ " " ^ string_of_op op ^ " " ^
string_of_typ type2 ^ " in " ^ string_of_expr(Binop(e1,op,e2))))
and check_cast (env,t,e) =
let sexpr,_ = generate_sexpr(env,e) in
let oldt = get_sexpr_type(sexpr) in
match (oldt,t) with
 (Int,Int) | (Float,Float) | (Bool,Bool) | (String,String)
 | (Float,Int) | (Int,Float) | (String,Int) | (String,Float)
 | (Float,String) | (Int,String) | (Bool,String) ->
S_Cast(oldt,sexpr,t)
| _ ->
raise(Failure("Covert from " ^ string_of_typ(oldt) ^ " to " ^ string_of_typ(t) ^ "is not per
and check_objaccess(env,e1,e2) =
let sexpr1,_ = generate_sexpr(env,e1) in
let objtyp = get_sexpr_type sexpr1 in
match(objtyp) with
   Object(cname) -> check_objaccess_class_property(env,cname,sexpr1,e1,e2)
|  _ -> raise (Failure ("illegal accessing:" ^ string_of_expr e1 ^ " is not an object"))
and check_objaccess_class_property(env,cname,sexpr1,e1,e2) =
match(e2) with
Id s -> check_objaccess_class_field(env,cname,sexpr1,s)
|   Call(s, exprs) -> check_objaccess_class_method(env,cname,sexpr1,e1,e2,s,exprs)
|  ObjAccess(e3,e4) -> let e1' = ObjAccess(e1,e3) in check_objaccess(env,e1',e4)
|  _ -> raise (Failure ("illegal accessing:" ^ string_of_expr e2 ^ " is not a class property
and check_objaccess_class_field(env,cname,sexpr1,s) =
let property_map = try StringMap.find cname env.env_class_map with | Not_found -> raise (Fai
let e2_typ = try StringMap.find s property_map.field_map with | Not_found -> raise (Failure(
let sexpr2 = S_Id(s,e2_typ) in S_ObjAccess(sexpr1, sexpr2, e2_typ)
and check_objaccess_class_method(env,cname,sexpr1,e1,e2,s,exprs) =
if cname = "constructor" then raise(Failure("illegal accessing: constructors cannot be acces
let property_map = try StringMap.find cname env.env_class_map with | Not_found -> raise (Fai
let e2_fdecl = try StringMap.find s property_map.method_map with | Not_found -> raise (Failu
let actuals,_ = exprl_to_sexprl(env,exprs) in
let match_params = fun (ft,_) actual ->
if (get_sexpr_type(actual) = ft) then
()
else raise(Failure ("illegal actual argument found " ^ string_of_typ(get_sexpr_type(actual))
" expected " ^ string_of_typ ft ^ " in " ^ string_of_expr(Call(s,exprs))))
  in
let _ = if List.length(exprs) != List.length(e2_fdecl.formals) then
raise (Failure ("expecting " ^ string_of_int
(List.length e2_fdecl.formals) ^ " argument(s) in " ^ string_of_expr e1 ^ "." ^ string_of_ex
else
List.iter2 match_params e2_fdecl.formals actuals
  in
S_ObjAccess(sexpr1,S_Call(s,actuals,e2_fdecl.typ),e2_fdecl.typ)

and is_typ_match(actuals,formals) =
(match actuals with [] -> Bool | h::l -> get_sexpr_type(h)) = (match formals with [] -> Bool
is_typ_match((match actuals with [] -> [] | h::l -> l),(match formals with [] -> [] | h::l -

and is_fdecl_match(env,fdecl,exprs) =
if List.length(exprs) != List.length(fdecl.formals) then false else
let actuals,_ = exprl_to_sexprl(env,exprs) in
is_typ_match(actuals,fdecl.formals)

and find_and_test_constructor(fname,cname,sel) =
let appended_constructor_name = cname ^ ".constructor" ^
if List.length sel > 0 then
"." ^ String.concat "." (List.map (fun sexpr -> string_of_typ (get_sexpr_type sexpr)) sel)
else
""
in
fname = appended_constructor_name
```

```
and check_objcreate(env,cname,el) =
let property_map = try StringMap.find cname env.env_class_map with | Not_found -> raise (Fai

let sel = List.rev(List.fold_left (fun sl e -> let sexpr, _ = generate_sexpr(env,e) in sexpr

if StringMap.exists (fun k v -> find_and_test_constructor(k,cname,sel)) property_map.method_
S_ObjCreate(cname,sel,Object(cname))
else
raise (Failure("No matching constructor for " ^ cname ^ "(" ^ String.concat "," (List.map st

and check_array_init(datatype,size) =
S_ArrayCreate(datatype, size)

and check_array_access(env,e,index) =
let se,_ = generate_sexpr(env, e) in
let typ = match e with
  Id s -> get_id_type(env,s)
in
S_ArrayAccess(se, index, typ)

and check_delete(env,e) =
let se,_ = generate_sexpr(env,e) in
let t = get_sexpr_type(se) in
match t with
Object(_) | Arraytype(_) -> S_Delete(se)
|  _ -> raise(Failure("illegal delete: " ^ string_of_expr e ^ " is not an object"))
and generate_sexpr (env,expr) =
match (expr) with
Literal i -> S_Literal(i), env
|  BoolLit b -> S_BoolLit(b), env
|  StringLit f -> S_StringLit(f), env
|  FloatLit f -> S_FloatLit(f), env
|  Id s -> S_Id(s, get_id_type(env,s)), env
|  Call(s, exprs) -> check_call_type(env,s,exprs), env
|  Assign(e1, e2) -> check_assign(env,e1,e2), env
|  Unop(op, e)-> check_unop(env,op,e), env
|  Binop(e1, op, e2)-> check_binop(env,e1,op,e2),env
|  Noexpr -> S_Noexpr,env
|  Nullexpr -> S_Null,env
| Cast(t,e) -> check_cast(env,t,e),env
    |  ObjAccess(e1, e2) -> check_objaccess(env,e1,e2),env
|  ObjCreate(cname, el) -> check_objcreate(env,cname,el), env
|   ArrayCreate(t, n) -> check_array_init(t,n),env
|   ArrayAccess(e, index) -> check_array_access(env,e,index),env
|   Delete(e) -> check_delete(env,e), env
```

```
let rec stmtl_to_sstmtl (env,stmt_list) =
let env_ref = ref(env) in
let rec iter = function
   head::tail ->
let a_head, env = generate_sstmt(!env_ref,head) in
env_ref := env;
a_head::(iter tail)
|  [] -> []
in
let sstmt_list = (iter stmt_list), !env_ref in
sstmt_list

and check_block (env,sl) =  match sl with
[] -> S_Block([S_Expr(S_Noexpr,Void)]), env
|  _  ->
let s_sl, _ = stmtl_to_sstmtl(env,sl) in
(S_Block(s_sl), env)

and check_expr_stmt (env,e) =
let se, env = generate_sexpr(env,e) in
let t = get_sexpr_type(se) in
S_Expr(se, t), env

and check_return (env,e) =
let se, _ = generate_sexpr (env,e) in
let t = get_sexpr_type(se) in
if (t = env.env_return_type) then
S_Return(se,t), env
else
raise (Failure ("return gives " ^ string_of_typ t ^ " expected " ^
string_of_typ env.env_return_type ^ " in " ^ string_of_s_expr se))

and check_if (env,p,b1,b2) =
let se, _ = generate_sexpr(env,p) in
let t = get_sexpr_type(se) in
let ifbody, _ = generate_sstmt (env,b1) in
let elsebody, _ = generate_sstmt (env,b2) in
if t = Bool
then S_If(se, ifbody, elsebody), env
    else raise (Failure ("expected Boolean expression in " ^ string_of_s_expr se))
and check_for (env,e1,e2,e3,s) =
let se1, _ = generate_sexpr(env,e1) in
let se2, _ = generate_sexpr(env,e2) in
let se3, _ = generate_sexpr(env,e3) in
let forbody, _ = generate_sstmt(env,s) in
let conditional = get_sexpr_type(se2) in
let sfor =
if (conditional = Bool) then (* Could be void too *)
S_For(se1, se2, se3, forbody)
else raise (Failure ("invaild statement type in For"))
in
(sfor, env)
and check_while (env,p,s) =
let se, _ = generate_sexpr(env,p) in
let t = get_sexpr_type(se) in
let sstmt, _ = generate_sstmt(env,s) in
let swhile =
if (t = Bool ) then (* Could be void too *)
S_While(se, sstmt)
else raise (Failure ("invaild statement type in While"))
in
(swhile, env)
and  generate_sstmt (env,stmt) =
 match (stmt) with
Block sl ->check_block(env,sl)
|  Expr e -> check_expr_stmt(env,e)
    |   Return e -> check_return(env,e)
    |  If(p, b1, b2) -> check_if(env,p,b1,b2)
    |  For(e1, e2, e3, st) -> check_for(env,e1,e2,e3,st)
    |   While(p, s) -> check_while(env,p,s)

let check_fbody(fname,fbody,return_type) =
let len = List.length fbody in
if len = 0 then () else
let final_stmt = List.hd (List.rev fbody) in
match return_type, final_stmt with
Void, _ -> ()
| Constructortyp, _ -> ()
|  _, S_Return(_, _) -> ()
|  _ -> raise(Failure ("Missing return statement: " ^ fname))

(*
let check_foramls (fname, formals) =
if (List.length(List.sort_uniq(formals)) <> List.length(formals)) then
raise(Failure ("duplicate formal in ")^fname)
else ()
*)

let generate_sfdecl (fdecl,function_map,global_map,class_map) =
let local_map = List.fold_left (fun m (t,n) ->
if (t = Void) then
raise(Failure("local " ^ n ^ " shouldn't be type void"))
else
StringMap.add n t m)
StringMap.empty fdecl.locals in
let param_map = List.fold_left (fun m (t,n) ->
if (t = Void) then
raise(Failure("formal " ^ n ^ " shouldn't be type void"))
else
StringMap.add n t m)
        StringMap.empty fdecl.formals in
let env =
{
env_class_map = class_map;
env_global_map = global_map;
env_local_map = local_map;
env_param_map = param_map;
env_function_map = function_map;
env_return_type = fdecl.typ;
} in
let sfbody = List.fold_left (fun lst stmt ->
let sstmt, _ = generate_sstmt(env,stmt) in
lst@[sstmt] ) [] fdecl.body in
ignore(check_fbody(fdecl.fname,sfbody,fdecl.typ));
ignore(report_duplicate (fun n -> "duplicate formals " ^ n) (List.map snd fdecl.formals));
ignore(report_duplicate (fun n -> "duplicate locals " ^ n) (List.map snd fdecl.locals));

(*ignore(check_foramls(fdecl.fname,fdecl.formals));*)
{
s_typ = fdecl.typ;
s_fname = fdecl.fname;
s_formals = fdecl.formals;
s_locals = fdecl.locals;
s_body = sfbody;
}

let get_main(functions) =
let find_main = (fun f -> match f.fname with "main" -> true  | _ -> false) in
let mains = (List.find_all find_main functions) in
if List.length mains < 1 then
raise (Failure("Main didn't defined"))
else if List.length mains > 1 then
raise (Failure("MultipleMainsDefined"))
else List.hd mains

let append_to_name(cname,formals,name) =
cname ^ "." ^ name ^
```

```
if List.length formals > 0 then
"." ^ String.concat "." (List.map (fun (typ,_) -> string_of_typ typ) formals)
else

""

let append_to_callname(formals,name) =
name ^
if List.length formals > 0 then
"." ^ String.concat "." (List.map (fun (typ,_) -> string_of_typ typ) formals)
else

""

(* type s_expr =
    S_Literal of int
  | S_BoolLit of bool
  | S_StringLit of string
  | S_FloatLit of float
  | S_Id of string * typ
  | S_Binop of s_expr * op * s_expr * typ
  | S_Unop of uop * s_expr * typ
  | S_Assign of s_expr * s_expr * typ
  | S_Call of string * s_expr list * typ
  | S_Noexpr
  | S_Null
  | S_ObjCreate of string * s_expr list * typ
  | S_ObjAccess of s_expr * s_expr * typ
  | S_Delete of s_expr *)

let rec implicit_this_in_constructor_sexpr(c_typ,s_expr,property_map) =
match (s_expr) with
S_Id(s,typ) -> if StringMap.mem s property_map.field_map then S_ObjAccess(S_Id("this",c_typ)
|   S_Binop(se1,op,se2,typ) -> S_Binop(implicit_this_in_constructor_sexpr(c_typ,se1,property
|   S_Unop(uop,se,typ) -> S_Unop(uop,implicit_this_in_constructor_sexpr(c_typ,se,property_ma
|   S_Assign(se1,se2,typ) -> S_Assign(implicit_this_in_constructor_sexpr(c_typ,se1,property_
|   S_Call(s,sel,typ) -> if StringMap.mem s property_map.method_map then
S_Call(s,List.rev (List.fold_left (fun lst se -> implicit_this_in_constructor_sexpr(c_typ,se
 else S_Call(s,sel,typ)
|   S_ObjCreate(s,sel,typ) -> S_ObjCreate(s,List.rev (List.fold_left (fun lst se -> implicit
|   S_ObjAccess(se1,se2,typ) -> (match se1 with
S_Id("this",_) -> S_ObjAccess(se1,se2,typ)
|  _ -> S_ObjAccess(implicit_this_in_constructor_sexpr(c_typ,se1,property_map),implicit_this
|   S_Delete(se) -> S_Delete(implicit_this_in_constructor_sexpr(c_typ,se,property_map))
|   _ -> s_expr
```

```
let rec implicit_this_in_constructor_stmt(c_typ,s_stmt,property_map) =
match (s_stmt) with
S_Block(sl) -> S_Block(implicit_this_in_constructor_body(c_typ,sl,property_map))
|   S_Expr(s,t) -> S_Expr(implicit_this_in_constructor_sexpr(c_typ,s,property_map),t)
|   S_Return(_,_) -> raise (Failure("Constructors cannot return a value"))
|   S_If(s,st1,st2) -> S_If(implicit_this_in_constructor_sexpr(c_typ,s,property_map),implici
implicit_this_in_constructor_stmt(c_typ,st2,property_map))
|   S_For(s1,s2,s3,st) -> S_For(implicit_this_in_constructor_sexpr(c_typ,s1,property_map),in
implicit_this_in_constructor_sexpr(c_typ,s3,property_map),implicit_this_in_constructor_stmt(
|   S_While(s,st) -> S_While(implicit_this_in_constructor_sexpr(c_typ,s,property_map),implic

and implicit_this_in_constructor_body (c_typ,s_body,property_map) =
let modified_s_body =
List.fold_left (fun lst s_stmt -> implicit_this_in_constructor_stmt(c_typ,s_stmt,property_ma
in
List.rev modified_s_body

let implicit_constructor_body (c_typ,s_body,property_map) =
[S_Expr(S_Assign(
S_Id("this",c_typ),
S_Call("cast",
[
S_Call("malloc",
[
S_Call("sizeof",
[
S_Id("this",c_typ)
],Int)
],Any)
],c_typ),
c_typ
),c_typ)]
@ implicit_this_in_constructor_body(c_typ,s_body,property_map) @
[S_Return(S_Id("this",c_typ),c_typ)]


let update_sconstrdecl(sfdecl,cname,property_map) =
{
s_typ = Object(cname);
s_fname = sfdecl.s_fname;
s_formals = sfdecl.s_formals;
s_locals =  sfdecl.s_locals;
s_body = implicit_constructor_body(Object(cname),sfdecl.s_body,property_map);
}
```

```
let update_sfdecl(sfdecl,cname) =
{
s_typ = sfdecl.s_typ;
s_fname = cname ^ "." ^ sfdecl.s_fname;
s_formals = sfdecl.s_formals;
s_locals = sfdecl.s_locals;
s_body = sfdecl.s_body;
}

let generate_scdecl (cdecl,class_map) =
let property_map = try StringMap.find cdecl.cname class_map with | Not_found -> raise (Failu
{
s_cname = cdecl.cname;
s_fields = cdecl.cbody.fields;
s_constructors = (let raw_s_constructors = List.fold_left (fun lst fdecl ->
generate_sfdecl(fdecl,property_map.method_map,property_map.field_map,class_map)::lst)
[] cdecl.cbody.constructors in
List.fold_left (fun lst sfdecl ->
  update_sconstrdecl(sfdecl,cdecl.cname,property_map)::lst)
  [] raw_s_constructors);
s_methods = (let raw_s_methods = List.fold_left (fun lst fdecl ->
generate_sfdecl(fdecl,property_map.method_map,property_map.field_map,class_map)::lst)
[] cdecl.cbody.methods in
List.fold_left (fun lst sfdecl ->
update_sfdecl(sfdecl,cdecl.cname)::lst)
[] raw_s_methods);
}

let generate_sast (globals,functions,builtins,classes,function_map,global_map,class_map) =
let smain = generate_sfdecl(get_main(functions),function_map,global_map,class_map) in
let sfdecls = List.fold_left (fun lst fdecl ->
if (fdecl.fname = "main") then lst else
generate_sfdecl(fdecl,function_map,global_map,class_map)::lst) [] functions in
let sbuiltins = List.fold_left (fun lst fdecl -> generate_sfdecl(fdecl,function_map,global_m
let scdecls = List.fold_left (fun lst cdecl -> generate_scdecl(cdecl,class_map)::lst) [] cla
(globals,sfdecls,scdecls,smain,sbuiltins)

let overload_class(classes) =
List.fold_left (fun lst cdecl ->
{
cname = cdecl.cname;
cbody =
{
fields = cdecl.cbody.fields;
methods = List.fold_left (fun lst fdecl -> {
typ = fdecl.typ;
```

```
fname = append_to_callname((Object(cdecl.cname),"this")::fdecl.formals,fdecl.fname);

formals = (Object(cdecl.cname),"this")::fdecl.formals;
locals = fdecl.locals;
body = fdecl.body;
}::lst) [] cdecl.cbody.methods;
constructors = List.fold_left (fun lst condecl -> {
typ = condecl.typ;
     fname = append_to_name(cdecl.cname,condecl.formals,condecl.fname);
     formals = condecl.formals;
     locals = (Object(cdecl.cname),"this") :: condecl.locals;
    body = condecl.body;
}::lst) [] cdecl.cbody.constructors;
}
}::lst)
[] classes

let overload_callname(s,el,env) =
let newS =
s ^
if List.length el > 0 then
"." ^ String.concat "." (List.map (fun expr -> let sexpr,_ = generate_sexpr(env,expr) in str
else
""
in
newS

let rec overload_expr(expr,env) =
match (expr) with
ObjAccess(e1,e2) -> (
match e2 with
Call(s,el) -> let modified_el = List.rev(List.fold_left(fun lst expr -> overload_expr(expr,e
  ObjAccess(e1,Call(overload_callname(s,modified_el,env),modified_el))
|   ObjAccess(e3,e4) -> let e1' = ObjAccess(e1,e3) in overload_expr(ObjAccess(e1',e4),env)
|  _ -> expr
)
|  Binop(e1,op,e2) -> Binop(overload_expr(e1,env),op,overload_expr(e2,env))
|   Unop(uop,e) -> Unop(uop,overload_expr(e,env))
|   Assign(e1,e2) -> Assign(overload_expr(e1,env),overload_expr(e2,env))
|  Call(s,el) -> Call(s,List.rev (List.fold_left (fun lst e -> overload_expr(e,env)::lst)[]
|  ObjCreate(s,el) -> ObjCreate(s,List.rev (List.fold_left (fun lst e -> overload_expr(e,env
|  _ -> expr

let rec overload_stmt(stmt,env) =
match (stmt) with
```

```
Block(sl) -> Block(overload_function_body(sl,env))
|  Expr(e) -> Expr(overload_expr(e,env))
|  Return(e) -> Return(overload_expr(e,env))
|   If(e,st1,st2) -> If(overload_expr(e,env),overload_stmt(st1,env),overload_stmt(st2,env))
|   For(e1,e2,e3,st) -> For(overload_expr(e1,env),overload_expr(e2,env),overload_expr(e3,env
|  While(e,st) -> While(overload_expr(e,env),overload_stmt(st,env))

and overload_function_body(body,env) =
let modified_body =
List.fold_left (fun lst stmt -> overload_stmt(stmt,env)::lst) [] body
in
List.rev modified_body

let overload_function(functions,function_map,global_map,class_map) =
List.fold_left (fun lst fdecl ->
let local_map = List.fold_left (fun m (t,n) ->
if (t = Void) then
raise(Failure("local " ^ n ^ " shouldn't be type void"))
else
StringMap.add n t m)
StringMap.empty fdecl.locals in
let param_map = List.fold_left (fun m (t,n) ->
if (t = Void) then
raise(Failure("formal " ^ n ^ " shouldn't be type void"))
else
StringMap.add n t m)
        StringMap.empty fdecl.formals in
let env =
{
env_class_map = class_map;
env_global_map = global_map;
env_local_map = local_map;
env_param_map = param_map;
env_function_map = function_map;
env_return_type = fdecl.typ;
} in
{
typ = fdecl.typ;
fname = fdecl.fname;
formals = fdecl.formals;
locals = fdecl.locals;
body = overload_function_body(fdecl.body,env);
}::lst)
[] functions
```

```
let overload_body_class(classes,class_map) =
List.fold_left (fun lst cdecl ->
let property_map = try StringMap.find cdecl.cname class_map with | Not_found -> raise  (Fai
{
cname = cdecl.cname;
cbody =
{
fields = cdecl.cbody.fields;
methods = overload_function(cdecl.cbody.methods,property_map.method_map,property_map.field_
constructors = overload_function(cdecl.cbody.constructors,property_map.method_map,property_
}
}::lst)
[] classes

(* Main method for analyzer *)
let analyze program =
match program with
(globals, functions, classes) ->
let builtins = define_builtin_functions in
let modified_classes = overload_class(classes) in
let function_map, global_map = build_maps(functions,builtins,globals) and class_map = build_

let modified_body_classes = overload_body_class(modified_classes,class_map) in
let modified_functions = overload_function(functions,function_map,global_map,class_map) in
let modified_function_map, _ = build_maps(modified_functions,builtins,globals) in
generate_sast(globals,modified_functions,builtins,modified_body_classes,modified_function_m
```

codegen.ml

```
open Llvm

module L = Llvm
module A = Ast
module S = Sast


module StringMap = Map.Make(String)

module H = Hashtbl

module P=Printf
let counter : int ref = ref 0
let object_types:(string, L.lltype) H.t = H.create 10;;
let object_field_indices:(string, int) H.t = H.create 50;;
let globals_table:(string, L.llvalue) H.t = H.create 50;;
let locals_table:(string, L.llvalue) H.t = H.create 50;;
let params_table:(string, L.llvalue) H.t = H.create 50;;
let new_obj_table:(int, L.llvalue) H.t = (H.create 50);;
let func_table:(string, L.llvalue * S.s_func_decl) H.t = H.create 50;;

let log_to_file str=
ignore()

let print_hashtable_int table =
ignore()

let print_hashtable table =
  H.iter (fun k v -> log_to_file (k^(L.string_of_llvalue v)^"\n")) table;;

let context = L.global_context ()
let the_module = L.create_module context "CGC"
and i32_t  = L.i32_type  context (* integer *)
and i8_t   = L.i8_type   context
and i1_t   = L.i1_type   context (* boolean *)
and i8_pt  = L.pointer_type (L.i8_type context) (* string *)
and void_t = L.void_type context  (* void *)
and f_t = L.double_type context;;


let rec find_object_type name =
  try Hashtbl.find object_types name
  with | Not_found -> raise (Failure("no class def of " ^ name))

(* convert sast type to underlying *)
and get_underlying_type_of_sexpr = function
    S.S_Literal(_) -> i32_t
  | S.S_BoolLit(_)  -> i1_t
  | S.S_FloatLit(_) -> f_t
  | S.S_StringLit(_) -> i8_pt
  | S.S_Id(_, data_type) -> ltype_of_typ data_type
  | S.S_Unop(_, _, data_type) -> ltype_of_typ data_type
  | S.S_Binop(_, _, _, data_type) -> ltype_of_typ data_type
  | S.S_Assign(_, _, data_type) -> ltype_of_typ data_type
  | S.S_Call(_, _, data_type) -> ltype_of_typ data_type
  | S.S_Noexpr -> void_t
  | S.S_Cast(_,_,data_type) ->  ltype_of_typ data_type
  | S.S_ObjAccess(_, _, data_type) -> ltype_of_typ data_type
  | S.S_ObjCreate(_, _, data_type)  ->  ltype_of_typ data_type
  (*| S.SArrayPrimitive(_, d) -> d*)
  | S.S_Null -> i32_t
  |  d -> raise(Failure ("Met unknown type in get_underlying_type_of_sexpr\n") )

and get_ast_type_of_sexpr = function
    S.S_Literal(_) -> A.Int
  | S.S_BoolLit(_)  -> A.Bool
  | S.S_FloatLit(_) -> A.Float
  | S.S_StringLit(_) -> A.String
  | S.S_Id(_, data_type) -> data_type
  | S.S_Unop(_, _, data_type) -> data_type
  | S.S_Binop(_, _, _, data_type) -> data_type
  | S.S_Assign(_, _, data_type) -> data_type
  | S.S_Call(_, _, data_type) -> data_type
  | S.S_Noexpr -> A.Void
  | S.S_ObjAccess(_, _, data_type) -> data_type
  | S.S_ObjCreate(_, _, data_type)  ->  data_type
  | S.S_Cast(_,_,data_type) ->  data_type
  (*| S.SArrayPrimitive(_, d) -> d*)
  | S.S_ArrayAccess(arrayName, index, t) -> t
  | S.S_ArrayCreate(typ, size) -> typ
  | S.S_Null -> A.Int
  |  d -> raise(Failure ("Met unknown type in get_ast_type_of_sexpr\n") )

and ltype_of_typ = function
    A.Int -> i32_t
  | A.Bool -> i1_t
  | A.Void -> void_t
  | A.String -> i8_pt
  | A.Float -> f_t
  | Object(name) ->  L.pointer_type(find_object_type name)
  | A.Arraytype(t) -> L.pointer_type (ltype_of_typ t)
  |  d -> raise(Failure ("Met unknown type in ltype_of_typ\n") )

(* Return the value for a variable or formal or global argument *)
and lookup the_name =
  try H.find locals_table the_name
  with | Not_found ->
    (try  H.find params_table the_name
     with | Not_found ->
       (try H.find globals_table the_name
        with | Not_found -> raise (Failure("Unbound var " ^ the_name))
       )
    )

(* Get the built function from the module *)
and  func_lookup fname =
  match (L.lookup_function fname the_module) with
```

```
    None  -> raise (Failure ("No function "^fname) )
  | Some f  -> f

let rec codegen_globals globals =
  ignore(log_to_file "Codegen globals\n");

  (* Declare each global variable; remember its value in a map *)
  let add_to_global_table table (t, n) =
    let init = L.const_int (ltype_of_typ t) 0 in
    H.add table n (L.define_global n init the_module)
  in
  ignore(List.map (add_to_global_table globals_table) globals);
  ignore(print_hashtable globals_table);
  log_to_file "Generated globals\n"

(* Define each function (arguments and return type) so we can call it *)
and codegen_function_decls functions =
  ignore(log_to_file "Codegen function declarations\n");
  ignore(List.map (fun f -> log_to_file ("This time build function "^f.S.s_fname^"\n") ) fun

  let add_to_func_table table fdecl =
    let name = fdecl.S.s_fname
    and formal_types =
      Array.of_list (List.map (fun (t,_) -> ltype_of_typ t) fdecl.S.s_formals)
    in
    let ftype = L.function_type (ltype_of_typ fdecl.S.s_typ) formal_types in
    H.add table name (L.define_function name ftype the_module, fdecl)
  in
  ignore(List.map (add_to_func_table func_table) functions);
  log_to_file "Generated function declarations\n"


and codegen_builtins func_decls =
  ignore(log_to_file "Codegen builtins\n");
  (* Declare printf(), which the print built-in function will call *)
  let printf_t = L.var_arg_function_type i32_t [| L.pointer_type i8_t |] in
  let printf_func = L.declare_function "printf" printf_t the_module in

  (* Declare malloc *)
  let malloc_ty = L.function_type (i8_pt) [| i32_t |] in
  let malloc_func = L.declare_function "malloc" malloc_ty the_module in

    (* Declare free *)
  let free_ty = L.function_type (void_t) [| i8_pt |] in
  let free_func = L.declare_function "free" free_ty the_module in
  (* Dump the module to check builtin functions *)
  ignore(log_to_file "Generated builtin functions\n")
(* L.dump_module the_module *)

(* Generate classes *)
and  codegen_struct_stub class_struct =
  let struct_t = L.named_struct_type context class_struct.S.s_cname in
  H.add object_types class_struct.S.s_cname struct_t


and codegen_struct class_struct =
  let struct_t = H.find object_types class_struct.S.s_cname in
  let type_list = List.map (fun (data_type,name) -> ltype_of_typ data_type) class_struct.S.s
  let name_list = List.map (fun (data_type, name) -> name) class_struct.S.s_fields in

  let type_array = (Array.of_list type_list) in
  List.iteri (fun index field ->
      let n = class_struct.S.s_cname ^ "." ^ field in
      H.add object_field_indices n index;
    ) name_list;
  L.struct_set_body struct_t type_array true

(* Pass constructor to this function *)
and codegen_obj_create fname expr_list data_type llbuilder =
  (* Call the full name of the function *)
  let class_name = match data_type with
      A.Object(obj_name) -> obj_name
    | d -> raise (Failure ("Data type in Obj_Create is not an object but "^(A.string_of_typ
  in
  let string_of_params params =
    String.concat "." (List.map (fun p -> (S.string_of_expr_type p ) ) params )
  in
  let param_str = (match expr_list with
        [] -> ""
      | _ -> "." ^ (string_of_params expr_list)
    )in
  let full_name = (class_name ^ ".constructor"^param_str) in
  let f = func_lookup full_name in
  let generate_param_func builder isReturn param =
    match param with
      S.S_Id(id,A.Object(obj_name)) ->
      lookup id
    | _ ->
      codegen_sexpr llbuilder false param
  in
  let params = List.map (generate_param_func llbuilder false) expr_list in
  let obj = L.build_call f (Array.of_list params) "tmp" llbuilder in
  obj

and codegen_id id d llbuilder =
  try  (
    let _val = H.find locals_table id in
    ignore(log_to_file ("Found "^id^" in local table\n") );
    L.build_load _val id llbuilder
  )
  with | Not_found ->
    (try let _val = H.find params_table id in
       ignore(log_to_file ("Found "^id^" in params\n") );
       _val
     with | Not_found ->
       (try let _val = H.find globals_table id in
          ignore(log_to_file ("Found "^id^" in global table\n") );
          L.build_load _val id llbuilder
        with | Not_found -> raise (Failure("Unbound var " ^ id))
       )
    )
and codegen_id_obj_in_param id d llbuilder =
  lookup id

(* Very possibly we need to rewrite this *)
and codegen_obj_access isAssign the_obj the_field data_type llbuilder =
  ignore(log_to_file "Building obj access\n");
  ignore(log_to_file ((S.string_of_s_expr the_obj)^"\n") );
  ignore(log_to_file ((S.string_of_s_expr the_field)^"\n") );

  ignore(log_to_file "Now we need object field index table\n");
  ignore(print_hashtable_int object_field_indices);

  (* LHS can be ID or a nested Obj_Access *)
  ignore(log_to_file "Generating parent\n");
  let codegen_lhs = function
      S.S_Id(id, data_type) ->
      ignore(log_to_file "LHS of obj access is just the obj\n");
      let _found = lookup id in
      _found
    | S.S_ArrayAccess(arrayName, index, t) -> generate_array_access true arrayName index llb
    | S.S_ObjAccess(o,f,t)->
      (* Need to load lhs before accessing further *)
      let _val = codegen_obj_access isAssign o f t llbuilder in
      L.build_load _val "load_nested_obj" llbuilder
    |  se  ->
      ignore(log_to_file "Got unhandled LHS in obj access:\n");
      ignore(log_to_file ("Type is "^(S.string_of_s_expr se)^"\n" ) );
      raise( Failure "Unhandled case in LHS in obj_access\n ")
  in
  let parent = codegen_lhs the_obj in
  ignore(log_to_file ("Generated parent: "^(L.string_of_llvalue parent)^"\n") );

  (* RHS can be ID *)
  ignore(log_to_file "Generating field\n");

  let rec codegen_rhs (parent_expr:L.llvalue) (parent_type:A.typ) field_expr=
    match field_expr with
      S.S_Id(id,data_type) ->
      (
        (* Find type of parent *)
        let search_term = ((S.string_of_ast_typ parent_type) ^ "." ^ id) in
        let field_index = Hashtbl.find object_field_indices search_term in
        let _val = L.build_struct_gep parent_expr field_index id llbuilder in
        let _mat = match data_type with
            A.Object(name)->
            let _store =
              if isAssign then
                _val
              else begin
                let _load = L.build_load _val id llbuilder in
                _load
              end
            in
            _store
          | _ ->
              _val
        in
        _mat
      )
    | S_ArrayAccess(arrayName, index, t) ->
      let ce  = codegen_rhs parent_expr parent_type arrayName in
      let index = L.const_int i32_t index in
      let index = L.build_add index (L.const_int i32_t 1) "temp_afterAdd1" llbuilder in
      let _val = build_gep ce [| index |] "tmp" llbuilder in
      if isAssign
          then _val
          else build_load _val "tmp" llbuilder

    | S.S_Call (fname, act, _) ->
      ignore(log_to_file ("Build object function "^fname^" with expr: \n") );
      ignore(List.map (fun expr -> log_to_file ((S.string_of_s_expr expr) ^ "\n") ) act );
      let class_name = A.string_of_typ parent_type in
```

```
      let func_name = class_name^"."^fname in
      ignore(log_to_file ("Function name is "^func_name^"\n"));
      let (fdef, fdecl) = H.find func_table func_name in
      let actuals = List.rev (List.map (codegen_sexpr llbuilder true) (List.rev act)) in
      let result = (match fdecl.S.s_typ with A.Void -> ""
                                        | _ -> func_name ^ "_result") in
      L.build_call fdef (Array.of_list actuals) result llbuilder
    | se ->
      ignore(log_to_file "Got unhandled RHS in obj access:\n");
      ignore(log_to_file ("Type is "^(S.string_of_s_expr se)^"\n" ) );
      raise( Failure "Unhandled case in RHS in obj_access\n ")
  in


  let lhs_type = get_ast_type_of_sexpr the_obj in
  let lhs = codegen_lhs the_obj in
  let rhs = codegen_rhs lhs lhs_type the_field in
  rhs

(* Check type of obj field *)

and get_value deref vname builder =
  if deref then
    let var = try H.find locals_table vname with
      | Not_found -> try H.find params_table vname with
        | Not_found -> try H.find globals_table vname with
          | Not_found -> raise (Failure("unknown variable name " ^ vname))
    in
    L.build_load var vname builder

  else
    let var = try H.find locals_table vname with
      | Not_found -> try H.find params_table vname with
        | Not_found -> try H.find globals_table vname with
          | Not_found -> raise (Failure("unknown variable name " ^ vname))
    in
    var

and initialise_array arr arr_len init_val start_pos llbuilder =
  let new_block label =
    let f = L.block_parent (L.insertion_block llbuilder) in
    L.append_block (context) label f
  in
  let bbcurr = L.insertion_block llbuilder in
  let bbcond = new_block "array.cond" in
  let bbbody = new_block "array.init" in
  let bbdone = new_block "array.done" in
  ignore (L.build_br bbcond llbuilder);
  L.position_at_end bbcond llbuilder;

  (* Counter into the length of the array *)
  let counter = L.build_phi [L.const_int i32_t start_pos, bbcurr] "counter" llbuilder in
  add_incoming ((L.build_add counter (L.const_int i32_t 1) "tmp" llbuilder), bbbody) counte
  let cmp = build_icmp Icmp.Slt counter arr_len "tmp" llbuilder in
  ignore (L.build_cond_br cmp bbbody bbdone llbuilder);
  position_at_end bbbody llbuilder;

  (* Assign array position to init_val *)
  let arr_ptr = build_gep arr [| counter |] "tmp" llbuilder in
  ignore (build_store init_val arr_ptr llbuilder);
  ignore (build_br bbcond llbuilder);
  position_at_end bbdone llbuilder

(* Generate 1 dimension array *)
and generate_one_d_array typ size builder =
  let t = ltype_of_typ typ in

  let size = (L.const_int i32_t size) in

  let size_t = L.build_intcast (L.size_of t) i32_t "1tmp" builder in

  let size = L.build_mul size_t size "2tmp" builder in  (* size * length *)
  let size_real = L.build_add size (L.const_int i32_t 1) "arr_size" builder in

  let arr = L.build_array_malloc t size_real "333tmp" builder in
  let arr = L.build_pointercast arr (L.pointer_type t) "4tmp" builder in

  let arr_len_ptr = L.build_pointercast arr (L.pointer_type i32_t) "5tmp" builder in

  ignore(L.build_store size_real arr_len_ptr builder);
  initialise_array arr_len_ptr size_real (L.const_int i32_t 0) 0 builder;
  arr

and generate_array_access deref arrayName index builder =
  (* let _ = print_int index in *)
  let index = L.const_int i32_t index in
  let index = L.build_add index (L.const_int i32_t 1) "temp_afterAdd1" builder in
  let arr = match arrayName with
    | S.S_Id(name, _) -> get_value true name builder
    | _ -> raise(Failure("No such arrayName:"))
  in
  let _val = L.build_gep arr [| index |] "2tmp" builder in
  if deref
  then L.build_load _val "3tmp" builder
  else _val

(* Construct code for an expression; return its value *)
and codegen_assign lhs rhs llbuilder =
  ignore(log_to_file ("Building assignment: "^(S.string_of_s_expr lhs)^" = "^(S.string_of_s_
  (match lhs with
     S.S_Id (s,t) ->
     (* ignore(print_string ("print type is " ^ A.string_of_typ t)); *)
     let e2' = codegen_sexpr llbuilder false rhs in
     if (String.compare s "this")=0 then begin
       ignore(log_to_file ("LHS of assign is "^s^". Ignore.\n"));e2'
     end
     else begin
       ignore(log_to_file ("LHS of assign is Id "^s^"\n") );
       (* If the rhs is object type, load it first *)
       let rhs_type = get_ast_type_of_sexpr rhs in

       (* Check AST type of expr *)
       ignore(log_to_file ("RHS has AST type "^(A.string_of_typ rhs_type)^"\n"));

       (* Decide whether to load by checking rhs type *)
       match rhs with
         S.S_Literal(_)| S.S_BoolLit(_)| S.S_FloatLit(_)| S.S_Unop(_, _, _)| S.S_Binop(_, _
       | S.S_Cast(_,_,_) ->
         ignore (L.build_store e2' (lookup s) llbuilder);
         e2'
       | S.S_Call(fname, _, data_type) ->
         ignore(log_to_file ("RHS of assign is function call of type "^(A.string_of_typ data
         let _val = L.build_store e2' (lookup s) llbuilder in
         ignore(log_to_file ("Load the value makes: "^(L.string_of_llvalue _val)^"\n"));
         _val
       | S.S_Id(name, data_type) -> (
           match data_type with
             A.Object(name) ->
             ignore(L.build_alloca (ltype_of_typ data_type) name llbuilder);
             ignore (L.build_store e2' (lookup s) llbuilder);
             e2'
           | _ ->
             ignore (L.build_store e2' (lookup s) llbuilder);
             e2'
         )
       | S.S_StringLit(_) | S.S_ArrayCreate(_, _) ->
         ignore (L.build_store e2' (lookup s) llbuilder);
         e2'
```

```
      | S.S_ObjAccess(obj,field,data_type) ->
        ignore(log_to_file ("RHS of assign is object access: "^(S.string_of_s_expr rhs)^"\n
        (* Construct function name to see if it exists *)
        let field_name_trans sexpr = match sexpr with
            S.S_Call(f, el,t) -> f
          | S.S_Id(s,t) -> s
        in
        let class_of_obj sexpr =
          ignore(log_to_file ("Matching sexpr "^(S.string_of_s_expr sexpr)^"\n"));
          match sexpr with
            S.S_Id(_,t) -> A.string_of_typ t
          | S.S_ObjAccess(_,_,t) -> A.string_of_typ t
        in
        let obj_type = class_of_obj obj in
        let pseudo_func_name = obj_type^"."^(field_name_trans field) in
        ignore(log_to_file ("Accessing "^pseudo_func_name^"\n"));

        (* If rhs is function call, do not load *)
        let obj_field =  (match (L.lookup_function pseudo_func_name the_module) with
              None  ->
              (* The field is an id. Load it. *)
              ignore(log_to_file ("Not a function. Load as id: "^(L.string_of_llvalue e2')^
              let _val = L.build_load e2' "tmp" llbuilder in
              ignore(log_to_file ("Loaded: "^(L.string_of_llvalue _val)^"\n"));
              ignore (L.build_store _val (lookup s) llbuilder);
              _val
            | Some f  ->
              (* The field is a function. Do not load it. *)
              ignore(log_to_file "A function. Do not load it.\n");
              ignore (L.build_store e2' (lookup s) llbuilder);
              e2'
          )
        in
        obj_field
      | _ ->
        ignore(log_to_file "The rhs to be assigned to lhs is not literal. Load it first. \n
        ignore(log_to_file ("RHS expr is "^(S.string_of_s_expr rhs)^"\n"));
        let _val = L.build_load e2' "tmp" llbuilder in
        ignore (L.build_store _val (lookup s) llbuilder);
        _val
    end
  | S.S_ObjAccess(the_obj,the_field,data_type) ->
    ignore(log_to_file ("Building object access in LHS of assign: type  "^(A.string_of_typ
    let e1' = codegen_obj_access true the_obj the_field data_type llbuilder in
    ignore(log_to_file ("Built LHS ObjAccess: "^(L.string_of_llvalue e1')^"\n"));
    let e2' = codegen_sexpr llbuilder true rhs in
```

```
      ignore(log_to_file ("Built RHS ObjAccess: "^(L.string_of_llvalue e2')^"\n"));
      let _val = L.build_store e2' e1' llbuilder in
      ignore(log_to_file ("Built store: "^(L.string_of_llvalue _val)^"\n"));
      e2'

    | S.S_ArrayAccess(e, index, typ) ->
      let vmemory = generate_array_access false e index llbuilder in
      let value = codegen_sexpr llbuilder false rhs in
      ignore (L.build_store value vmemory llbuilder);
      value
    | _ ->
      let e1' = codegen_sexpr llbuilder false lhs and e2' = codegen_sexpr llbuilder false rhs
      ignore (L.build_store e2' e1' llbuilder); e2'
  )

(* Construct code for an expression; return its value *)
and codegen_sexpr builder isReturn (sexpr:S.s_expr) =
  ignore(log_to_file ("Building expression: "^(S.string_of_s_expr sexpr)^"\n") );

  match sexpr with
    S.S_Literal i ->
      L.const_int i32_t i
  | S.S_BoolLit b -> L.const_int i1_t (if b then 1 else 0)
  | S.S_FloatLit(f)           -> L.const_float f_t f
  | S.S_StringLit s  ->L.build_global_stringptr s "tmp_string" builder
  | S.S_Noexpr -> L.const_int i32_t 0
  | S.S_Null -> L.const_null i32_t
  | S.S_Id (s,data_type) ->
    if isReturn then begin
      ignore(log_to_file ("Id "^s^" is to be returned.\n"));
      let to_ret = match data_type with
          A.Object(name) ->
          lookup s
        |_ ->
          L.build_load (lookup s) s builder
      in
      to_ret
    end
    else begin
      ignore(log_to_file ("The Id is "^s^"\n"));
      L.build_load (lookup s) s builder
    end
  | S.S_ArrayCreate(typ, size) -> generate_one_d_array typ size builder
  | S.S_ArrayAccess(arrayName, index, t) -> generate_array_access true arrayName index build
  | S.S_Binop (e1, op, e2,_) ->
    ignore(log_to_file ("Building binop "^(S.string_of_s_expr e1)^","^(A.string_of_op op)^"
    let e1' = codegen_sexpr builder false e1
    and e2' = codegen_sexpr builder false e2 in

    ignore(log_to_file ( ("\n"^L.string_of_llvalue e1')^"\n" ) );
    ignore(log_to_file ( (L.string_of_llvalue e2')^"\n" ) );

    if (Semant.get_sexpr_type(e1) = A.String ) then
      (match op with
         A.Add -> codegen_strcat e1' e2' builder
       | _ -> raise(Failure("unsupported string opreators"))
      )
    else if (Semant.get_sexpr_type(e1) = A.Float ) then
      (match op with
         A.Add     -> L.build_fadd
       | A.Sub     -> L.build_fsub
       | A.Mult    -> L.build_fmul
       | A.Div     -> L.build_fdiv
       | A.Equal   -> L.build_fcmp L.Fcmp.Oeq
       | A.Neq     -> L.build_fcmp L.Fcmp.One
       | A.Less    -> L.build_fcmp L.Fcmp.Olt
       | A.Leq     -> L.build_fcmp L.Fcmp.Ole
       | A.Greater -> L.build_fcmp L.Fcmp.Ogt
       | A.Geq     -> L.build_fcmp L.Fcmp.Oge
       | _ -> raise(Failure("unsupported float opreators"))
      ) e1' e2' "tmp" builder
    else
      (match op with
         A.Add     -> L.build_add
       | A.Sub     -> L.build_sub
       | A.Mult    -> L.build_mul
       | A.Div     -> L.build_sdiv
       | A.And     -> L.build_and
       | A.Or      -> L.build_or
       | A.Equal   -> L.build_icmp L.Icmp.Eq
       | A.Neq     -> L.build_icmp L.Icmp.Ne
       | A.Less    -> L.build_icmp L.Icmp.Slt
       | A.Leq     -> L.build_icmp L.Icmp.Sle
       | A.Greater -> L.build_icmp L.Icmp.Sgt
       | A.Geq     -> L.build_icmp L.Icmp.Sge
      ) e1' e2' "tmp" builder
  | S.S_Unop(op, e, _) ->
    ignore(log_to_file ("Building unop "^(S.string_of_s_expr e)^"\n" ) );
    let e' = codegen_sexpr builder false e in
    (match (op)  with
       A.Neg     ->
```

```
        if (Semant.get_sexpr_type(e) = A.Float ) then
          L.build_fneg
        else
          L.build_neg
      | A.Not     -> L.build_not
     ) e' "tmp" builder

   | S.S_Assign (lhs,rhs, _) ->
     codegen_assign lhs rhs builder
   | S.S_Cast(t1,e,t2) ->
     (
       match (t1,t2) with
         (Int,Int) | (Float,Float) | (Bool,Bool) | (String,String) -> codegen_sexpr builder f
       | (Float,Int) ->
         let e' = codegen_sexpr builder false e in
         L.build_fptosi e' i32_t "float_to_int" builder
       | (Int,Float) ->
         let e' = codegen_sexpr builder false e in
         L.build_sitofp e' f_t "int_to_float" builder
       | (String,Int) ->
         let atoi_func = func_lookup "atoi" in
         L.build_call atoi_func [| (codegen_sexpr builder false e) |] "atoi" builder
       | (String,Float) ->
         let atof_func = func_lookup "atof" in
         L.build_call atof_func [| (codegen_sexpr builder false e) |] "atof" builder
       | (Float,String) -> codegen_tostring e builder
       | (Int,String) -> codegen_tostring e builder
       | (Bool,String) -> codegen_tostring e builder
       | _ -> raise(Failure("cast not implemented"))
     )
   |  S.S_Call ("print",[e],_ ) ->
     let printf_format =
       let temp = Semant.get_sexpr_type(e) in
       (match temp with
          A.Int | A.Bool -> int_format_str
        | A.String -> string_format_str
        | A.Float -> float_format_str
        | A.Arraytype(t) -> (match t with
              A.Int | A.Bool -> int_format_str
            | A.String -> string_format_str
            | A.Float -> float_format_str
          )
        | _ -> raise(Failure("print only supports primitive types") )
       ) in
     let printf_func = func_lookup "printf" in
     let _val =
       match e with
         S.S_ObjAccess(_,_,_) ->
         L.build_load ((codegen_sexpr builder false e)) "var_for_print" builder
       | _ ->
         codegen_sexpr builder false e
     in
     L.build_call printf_func [| (printf_format builder) ; _val |]
       "printf" builder

   | S.S_Call ("toString",[e],_) ->
     codegen_tostring e builder
   | S.S_Call ("strcmp",e,_) ->
     let strcmp_func = func_lookup "strcmp" in
     let actuals = List.rev (List.map (codegen_sexpr builder false) (List.rev e)) in
     L.build_call strcmp_func (Array.of_list actuals) "strcmp" builder
   (* Add sizeof *)
   | S.S_Call ("sizeof",[e],data_type) ->
     codegen_sizeof e data_type builder
   (* Add malloc *)
   | S.S_Call ("malloc",[e],data_type) ->
     codegen_malloc e data_type builder
   (* Add cast *)
   | S.S_Call ("cast",[e],data_type) ->
     codegen_cast e data_type builder


   | S.S_Call (fname, act, _) ->
     ignore(log_to_file ("Build function "^fname^" with expr: \n") );
     ignore(List.map (fun expr -> log_to_file ((S.string_of_s_expr expr) ^ "\n") ) act );

     let (fdef, fdecl) = H.find func_table fname in
     let actuals = List.rev (List.map (codegen_sexpr builder true) (List.rev act)) in
     let result = (match fdecl.S.s_typ with A.Void -> ""
                                          | _ -> fname ^ "_result") in
     L.build_call fdef (Array.of_list actuals) result builder
   |   S_ObjCreate(id, expr_list, data_type) -> codegen_obj_create id expr_list data_type bui
   |   S_ObjAccess(e1, e2, d) -> codegen_obj_access (not isReturn) e1 e2 d builder


 (* sizeof *)
 and codegen_sizeof sexpr data_type llbuilder =
   ignore(log_to_file ("Building sizeof with data_type:"^(A.string_of_typ data_type)^"\n") );
   ignore(log_to_file ("Expr: "^(S.string_of_s_expr sexpr )^"\n"   ) );

   let type_of_el = get_underlying_type_of_sexpr sexpr in
   let size_of_el = L.size_of type_of_el in
   L.build_bitcast size_of_el i32_t "tmp" llbuilder


 and codegen_tostring e llbuilder =
   let ll_expr = codegen_sexpr llbuilder false e in
   let t = Semant.get_sexpr_type(e) in
   let (size, sprintf_format) =
     (match (t) with
        A.Int | A.Bool ->
        (L.const_int i32_t 20, int_sprintf_str(llbuilder))
      | A.Float ->
        (L.const_int i32_t 20, float_sprintf_str(llbuilder))
      | A.String ->
        let strlen_func = func_lookup "strlen" in
        let len =  L.build_call strlen_func [| (ll_expr); |] "strlen" llbuilder in
        let len = L.build_add len (L.const_int i32_t 1) "tmp" llbuilder in
        (L.const_int i32_t 20, string_sprintf_str(llbuilder))
      | _ ->  raise (Failure ("toString() function only supports primitive types") )
     ) in
   let t = i8_t in
   let buf = L.build_array_malloc t size "to_string_buf" llbuilder in
   let buf = L.build_pointercast buf (L.pointer_type t) "to_string_buf" llbuilder in
   let sprintf_func = func_lookup "sprintf" in
   let _ = L.build_call sprintf_func [| buf; sprintf_format ; ll_expr; |]
       "sprintf" llbuilder in
   buf

 and codegen_strcat e1' e2' llbuilder =
   let ll_expr1 = e1' in
   let ll_expr2 = e2' in
   let strlen_func = func_lookup "strlen" in
   let strcpy_func = func_lookup "strcpy" in
   let strcat_func = func_lookup "strcat" in
   let len1 =  L.build_call strlen_func [| (ll_expr1); |] "strlen" llbuilder in
   let len2 =  L.build_call strlen_func [| (ll_expr2); |] "strlen" llbuilder in
   let len_new = L.build_add len1 len2 "tmp" llbuilder in
   let size = L.build_add len_new (L.const_int i32_t 1) "tmp" llbuilder in
   let t = i8_t in
   let buf = L.build_array_malloc t size "to_string_buf" llbuilder in
   let buf = L.build_pointercast buf (L.pointer_type t) "to_string_buf" llbuilder in
   let buf = L.build_call strcpy_func [| buf; ll_expr1; |] "strcpy" llbuilder in
   let buf = L.build_call strcat_func [| buf; ll_expr2; |] "strcat" llbuilder in
   buf

 and codegen_malloc sexpr data_type llbuilder =
   let f = func_lookup "malloc" in
   let params = List.map (codegen_sexpr llbuilder false ) [sexpr] in
   match data_type with
   A.Void -> L.build_call f (Array.of_list params) "" llbuilder
   |  _ ->
   let c = !counter in
   let ptr = L.build_call f (Array.of_list params) (string_of_int c) llbuilder in
   counter := c + 1;
   H.add new_obj_table c ptr;
   ptr

 and codegen_cast sexpr data_type llbuilder =
   ignore(log_to_file ("Building cast with data_type:"^(A.string_of_typ data_type)^"\n") );
   ignore(log_to_file ("Expr: "^(S.string_of_s_expr sexpr )^"\n"   ) );

   let cast_malloc_to_objtype lhs currType newType llbuilder = match newType with
       A.Object(name)->
       let obj_type = ltype_of_typ (A.Object(name)) in
       ignore(log_to_file ("In cast obj type is "^(L.string_of_lltype obj_type)^"\n")) ;
       L.build_pointercast lhs obj_type "tmp" llbuilder
     |   _ as t -> raise (Failure("I don't know why Im here but this is wrong"))
   in
   let t = Semant.get_sexpr_type sexpr in
   let lhs = match sexpr with
     |   S.S_Id(id, data_type) ->
       lookup id
     | _ -> codegen_sexpr llbuilder false sexpr
   in
   cast_malloc_to_objtype lhs t data_type llbuilder


 (* Need to be moved*)
 and int_format_str builder = L.build_global_stringptr "%d\n" "fmt" builder
 and string_format_str builder = L.build_global_stringptr "%s\n" "fmt" builder
 and float_format_str builder = L.build_global_stringptr "%f\n" "fmt" builder
 and int_sprintf_str builder = L.build_global_stringptr "%d" "tostr_fmt" builder
 and string_sprintf_str builder = L.build_global_stringptr "%s" "tostr_fmt" builder
 and float_sprintf_str builder = L.build_global_stringptr "%f" "tostr_fmt" builder


 and codegen_classes (classes:S.s_class_decl list) =
   ignore(log_to_file "Codegen classes\n");
   ignore(List.map (fun c -> log_to_file ("To build class "^c.S.s_cname^"\n") ) classes);

   let codegen_one_class (the_class:S.s_class_decl) =
     ignore(log_to_file ("Generating class "^the_class.s_cname^"\n") );
     (* Generate struct decl *)
     let _ = codegen_struct_stub the_class in

     (* Generate class local vars *)
     let _ = codegen_struct the_class in

     (* Generate functions *)
     ignore(log_to_file "Finished class definition. Now start with functions\n");
     ignore(List.map (fun f -> log_to_file ("To build constructor "^f.S.s_fname^"\n") ) the_c
     ignore(List.map (fun f -> log_to_file ("To build method "^f.S.s_fname^"\n") ) the_class.

     let _ = codegen_function_decls the_class.s_constructors in
     let _ = codegen_function_decls the_class.s_methods in
     let _ = codegen_fbody the_class.s_constructors in

     let _ = codegen_fbody the_class.s_methods in
     ignore(log_to_file ("Generated class "^the_class.s_cname^"\n") );
   in
   List.map codegen_one_class classes;
   ignore(log_to_file "Finished generating classes\n")

 and find_struct name =
   try Hashtbl.find object_types name
   with | Not_found -> raise(Failure ("Cannot find object type "^name) )

 and codegen_fbody functions =

   (* Fill in the body of the given function *)
   let build_function_body fdecl =
     let _ = log_to_file ("Building function " ^ fdecl.S.s_fname ^ "\n" ) in

     (* First clear local and param tables *)
     let _ = H.clear locals_table; H.clear params_table in

     (* Find target function *)
     let (the_function, _) = H.find func_table fdecl.S.s_fname in
     let builder = L.builder_at_end context (L.entry_block the_function) in

     (* Construct the function's "locals": formal arguments and locally
        declared variables.  Allocate each on the stack, initialize their
        value, if appropriate, and remember their values in the "locals" map *)
     let codegen_locals local_vars =
       let add_formal table (the_type,name) value =
         L.set_value_name name value;
         let local = match the_type with
             A.Object(name) ->
             value
           | _ ->
             let _val = L.build_alloca (ltype_of_typ the_type) name builder in
             ignore (L.build_store value _val builder); (* local is the pointer we store at *
             _val
         in
         H.add table name local
       in
       let add_local table (the_type,name) =
         let t =
           match the_type with
             A.Object(cname) ->
             find_struct cname
           | _ ->
             ltype_of_typ the_type
         in

         let local_var = L.build_alloca t name builder
         in
         H.add table name local_var
       in
       let _ = List.map2 (add_formal params_table) fdecl.S.s_formals (Array.to_list (L.params
       in
       ignore(List.map (add_local locals_table) local_vars);

       (* Log to file all the locals and formals *)
       ignore(log_to_file "Logging local table\n");
       print_hashtable locals_table
     in
     codegen_locals fdecl.S.s_locals;

     (* Invoke "f builder" if the current block doesn't already
        have a terminal (e.g., a branch). *)
     let add_terminal builder f =
       match L.block_terminator (L.insertion_block builder) with
         Some _ -> ()
       | None -> ignore (f builder)
     in
     (* Build the code for the given statement; return the builder for
        the statement's successor *)
     let rec codegen_stmt builder stmt =
       match stmt with
         S.S_Block sl ->
         List.fold_left codegen_stmt builder sl
       | S.S_Expr (e,_) ->
         ignore (codegen_sexpr builder false e); builder
       | S.S_Return (e,_) ->
         ignore(log_to_file ("Building return expr "^(S.string_of_s_expr e)^"\n" ) );
         ignore (match fdecl.S.s_typ with
               A.Void -> L.build_ret_void builder
             | _ -> L.build_ret (codegen_sexpr builder true e) builder); builder
       | S.S_If (predicate, then_stmt, else_stmt) ->
         let bool_val = codegen_sexpr builder false predicate in
         let merge_bb = L.append_block context "merge" the_function in

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

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

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

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

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

         let pred_builder = L.builder_at_end context pred_bb in
         let bool_val = codegen_sexpr pred_builder false predicate in

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

       | S.S_For (e1, e2, e3, body) -> codegen_stmt builder
                                         ( S.S_Block [S.S_Expr(e1,Semant.get_sexpr_type(e1))
                                                      S.S_While (e2, S.S_Block [body ; S.S_E
     in
     (* let ff = func_lookup "free" in
     let counters = H.to_seq_values new_obj_table in
   (* each element in the table -> llvalue, which is a pointer*)
     let _ = Format.print_string fdecl.S.s_fname in
     let _ = if fdecl.S.s_fname == "main" then
     (Seq.iter (fun c -> L.build_call ff (Array.of_list [c]) "" builder; () ) counters)
     else () in *)
```

```
    (* Build the code for each statement in the function *)
    let builder_stmt_built = codegen_stmt builder (S.S_Block fdecl.S.s_body)
    in
    (* Add a return if the last block falls off the end *)
    add_terminal builder_stmt_built (match fdecl.S.s_typ with
          A.Void -> L.build_ret_void
        | t -> L.build_ret (L.const_int (ltype_of_typ t) 0))
  in
  List.iter build_function_body functions

let codegen_sprogram (program:S.s_program) =
  (* First wipe out log file *)


  (* Build builtin function declarations *)
  let _ = codegen_builtins program.S.builtins in

  let _ = codegen_globals program.S.global_vars in

  let _ = codegen_classes program.S.classes in

  (* Generate functions here *)
  let _ = codegen_function_decls program.S.functions in
  let _ = codegen_function_decls (program.S.main::[]) in
  let _ = codegen_fbody program.S.functions in
  let _ = codegen_fbody (program.S.main::[]) in
  the_module
```

8.2 Test files

testall.sh

```
#!/bin/sh

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

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

# Path to the LLVM compiler
LLC="/usr/local/opt/llvm/bin/llc"
```

```
# Path to the C compiler
CC="cc"

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

# Set time limit for all operations
ulimit -t 30

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

keep=0

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

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

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

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

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

Check() {
    error=0
    basename=‘echo $1 | sed ’s/.*\\///
                             s/.mc//’‘
    reffile=‘echo $1 | sed ’s/.mc$//’‘
    basedir="‘echo $1 | sed ’s/\/[ˆ\/]*$//’‘/."

    echo -n "$basename..."

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

    generatedfiles=""

    generatedfiles="$generatedfiles ${basename}.ll ${basename}.s ${basename}.exe ${basename
    Run "$MICROC" "<" "$1" ">" "${basename}.ll" &&
    Run "$LLC" "-relocation-model=pic" "${basename}.ll" ">" "${basename}.s" &&
    Run "$CC" "-o" "${basename}.exe" "${basename}.s" "printbig.o" &&
    Run "./${basename}.exe" > "${basename}.out" &&
    Compare ${basename}.out ${reffile}.out ${basename}.diff

    # Report the status and clean up the generated files

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

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

    echo -n "$basename..."

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

    generatedfiles=""

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

    # Report the status and clean up the generated files

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

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

```
h) # Help
    Usage
    ;;
    esac
done

shift `expr $OPTIND - 1`

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

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

if [ ! -f printbig.o ]
then
    echo "Could not find printbig.o"
    echo "Try \"make printbig.o\""
    exit 1
fi

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

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

```
exit $globalerror
```

test-add1.mc

```
int add(int x, int y)
{
  return x + y;
}

int main()
{
  print( add(17, 25) );
  return 0;
}
```

test-add1.out

```
42
```

test-arith-mulAndDiv.mc

```
int main()
{
  print(2*3);
  print(5/1);
  print(2/3);
  return 0;
}
```

test-arith-mulAndDiv.out

```
6
5
0
```

test-arith1.mc

```
int main()
{
  print(39 + 3);
  return 0;
}
```

test-arith1.out

```
42
```

test-arith2.mc

```
int main()
{
  print(1 + 2 * 3 + 4);
  return 0;
}
```

test-arith2.out

```
11
```

test-arith3.mc

```
int foo(int a)
{
  return a;
}

int main()
{
  int a;
  a = 42;
  a = a + 5;
  print(a);
  return 0;
}
```

test-arith3.out

```
47
```

test-array.mc

```
int main() {
  int[] array;
  array = new int[5];
  array[0] = 13;
  array[1] = 19;
  array[2] = 22;
  array[3] = 33;
  array[4] = 67;
  print(array[0]);
  print(array[1]);
  print(array[2]);
  print(array[3]);
  print(array[4]);
  return 0;
}
```

test-array.out

```
13
19
22
33
67
```

test-array1.mc

```
int main() {
  int[] array;
  array = new int[5];
  array[0] = 0;
  array[1] = 1;
  print(array[0]);
  print(array[4]);
  return 0;
}
```

test-array1.out

```
0
0
```

test-array2.mc

```
int main() {
  int[] array;
  array = new int[5];
  array[0] = 13;
  array[1] = 19;
  array[2] = 22;
  array[3] = 33;
  array[4] = 67;
  print(array[0]);
  print(array[1]);
  print(array[2]);
  print(array[3]);
  print(array[4]);
  return 0;
}
```

test-array2.out

```
13
19
22
33
67
```

test-array3.mc

```
int main()
{
  float[] array;
  array = new float[10];
  array[5] = 3.14;
  print(array[0]);
  print(array[5]);
  return 0;
}
```

test-array3.out

```
0.000000
3.140000
```

test-array4.mc

```
int main()
{
  string[] array;
  array = new string[5];
  array[3] = "plt is a nice course";
  print(array[0]);
  print(array[3]);
  return 0;
}
```

test-array4.out

```
(null)
plt is a nice course
```

test-assign-float1.mc

```
int main()
{
  /* assign string */
  float a;
  a = 3.5;
  print(a);

  return 0;
}
```

test-assign-float1.out

```
3.500000
```

test-assign-float2.mc

```
int main()
{
  /* assign string */
  float a;
  float b;
  a = -3.5;
  b = a;
  print(b);

  return 0;
}
```

test-assign-float2.out

```
-3.500000
```

test-class-constructor.mc

```
class Circle{
  int x;
  int y;
  constructor(){}
  constructor(int a, int b){
    x=a;
    y=b;
  }
}
int main(){
  class Circle c;
  int the_x;
  c = new Circle(3,5);
  the_x=c.x;
  print(the_x);
  return 0;
}
```

test-class-constructor.out

```
3
```

test-class-func.mc

```
class Test{
    constructor(){}
    int getInt(){
        return 3;
    }
}
int main(){
    class Test t;
    int i;
    t = new Test();
    i = t.getInt();
    print(i);

    return 0;
}
```

test-class-func.out

```
3
```

test-class-funcWithThis.mc

```
class Line {
    int x;
constructor(int a){
        x = a;
    }
    int getX() {
      int a;
      a = this.x;
      return a;
    }
}


int main() {
    class Line e;
    int i;
    e = new Line(5);
i = e.getX();
    print(i);
    return 0;
}
```

test-class-funcWithThis.out

```
5
```

test-class-inClassFor.mc

```
class Test{
    int value;
    constructor(int v){
        value = v;
    }
    int getValue(){
        int v;
        int i;

        v = this.value;
        for(i=0;i<3;i = i+1){
            v = v*2;
        }

        return v;
    }
}
int main(){
    class Test t1;
    class Test t2;
    int v1;
    int v2;

    t1 = new Test(3);
    t2 = new Test(-1);
    v1 = t1.getValue();
    v2 = t2.getValue();

    print(v1);
    print(v2);

    return 0;
}
```

test-class-inClassFor.out

```
24
-8
```

test-class-inClassIf.mc

```
class Test{
    int value;
    constructor(int v){
        value = v;
    }
    int getValue(){
        int v;

        v = this.value;
        if(v < 3)
            v = v*2;
        return v;
    }
}
int main(){
    class Test t1;
    class Test t2;
    int v1;
    int v2;

    t1 = new Test(1);
    t2 = new Test(3);
    v1 = t1.getValue();
    v2 = t2.getValue();

    print(v1);
    print(v2);

    return 0;
}
```

test-class-inClassIf.out

```
2
3
```

test-class-inClassWhile.mc

```
class Test{
    int value;
    constructor(int v){
        value = v;
    }
    int getValue(){
        int v;

        v = this.value;
        while(v>10){
            v = v/2;
        }

        return v;
    }
}
int main(){
    class Test t1;
    class Test t2;
    int v1;
    int v2;

    t1 = new Test(100);
    t2 = new Test(9);
    v1 = t1.getValue();
    v2 = t2.getValue();

    print(v1);
    print(v2);
    return 0;
}
```

test-class-inClassWhile.out

```
6
9
```

test-class-nestedAccess.mc

```
class A{
  int x;
  constructor(int the_x){
    x = the_x;
  }
}
class B{
  class A a;
  constructor(class A the_a){
    a = the_a;
  }
}

int main(){
  int c;
  class A a;
  class B b;
  a = new A(3);
  b = new B(a);
  c = b.a.x;
  print(c);
  return 0;
}
```

test-class-nestedAccess.out

```
3
```

test-class-nestedObj.mc

```
class Dot{
  int x;
  int y;
  constructor(int a, int b){
    x = a;
    y = b;
  }
}
class Circle{
  class Dot center;
  constructor(class Dot d){
    center = d;
  }
}

int main(){
  class Dot d;
  class Circle c;
  d = new Dot(3,5);
  c = new Circle(d);
  print(c.center.x);
  print(c.center.y);
  return 0;
}
```

test-class-nestedObj.out

```
3
5
```

test-class-noParamConstructor.mc

```
class Circle{
  int x;
  int y;
  constructor(){}
}
int main(){
  class Circle c;
  c = new Circle();
  return 0;
}
```

test-class-noParamConstructor.out

---

test-class-objArrayField.mc

```
class Test{
    int[] array;
    constructor(){
        array = new int[10];
    }
}

int main(){
    class Test t;
    int[] a;
    t = new Test();
    a = t.array;
    a[3] = 33;
    print(a[0]);
    print(a[3]);
    print(a[9]);
    return 0;
}
```

test-class-objArrayField.out

```
0
33
0
```

test-class-objAsParam.mc

```
class Line {
constructor(){}
}


int foo(class Line e) {
return 0;
}

int main() {
    class Line e;
    e = new Line();
print(foo(e));
    return 0;
}
```

test-class-objAsParam.out

```
0
```

test-class-printObjAccess.mc

```
class Circle{
  int x;
  int y;
  constructor(int a, int b){
    x = a;
    y = b;
  }
}
int main(){
  class Circle c;
  c = new Circle(3,5);
  print(c.x);
  return 0;
}
```

test-class-printObjAccess.out

```
3
```

test-class1.mc

```
class Example {
  float ret5(){
    return 5.0;
  }
constructor(){}
}


float foo(class Example example) {
return example.ret5();
}

int main() {
    class Example e;
    e = new Example();
  print(foo(e));
    return 0;
}
```

test-class1.out

```
5.000000
```

test-class2.mc

```
class Example{
  int x;
  int y;
  constructor(int a, int b){
    x=a;
    y=b;
  }
}
int main(){
  class Example e;
  int x;
  e = new Example(3,5);
  x=e.x;
  print(x);
  return 0;
}
```

test-class2.out

```
3
```

test-expr1.mc

```
int a;
bool b;

void foo(int c, bool d)
{
  int dd;
  bool e;
  a + c;
  c - a;
  a * 3;
  c / 2;
}

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

test-expr1.out

---

test-fib.mc

```
int fib(int x)
{
  if (x < 2) return 1;
  return fib(x-1) + fib(x-2);
}

int main()
{
  print(fib(0));
  print(fib(1));
  print(fib(2));
  print(fib(3));
  print(fib(4));
  print(fib(5));
  return 0;
}
```

test-fib.out

```
1
1
2
3
5
8
```

test-float-arith-cmp1.mc

```
int main()
{
  float a;
  float b;
  a = 1.0;
  b = 1.0;
  if (a == b)
print("correct");
  else
print("not correct");
```

```
  return 0;
}
```

test-float-arith-cmp1.out

```
correct
```

test-float-arith-cmp2.mc

```
int main()
{
  float a;
  float b;
  a = 1.0;
  b = 1.2;
  if (a < b)
print("correct");
  else
print("not correct");

  return 0;
}
```

test-float-arith-cmp2.out

```
correct
```

test-float-arith-cmp3.mc

```
int main()
{
  float a;
  float b;
  a = 1.0;
  b = 1.0;
  if (a <= b)
print("correct");
  else
print("not correct");

  return 0;
}
```

test-float-arith-cmp3.out

```
correct
```

test-float-arith-cmp4.mc

```
int main()
{
  float a;
  float b;
  a = 2.0;
  b = 1.0;
  if (a > b)
print("correct");
  else
print("not correct");

  return 0;
}
```

test-float-arith-cmp4.out

```
correct
```

test-float-arith-cmp5.mc

```
int main()
{
  float a;
  float b;
  a = 1.0;
  b = 1.0;
  if (a >= b)
print("correct");
  else
print("not correct");

  return 0;
}
```

test-float-arith-cmp5.out

```
correct
```

test-float-arith-cmp6.mc

```
int main()
{
  float a;
  float b;
  a = 2.0;
  b = 1.0;
  if (a != b)
print("correct");
  else
print("not correct");

  return 0;
}
```

test-float-arith-cmp6.out

```
correct
```

test-float-arith1.mc

```
int main()
{
  /* assign string */
  float a;
  float b;
  a = -3.5 + 1.0;
  b = a + 1.0;
  print(b);

  return 0;
}
```

test-float-arith1.out

```
-1.500000
```

test-float-arith2.mc

```
int main()
{
  /* assign string */
  float a;
  float b;
  a = -3.5 + 1.0;
  b = a + 1.0;
  print(1.0+b*a);

  return 0;
}
```

test-float-arith2.out

```
4.750000
```

test-float-arith3.mc

```
int main()
{
  /* assign string */
  float a;
  float b;
  a = -3.5 + 1.0;
  b = a + 1.0;
  print(b/a);

  return 0;
}
```

test-float-arith3.out

```
0.600000
```

test-float-arith4.mc

```
int main()
{
  /* assign string */
  float a;
  float b;
  a = -3.5 + 1.0;
  b = a + 1.0;
  print(b-a);

  return 0;
}
```

test-float-arith4.out

```
1.000000
```

test-float-arith5.mc

```
int main()
{
  /* assign string */
  float a;
  float b;
  a = -3.5 + 1.0;
  b = 0.0;
  print(a/b);

  return 0;
}
```

test-float-arith5.out

```
-inf
```

test-float1.mc

```
int main()
{
  float a;
  a = 3.14159267;
  print(a);
  return 0;
}
```

test-float1.out

```
3.141593
```

test-float2.mc

```
int main()
{
  float a;
  float b;
  float c;
  a = 3.14159267;
  b = -2.71828;
  c = a + b;
  print(c);
  return 0;
}
```

test-float2.out

```
0.423313
```

test-float3.mc

```
void testfloat(float a, float b)
{
  print(a + b);
  print(a - b);
  print(a * b);
  print(a / b);
  print(a == b);
  print(a == a);
  print(a != b);
  print(a != a);
  print(a > b);
  print(a >= b);
  print(a < b);
  print(a <= b);
}

int main()
{
  float c;
  float d;

  c = 42.0;
  d = 3.14159;

  testfloat(c, d);

  testfloat(d, d);

  return 0;
}
```

test-float3.out

```
45.141590
38.858410
131.946780
13.369027
0
1
1
0
1
1
0
0
6.283180
0.000000
9.869588
1.000000
1
1
0
0
0
1
0
1
```

test-for1.mc

```
int main()
{
  int i;
  for (i = 0 ; i < 5 ; i = i + 1) {
    print(i);
  }
  print(42);
  return 0;
}
```

test-for1.out

```
0
1
2
3
4
42
```

test-for2.mc

```
int main()
{
  int i;
  i = 0;
  for ( ; i < 5; ) {
    print(i);
    i = i + 1;
  }
  print(42);
  return 0;
}
```

test-for2.out

```
0
1
2
3
4
42
```

test-func1.mc

```
int add(int a, int b)
{
  return a + b;
}

int main()
{
  int a;
  a = add(39, 3);
  print(a);
  return 0;
}
```

test-func1.out

```
42
```

test-func2.mc

```
/* Bug noticed by Pin-Chin Huang */

int fun(int x, int y)
{
  return 0;
}

int main()
{
  int i;
  i = 1;

  fun(i = 2, i = i+1);

  print(i);
  return 0;
}
```

test-func2.out

```
2
```

test-func3.mc

```
void printem(int a, int b, int c, int d)
{
  print(a);
  print(b);
  print(c);
  print(d);
}

int main()
{
  printem(42,17,192,8);
  return 0;
}
```

test-func3.out

```
42
17
192
8
```

test-func4.mc

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

int main()
{
  int d;
  d = add(52, 10);
  print(d);
  return 0;
}
```

test-func4.out

```
62
```

test-func5.mc

```
int foo(int a)
{
  return a;
}

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

test-func5.out

test-func6.mc

```
void foo() {}

int bar(int a, bool b, int c) { return a + c; }

int main()
```

```
{
  print(bar(17, false, 25));
  return 0;
}
```

test-func6.out

```
42
```

test-func7.mc

```
int a;

void foo(int c)
{
  a = c + 42;
}

int main()
{
  foo(73);
  print(a);
  return 0;
}
```

test-func7.out

```
115
```

test-func8.mc

```
void foo(int a)
{
  print(a + 3);
}

int main()
{
  foo(40);
  return 0;
}
```

test-func8.out

```
43
```

test-func9.mc

```
void foo(int a)
{
  print(a + 3);
  return;
}

int main()
{
  foo(40);
  return 0;
}
```

test-func9.out

```
43
```

test-gcd.mc

```
int gcd(int a, int b) {
  while (a != b) {
    if (a > b) a = a - b;
    else b = b - a;
  }
  return a;
}

int main()
{
  print(gcd(2,14));
  print(gcd(3,15));
  print(gcd(99,121));
  return 0;
}
```

test-gcd.out

```
2
3
11
```

test-gcd2.mc

```
int gcd(int a, int b) {
  while (a != b)
    if (a > b) a = a - b;
    else b = b - a;
  return a;
}

int main()
{
  print(gcd(14,21));
  print(gcd(8,36));
  print(gcd(99,121));
  return 0;
}
```

test-gcd2.out

```
7
4
11
```

test-global1.mc

```
int a;
int b;

void printa()
{
  print(a);
}

void printbb()
{
  print(b);
}

void incab()
{
  a = a + 1;
  b = b + 1;
}

int main()
{
  a = 42;
  b = 21;
  printa();
  printbb();
  incab();
  printa();
  printbb();
  return 0;
}
```

test-global1.out

```
42
21
43
22
```

test-global2.mc

```
bool i;

int main()
{
  int i; /* Should hide the global i */

  i = 42;
  print(i + i);
  return 0;
}
```

test-global2.out

```
84
```

test-global3.mc

```
int i;
bool b;
int j;

int main()
{
  i = 42;
  j = 10;
  print(i + j);
  return 0;
}
```

test-global3.out

```
52
```

test-hello.mc

```
int main()
{
  print(42);
  print(71);
  print(1);
  return 0;
}
```

test-hello.out

```
42
71
1
```

test-if1.mc

```
int main()
{
  if (true) print(42);
  print(17);
  return 0;
}
```

test-if1.out

```
42
17
```

test-if2.mc

```
int main()
{
  if (true) print(42); else print(8);
  print(17);
  return 0;
}
```

test-if2.out

```
42
17
```

test-if3.mc

```
int main()
{
  if (false) print(42);
  print(17);
  return 0;
}
```

test-if3.out

```
17
```

test-if4.mc

```
int main()
{
  if (false) print(42); else print(8);
  print(17);
  return 0;
}
```

test-if4.out

```
8
17
```

test-if5.mc

```
int cond(bool b)
{
  int x;
  if (b)
    x = 42;
  else
    x = 17;
  return x;
}

int main()
{
 print(cond(true));
 print(cond(false));
 return 0;
}
```

test-if5.out

```
42
17
```

test-if6.mc

```
int cond(bool b)
{
  int x;
  x = 10;
  if (b)
    if (x == 10)
      x = 42;
  else
    x = 17;
  return x;
}

int main()
{
 print(cond(true));
 print(cond(false));
 return 0;
}
```

test-if6.out

```
42
10
```

test-local1.mc

```
void foo(bool i)
{
  int i; /* Should hide the formal i */

  i = 42;
  print(i + i);
}
```

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

test-local1.out

```
84
```

test-local2.mc

```
int foo(int a, bool b)
{
  int c;
  bool d;

  c = a;

  return c + 10;
}

int main() {
 print(foo(37, false));
 return 0;
}
```

test-local2.out

```
47
```

test-ops1.mc

```
int main()
{
  print(1 + 2);
  print(1 - 2);
  print(1 * 2);
  print(100 / 2);
  print(99);
  print(1 == 2);
  print(1 == 1);
  print(99);
  print(1 != 2);
  print(1 != 1);
  print(99);
  print(1 < 2);
  print(2 < 1);
  print(99);
  print(1 <= 2);
  print(1 <= 1);
  print(2 <= 1);
  print(99);
  print(1 > 2);
  print(2 > 1);
  print(99);
  print(1 >= 2);
  print(1 >= 1);
  print(2 >= 1);
  return 0;
}
```

test-ops1.out

```
3
-1
2
50
99
0
1
99
1
0
99
1
0
99
1
1
0
99
0
1
99
0
1
1
```

test-recur.mc

```
int fib(int x)
{
  if (x < 2) return 1;
  return fib(x-1) + fib(x-2);
}

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

test-recur.out

```
1
1
2
3
5
8
13
```

test-var1.mc

```
int main()
{
  int a;
  a = 42;
  print(a);
  return 0;
}
```

test-var1.out

```
42
```

test-var2.mc

```
int a;

void foo(int c)
{
  a = c + 42;
}

int main()
{
  foo(73);
  print(a);
  return 0;
}
```

test-var2.out

```
115
```

test-while1.mc

```
int main()
{
  int i;
  i = 5;
  while (i > 0) {
    print(i);
    i = i - 1;
  }
  print(42);
  return 0;
}
```

test-while1.out

```
5
4
3
2
1
42
```

test-while2.mc

```
int foo(int a)
{
  int j;
  j = 0;
  while (a > 0) {
    j = j + 2;
    a = a - 1;
  }
  return j;
}

int main()
{
  print(foo(7));
  return 0;
}
```

test-while2.out

```
14
```