Introduction
Features
Comparison with C
Language Tutorial
Lexical Conventions
- Identifiers
- Keywords
- Literals
- Separators
- Operators
- Comments

Data Types
- Basic Data Types
- Collection Data Types
- Custom Data Types

Expressions
Statements
- Expression Statement
- Compound Statement
- Control Flow Statement
- Loop Statement

Function
- Function Definitions
- Calling Functions

Anonymous Functions
Scope Rules
Inheritance
Interfaces
Memory Allocation

Project Planning
    Planning Process
    Specification
    Development and Testing
    Roles and Responsibilities
    Test Plan
    - Unit Tests
    - Integration Tests
    - Automation
    - Source to Target

Introduction

The Cimple programming language is a statically-typed, low-level compiled language that is syntactically similar to C. Its purpose is to offer the fast execution speeds of C while simplifying the process of writing and maintaining code by offering additional language constructs that free the programmer to organize logic in a variety of ways. It is based off of C with features that enable the programmer to write expressive server side programs with the same ease as he or she would with a language like python or ruby. These features include inheritance, interface types, anonymous functions and closures.

We have chosen C as the basis of our language due to its status as a "low-level" language. The ideas expressible in the language have close relationships with the resulting CPU instructions the compiler generates. This proximity of the language to the computer hardware in this sense promotes fast execution speed. Therefore, our language would benefit programs that require fast execution speed, such as any software that processes or renders signals in real time, such as digital audio production workstations and plugins, video editing and rendering, real-time (or otherwise) 3D graphics, video games, etc.

Additionally, we are introducing some higher level language features, including object-oriented principles such as structs with inheritance and methods, anonymous functions with closures, and interfaces. This extends the ways a programmer may organize code, and increase the speed with which more abstract concepts can be represented in code.

Types of programs that can benefit from this include those that rely on asynchronous communication. An example would be client software interacting with a server, and vice verse.

Comparison with C

Inheritance

Cimple supports inheritance, which is a new feature compared to C. User-defined data-types i.e. structs (analogous to classes in Java) are allowed to inherit the members of other structs, thus providing the benefit of code-reuse. This follows a typical parent-child hierarchy observed in most object-oriented languages. This speeds up program development and ensures validity to the child struct instances such that what is valid and consistent about a struct will also work for other structs that inherit from it. More than one struct can inherit from a struct but not the other way round i.e. inheritance in Cimple works the way it does in Java but mupltiple inheritance not in C++.

Interfaces

An interface is an abstract type which defines a group of related methods with empty bodies. It is a structure which enforces certain properties on an object, which implements one. They are useful for situations when a rigid type hierarchy is not desired. Cimple allows more flexibility in inheritance than what a parent-child hierarchy can allow, by virtue of interfaces.
Interfaces provide structs with the ability to be formal about the behavior they promise to exhibit. They form a contract between the struct and the outside world, which is enforced at build time by the compiler. If a struct claims to implement an interface, all methods defined by that interface must appear in its definition for the struct to successfully compile. Interfaces are not permitted to contain method implementations. Still using interfaces provides implementation assistance to programmers.

Anonymous functions

These are a new feature in Cimple, not found in C. These allow for a more intuitive scheme of declaration of function arguments to other functions than what would be done in C, using a function pointer instead. They provide compactness to the program and enhance the readability.

The benefit of compactness stems from the fact that they are used unnamed, because such a function is only ever called in one place, and so no need to add a name to the scope they are in. Anonymous functions are declared inline and inline functions have advantages in that they can access variables in the parent scopes. The code becomes more readable when handlers are defined right inside the code that's calling them. Reading the code can be done almost sequentially, rather than having to go and find the function with that name.

Memory Management Convenience

Cimple follows a memory management scheme much similar to C in terms of the decision-making about the place in memory that variables land up in, depending on their type. But it gives a much more convenient functionality to the programmer in practicing memory allocation and de-allocation than C does.
As in C, variables can be stored on the stack (“automatic variables”) or on the heap. Automatic variables, as the name implies, are automatically de-allocated upon exiting scope. However, storage for heap-allocated variables is at the will of the programmer, and is allotted or freed with the use of keywords `make` and `clean`. Syntactic details are discussed in a later section.

Tutorial

This section is intended to help get the reader started on using the Cimple language, by writing simple programs in it. But before getting to that, it is required that the project be downloaded and its dependencies be installed.
The two primary requirements for the project to work are:

Ocaml - version 4.02.3 - better installed through OPAM
GCC

Once these are installed and the project is downloaded, the following steps are to be followed to run your first program in Cimple.

1. Enter the project folder, and run the command **`make .`** This is needed so that an executable named 'cimple' is generated which is the entry-point to the compiler.

2. Write a Cimple program:

Clearly, at this time, there isn't enough detail discussed for a beginner to be able to whip out a Cimple program. But to get started, copy the following code verbatim into a new file:

```
int main(){

     int quarter = 25;
     int dime = 10;
     return quarter+dime;

}
```

Save the file with the name first.cpl'.

3. Now run the following command :

```
./runProgram.sh first.cpl
```

4. If all done correctly, the following should be the output:- `35`

An important thing to remember is that every program in Cimple must have the `main()` function, which is the entry-point of control into a Cimple program. Cimple programs without a `main()` function are not executable.

Lexical Conventions

Identifiers

Identifiers are case-sensitive sequences of characters that represent the names of various entities, such as numeric values, custom data types and

functions. Identifiers contain a sequence of letters, digits and underscore ‘_’, but should always start with a letter. However, they cannot be the same as a keyword in Cimple, to make it through compilation.

Being case-sensitive means that two identifiers composed of lowercase letters and uppercase letters even if they have the same spellings are different. Identifiers in Cimple are of two types:

(a) Value and Function Identifiers - These are the names given to things such as integer, floating-point or string values and the functions defined in a program. They must begin with a lower-case letter, and can be followed by any number of letters of either case, digits from 0 to 9 as well as the underscore ('_').

(b) Struct Identifiers - These are the names given to structs in a Cimple program. They must begin with an upper-case letter, and can be followed by any number of letters of either case, digits from 0 to 9 as well as the underscore ('_').

Also, there is something called method receivers that are a very handy concept in the interfaces domain in Cimple. What they are is discussed later in this document. For the purposes of naming, receivers share the same rules as structs i.e. their names should begin with a capital letter and can be followed by any other letters, digits or an underscore.

Keywords

Keywords are words that are reserved for use naming of entities that hold a special meaning for the Cimple compiler. These words cannot be used for any other purpose. The following is a listing of all the keywords recognized by Cimple :

```
auto        long        if          interface
register    float       else        extends
static      double      while       implements
extern      struct      for         make
void        unsigned    break       clean
char        signed      return      super
short       const       func
```

Literals

Literals are the source code representation of a value of some primitive types. Following are the types of literals in Cimple:

Integer Literal - This is a sequence of one or more digits in decimal. Negative integers are represented by a negation operator prefixed to a sequence such as the aforementioned one. Examples: 235, 89

Float Literal - This is a sequence consisting of an integer part, a decimal point and a fraction part. For representing negative numbers, a negation operator is prefixed. Examples:13.5, 9098.765

Character Literal - Character literals are characters that are enclosed by single quotes. Examples: 'c', '$', '2', '*' etc. There are some special character literals that hold specific meanings for the compiler, and act as escape sequences. They are listed below along with their meanings:

'\n' - Newline
'\t' - Horizontal Tab
'r' - Carriage Return

Character literals with more than one character are inherently machine-dependent and should be avoided.

String Literal - This is a double-quoted sequence of ASCII characters. A string can also be empty. Examples are : "Cimple is fun", "good", "I am 20 years old!", "".

Separators - A separator separates tokens. Separators themselves are simply single-character tokens. Following is a list of separators in Cimple :

Character	Token
'('	LPAREN
')'	RPAREN

'{'	LBRACKET
'}'	RBRACKET
'['	LBRACKET_SQUARE
']'	RBRACKET_SQUARE
','	COMMA
';'	SEMICOLON

Operators

Operators are symbols that command the language to carry out specific mathematical, relational or logical computations over a set of data items to produce a final result. Cimple recognizes many types of operators, which are distinguished into categories in two ways:

(a) Based on the number of operands an operator takes :- There are two types of operators on this basis - Unary (operators taking only one operand) and Binary (operators acting on two operands).

(b) Based on the type of computation an operator performs :- There are categories such as arithmetic, assignment, logical, relational, bit-wise among which operators can be segregated on this basis.

Following is a description of all the types:

1. Assignment operators :- These operators deal with operations of assigning values to variables, and are always binary. Examples: = denoting equals, += denoting plus equals, -= denoting minus equals, *= denoting multiply equals, etc.

2. Arithmetic operators :- These represent arithmetic operations on numeric values. Example of binary arithmetic operators are: + denoting addition, - denoting subtraction, * denoting product, / denoting division, % denoting modulus. In unary usage, the value of an expression formed by the unary plus (+) and its operand is the positive value of its operand, while the value of an expression formed by the unary minus (-) and its operand is the negated operand.

3. Relational operators :- These stand for operations such as comparison of numeric values and are binary, examples:- < denoting 'less than', > denoting 'greater than', <= denoting 'less than or equals', == denoting 'equals', != denoting 'not equals'.

4. Logical operators :- These are used to perform boolean condition checks on expressions and are binary, examples: && denotes 'and' i.e. operator evaluates to true if both of its boolean-typed operands evaluate to true, || denotes 'or' i.e. the result is true if either of its boolean-typed operands evaluate to true.

5. Bitwise operators :- These are used for operations on individual bits of the operands, and can be unary or binary. Examples:- & denoting the binary AND, ^ meaning the bitwise XOR, ~ meaning negation or the bitwise NOT, >> meaning right-shift, << meaning right-shift.

6. Increment and Decrement Operators :- These are C-style operators which are used to increase and decrease the value of the operands by 1, and are unary. These include ++ and --, standing for incrementation and decrementation respectively. These are used as postfix operators that act on an identifier (variable name) representing a numeric value. in such usage, the value of the identifier is incremented or decremented, but the value of the expression is equal to the value of the identifier before having been altered.

Following is a table of all the Cimple operators, listed in the decreasing order of precedence :

Operators	Associativity
( ) [ ] .	left to right
! ~ ++ -- + -	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
|	left to right
&&	left to right
||	left to right
?:	right to left
= += -= *= /= %= &= ^= |= <<= >>=	right to left

The + and - operators appearing first are the unary versions whereas the lower + and - are binary.

Comments

Comments in Cimple are anything which is enclosed within the character-pairs /* and */, or that which follows a pair of forward-slashes i.e. '//' in the same line. Following are the examples of the both types:

```
/* This is a
multi-line comment */

// Whereas this is a single line comment
```

Comments contain information which is ignored by the compiler, but that is helpful to the programmer writing and maintaining the code to which it refers. Comments cannot be nested.

Data Types

Data types in Cimple refer to the different forms of data that the compiler accepts to perform operations on. These come in the following two flavors:

Basic Data Types

The following primitive data-types are supported by the language.

`int, short, long` - These are the reserved words that refer to 32, 16, and 64-bit integer values in Cimple.

`float, double` - There are the reserved names for 32 and 64-bit floating-point numeric values in Cimple.

`char` - The char type is used to store a single character from the language character set, and occupies 8-bits of memory.

`string` - This is a built-in data type in Cimple, which is recognized as an array of characters. It is specified by the `string` keyword.

`void` - The void type indicates an absence of a value, and is used primarily as a return type of a function to indicate that it is used purely for its side effects, such as printing.

Collection Data Types

These are data-types that are formed from a collection of values of one of the basic data-types. All the values in the collection necessarily belong to the same primitive data-type. They are called arrays in Cimple.

Arrays are 0-indexed lists of the same type of values. They are declared by writing the number of items the array holds, enclosed by square brackets at the end of the name of an identifier, example :- int integerList[10]. Individual elements of the array can be accessed by

specifying an index into the array enclosed by square brackets at the end of the name of the identifier, example :

```
// Set the tenth element in the array to the value 10

integerList[9] = 10;
```

Custom Data Types

These are data-types that are user-defined and are not already included in the language specification. In Cimple, they are called structures.

Structures or `struct` as they are recognized by the compiler, are the primary means of creating custom data types. Structs encapsulate data members of arbitrary type and functions that act on or with these data members. Their size is variable and dependent on the data members defined by the user. Constructors can be defined that initialize the state of a struct by implicitly being called whenever an instance of that struct is allocated.

Struct Declaration

The declaration of a struct is illustrated in this example:

```
struct Vehicle{

    string color;

    float speed;

}
```

It can be seen in the above example that a struct having name 'Vehicle' is declared with the keyword `struct,` and holds two member variables of heterogeneous data-types. One thing which is important to note here is that the identifier used as the name of the struct begins with an upper-case letter. This is a requirement for the compiler to accept the identifier

as the name of the struct, whereas names for other things such as variables and functions are acceptable if beginning with a lower-case letter.

Struct Instantiation

Unlike the primitive data-types, custom data types need to be instantiated before they can be leveraged for data operations. The syntax observed in creating instances of structs is:

struct struct-identifier pointer-identifier = make struct-identifier();

`struct` and `make` are keywords here.

Example:-
```
struct  Employee *teacher = make Employee();
```

for some struct Employee defined as :-
```
struct Employee{
        int age;
        double wage;
        double getWage(){
            // function body
        }
}
```

Expressions

Programs in Cimple are built with expressions, and a program can be viewed as a list of expressions executed in sequence.
The types of expressions in Cimple are elaborated in the following table.

Name	Syntax	Examples

Identifiers	`identifier`	a, computeSum, Student
Constants	`integer` or `floating-point` or `character` or `string`	25000, 35.85, c, name
Address	`& identifier`	&num
Pointers	`* identifier` or `** identifier`	*ptr **doublePtr
Arrays	`identifier [ size ]`	numbers [ ] groceryList[quantity] books[10]

Arithmetic, Relational, Logical operations	`expression arithmetic-operator expression`
	or
	`expression relational-operator expression`
	or
	`expression logical-operator expression`
		34 + 5, (234 / 4 * 8) - 12
		25<=100, area/length != width
		3 || 8
		(a>b)&&(c==50)

Assignment operations	`expression assignment-operator expression`	height = 10 amount += (P*R*T)/100 age *= 5

Incrementation, Decrementation operations	`expression increment-operator`
	or
	`expression decrement-operator`
		num ++
		(5*13) --

Declaration	`declaration-specifier declarator-list`	int a, d string *pointer

Function call	`identifier ( )`
	or
	`identifier ( parameter-list )`
		add(one, two)
		getLength(array)

Memory management	`make struct-identifier ( )` `clean struct-identifier`	make Vehicle() (in a statement such as `struct Vehicle *car = make Vehicle();` discussed more in detail later), clean car
Struct member Dereferencing	`expression dereference-operator expression`	square.area (for a struct as the one below: `struct Shape{ int area; int volume; }` `struct Shape *square = make Square();`)

Anonymous function definition	`func return_type parameters statements`	func (int) (int a, int b){
		int product = a*b;
		return product;
		}

Statements

A statement is the smallest unit of execution of code. The following types of statements are supported in Cimple.

Expression Statements

An expression statement is basically just an expression followed by a semi-colon. It follows this syntax :-

expression;

A few examples:

```
12+60;

length -= 100;

char *p = &c;

string text;

int newArea = square.area--;

struct Employee *teacher = make Employee();
```

Being the smallest unit of execution of code, an expression statement gets executed with all the expressions that constitute it getting evaluated.

Compound Statements

Compound statement is nothing but a set of statements written one following the other, and enclosed within a pair of braces, in the following syntax:

{

statement_1

statement_2

```
        .
        .
        .
    statement_n
}
```

Here is an example:

```
{
    double r = 100.56;
    double s = 765.90;
    double result = r-s;
}
```

Conditional Statements

These statements are used for the conditional execution of another statement or a block of statements.

- `if` statement - This is a conditional statement, which is written in the following syntax:

`if` *(expression) statement*

The purpose of this statement is to evaluate whether the expression inside the parentheses is true, and if it is, to execute the statement that follows. Thus, execution of the statement is dependent on the validity of the preceding expression i.e. the execution is conditional.

Example:

```
if( num != 100){
    num+=50;
    int newNum = num;
}
```

- `else` statement - This is another conditional statement, which follows an `if` statement. this syntax is followed:

`if` *(expression) statement_1* `else` *statement_2*

The difference between this construct and the previous one having only an `if` clause is that if the expression enclosed within the parentheses is true, statement_1 will execute, but if it is not true (the case with which `else` corresponds), statement_2 will execute.

Example: if(weight>200) string message = "Heavy";

else string message = "Light";

Loop Statements

This is a type of statements which are used to execute the same statement or block of statements repeatedly as long as some specific condition is met. They are named 'loops' because they execute one pass of statements, circling back to the beginning and going again. There are three variants of these:

- `for` statement - This statement assumes the following syntax:

for (expression_1; expression_2; expression_3) statement

The way it works is, *expression_1* is an initialization expression, which behaves as the initiation of the loop. *expression _2* is a test expression which runs every time a new iteration of the loop is going to begin, in order to decide if the iteration will take place. *expression_3* is an increment or decrement expression, run after each loop iteration,

which causes the progression of the value of *expression_2* toward its final value, where the loop is supposed to stop.

It is worth noting that none of the three expressions in a loop's body are necessary, i.e. this syntax is totally acceptable:

for (; ;) statement

Example:

```
for(int i=10; i>0; i--){
     int j = 2*i;
}
```

- **`while`** statement - The **`while`** loop has the following syntax:

expression-1;
while (expression-2) {
statement
expression-3 ;
}

This loop statement works very similarly to the **`for`** loop. All expressions have the same purpose here as they did in the case of **`for`** loop, and they are all optional here as well. If *expression_2* is missing in a **`while`** statement, the statement is interpreted to be equivalent to **`while(1)`** or **`while(true)`**, which means that the test-condition always stays true and thus causes a never-ending execution of the associated statements.

Example:-

```
int m = 0;
int n = 8;
while(m>n){
     n=4*n;
     m+=5;
}
```

Control Flow Statements

This is a category of statements which are used to maneuver the flow of control during the execution of a Cimple program.

- `return` statement - This is a statement that allows the control to jump from a function in execution back to the code that invoked the said function.

  It follows either of the following syntax:

  return ;

  return (expression);

The difference between the two types is that some value is returned by the second statement while nothing is returned by the first. The value that gets returned in the second case has the same type as the type of the expression on the right side.

Example:-

```
{
    float radius = 25;
    return 3.14*radius*radius;
}
```

- `break` statement - This statement is used inside a loop to terminate the execution of it, before the test expression evaluates to `false` and terminates the loop. When a `break` is executed, the control transfers to the statement that follows the terminated statement or block.

  The syntax is very simple:- *break ;*

  Example :-

```
int value = 50;
while(value<100){
    value+=10;
    if (value==90) break;
}
```

Functions

A function in Cimple is a block of statements that execute when the function is invoked. They differ from a usual block of statements in that they can be executed in multiple places in a program, without having to be re-written, with a simple invocation statement called on an identifier reference to the function.

Function definition

A function in Cimple is defined using the following syntax:

return-type function-name (parameter declarations) {

declarations

statements

}

In a function definition, the 'return-type' , 'function-name' and the pair of matched parentheses are all required, while the other things like 'parameter declarations', 'declarations' and 'statements' are optional.

Return type of a function refers to the data-type of the value returned to the calling code, when that function is called. It can be any of the primitive types like `int`, `float`, `char`, `string` or a user-defined type or a pointer. A function may not return any value. In that case the return type needs to be specified as `void`. Although the body of a function, which is the block of statements that are enclosed in a pair of braces { } and follow the parentheses in a function definition, is optional when defining a function, if the function has a return type other than `void`, it should compulsorily have a `return` statement with a value having the same type as the return type of the function. This can be better explained with examples.

The following function definitions are correct.

```
double getVolume(double length, double width, double height){
```

```
            double volume = length*width*height;

            return volume;

}

void computeSum(int p, int q, int r, int s){

            int sum = p+q+r+s;

}
```

The following function definition is not correct, because the function will not return a value of the same type as the return type.

```
int findAverage(int num_1, int num_2){

     int avg = (num_1+num_2)/2;

}
```

Parameters, or arguments, of a function are values that are supplied to it to use them in some computation or to perform some operation on them. These can be of any type, primitive, user-defined or pointers. These are optional and can be left out of a function definition.

Function Call

The way to bring a function into execution is to call its name with the proper arguments. The syntax followed is:

function-name (arg_1, arg_2, ...)

Here, arg_1, arg_2 etc are values of the same type as the types of the parameters mentioned in the definition of the function, and they appear in the invocation in the same order as their counter-parts in the definition.

Example:-

```
void computeSum(int p, int q, int r, int s){
```

```
        int sum = p+q+r+s;
    }
```

For the above function definition, the function computeSum can be invoked by a statement like this:

```
computeSum(10, 8, 72, 3);
```

The rule for function calling changes a little for functions that are members of structs. Structs are Cimple's way of encapsulating data and methods in order to provide object-oriented programmability. Thus, a function which is defined inside a struct can only be invoked by an instance of that struct. This is done using the member-dereference operator, according to this syntax :-

struct-instance-name.function-name(parameter-list);

Here is an example for this scenario:

```
struct Course{
    string name;
    int id;
    void enroll(){
        // function body
    }
}
```

Invoking the function 'enroll':-

```
struct Course *literature = make Course();
literature.enroll();
```

Anonymous Functions

As mentioned earlier, anonymous functions are a new concept in Cimple, compared to C. They differ from the usual functions in that they are defined inline with another function definition, without a name. Because this means that a function could begin its definition within some scope other than global, it is valid for the scope of the anonymous function to reference variable names (identifiers) declared outside of its scope. In this case, the compiler will copy the values of these variables to a data structure that is passed as an implicit argument to the function.
One important feature of anonymous functions is that although they work as arguments to other functions, they are not returnable i.e. they cannot be used as return types of functions.

Anonymous Function Definition

An anonymous function is defined as follows:

func (type)(arg_1, arg_2, …) { statements }

The first set of parentheses can only have one type indicated, and represents the return type of the anonymous function. The second set of parentheses contains the parameter list, as with the named functions.

Example of anonymous function definition:-

```
func(int)(int i) {
    // Do something with i
    return i;
}
```

Anonymous Function Call

An anonymous function is used as an argument to another function, thus the anonymous function gets called during the execution of the function that it is serving as an argument to. The use of an anonymous function as an argument is specified according to the following syntax:

*return-type outer-function-name (arg_1, func(return-type)(arg_1, arg_2) *anon-function-name, arg_2, ...)*

anon-function-name is an alias for the anonymous function which takes an int parameter and returns a void.

An example will make this clearer.

```
void some_function(int x, func(int)(int) *fn)
{
    int result = fn(x);
     printf("%d\n", result);
}

int main(int argc, string* argv)
{
     for (int i = 0; i < 5; ++i) {
         some_function(i, func(int)(int j) {
             return j + 1;
          });
     }
}
```

Scope Rules

The accessibility of a variable in different parts of a program is called its scope. Not all variables can be accessed everywhere in a program, and the scope of a variable is dependent upon where in the program it has been declared.

Variables that are declared at the head of a Cimple program or elsewhere but not within any function or looping construct, are called global variables, and they can be accessed all through the entire program. Variables that are declared within a function or a `for` or `while` loop are called local variables, and their scope is limited to within the function or loop statement where they are declared. The scope of variables that are arguments to functions is also the function itself.

The following example illustrates the scope of various types of variables.

```
int numOfSides = 3;
```

```
struct Triangle {
     float sides[numOfSides-1];
     float perimeter;
}
int main(){
  struct Triangle *equilateral = make Triangle();
     for(int i=0;i< numOfSides;i++){
          equilateral.sides[i] = 10;
     }
   equilateral.perimeter = findPerimeter(equilateral);
   return 0;
}
float findPerimeter(Triangle t){
     int j,  peri =0;
     while(j< numOfSides){
          peri+=t.sides[j];
          j++;
     }
     return peri;
}
```

1. Variable 'numOfSides' is declared at the very beginning of the program, or otherwise, outside any function. Thus, it is accessible anywhere in the program till the end.

2. Variable `i` is declared in the `for` loop, and can be used in the same context in the loop alone, i.e. the scope of variable `i` extends from the body of the `for` loop following the expression `int i=0;` to the closing brace .

3. Variable `t` is an argument to the function `findPerimeter`. Thus, the scope of `t` is the body of `findPerimeter`.

4. Variables `i` and `peri` are declared within function `findPerimeter`, so they can only be used within that function.

5. Other identifiers such as `Triangle`, `equilateral`, `sides` and `perimeter` are struct instance or member identifiers, and they observe different scoping rules than the types discussed so far.

Inheritance

In Cimple, a struct can inherit from another struct. This is done with the use of `extends` keyword, such that a child struct "extends" the functionality of another struct, which will be its parent. The syntax is:

struct struct-identifier_1 extends struct-identifier_2 { member variables and functions }

Here is an illustration of this:

```
struct Person {
        string name;
        Person(string theName)
        {
            name = theName;
        }
    }

struct Student extends Person{
     string uni;
     Student(string name, string theUni){
       super(name);
       uni = theUni;
   }
```

```
}
```

Another keyword worth noting here is `super`. It allows access to a parent struct's methods and can only be used within the struct definition. In the above example, the constructor of the parent class is invoked by calling `super(name);`.
To access a method, this could be written:- `super.getName();` .

Interfaces

To benefit from a flexible inheritance scheme, structs can inherit the methods declared in other structs called interfaces, but provide native implementation to them. An interface is an abstract type which defines a *Method Set*, which is a collection of methods encapsulated in scope. A method set is a list of methods belonging to the same receiver. A method set is defined internally.
For example :-

```
{
    ReturnType1 (Receiver) func_name1();
    ReturnType2 (Receiver) func_name2 ();
}
```

The Receiver must begin with a capital letter or maybe empty. It is a semantic error for it to be empty for a non interface type.

A method is a function with a receiver associated with it.

Syntax:- *ReturnType (Receiver_opt) func_name ()*

Interface Declaration

An interface is declared in much the same way as a struct. The only difference lies in the keywords used to identify the two. This is the syntax:

interface struct-identifier{ method-set }

Example :-

```
interface Pet{
     void feed();
```

```
    void walk();

}
```

Interface Implementation

As in the case of a struct, an interface can be 'implemented' by a struct with the use of the keyword `implements`. The syntax followed is:

struct struct-identifier_1 implements struct-identifier_2 { member variables and method-sets }

Example:-

```
struct Dog extends Animal implements Pet{
    void feed(){
     // dog-specific feeding implementation
    }
    void walk(){
    // native implementation
    }
}
```

It is a structure which enforces certain properties on an object, which implements one. The receiver for this method set is *empty*. In particular all interfaces have the same receiver.

Memory Management

As mentioned earlier, Cimple uses two types of memory management for variables. The variables that come into action on the activation of their scope are automatically allocated space on the stack memory. Other variables, specifically those that refer to user-defined custom data-types are given space on the heap, whenever instances of those types are created. This is done with use of the keyword `make` followed by the identifier used for the custom type, as discussed in the above section dedicated to structs. A call to the constructor method of the struct can be used to initialize the data members to their default values.

Likewise, Cimple manages memory de-allocation skillfully. For heap variables, memory is not de-allocated until the programmer specifies such action with the use of `clean` keyword. That can be done in the following syntax:

clean struct-instance-identifier

Project Planning

Planning Process

We used to meet our TA, Alexandra Medway, every week on Tuesdays, to get a progress check about the tasks handled since the last meeting as well as to decide the goals for the upcoming week. Other team meetings used to be held on Tuesdays or Fridays to collaborate on the tasks more granularly. Even though a few soft deadlines were missed, the progress of the project remained largely steady throughout the semester. The timeline followed in completing the different milestones of the project is laid out in the 'Project Timeline' section.

Specification

The beginning of the project was marked by a decision about the type of language that we wanted to implement, worded down in our language proposal. We wanted the language to have the execution speed of C, while supporting some empowering features such as inheritance and anonymous functions, which meant that the language would extend the functionality of C somewhat into the realm of Java and Javascript. Our initial meeting with our TA revealed that the language, being an extension of C, would be better implemented if brought to compile down to C rather than LLVM.

Development and Testing

We started with a simple goal of defining the basic grammar, and getting an abstract syntax tree built out of the basic arithmetic operations. Successively, we expanded the grammar and re-iterated the above steps. Then we moved on to the further steps of getting static semantic checking and code generation working. Once the 'Hello World' milestone

was met, we moved on to the implementation of the larger 'non-C' features such as interfaces and anonymous functions. Every new feature was accompanied by a test, to make sure that the syntax tree was generated correctly and code generation was happening without glitches.

Roles and Responsibilities

After a discussion with the TA, it was made clear that even though all the team members were to shoulder responsibility for the entire project, assigning specific roles to each member would make sure that one person was answerable for falling behind on at least their individual responsibility, in case that happened. Accordingly, roles were allotted as follows:

Graham Barab - Manager and System Architect
Shankara Pailoor - Language Guru
Panchampreet Kaur - Tester

Test Plan

Unit Tests

We started testing the scanner, parser and ast with unit-tests, to determine whether the basic arithmetic and assignment expressions were being parsed correctly into a syntax tree. At first, the testing was limited to the use of menhir. However, in subsequent meetings with the TA, we decided to setup a dedicated test-suite with a structure to support four kinds of tests:

1. Parse-tree tests - Pass and Fail

2. Build tests - Pass and Fail

The parse tree tests were setup to make sure that the syntax tree was getting created properly and the build tests were to test whether code-generation was taking place without hassles. The pass sub-category pertains to the syntactically correct programs which were expected to pass the various front-end and back-end components of the compiler

while the fail category refers to the incorrect programs which were supposed to generate a compilation error.

Besides, we also made use of the Travis automated testing suite, which was a big help in making sure commits on all different branches were resolved for errors.

Integration Tests

As the project progressed to the later stages where we moved to the implementation of interfaces and anonymous functions, we started writing more detailed programs for testing purposes. The integration testing was carried out using the same test suite as described in the above section.

Automation

As mentioned earlier, Travis CI was an automated testing tool that we incorporated into our test suite to make use of at the level of version control. Besides, we added scripts to the folder containing each type of tests, which were added to the project Makefile and would run comparing the outputs of all test programs against their expected outputs.

Source to Target

The code comprising the test-suite is included in the appendix section of this report.

Software Development Environment

The following tools were used in the development of this project:

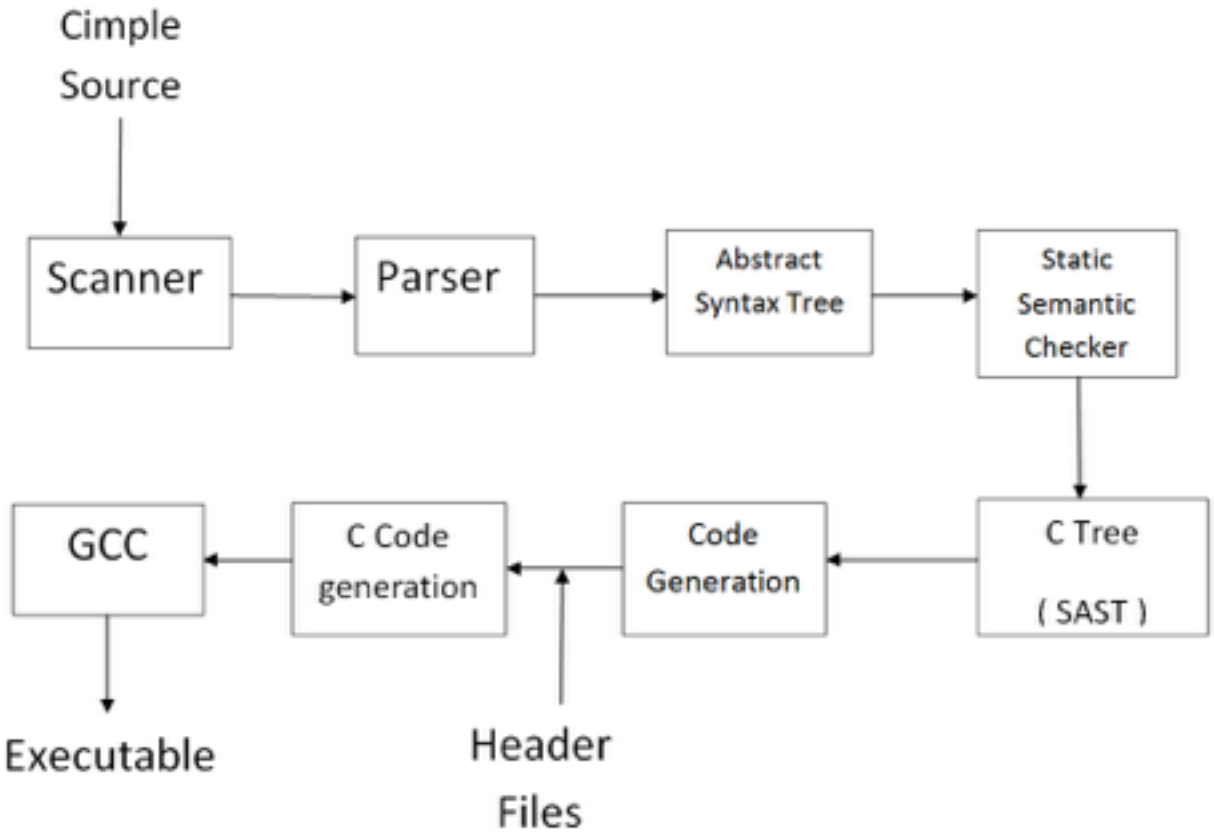
Purpose	Tool

Programming language Compiler front-end C output generation from Cimple code Operating systems Version Control Text Editors	Ocaml version 4.02.3 Ocamllex, Ocamlyacc GCC version 5.4.0 Mac OS X, Ubuntu Github Nano, Vim

Project Timeline

Date	Milestone
September 15	Brainstorming about basic language features
September 28	Language Proposal submitted
October 26	Language Reference Manual submitted
November 15	Scanner and Parser complete
November 18	*Hello World* working
December 9	Interfaces implemented
December 16	Anonymous functions and C codegen implemented
December 19	Submission of Final report and Project Demo

Architectural Design



The point of entry into the compiler is file in the project folder, called 'cimple.ml', which takes in a Cimple source file and routes it through the correct components of the compiler. The parts through which a source file goes are explained in sequence below.

Scanner - This part accepts the stream of characters that make up the source of the program, and outputs tokens made out of those characters, with characters such as spaces and user comments filtered out. The tokens then go into the parser. The code associated with this component is written in scanner.mll.

Parser - This part accepts tokens and emits an abstract syntax tree from those tokens in accordance with the grammar of the language. This part is coded in parser.mly.

Abstract Syntax Tree - This is a tree structure of all things such as data types, expressions and statements that are accepted by the language,

and is further used for static semantic checking. The code for the syntax tree resides inside ast.mli.

Semantic Checker - This part deals with the verification and validation of the variables, types and expressions forming the program. The code for this part is contained in semant.ml.

C-Tree - This part deals with the verification of the correctness of the types, values, expressions of the intermediary C program which results from the Cimple source, after the latter is passed by the semantic checking stage. The code for this can be found in ctree.ml.

Code Generation - This part outputs a C program after the Cimple program is verified for its semantic validity and C header files are linked on. The relevant file is ccodegen.ml.

In the end, the C program that has so far formed, is fed to GCC, which compiles it and returns an executable to be run.

Lessons learnt

Shankara

When designing a compiler it is important that everyone has a very clear understanding of all the components of the language. To achieve that I really recommend taking professor Edward's plea to have your parser and scanner done by Nov 1 or earlier. Having a complete parser and scanner is crucial because last minute changes to it could be disastrous. We had some near close calls and having to suddenly deal with shift/reduce errors can be stress inducing. I think you should choose a language that will motivate you to complete it. For me, I was very interested in the components we developed, like classes, inheritance, anonymous functions and interfaces and I wanted to get them to work. It is also critical that you maintain a diverse and extensive set of tests as you develop. Without it, you simply cannot guarantee you changes haven't broken something else. Having that piece of mind as the semester nears to a close is invaluable. Travis CI was a great tool for us, but there are many such out there. Lastly, don't fight OCaml, embrace the way it is. Being open to a

new way of programming will make this class much easier and you will have a lot more fun.

Graham

Obviously, as many have said before, it is critical that the project be started as early as possible, and worked on regularly. In particular, the parser should be air tight as early as possible, because it is very easy to introduce subtle errors by modifying it later into the project.

Since I come from an imperative and object oriented programming background, debugging a functional programming language such as OCaml proved to be a bit of a challenge. Over the course of this project, I have found that a rudimentary, but effective, way to debug and troubleshoot is to use the sequencing feature of the language (statements followed by a semicolon) to print debug messages at runtime. The failure of a particular print statement to appear can be very revealing.

As the manager, another very important lesson I have learned is the importance of staying in regular communication with all temamembers, and make sure contributions occurr regularly.

Graham Barab (Manager, System Architect)
Pancham

Through this project, I have learnt that it is very crucial that one takes sufficient input from people who have done such a project in a similar environment, like students from previous years or the TAs, about the scope and gravity of the project. This would help in setting realistic yet challenging enough deadlines for the various milestones of the project and keep unpleasant surprises from coming your way through the semester, especially toward the end.
Another take away has been that one should put off pushing code until all build tests pass, and to also keep your code in your separate branch first before pushing to the common branch. Having to spend time on merge conflicts is one thing, but taking other team members down with you by messing with their branch is an even worse scenario. In the end, among

the many widely applicable skills that this class can equip you with, getting to know and adopt functional programming has definitely been the highlight.

Project Log

```
ec232739b0dcafbd0156129809678d377be0a760 Merge branch 'graham' of https://github.com/TeamCimple/Cimple into graham
4c46bb19038fa1050055bb68fb61be4090302188 Cleanup
bc9bb35d795f6033d9f1e71b9863c530599dc5d9 Adding dsp demo
dc98d880cb78d0b18997ff018736399b08bc3c36 adding demo folder with programs from demo
74eae35fe5cdf0951d500a84c8deae94dea7522e Caught match error
42e7e994b176b5f54bbe16f3122ac2c190e8aaa9 Fixed duplicate symbols bug for real
b869cd2f591d8fda7a1a025b4d35d92c27b481f2 Fixed order of declarations bug
fa5c1b2844c0b03741edb659be2073ad2f335283 Tutorial script
13df3d74eab1dc0d48cc8528d08428a1154eaee6 Fixed bug in members_from* causing duplicate symbols to be generated in c
9078c91a4d6ca0eab3422670398e1641a0d4fb6e Got rid of first c compilation errors
c9c4d8458aae7226bf45a894fd31f0ef3d709c43 Closer, still compilation errors in c file output
7c90c5a37a37d62df896082aa8bbf041d39f0181 Fixed undeclared receiver symbol error
edc8a17fa7d27e951e5df5e2a5ad9f05dee59083 fixing merge conflict
2c609db9e97cdacb987ec6c42598e0c867d8cb7a adding changes
704b832d2442393b39726abe4b4f0d2bbe0bfd7e binary search out file
f73e4520b4981f9a538b1ab0369acb6a73efcf9c Super statements support anonymous functions
fc4613ee42039d0fe32723bad04fbb9133ac10d5 Fixed order of argument bug
515c0acf901015f573845c1c4f5991f226698d00 Merge branch 'graham' of https://github.com/TeamCimple/Cimple into graham
88fbc278a5781574950e1e96c10ae4cfc3e12398 Support for anon functions in methods
845977a8c4ac2dae48e1f12157296193497ddad9 changing back order of arguments to function. It broke the anonbuild
ac58430414843216edc43114fb2e1095661d2b79 fixed bug with function not allowing pointer return types
470217109915c451fa6a243a7a77b4cff226337b adding parse tree tests
9c901ab0ee3ac050109654dd56787716ac3568c3 merged
7cb855a5c3abfbe674dbf694db9e809f521c629d converting globals
5ed98256895c128e773c66c1abb29c2258473d64 Added semantic check for printf, math functions
eeab3b034463e34666597b5730825414c583d414 Merge branch 'graham' of https://github.com/TeamCimple/Cimple into graham
38219aa234bb1c8fdefefa0d82a59ca2f768c8b0 printf works, still need semantic checking
0692e22b9fa33d945c41b8bf98e75c8fbdd436db added destructors
77252fdd27ba348be81cdcb820a2bcbefea9996a Merge branch 'graham' of https://github.com/TeamCimple/Cimple into graham
e068d36b5298b1c64460b7cafe320d7f62c46ca1 Started work on printf
```

Code

scanner.mll

```
(*{ open Parser open Test }*)
{ open Parser }

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf }
| ';' { SEMICOLON }
| "++" { PLUSPLUS }
| '+' { PLUS }
| "+=" { PLUS_ASSIGN }
| "--" { MINUSMINUS }
| '-' { MINUS }
| "-=" { MINUS_ASSIGN }
| "/*" { comment 1 lexbuf}
| '*' { TIMES }
| "*=" { TIMES_ASSIGN }
| '/' { DIVIDE }
| "/=" { DIVIDE_ASSIGN }
| '%' { MOD }
| "%=" { MOD_ASSIGN }
| "<<" { LSHIFT }
| "<<=" { LSHIFT_ASSIGN }
| ">>" { RSHIFT }
| ">>=" { RSHIFT_ASSIGN }
| "&&" { AND }
| '&' { BITWISE_AND }
| "&=" { AND_ASSIGN }
| '^' { XOR }
| "^=" { XOR_ASSIGN }
| "||" { OR }
```

```
| "|" { BITWISE_OR }
| "|=" { OR_ASSIGN }
| "~" { NOT }
| "!=" { NOT_EQUALS }
| "==" { EQUALS }
| "<" { LESS_THAN }
| "<=" { LESS_THAN_EQUALS }
| ">" { GREATER_THAN }
| ">=" { GREATER_THAN_EQUALS }
| "NIL" { NIL }
| "super" { SUPER }
| "clean" { CLEAN }
| ['0'-'9']+ as lit { INT_LITERAL(int_of_string lit) }
| ['0'-'9']+"."+['0'-'9']* as lit { FLOAT_LITERAL(float_of_string lit) }
| '"' + [' ' '_' '%' '^' '/' '[' ']' '(' ')' '#' '&' '$' '@' '|' '~' '+' '-' ' ' '\'' '0'-'9' 'a'-'z' 'A'-'z' '\\']* + '"' as lit { STRING_LITERAL(lit) }
| "extends" { EXTENDS }
| "make" { MAKE }
| "implements" { IMPLEMENTS }
| "interface" { INTERFACE }
| "auto" { AUTO }
| "register" { REGISTER }
| "static" { STATIC }
| "extern" { EXTERN }
| "typedef" { TYPEDEF }
| "void" { VOID }
| "char" { CHAR }
| "short" { SHORT }
| "int" { INT }
| "string" { STRING }
| "long" { LONG }
```

```
| "float" { FLOAT }
| "double" { DOUBLE }
| "signed" { SIGNED }
| "unsigned" { UNSIGNED }
| "func" { FUNC }
| "const" { CONST }
| "volatile" { VOLATILE }
| "struct" { STRUCT }
| "union" { UNION }
| "enum" { ENUM }
| "case" { CASE }
| "default" { DEFAULT }
| "if" { IF }
| "else" { ELSE }
| "switch" { SWITCH }
| "while" { WHILE }
| "do" { DO }
| "for" { FOR }
| "goto" { GOTO }
| "continue" { CONTINUE }
| "break" { BREAK }
| "return" { RETURN }
| '{' { LBRACKET }
| '}' { RBRACKET }
| '[' { LBRACKET_SQUARE }
| ']' { RBRACKET_SQUARE }
| '(' { LPAREN }
| ')' { RPAREN }
| '.' { PERIOD }
| ',' { COMMA }
| '=' { ASSIGN }
| '?' { QUESTION }
| ':' { COLON }
| '*' { ASTERISK }
| "..." { ELLIPSIS }
| ['a'-'z''_']+['a'-'z''A'-'Z''_''0'-'9']* as lit { IDENTIFIER(lit) }
| ['A'-'Z']+['a'-'z''A'-'Z''_''0'-'9']* as structLit {
        STRUCT_IDENTIFIER(structLit) }
| eof { EOF }
```

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

and comment depth = parse
  "/*"  {comment (depth+1) lexbuf}
|   "*/" { if (depth = 1) then token lexbuf else comment (depth-1) lexbuf}
| _ { comment depth lexbuf }
```

parser.mly

```
%{ open Ast %}

%token SEMICOLON
%token <int> INT_LITERAL
%token <float> FLOAT_LITERAL
%token <string> STRING_LITERAL
%token <string> IDENTIFIER
%token <string> STRUCT_IDENTIFIER
%token ASSIGN
%token RETURN
%token NIL
%token PLUS MINUS TIMES DIVIDE MOD PLUSPLUS MINUSMINUS
%token AND OR BITWISE_AND BITWISE_OR XOR NOT LSHIFT RSHIFT
%token EQUALS NOT_EQUALS LESS_THAN LESS_THAN_EQUALS GREATER_THAN GREATER_THAN_EQUALS
%token TIMES_ASSIGN DEVIDE_ASSIGN MOD_ASSIGN PLUS_ASSIGN MINUS_ASSIGN LSHIFT_ASSIGN RSHIFT_ASSIGN AND_ASSIGN XOR_ASSIGN OR_ASSIGN
%token AUTO REGISTER STATIC EXTERN TYPEDEF
%token VOID CHAR SHORT INT LONG FLOAT DOUBLE SIGNED UNSIGNED STRING FUNC
%token CONST VOLATILE
%token STRUCT UNION INTERFACE MAKE SUPER CLEAN
%token SWITCH CASE ENUM DEFAULT IF ELSE
%token LBRACKET RBRACKET LBRACKET_SQUARE RBRACKET_SQUARE LPAREN RPAREN COMMA
COLON ELLIPSIS ASTERISK PERIOD
%token WHILE DO FOR GOTO CONTINUE BREAK
%token EXTENDS IMPLEMENTS
%token QUESTION
%token EOF

%nonassoc NOELSE
%nonassoc NOPOINTER
%nonassoc NOCALL
%nonassoc ELSE
%nonassoc DELENIATOR
%nonassoc LPAREN
%start program
%type <Ast.tProgram> program

%%
```

```
statement_list:
  /* nothing */ { [] }
  | statement_list statement { $2::$1 }

statement:
  expr_opt SEMICOLON { Expr $1 }
  | selection_statement { $1 }
  | compound_statement { $1 }
  | iteration_statement { $1 }
  | BREAK SEMICOLON { Break }
  | RETURN expr_opt SEMICOLON { Return $2 }

selection_statement:
    IF LPAREN expr RPAREN statement %prec NOELSE{ If($3, $5, EmptyElse)}
  | IF LPAREN expr RPAREN statement ELSE statement  {If($3, $5, $7)}

iteration_statement:
    WHILE LPAREN expr RPAREN statement { While($3, $5) }
  | FOR LPAREN expr_opt SEMICOLON expr_opt SEMICOLON expr_opt RPAREN statement { For($3, $5, $7, $9) }

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

expr:
  assignment_expression { $1 }
  | make_expr { $1 }
  | anon_func_def { AnonFuncDef($1) }

assignment_expression:
    postfix_expr assignment_operator expr { AsnExpr($1, $2, $3) }
  | logical_or_expression { $1 }


logical_or_expression:
  | logical_and_expression { $1 }
  | logical_or_expression OR logical_and_expression { CompareExpr($1, LogicalOr,
  $3) }
```

```
logical_and_expression:
  | equality_expression { $1 }
  | logical_and_expression AND equality_expression { CompareExpr($1, LogicalAnd,
  $3) }

equality_expression:
  | relational_expression { $1 }
  | equality_expression EQUALS relational_expression { CompareExpr($1, Eql,
  $3) }
  | equality_expression NOT_EQUALS relational_expression { CompareExpr($1,
  NotEql, $3) }

relational_expression:
  | add_expr { $1 }
  | relational_expression LESS_THAN add_expr { CompareExpr($1, Less, $3) }
  | relational_expression GREATER_THAN add_expr { CompareExpr($1, Greater, $3) }
  | relational_expression LESS_THAN_EQUALS add_expr { CompareExpr($1, LessEql,
  $3) }
  | relational_expression GREATER_THAN_EQUALS add_expr { CompareExpr($1,
  GreaterEql, $3) }

unary_expr:
  | postfix_expr { $1 }
  | MINUS postfix_expr { Neg($2) }

postfix_expr:
    primary_expr { $1 }
  | postfix_expr PLUSPLUS { Postfix($1, PostPlusPlus) }
  | postfix_expr MINUSMINUS { Postfix($1, PostMinusMinus) }
  | postfix_expr LPAREN expr_list RPAREN { Call(Noexpr, $1, $3) }
  | postfix_expr LBRACKET_SQUARE postfix_expr RBRACKET_SQUARE { ArrayAccess($1, $3) }
  | SUPER LPAREN expr_list RPAREN { Super($3) }
  | CLEAN primary_expr { Clean($2) }
  | postfix_expr PERIOD  IDENTIFIER LPAREN expr_list RPAREN  { Call($1,
  Id(Identifier($3)), $5) }
  | postfix_expr PERIOD IDENTIFIER %prec NOCALL{ MemAccess($1, Identifier($3)) }
```

```
make_expr:
        MAKE STRUCT_IDENTIFIER LPAREN expr_list RPAREN { Make(CustomType($2),
        $4)}
   | MAKE type_ LBRACKET_SQUARE primary_expr RBRACKET_SQUARE  {
          Make(ArrayType($2, NoPointer, $4), []) }
   /*| MAKE type_ pointer LBRACKET_SQUARE primary_expr RBRACKET_SQUARE {
          Make(ArrayType($2, $3, $5), [])}*/

expr_list:
  /* Nothing */ { [] }
| expr    { [$1] }
| expr_list COMMA expr { $3 :: $1 }

assignment_operator:
   ASSIGN { Asn }
 | TIMES_ASSIGN { MulAsn }
 | DIVIDE_ASSIGN { DivAsn }
 | MOD_ASSIGN { ModAsn }
 | PLUS_ASSIGN { AddAsn }
 | MINUS_ASSIGN { SubAsn }
 | LSHIFT_ASSIGN { LshAsn }
 | RSHIFT_ASSIGN { RshAsn }
 | AND_ASSIGN { AndAsn }
 | XOR_ASSIGN { XorAsn }
 | OR_ASSIGN { OrAsn }

binary_operator:
   AND { And }
 | OR { Or }
 | BITWISE_AND { BitAnd }
 | BITWISE_OR { BitOr }
 | XOR { Xor }
 | NOT { Not }
 | LSHIFT { Lsh }
 | RSHIFT { Rsh }
```

```
logical_opeator:
   EQUALS { Eql }
 | NOT_EQUALS { NotEql }
 | LESS_THAN { Less }
 | LESS_THAN_EQUALS { LessEql }
 | GREATER_THAN { Greater }
 | GREATER_THAN_EQUALS { GreaterEql }

add_expr:
   add_expr PLUS mult_expr { Binop($1, Add, $3) }
 | add_expr MINUS mult_expr { Binop($1, Sub, $3) }
 | mult_expr  { $1 }

mult_expr:
   mult_expr TIMES unary_expr { Binop($1, Mul, $3) }
 | mult_expr DIVIDE unary_expr { Binop($1, Div, $3) }
 | mult_expr MOD unary_expr { Binop($1, Mod, $3) }
 | unary_expr             { $1 }

primary_expr:
 LPAREN expr RPAREN         { $2 }
 | FLOAT_LITERAL            { FloatLiteral($1) }
 | INT_LITERAL              { Literal($1) }
 | STRING_LITERAL           { StringLiteral($1) }
 | IDENTIFIER               { Id(Identifier($1))}
 | NIL                      { Nil }
 | BITWISE_AND primary_expr { Pointify($2) }
 | TIMES primary_expr       { Deref($2) }

type_specifier:
   VOID { Void }
 | CHAR { Char }
 | SHORT { Short }
 | INT { Int }
 | LONG { Long }
 | FLOAT { Float }
 | DOUBLE { Double }
 | SIGNED { Signed }
 | UNSIGNED { Unsigned }
```

```
 | STRING { String }

storage_class_specifier:
   AUTO   { Auto }
 | REGISTER { Register }
 | STATIC   { Static }
 | EXTERN  { Extern }
 | TYPEDEF { Typedef }

declaration_specifiers:
   type_ { DeclSpecTypeSpecAny($1) }
 | declaration_specifiers type_ { DeclSpecTypeSpecInitlist($2, $1) }

type_:
  type_specifier pointer { let rec num_ptrs ptr = match ptr with
                               | PtrType(_, next_ptr) -> 1 + num_ptrs
                               next_ptr
                               | Pointer -> 1
                               | NoPointer -> 0 in PointerType(PrimitiveType($1),
 num_ptrs $2) }
 | type_specifier %prec NOPOINTER { PrimitiveType($1) }
 | STRUCT STRUCT_IDENTIFIER pointer { let rec num_ptrs ptr = match ptr with
                               | PtrType(_, next_ptr) -> 1 + num_ptrs
                               next_ptr
                               | Pointer -> 1
                               | NoPointer -> 0 in PointerType(CustomType($2),
 num_ptrs $3) }
 | STRUCT STRUCT_IDENTIFIER %prec NOPOINTER{ CustomType($2) }
 | INTERFACE STRUCT_IDENTIFIER { CustomType($2) }

init_declarator_list:
   init_declarator { InitDeclList([$1]) }
 | init_declarator_list COMMA init_declarator { InitDeclList($3::[$1])}

init_declarator:
   declarator  { InitDeclarator($1) }
 | declarator ASSIGN expr { InitDeclaratorAsn($1, Asn, $3) }
```

```
 pointer:
    TIMES pointer { PtrType(Pointer, $2) }
  | TIMES { Pointer }

 declarator:
    direct_declarator { DirectDeclarator($1) }

 record_initializer_list:
    | record_initializer { Initializer($1) }
    | record_initializer_list COMMA record_initializer {InitializerList($1 ::
            [$3])}

 record_initializer:
    assignment_expression { InitializerExpr($1) }
    | LBRACKET record_initializer_list RBRACKET { $2 }
    | LBRACKET record_initializer_list COMMA RBRACKET { $2 }

 direct_declarator:
     IDENTIFIER { Var(Identifier($1)) }

 declaration:
    declaration_specifiers init_declarator_list SEMICOLON { Declaration($1, $2)}

 declaration_list:
    /* Nothing */ { [] }
    | declaration_list declaration { $2 :: $1 }


 struct_declaration:
  STRUCT STRUCT_IDENTIFIER struct_inheritence_opt struct_interface_opt LBRACKET
  declaration_list constructor_destructor_opt RBRACKET SEMICOLON { {
          members = (List.rev $6);
          struct_name = $2;
          extends = $3;
          children = [""];
          methods = [];
          Implements = $4;
          constructor = (fst $7);
          destructor = (snd $7);
```

```
}}

struct_inheritence_opt:
  EXTENDS STRUCT_IDENTIFIER { $2 }
  | { "" }

struct_interface_opt:
   | IMPLEMENTS STRUCT_IDENTIFIER { $2 }
   |  { "" }

interface:
   INTERFACE STRUCT_IDENTIFIER LBRACKET func_decl_list RBRACKET SEMICOLON{{
                                name = $2;
                                funcs = $4;
                            }}

compound_statement:
     LBRACKET declaration_list statement_list RBRACKET { CompoundStatement((List.rev $2),
     (List.rev $3)) }

func_params:
    declaration_specifiers declarator { FuncParamsDeclared($1, $2) }
  | declaration_specifiers { ParamDeclwithType($1) }
  | anon_func_decl { AnonFuncDecl($1) }


func_params_list:
   /* Nothing */ { [] }
   | func_params { [$1] }
   | func_params_list COMMA func_params { $3 :: $1 }


receiver:
        STRUCT_IDENTIFIER IDENTIFIER (($1, $2))
        | STRUCT_IDENTIFIER TIMES IDENTIFIER (($1, $3))
        | (("", ""))

constructor_destructor_opt:
     STRUCT_IDENTIFIER LPAREN func_params_list RPAREN compound_statement NOT
```

```
            constructor_name = $1;
            constructor_params = $3;
            constructor_body = $5;
    }, {destructor_name = $1; destructor_body = $10}))
    | /* Nothing */ {{{constructor_name = ""; constructor_params = [];
    constructor_body = CompoundStatement([],
    [])}, {destructor_name = ""; destructor_body = CompoundStatement([], [])}}}

func_decl:
    declaration_specifiers declarator LPAREN func_params_list RPAREN compound_statement { {
            return_type = $1;
            func_name = $2;
            receiver = ("", "");
            params = ($4);
            body = $6 }}
    | declaration_specifiers LPAREN receiver RPAREN declarator LPAREN
    func_params_list RPAREN compound_statement {{
            return_type = $1;
            func_name = $5;
            receiver = $3;
            params = $7;
            body = $9
    }}
    | declaration_specifiers LPAREN receiver RPAREN declarator
    LPAREN func_params_list RPAREN SEMICOLON {{
           return_type = $1;
           func_name = $5;
           receiver = $3;
           params = $7;
           body = CompoundStatement([], [])
    }}
    | declaration_specifiers declarator LPAREN func_params_list RPAREN
    SEMICOLON {{
           return_type = $1;
           func_name = $2;
           receiver = ("", "");
           params = $4;
           body = CompoundStatement([], [])
    }}
```

```
func_decl_list:
        /*Nothing */ { [] }
        | func_decl_list  func_decl { $2::$1 }

anom_func_def:
   FUNC LPAREN RPAREN LPAREN func_params_list RPAREN compound_statement { {
    anon_name = "";
    anon_return_type = PrimitiveType(Void);
    anon_params = ($5);
    anon_body = $7}
   }

 | FUNC LPAREN type_ RPAREN LPAREN func_params_list RPAREN compound_statement { {
    anon_name = "";
    anon_return_type = $3;
    anon_params = ($6);
    anon_body = $8}
 }

anom_func_decl:
   FUNC LPAREN RPAREN LPAREN func_params_list RPAREN IDENTIFIER { {
    anon_decl_return_type = PrimitiveType(Void);
    anon_decl_params = ($5);
    anon_decl_name = Identifier($7);}
  }
 | FUNC LPAREN type_ RPAREN LPAREN func_params_list RPAREN IDENTIFIER { {
    anon_decl_return_type = $3;
    anon_decl_params = ($6);
    anon_decl_name = Identifier($8);}
  }

decls:
        /* Nothing */ { { globals = []; structs = []; functions = []; interfaces
        = [] }}
   | decls func_decl { {functions = $2 :: ($1.functions); globals = $1.globals;
   structs = $1.structs; interfaces = $1.interfaces} }
   | decls declaration { { functions = $1.functions; globals =  ($2 ::
           $1.globals); interfaces = $1.interfaces;
   structs = $1.structs }}
```

```
   | decls struct_declaration { {functions = $1.functions; globals =
           $1.globals; interfaces = $1.interfaces; structs = ($2 :: $1.structs)}}
   | decls interface { {functions = $1.functions; globals = $1.globals; structs
   = $1.structs; interfaces = $2 :: ($1.interfaces) } }

program:
   decls EOF { $1 }
```

ast.mli

Semant.ml

```
open Ast


module StringMap = Map.Make(String)




let add_symbol_list_to_symtable symlist symtable =

    List.fold_left (fun tbl x ->

        let symbolStr = (Astutil.string_of_symbol_simple x) in

        if

             (StringMap.mem symbolStr tbl) then raise(Failure("Error, redefining symbol
" ^ symbolStr))

        else

                  (StringMap.add (Astutil.string_of_symbol_simple x) x tbl)) symtable
symlist


let stdlib_funcs =

        [{

               return_type = (DeclSpecTypeSpec(Int));

               func_name = DirectDeclarator(Var(Identifier("printf")));

               params = [FuncParamsDeclared(DeclSpecTypeSpec(String),

               DirectDeclarator(Var(Identifier("x"))))];

               receiver = ("", "");

               body = CompoundStatement([], []);

       };

       {

              return_type = (DeclSpecTypeSpec(Float));

              func_name = DirectDeclarator(Var(Identifier("cos")));

              params = [ParamDeclWithType(DeclSpecTypeSpec(Float))];

              receiver = ("", "");

              body = CompoundStatement([], []);
```

```
        };
        {
                return_type = (DeclSpecTypeSpec(Float));
                func_name = DirectDeclarator(Var(Identifier("sin")));
                params = [ParamDeclWithType(DeclSpecTypeSpec(Float))];
                receiver = ("", "");
                body = CompoundStatement([], []);
        };
        {
                return_type = (DeclSpecTypeSpec(Float));
                func_name = DirectDeclarator(Var(Identifier("exp")));
                params = [ParamDeclWithType(DeclSpecTypeSpec(Float))];
                receiver = ("", "");
                body = CompoundStatement([], []);
        };
        {
                return_type = (DeclSpecTypeSpec(Float));
                func_name = DirectDeclarator(Var(Identifier("log")));
                params = [ParamDeclWithType(DeclSpecTypeSpec(Float))];
                receiver = ("", "");
                body = CompoundStatement([], []);
        }
        ]


let rec string_of_type tp =
    let string_of_primitive_type = function
       | Void -> "Void"
       | Char -> "Char"
       | Short -> "Short"
       | Int -> "Int"
```

```
        | Long -> "Long"
        | Float -> "Float"
        | Double -> "Double"
        | Unsigned -> "Unsigned"
        | Signed -> "Signed"
        | String -> "String"
    in match tp with
     PrimitiveType(t) -> string_of_primitive_type t
   | CustomType(s) -> "CustomType(" ^ s ^ ")"
    | AnonFuncType(t, tlist) -> "AnonFuncType(ReturnType: " ^ string_of_type t ^ ",
ParamTypes: " ^ string_of_type_list tlist ^ ")"
   | PointerType(base, num) -> "PointerType(" ^ string_of_type base ^ "," ^
   string_of_int num ^ ")"
   | ArrayType(_, _, _) -> ""
   | NilType -> ""

and string_of_type_list = function
    [] -> ""
  | [x] -> string_of_type x
  | h::t -> string_of_type h ^ " " ^  string_of_type_list t

let var_name_from_direct_declarator = function
     DirectDeclarator(Var(Identifier(s))) -> s
   | PointerDirDecl(_, Var(Identifier(s))) -> s


let var_name_from_declaration = function
     Declaration(_ , InitDeclarator(dd)) -> var_name_from_direct_declarator dd
   | Declaration(_, InitDeclaratorAsn(dd, _, _)) -> var_name_from_direct_declarator dd
       | Declaration(_, InitDeclList([InitDeclarator(dd)])) ->
var_name_from_direct_declarator dd
```

```
   | Declaration(_, InitDeclList([InitDeclaratorAsn(dd, _, _)])) ->
                 var_name_from_direct_declarator dd
   | _ -> raise(Failure("var_name_from_declaration: not yet supported"))

let var_name_from_anon_decl adecl = match adecl.anon_decl_name with
    Identifier(s) -> s

let var_name_from_func_param = function
    FuncParamsDeclared(_, x) -> var_name_from_direct_declarator x
   | AnonFuncDecl(adecl) -> var_name_from_anon_decl adecl
   | _ -> raise(Failure("pointers not supported"))

let rec type_from_declaration_specifiers = function
   DeclSpecTypeSpec(tspec) -> PrimitiveType(tspec)
 | DeclSpecTypeSpecAny(t) -> t
 | _ -> raise(Failure("type_from_declaration_specifiers: invalid specifiers"))
  (*| DeclSpecTypeSpecInitList(t, tDeclSpecs) -> CompoundType(t,
type_from_declaration_specifiers tDeclSpecs)*)

let rec get_num_pointers ptrs = match ptrs with
    | PtrType(ptr1, ptr2) -> (get_num_pointers ptr1) + (get_num_pointers ptr2)
    | Pointer -> 1
    | NoPointer -> 0

let get_func_name fdecl = var_name_from_direct_declarator fdecl.func_name

let type_from_declaration = function
      | Declaration(decl_spec, _) -> type_from_declaration_specifiers decl_spec


let is_assignment_declaration decl = match decl with
```

```
      | Declaration(decl_spec,
      InitDeclList([InitDeclaratorAsn(_, _, _)])) -> true
      | Declaration(decl_spec, InitDeclaratorAsn(PointerDirDecl(ptr, _), _ , _))
      -> true
      | _ -> false


let rec type_from_func_param = function
     FuncParamsDeclared(t, PointerDirDecl(ptr, _)) ->
             PointerType(type_from_declaration_specifiers t, get_num_pointers
             ptr)
   | FuncParamsDeclared(t, _) -> type_from_declaration_specifiers t
   | ParamDeclWithType(declspecs) -> type_from_declaration_specifiers declspecs
   | AnonFuncDecl(adecl) -> type_from_anon_decl adecl


and type_list_from_func_param_list l = List.map type_from_func_param l


and param_list_has_void params func_name =
        List.map (fun param -> if ((type_from_func_param param) = PrimitiveType(Void))
                               then
                                raise(Failure("Using void as a function
                                       parameter for function: " ^ func_name))
                                else ()) params


and type_from_anon_decl d = AnonFuncType(d.anon_decl_return_type,
type_list_from_func_param_list d.anon_decl_params)


let type_from_anon_def d = AnonFuncType(d.anon_return_type,
type_list_from_func_param_list d.anon_params)


let symbol_from_func_param p = match p with
   | FuncParamsDeclared(decl_specs, decl) ->
```

```
                    VarSymbol(var_name_from_direct_declarator decl,
                    type_from_func_param p)
   | AnonFuncDecl(d) ->
                          AnonFuncSymbol(Astutil.string_of_identifier  d.anon_decl_name,
type_from_anon_decl d)
    | _ -> raise(Failure("symbol_from_func_param not fully implemented. Cannot handle
declarations with parameters as types alone"))


let symbol_from_declaration decl = match decl with
    Declaration(declspec, _) -> VarSymbol(var_name_from_declaration decl,
    type_from_declaration decl)


let symbol_table_key_for_method struct_name func_name = let cstruct_name =
        String.concat "" ["_struct";struct_name] in String.concat "_"
[cstruct_name;func_name]


let symbol_from_fdecl fdecl =
        let func_name = var_name_from_direct_declarator fdecl.func_name
        in
        if (fdecl.receiver = ("", "")) then
            FuncSymbol(func_name, fdecl)
        else
            FuncSymbol((symbol_table_key_for_method (fst fdecl.receiver)
            func_name), fdecl)


let symbol_from_struct struct_decl = StructSymbol(struct_decl.struct_name,
struct_decl)


let symbols_from_structs struct_decls = List.map symbol_from_struct struct_decls


let symbol_from_interface interface = InterfaceSymbol(interface.name, interface)
```

```
let symbols_from_interfaces interfaces = List.map symbol_from_interface
interfaces

let lookup_symbol_by_id symbols id = try StringMap.find
(Astutil.string_of_identifier id) symbols with
        Not_found -> raise(Failure("undeclared identifier: " ^
        Astutil.string_of_identifier id))

let get_interface symbol_table name =
        let sym = lookup_symbol_by_id symbol_table (Identifier(name)) in
        match sym with
        | InterfaceSymbol(_, interface) -> interface
        | _ -> raise(Failure("cannot find interface"))

let is_interface symbol_table name =
        let sym = lookup_symbol_by_id symbol_table (Identifier(name)) in
        match sym with
        | InterfaceSymbol(_, interface) -> true
        | _ -> false

let func_decl_from_anon_def anonDef = {
    return_type = DeclSpecTypeSpecAny(anonDef.anon_return_type);
    func_name = DirectDeclarator(Var(Identifier("")));
    receiver = ("", "");
    params = anonDef.anon_params;
    body = anonDef.anon_body
}

let type_from_identifier symbols id =
```

```
        let x = lookup_symbol_by_id symbols id in match x with
        | VarSymbol(_, t) -> t
        | FuncSymbol(_, func) -> type_from_declaration_specifiers
        func.return_type
        | StructSymbol(_, struct_decl) -> CustomType(struct_decl.struct_name)
        | InterfaceSymbol(name, _) -> CustomType(name)
        | AnonFuncSymbol(_, t) -> t


let get_parameter_list symbol = match symbol with
        FuncSymbol(_, func) -> type_list_from_func_param_list func.params
      | AnonFuncSymbol(_, t) -> (match t with
              AnonFuncType(_, tlist) -> tlist
                    | _ -> raise(Failure("get_parameter_list: Error, invalid type for
Anonymous function")))
      | _ -> raise(Failure("Shouldn't get parameter list for Var Symbol"))


let get_struct symbol = match symbol with
        StructSymbol(_, struct_) -> struct_
      | _ -> raise(Failure("Attempting to get non struct symbol"))


let get_func symbol = match symbol with
        FuncSymbol(_, func) -> func
      | _ -> raise(Failure("Attempting to get non func symbol"))


let get_type_from_struct_member cust_type symbols t =
        let cust_symb = lookup_symbol_by_id symbols
                        (Identifier(cust_type)) in


        match cust_symb with
        | StructSymbol(_, struct_decl) ->
```

```
                let list_decl = List.map var_name_from_declaration struct_decl.members
in
              if (List.mem t list_decl) then
                  let dec = List.find (fun decl ->
                                        (var_name_from_declaration decl) = t)
                  struct_decl.members in  (type_from_declaration dec)
             else
                 raise(Failure("Invalid member of  struct"))
       | _ -> raise(Failure("Attemtping to get type for struct member for non struct
              member"))



let type_from_mem_access type_ t symbols  =
        match type_ with
                | CustomType(cust_type) -> get_type_from_struct_member cust_type
                symbols t

                | PointerType(CustomType(cust_type), 1) -> get_type_from_struct_member
                cust_type symbols t
                | _ -> raise(Failure("Non Custom type trying to access
                 member"))



let is_interface symbols id =
        let sym = lookup_symbol_by_id symbols id in match sym with
        | InterfaceSymbol(_, _) -> true
        | _ -> false

let rec t1_inherits_t2 t1 t2 symbols =
        let sym1 = (lookup_symbol_by_id symbols (Identifier(t1))) in
                match sym1 with
```

```
                | StructSymbol(type_, struct_) -> (let sym2 = (lookup_symbol_by_id
                        symbols (Identifier(t2))) in match sym2 with
                        | StructSymbol(type_2, struct2_) ->
                              if (struct_.extends <> "") then
                                      (if (struct_.extends <> struct2_.struct_name)


                                              then (t1_inherits_t2
                                              struct_.extends t2 symbols)
                                              else false
                                       ) else (if (type_ = type_2) then true
                              else false)
                         | _ -> false)
                | _ -> false


let rec get_interface_for_struct t1 symbols =
        let sym1 = (lookup_symbol_by_id symbols (Identifier(t1))) in
                match sym1 with
                | StructSymbol(typ_, struct_) -> (if (struct_.implements <> "") then
                                struct_.implements else (if (struct_.extends <> "")
                                  then get_interface_for_struct struct_.extends symbols
else
                                        ""))
               | _ -> raise(Failure("Not supported"))




let rec t1_implements_t2 t1 t2 symbols =
        let sym1 = (lookup_symbol_by_id symbols (Identifier(t1))) in
                match sym1 with
                | StructSymbol(typ_, struct_) -> (let sym2 =
                        (lookup_symbol_by_id symbols (Identifier(t2))) in match
```

```
                        sym2 with
                        | InterfaceSymbol(typ_, _) -> if (struct_.implements =
                                t2) then true else (if (struct_.extends <> "")
                                then t1_implements_t2 struct_.extends t2 symbols else
                                        false)
                        | _ -> raise(Failure("Not supported")))
                | _ -> raise(Failure("Not supported"))


let rec t1_inherits_t2 t1 t2 symbols =
        let sym1 = (lookup_symbol_by_id symbols (Identifier(t1))) in
                match sym1 with
                | StructSymbol(typ_, struct_) -> (let sym2 =
                        (lookup_symbol_by_id symbols (Identifier(t2))) in match
                        sym2 with
                        | StructSymbol(typ2_, struct2_) -> if (struct_.extends
                        <> "") then (if (struct_.extends = struct2_.struct_name)
                        then true else t1_inherits_t2 struct_.extends t2 symbols) else
                                false
                        | _ -> raise(Failure("cannot inherit from non struct
                                type"))


                        )
                | _ -> raise(Failure("Only struct types can inherit"))


let check_compatible_custom_types symbols t1 t2 =
        let t1_sym = lookup_symbol_by_id symbols t1 in
        let t2_sym  = lookup_symbol_by_id symbols t2 in
        match (t1_sym, t2_sym) with
        | (StructSymbol(t1_name, t1_struct), StructSymbol(t2_name, t2_struct))
        -> if (t1_name = t2_name) then () else (
```

```
                if (t1_inherits_t2 t2_name t1_name symbols) then () else
                raise(Failure("Incompatible types:" ^ t1_name ^ "," ^ t2_name)))
        | (StructSymbol(t1_name, t1_struct), InterfaceSymbol(name, _)) ->
                        raise(Failure("Incompatible types:" ^ t1_name ^ "," ^
                        name))
        | (InterfaceSymbol(name, _), StructSymbol(t2_name, t2_struct)) ->  if
                (t1_implements_t2 t2_name name symbols) then () else
                        raise(Failure("Incompatible types:" ^ t2_name ^ "," ^
                        name))
        | (_, _) -> raise(Failure("attempting to check compatible types for non
        custom types"))

(* This is meant to check assignments of custom type to pointer type
 * The only case this is valid is if a is pointer and b is interface
 * which a satisfies *)

let check_pointer_and_custom_types a b symbols =
     let sym_a = lookup_symbol_by_id symbols (Identifier(a)) in

     let sym_b = lookup_symbol_by_id symbols (Identifier(b)) in

     match (sym_a, sym_b) with
     | (StructSymbol(_, _), StructSymbol(_, _)) ->
             raise(Failure("Assigning:" ^ b ^"to" ^a ^ "which
                                is pointer type"))
     | (StructSymbol(strct, _), InterfaceSymbol(name, _)) -> if
                        (t1_implements_t2 strct name symbols) then () else
                                raise(Failure(strct ^ "does not implement" ^
                                name))
     | _ -> raise(Failure("Assigning incompatible custom types,
```

```
                pointer and non pointer"))



let rec check_compatible_anon_types symbols t1 t2 =

        let f a b =

             let error_str = "t1 = " ^ string_of_type t1 ^ ", t2 = " ^ string_of_type
t2

            in

              if a = b then () else raise(Failure("check_compatible_anon_types: Error,
param types not equal: " ^ error_str))  in

        let check_lists_are_equal l1 l2 = List.iter2 f l1 l2 in

        match (t1, t2) with

           (AnonFuncType(rType1, plist1), AnonFuncType(rType2, plist2)) ->

                  check_compatible_types symbols rType1 rType2;

                  check_lists_are_equal plist1 plist2

          | (_, _) -> raise(Failure("check_compatible_anon_types: Error, invalid anon
types passed as arguments"))


and check_compatible_types symbols t1 t2 = match (t1, t2) with

       (PrimitiveType(pt1), PrimitiveType(pt2)) -> (match pt1, pt2 with

       | Void, Void -> ()

       | Int, Float -> ()

       (*| Int, Float -> raise(Failure("assigning float to int"))*)

       | Float, Int -> ()

       (*| Float, Int -> raise(Failure("assigning int to float"))*)

       | String, Float -> raise(Failure("assigning float to string"))

       | String, Int -> raise(Failure("assigning int to string"))

       | Int, String -> raise(Failure("assigning string to int"))

       | Float, String -> raise(Failure("assigning string to float"))

       | Int, Int -> ()

       | Float, Float -> ()

       | String, String -> ()
```

```
      | _ -> raise(Failure("Incompatible types")))
  | (PointerType(typ1_, c1), PointerType(typ2_, c2)) ->
                        ignore(check_compatible_types symbols typ1_ typ2_);
                        if (c1 = c2) then () else raise(Failure("Incompatible
                        pointer depths " ^ (string_of_int c1) ^ " " ^
                        (string_of_int c2)))
  | (PrimitiveType(_), CustomType(_)) -> raise(Failure("Primitive type
  incompatible with custom type"))
  | (CustomType(_), PrimitiveType(_)) -> raise(Failure("Custom type incompatible
  with primitive type"))
    | (CustomType(a), CustomType(b)) -> check_compatible_custom_types symbols
(Identifier(a))
  (Identifier(b))
  | (PointerType(CustomType(a), 1), CustomType(b)) ->
                  check_pointer_and_custom_types a b symbols
  | (CustomType(a), PointerType(CustomType(b), 1)) -> if (t1_implements_t2 b a
  symbols) then () else raise(Failure("Incompatible types, pointer and custom type: "
  ^ b ^ " " ^ a))
  (*| (PointerType(_, _), PrimitiveType(_)) -> ()*)
  (*| (PrimitiveType(_), PointerType(_, _)) -> () *)
  | (PointerType(_, _), PrimitiveType(_)) -> raise(Failure("Cannot compare
  pointer and primitive"))
  | (PrimitiveType(_), PointerType(_, _)) -> raise(Failure("Cannot compare
  pointer and primitive"))
  | (PointerType(_, _), NilType) -> ()
  | (PrimitiveType(_), NilType) -> raise(Failure("Incompatible types: primitive
  and nil. "))
  | (CustomType(s), NilType) -> raise(Failure("Incompatible types: " ^ s ^ "
  and " ^ "nil"))
  | AnonFuncType(_, _), AnonFuncType(_, _) -> check_compatible_anon_types
  symbols t1 t2
```

```
   | (PrimitiveType(_), AnonFuncType(rtype, _)) -> check_compatible_types symbols t1
rtype

  | _ ->

          let t1Str = string_of_type t1 in

          let t2Str = string_of_type t2 in

           let errorStr = "check_compatible_types: Error - " ^ t1Str ^ " and " ^ t2Str
^ " not yet supported" in

          raise(Failure(errorStr))


let rec get_fdecl_for_receiver typ_ tSymbol_table func_name =

        let object_symbol = (lookup_symbol_by_id tSymbol_table (Identifier(typ_))) in

        let rec find_func symbol func_name = match symbol with

                                      | StructSymbol(type_, struct_) -> (let key =
symbol_table_key_for_method struct_.struct_name func_name in

                                    if (StringMap.mem key tSymbol_table) then

                                           (let sym = StringMap.find key

                                           tSymbol_table in (match sym with

                                           |FuncSymbol(_, fdecl) -> fdecl

                                           | _ -> raise(Failure("found a non func

                                           symbol matching call"))))

                                    else (

                                            if (struct_.extends <> "") then

                                                    get_fdecl_for_receiver

                                                    struct_.extends

                                                    tSymbol_table func_name

                                            else

                                                   raise(Failure("Receiver doesn't

                                                   have function"))))

                | InterfaceSymbol(type_, interface) -> (if (List.exists (fun

                           fdecl -> if (get_func_name fdecl = func_name) then true else
false)

                interface.funcs) then
```

```
                                (List.find (fun fdecl -> if(get_func_name fdecl =
                                        func_name) then true else false)
                                interface.funcs)
                                 else
                                        raise(Failure("Interface doesn't have
                                        function")))
                | _ -> raise(Failure("Cannot get method for non struct or
                interface")) in
        find_func object_symbol func_name



let type_of_array_type symbols = function
        | ArrayType(type_of_array, pointer, expr) -> (match type_of_array with
                                                        | PointerType(base, num)
                                                        -> PointerType(base,
                                                        num+1)
                                                                                | _ ->
PointerType(type_of_array,
                                                                                1))

        | _ -> raise(Failure("type_of_array_type should not be called on non
        array_type"))

let rec get_array_access_depth depth expr = match expr with
        | ArrayAccess(expr, _) -> 1 + (get_array_access_depth 1 expr)
        | _ -> 1

let rec type_from_array_access symbols expr = match expr with
  | ArrayAccess(e1, e2) -> type_from_array_access symbols e1
  | _ -> type_from_expr symbols expr
```

```
and type_from_expr symbols expr = match expr with
   Literal(_) -> PrimitiveType(Int)
  | FloatLiteral(_) -> PrimitiveType(Float)
  | StringLiteral(_) -> PrimitiveType(String)
  | Nil -> NilType
  | DeclExpr(e) -> type_from_declaration e
  | Neg(e) -> (let t = type_from_expr symbols e in match t with
                        | PrimitiveType(Int) -> PrimitiveType(Int)
                        | PrimitiveType(Float) -> PrimitiveType(Float)
                        | _ -> raise(Failure("Cannot take negative of type other
                        than int or float")))
  | Unop(e, _) -> type_from_expr symbols e
  | ArrayAccess(e1, e2) -> (ignore(let t2 = type_from_expr symbols e2 in match t2 with
                                    | PrimitiveType(Int) -> ()
                                    | _ -> raise(Failure("Index for array must be
                                    int primitive")));

                                    let t1 = type_from_array_access symbols expr in
                                    let depth = (get_array_access_depth 1 e1) in
                                    match (t1) with
                                    | PointerType(base, num) ->
                                            if (num < depth) then
                                            ( raise(Failure("Too deep array
                                            access"))) else ( if (depth = num)
                                            then (base) else (PointerType(base,
                                            num - depth)))


                                    | _ -> raise(Failure("Attempting array access
                                    for non pointer type")))
```

```
  | Binop(e1, _, e2) -> let t1 = type_from_expr symbols e1 in
                        let t2 = type_from_expr symbols e2 in
                        ignore (check_compatible_types symbols t1 t2);
                        type_from_expr symbols e1
  | Make(typ_, _) -> (match typ_ with
                        | ArrayType(type_of_array, pointer, expr) -> (
                                 ignore(let type_of_expr = type_from_expr symbols expr
in (
                                match type_of_expr with
                                | PrimitiveType(Int) -> ()
                                   | _ -> raise(Failure("Attempting to  allocate memory
of
                                non integral size"))
                                ));


                                type_of_array_type symbols typ_
                         )
                        | PointerType(base_type, count) -> ignore(Printf.printf
                        "%s" (string_of_int count));
                                        PointerType(base_type, count + 1)
                        | _ -> PointerType(typ_, 1))
  | Clean(e) -> ignore(let t1 = type_from_expr symbols e in match t1 with
                        | PointerType(base_type, count) -> ()
                                 | _ -> raise(Failure("Cannot clean a non pointer")));
PrimitiveType(Void)
  | Call(e, Id(Identifier(id)), _) -> (match e with
                | Noexpr -> (


                                if (StringMap.mem id symbols) then
                                        let sym = StringMap.find id symbols in
                                        match sym with
                                            FuncSymbol(_, fdecl) ->
```

```
                                                    type_from_declaration_specifiers
fdecl.return_type
                                        | AnonFuncSymbol(_, t) -> t
                                        | _ -> raise(Failure("Non function
                                                symbol associated with call"))
                                else
                                        raise(Failure("Calling function: " ^ id
                                        ^ "which is undefined"))
                                )
                        | _ ->  (
                                let typ_ = type_from_expr symbols e in
                                (match(typ_) with
                                | CustomType(name) -> (
                                let fdecl =
                                        get_fdecl_for_receiver name symbols id in
                                type_from_declaration_specifiers fdecl.return_type)
                                | PointerType(CustomType(name), 1) -> (let fdecl =
                                        get_fdecl_for_receiver name symbols id in
                                type_from_declaration_specifiers fdecl.return_type)
                                | _ -> raise(Failure("Invalid type making method
                                call")))))

  | CompareExpr(_, _, _) -> PrimitiveType(Int)
  | Postfix(e1, _) -> type_from_expr symbols e1
  | MemAccess(expr, Identifier(t)) -> let typ_ = type_from_expr symbols expr in
                                                type_from_mem_access typ_ t symbols
  | Id(id) -> type_from_identifier symbols id
  | AsnExpr(expr, _, _) -> type_from_expr symbols expr
  | Super(_) -> raise(Failure("Super defined outside of head of constructor"))
  | Deref(e) -> (let typ_ = type_from_expr symbols e in
                        match typ_ with
```

```
                        | PointerType(base_type, count) ->
                                        if (count = 1) then
                                                base_type
                                        else
                                            PointerType(base_type, count-1)
                                | _ -> raise(Failure("Dereferencing non pointer type")))
  | Pointify(e) -> (let typ_ = type_from_expr symbols e in
                                    match typ_ with
                                    | PointerType(base_type, count) ->
                                                    PointerType(base_type, count+1)
                                                    (* Because this isn't recursive
                                                     * we need to check if the
                                                     * existing type is a pointer
                                                     * and then add count to the
                                                     * existing pointer type *)
                                    | CustomType(s) -> if (is_interface symbols
                                    (Identifier(s))) then raise(Failure("Cannot make
                                    pointer out of interface")) else
                                            PointerType(typ_, 1)
                                    | _ -> (match e with
                                                    | Id(id) ->  PointerType(typ_, 1)
                                                            | _ -> raise(Failure("Cannot make
pointer out of non-identifier"))))
  | AnonFuncDef(adef) -> type_from_anon_def adef
  | Noexpr -> PrimitiveType(Void)

let rec type_list_from_expr_list symbols elist = match elist with
      [] -> []
    | [e] -> [type_from_expr symbols e]
    | h::t -> [type_from_expr symbols h]@(type_list_from_expr_list symbols t)
```

```
let receiver_has_func typ_ symbols func =
        let object_symbol = (lookup_symbol_by_id symbols (Identifier(typ_))) in
        let rec has_func symbol func = match symbol with
                | StructSymbol(type_, struct_) -> (match func.receiver with
                                (type2_, id) -> if (type2_ == type_) then () else
                                        if (struct_.extends <> "") then ( let
                                        parent = lookup_symbol_by_id symbols
                                                (Identifier(struct_.extends)) in has_func
parent func)
                                        else raise(Failure("method does not have
                                        receiver")))
                | InterfaceSymbol(type_, interface) -> (if (List.mem func
                interface.funcs) then () else raise(Failure("method isn't part
                of interface")))
                | _ -> raise(Failure("Receiver must be either struct or
                interface"))
        in

        has_func object_symbol func

let check_constructor symbols struct_name params =
                  let struct_symbol = if (StringMap.mem struct_name symbols) then
(lookup_symbol_by_id symbols
        (Identifier(struct_name))) else raise(Failure("Calling constructor for
        undeclared struct: " ^ struct_name)) in

        match struct_symbol with

        StructSymbol(typ_, struct_) ->
                let constructor = struct_.constructor in
                let param_list = constructor.constructor_params in
```

```
                if List.length param_list != List.length params then
                     if (constructor.constructor_name = "") then
                         raise(Failure("Parameters for constructor not defined"))
                else
                     List.iter2 (check_compatible_types symbols)
                     (type_list_from_func_param_list param_list)
                          (List.map (type_from_expr symbols) params)


       | _ -> raise(Failure("not handled\n"))



let rec check_compatible_type_lists symbols tl1 tl2 = match (tl1, tl2) with
    ([], []) -> ()
  | ([x], [y]) -> check_compatible_types symbols x y
  | (h1::t1, h2::t2) -> check_compatible_types symbols  h1 h2;
                        check_compatible_type_lists symbols t1 t2
  | _ -> raise(Failure("check_compatible_type_lists: type lists are incompatible"))


let validate_call_expr expr_list symbols params =
        let func_param_types = type_list_from_func_param_list params in
        let exprList = List.map (type_from_expr symbols) expr_list in
        List.iter2 (check_compatible_types symbols) exprList func_param_types


(* TODO: validate this *)
let rec validate_anon_call_expr expr expr_list symbols program anonSym = match anonSym
with
      AnonFuncSymbol(name, _) ->
          let anonDef = anon_def_from_tsymbol program anonSym in
```

```
          check_anon_body anonDef symbols program anonDef.anon_body

    | _ -> ()


and check_format_string_with_expr_list symbols fmtStr elist =

    let type_from_fmtSpec fmtSpec = match fmtSpec with

          "%s" -> PrimitiveType(String)

        | "%d" -> PrimitiveType(Int)

        | "%f" -> PrimitiveType(Float)

        | "%c" -> PrimitiveType(Char)

            | _ -> raise(Failure("check_format_string_with_expr_list: Error - Invalid
format string"))

    in

    let rec get_fmtSpec_list_from_string str =

        try

            let firstOccurrence = String.index str '%' in

            let substr = String.sub str firstOccurrence 2 in

               let remainderStr = String.sub str (firstOccurrence + 1) ((String.length
str) - (firstOccurrence + 1))  in

            [substr]@(get_fmtSpec_list_from_string remainderStr)

        with Not_found ->

            []

    in

    let rec type_list_from_fmtSpec_list specList = match specList with

          [] -> []

        | [s] -> [type_from_fmtSpec s]

        | h::t -> [type_from_fmtSpec h]@(type_list_from_fmtSpec_list t)

    in

    let fmtSpecList = get_fmtSpec_list_from_string fmtStr in

    let argTypeList = type_list_from_expr_list symbols elist in

    let fmtTypeList = type_list_from_fmtSpec_list fmtSpecList in

    try
```

```
          List.iter2 (fun t1 t2 -> (check_compatible_types symbols t1 t2)) argTypeList
fmtTypeList

    with

             Invalid_argument(_) -> raise(Failure("check_format_string_with_expr_list:
Error - Number of format specifiers in format string does not match number of
arguments"))

         | _ -> raise(Failure("check_format_string_with_expr_list: expression types do
not match format specifier list"))



and check_call_to_printf symbols exprList =

    match (List.rev exprList) with

    |  [] -> raise(Failure("Printf needs at least one argument"))

    |  [e] -> if ((type_from_expr symbols e) <> PrimitiveType(String)) then

                  raise(Failure("check_call_to_printf: Error - If only 1 argument, must
be string!"))

              else

                  ()

    | h::t ->

             if ((type_from_expr symbols h) <> PrimitiveType(String)) then

                  raise(Failure("check_call_to_printf: Error - If only 1 argument, must
be string!"))

              else (match h with

                      StringLiteral(s) ->

                          check_format_string_with_expr_list symbols s t

                    | _ ->

                                  let errorStr = "check_call_to_printf: Error - h is " ^
(Astutil.string_of_expr h) in

                        raise(Failure(errorStr)))



and is_literal expr = match expr with

| Literal(_) -> true

| StringLiteral(_) -> true

| FloatLiteral(_) -> true
```

```
| _ -> false

and check_expr symbols program e = match e with
     Id(Identifier(name)) -> if (StringMap.mem name symbols) == false then
                                raise(Failure("Undeclared identifier"))
                             else ()

  | Binop(e1, _, e2) -> let t1 = type_from_expr symbols e1 in
                         let t2 = type_from_expr symbols e2 in
                         check_compatible_types symbols t1 t2
   | Pointify(expr) -> ()
   | Neg(expr) -> ignore(type_from_expr symbols expr);

   | Literal(_) -> ()
   | StringLiteral(_) -> ()
   | FloatLiteral(_) -> ()
   | ArrayAccess(e1, e2) -> ignore(type_from_expr symbols e1); if (is_literal
   e1) then raise(Failure("Literal expression is not an array")) else ()
   | Deref(expr) -> (let t1 = type_from_expr symbols expr in
                        match t1 with
                        | PointerType(base_type, _) -> ()
                        | _ -> raise(Failure("Dereferencing non pointer")))
   | Super(_) -> raise(Failure("Super is not at the head of a constructor"))
   | Clean(expr) -> (let t1 = type_from_expr symbols expr in
                        match t1 with
                        | PointerType(base_type, num) -> ()
                        | _ -> raise(Failure("Cleaning a non pointer type")))
   | Call(expr, Id(Identifier(id)), expr_list) -> (match expr with
               | Noexpr ->
                        (if (StringMap.mem id symbols) then
```

```
                        let s =  StringMap.find id symbols in
                        match s with
                             FuncSymbol(_, fdecl) ->
                                 if (id = "printf") then
                                     check_call_to_printf symbols expr_list
                                 else
                                   validate_call_expr
                                   expr_list symbols
                                   fdecl.params
                            | AnonFuncSymbol(name, t) ->
                                     ()
                                           (*validate_anon_call_expr expr expr_list
symbols program s*)
                            | _ -> raise(Failure("Non function symbol cannot
                            make call"))
                       else
                           raise(Failure("Calling function: " ^ id
                           ^ "which is undefined"))
                      )
                | _ ->  (
                           let typ_ = type_from_expr symbols expr in
                           (match(typ_) with
                           | CustomType(name) -> (let fdecl =
                                  get_fdecl_for_receiver name symbols id in
                           validate_call_expr expr_list symbols
                           fdecl.params)
                           | PointerType(CustomType(name), 1) -> (let fdecl =
                                  get_fdecl_for_receiver name symbols id in
                                          validate_call_expr expr_list symbols
                                          fdecl.params)
                           | _ -> raise(Failure("Invalid type making method
```

```
                        call")))))
   | Unop(e, unop) -> check_expr symbols program e;
                      let te = type_from_expr symbols e in
                      (match (te, unop) with
                              (PrimitiveType(Void), _) -> raise(Failure("Cannot apply
unary operator to void type"))
                        | (PrimitiveType(_), _) -> ()
                        | _ -> raise(Failure("Type/Unary Operator mismatch")))
   | CompareExpr(e1, op, e2) -> (let t1 = type_from_expr symbols e1 in
                                 let t2 = type_from_expr symbols e2 in
                                 match (t1, t2) with
                                 | (CustomType(a), _) ->
                                                raise(Failure("Cannot
                                                compare custom types"))
                                 | (_, CustomType(s)) -> raise(Failure("Cannot
                                 compare custom types"))
                                 | _ -> check_compatible_types symbols
                                 (type_from_expr symbols e1) (type_from_expr
                                 symbols e2))
   | Postfix(e1, op) -> check_expr symbols program e1;
                      let te = type_from_expr symbols e1 in
                      (match (te, op) with
                          (PrimitiveType(Void), _) -> raise(Failure("Cannot
                          apply postfix operator to void type"))
                        | (PrimitiveType(_), _) -> ()
                        | _ -> raise(Failure("Type/Postfix Operator mismatch")))

   | MemAccess(s, Identifier(t)) -> ignore(let typ_ = type_from_expr symbols s in
                                           type_from_mem_access typ_ t
                                           symbols);
   | Make(typ_, expr_list) -> (match (typ_, expr_list)  with
```

```
                        | (PrimitiveType(s), [a]) -> ()
                        | (ArrayType(array_time, ptr, expr), []) ->
                                        ignore(type_from_expr symbols
                                        (Make(typ_, expr_list)));
                        | (CustomType(s), e) -> (check_constructor symbols s e)
                        | _ -> raise(Failure("Invalid make")))
   | AsnExpr(expr, asnOp, e) -> ignore(
                                    if (is_literal expr) then
                                            raise(Failure("Cannot assign to
                                            literal"))
                                    else ()
                                 );
                                let t1 = type_from_expr symbols e in
                                let t2 = type_from_expr symbols expr in
                                (match (t1, t2) with
                                    (PrimitiveType(Void), _) | (_, PrimitiveType(Void)) -
> raise(Failure("Cannot assign to type void"))
                                                        | (PrimitiveType(_), CustomType(_)) ->
raise(Failure("Cannot assign a struct to a primitive type"))
                                                        | (CustomType(_), PrimitiveType(_)) ->
raise(Failure("Cannot assign a primitive type to a struct"))
                                 | _ -> check_compatible_types symbols t2 t1)
   | Noexpr -> ()
   | _ -> raise(Failure("unmatched expression"))


and symbols_from_decls decls = List.map symbol_from_declaration decls


and symbols_from_func_params func_params = List.map symbol_from_func_param func_params


and symbol_from_receiver receiver = match receiver with
     | (type_, id) -> VarSymbol(id, PointerType(CustomType(type_), 1))
```

```
and type_from_receiver receiver = match receiver with
     | (typ_, _) -> typ_

and id_from_receiver receiver = match receiver with
     | (_, id) -> id

and compare_func_params symbols p1 p2 =
        let p1_types = List.map type_from_func_param p1 in
        let p2_types = List.map type_from_func_param p2 in
        if (List.length p1_types <> List.length p2_types) then
                raise(Failure("Func Param length mismatch")) else
        List.iter2 (check_compatible_types symbols) p1_types p2_types

and compare_func_names n1 n2 =
        let n1_name = var_name_from_direct_declarator n1 in
        let n2_name = var_name_from_direct_declarator n2 in
        if (n1_name = n2_name) then () else raise(Failure("Function
        Names do not match"))

and compare_func_return_types symbols r1 r2 =
       check_compatible_types symbols (type_from_declaration_specifiers r1)
  (type_from_declaration_specifiers r2)

and compare_functions symbols f1 f2 =
        let _ = compare_func_names f1.func_name f2.func_name in
        let _ = compare_func_params symbols f1.params f2.params in
        compare_func_return_types symbols f1.return_type f2.return_type

and symbols_from_fdecls fdecls = List.map symbol_from_fdecl fdecls
```

```
and get_id_from_symbol = function

     VarSymbol(id, _) -> id

   | FuncSymbol(id, _) -> id

   | StructSymbol(id, _) -> id

   | InterfaceSymbol(id, _) -> id

   | AnonFuncSymbol(id, t) -> id


and symtable_from_symlist symlist =

      List.fold_left (fun m symbol ->

          if (StringMap.mem (get_id_from_symbol symbol) m) then

              raise(Failure("symlist_to_symtable: Error - redefining variable"))

          else

                  StringMap.add (get_id_from_symbol symbol) symbol   m) StringMap.empty
symlist



(*let symbols_from_anon_def anonDef = *)

    (*let paramSymbols = symbols_from_func_params anonDef.anon_params in*)

    (*let bodySymbols = symbols_from_s anonDef.anon_body in*)

    (*symtable_from_symlist (paramSymbols@bodySymbols)*)


and merge_symtables s1 s2 =

    StringMap.merge (fun key v1 v2 ->

        (match v2, v2 with

           x, y -> if x != y then

                     raise(Failure("merge_symtables: Error - duplicate symbol"))

                   else x

        | None, y -> y

        | x, None -> x)) s1 s2


and get_decls_from_compound_stmt stmt = match stmt with
```

```
        CompoundStatement(x, y) -> x
   | _ -> []

and get_stmts_from_compound_stmt stmt = match stmt with
        CompoundStatement(x, y) -> y
   | _ -> []

and check_local_declaration symbols decl = match decl with
     Declaration(declspec, InitDeclList([InitDeclaratorAsn(declarator, asnOp,
     expr)])) ->

             let sym = symbol_from_declaration decl in
             (match sym with
                 VarSymbol(_, t1) -> let t2 = type_from_expr symbols expr in
                                     check_compatible_types symbols t1 t2
               | FuncSymbol(_, func) -> let t1 =
type_from_declaration_specifiers func.return_type in let
                        t2 =  type_from_expr symbols expr
               in check_compatible_types symbols t1 t2)
    | Declaration(declspec, InitDeclList([InitDeclarator(decl)])) -> ()
    | _ -> raise(Failure("check_local_declaration not supported"))


and check_bool_expr symbols expr = check_compatible_types symbols (type_from_expr
symbols
expr) (PrimitiveType(Int))

and add_to_symbol_table tbl decls =
    List.fold_left (fun m symbol ->
        if StringMap.mem (get_id_from_symbol symbol) m then
            raise(Failure("redefining variable: " ^ get_id_from_symbol symbol))
```

```
        else StringMap.add (get_id_from_symbol symbol) symbol m)
    tbl (symbols_from_decls decls)

and add_symbol_to_symbol_table tbl sym =
    if (StringMap.mem (Astutil.string_of_symbol_simple sym) tbl) then
       raise(Failure("redefining variable: " ^ get_id_from_symbol sym))
    else
       StringMap.add (Astutil.string_of_symbol_simple sym) sym tbl

and add_symbol_list_to_symbol_table tbl symlist = match symlist with
      [] -> tbl
    | [x] -> add_symbol_to_symbol_table tbl x
    | h::t -> let htbl = add_symbol_to_symbol_table tbl h in
              add_symbol_list_to_symbol_table htbl t

and check_statement func symbol_table program stmt = match stmt with
     Expr(e) -> (match e with
                       | Make(_, _) -> raise(Failure("Cannot have stand
                       alone make."))
                       | _ -> check_expr symbol_table program e)
    | Return(e) -> check_expr symbol_table program e; check_compatible_types
symbol_table (type_from_expr symbol_table e)
  (type_from_declaration_specifiers func.return_type)
  | If(e, s1, s2) -> check_bool_expr symbol_table e; check_statement
  func symbol_table program s1;  check_statement func symbol_table program s2
  | EmptyElse -> ()
  | For(e1, e2, e3, st) -> ((match (e1, e3) with
                                  | (Make(_, _), _) -> raise(Failure("Cannot have
                                  stand alone make"))
                                  | (_, Make(_, _)) -> raise(Failure("Cannot have
                                  stand alone make"))
```

```
                                | _ -> ());

                                   check_expr symbol_table program e2; check_bool_expr
symbol_table e2; check_statement

                                func symbol_table program st)

   | While(e, s) -> check_bool_expr symbol_table e; check_statement func symbol_table
program  s

  | CompoundStatement(dl, sl) -> let tbl = add_to_symbol_table symbol_table dl

   in List.iter (check_local_declaration tbl) dl; List.iter (check_statement func tbl
program ) sl

  | Break -> ()



and check_anon_body anonDef symbol_table program stmt = match stmt with

     Expr(e) -> (match e with

                        | Make(_, _) -> raise(Failure("Cannot have stand

                        alone make."))

                        | _ -> check_expr symbol_table program e)

  | Return(e) ->

        check_expr symbol_table program e;

                 check_compatible_types  symbol_table  (type_from_expr  symbol_table  e)
anonDef.anon_return_type

  | If(e, s1, s2) ->

        check_bool_expr symbol_table e;

        check_anon_body anonDef symbol_table program s1;

        check_anon_body anonDef symbol_table program s2

  | EmptyElse -> ()

  | For(e1, e2, e3, st) -> ( (match (e1, e3) with

                                | (Make(_, _), _) -> raise(Failure("Cannot have

                                stand alone make"))

                                | (_, Make(_, _)) -> raise(Failure("Cannot have

                                stand alone make"))

                                | _ -> ());

                                check_expr symbol_table program e2;
```

```
                                check_bool_expr symbol_table e2;
                                check_anon_body anonDef symbol_table program st)
  | While(e, s) ->
        check_bool_expr symbol_table e;
        check_anon_body anonDef symbol_table program  s
  | CompoundStatement(dl, sl) ->
        let tbl = add_to_symbol_table symbol_table dl in
        List.iter (check_local_declaration tbl) dl;
        List.iter (check_anon_body anonDef tbl program ) sl
  | Break -> ()

and func_decl_from_anon_func_def anonDef = {
    return_type = DeclSpecTypeSpecAny(anonDef.anon_return_type);
    func_name = DirectDeclarator(Var(Identifier("")));
    receiver = ("", "");
    params = anonDef.anon_params;
    body = anonDef.anon_body
}

and check_anon_func_def symbol_table program anonDef =
         check_statement  (func_decl_from_anon_func_def  anonDef)  symbol_table  program
anonDef.anon_body

(* This function checks 1) Is there a cycle in the inheritence tree and 2)
 * checks that all extensions are valid i.e. no extending oneself or a non
 * existenct struct *)

and validate_all_inheritence symbols structs =
        let validate_inheritence symbols strct =
                (* It could be that the struct inside of the symbols map is
                 * different from the struct we pass into this function is not
```

```
                 * the same as the one in our symbol table since the symbol
                 * table could be modified.
                 * *)
                      let StructSymbol(_, struct_) = StringMap.find strct.struct_name
symbols in
                (* Check to see if the parent struct is defined*)
                if (StringMap.mem struct_.extends symbols)
                then
                               let StructSymbol(name, parent_struct) = StringMap.find
struct_.extends
                        symbols in

                        (* Check if parent struct is a member of struct_'s
                         * children list. If it is then we have circular
                         * definition *)

                        if (List.mem parent_struct.struct_name struct_.children)
                        then
                                raise(Failure("Circular inheritence: " ^
                                parent_struct.struct_name ^ " extends " ^
                                struct_.struct_name ^ " but is also a parent of
                                " ^ struct_.struct_name))
                      else
                               (* We found our parent struct and are about to
                                * add ourselves and our children to its children
                                * list. Sanity check we aren't extending
                                * ourselves
                                *  *)
                               if (struct_.struct_name = name)

                               then
```

```
                                    raise(Failure("Struct: " ^ struct_.struct_name
^"
                                    cannot extend itself"))
                            else
                                    let list_to_add = struct_.children @
                                      [struct_.struct_name] in

                                    let updated_struct = {
                                            members = parent_struct.members;
                                                                    struct_name =
parent_struct.struct_name;
                                            extends = parent_struct.extends;
                                            methods = [];
                                            implements = parent_struct.implements;
                                               children = (parent_struct.children @
list_to_add);
                                                                    constructor =
parent_struct.constructor;
                                            destructor =
                                                    parent_struct.destructor;
                                    } in

                                    let new_symbol = StructSymbol(name,
                                    updated_struct) in

                                    StringMap.add name
                                    new_symbol symbols

                else
                        if (struct_.extends = "") then symbols else
                                raise(Failure("extending a struct that isn't
                                defined: " ^ struct_.extends))
```

```
                in

        List.fold_left validate_inheritence symbols structs

(* Assumes that the symbol table has been validated for
 * inheritence rules and duplicate entries *)

and get_parents symbols struct_ =
        if (struct_.extends <> "")

        then
                let StructSymbol(_, parent_struct) = StringMap.find
                struct_.extends symbols in

                [parent_struct] @ (get_parents symbols parent_struct)
       else
                []
(* This function gets the constructor of the closest
 * ancestor of struct_.  *)
and get_ancestors_constructor symbols struct_ =
        if (struct_.extends = "")

        then struct_.constructor

        else
                    let StructSymbol(_, parent_struct) = StringMap.find struct_.extends
symbols in
                        if (parent_struct.constructor.constructor_name = "")
                                then get_ancestors_constructor symbols parent_struct
                        else
                                parent_struct.constructor
```

```
and get_ancestors_destructor symbols struct_ =
        if (struct_.extends = "")

        then struct_.destructor

        else
                    let StructSymbol(_, parent_struct) = StringMap.find struct_.extends
symbols in
                         if (parent_struct.destructor.destructor_name = "")
                                 then get_ancestors_destructor symbols parent_struct
                         else
                                 parent_struct.destructor



and check_void_decl decl = match decl with
        | Declaration(decl_spec, _) ->
                         if (type_from_declaration_specifiers decl_spec =
                                 PrimitiveType(Void)) then
                                         raise(Failure("Invalid Declaration of
type Void. Trying to declare variable: " ^ var_name_from_declaration decl ^ " as
void"))
                         else
                                 ()
        | _ -> raise(Failure("Unhandled Declaration"))

and remove_duplicate_strings string_list =
        let str_map = List.fold_left (fun acc str_ -> StringMap.add str_ 1 acc)
        StringMap.empty string_list in

        StringMap.fold (fun str _ acc -> acc @ [str]) str_map []
```

```
and get_method_names_for_struct tSymbol_table struct_ =
       if (struct_.extends = "") then List.map (fun fdecl ->
               var_name_from_direct_declarator fdecl.func_name) struct_.methods
       else (
              let StructSymbol(_, parent_struct) = lookup_symbol_by_id
              tSymbol_table (Identifier(struct_.extends)) in List.rev (
              (List.map (fun fdecl -> var_name_from_direct_declarator
              fdecl.func_name) struct_.methods) @ (get_method_names_for_struct
              tSymbol_table
              parent_struct)))

and get_unique_method_names_for_struct tSymbol_table struct_ =
        remove_duplicate_strings (get_method_names_for_struct tSymbol_table struct_)

and check_child_methods_against_parent symbols child_methods parent_methods =
        let parent_method_map = List.fold_left (fun m parent_decl -> let
        parent_method = var_name_from_direct_declarator parent_decl.func_name in if
                (StringMap.mem parent_method m) then raise(Failure("Redeclaring
       method: " ^ parent_method)) else (StringMap.add parent_method parent_decl m))
        StringMap.empty parent_methods in

        List.map (fun child_method ->
                let func_name =
                        var_name_from_direct_declarator child_method.func_name in
        if (StringMap.mem func_name
        parent_method_map) then (let parent_method = StringMap.find
        func_name parent_method_map in
        try
        compare_functions symbols child_method
```

```
        parent_method
        with
        _ -> raise(Failure("Declared child method: " ^ func_name ^ " is
        incompatible with parent declaration"))
        ) else ()) child_methods

and update_fields functions symbols structs  =

        let _ = validate_all_inheritence symbols structs in

        let update_field functions symbols strct =
                let StructSymbol(_, struct_) = StringMap.find strct.struct_name
symbols in
                let strct_methods = List.filter (fun func -> if
                        (type_from_receiver func.receiver = strct.struct_name)
                        then true else
                                false) functions in
                if (struct_.extends = "")
                then
                        let updated_child_struct = {
                                struct_name = struct_.struct_name;
                                members = struct_.members;
                                children = struct_.children;
                                methods = strct_methods;
                                constructor = struct_.constructor;
                                destructor = struct_.destructor;
                                implements = struct_.implements;
                                extends = struct_.extends;
                        } in
                        StringMap.add struct_.struct_name
                        (StructSymbol(struct_.struct_name,
```

```
                updated_child_struct)) symbols
        else

                let parents = get_parents symbols struct_ in

                                        match parents with

                | [] -> symbols
                | _ -> List.fold_left (fun sym parent_struct ->

                                        let parent_methods =
                                                List.filter (fun func ->
                                                        if
(type_from_receiver
                                                                        func.receiver
                                                                        =
parent_struct.struct_name)
                                                                then true else
                                                                        false)
                                                functions in
                                                                        ignore
(check_child_methods_against_parent symbols
                                                strct_methods parent_methods);

                                        let StructSymbol(_,
                                        current_struct)  =
                                                StringMap.find
                                                strct.struct_name
                                                sym in
```

```
                                                let updated_child_struct = {
                                                        struct_name =
                                                                struct_.struct_name;
                                                        members =
                                                                parent_struct.members
                                                                @
current_struct.members;
                                                        children =
current_struct.children;
                                                        methods = strct_methods;
                                                        constructor = (if
(struct_.constructor.constructor_name
                                                                = "") then
(get_ancestors_constructor
                                                                  sym
                                                                  struct_)
                                                                else
struct_.constructor);
                                                        implements =
                                                                struct_.implements;
                                                        extends =
                                                                struct_.extends;
                                                        destructor =
                                                                struct_.destructor;
                                                } in
```

```
                                                StringMap.add
                                                struct_.struct_name
                                                (StructSymbol(struct_.struct_name,
                                                updated_child_struct))
                                                sym) symbols parents in

                List.fold_left (fun symbols struct_ -> (update_field functions
                symbols struct_)) symbols structs

(* Updates structs in the program object with the ones populated in symbol table *)
and update_structs_in_program program  =
        let fdecls = program.functions@stdlib_funcs
        in

            let symbol_table = update_fields program.functions (List.fold_left (fun m
symbol -> if StringMap.mem
                (get_id_from_symbol symbol) m then
                        raise(Failure("redefining identifier")) else StringMap.add
                        (get_id_from_symbol symbol) symbol m) StringMap.empty
                         (symbols_from_decls program.globals @
                         symbols_from_fdecls (List.filter (fun func -> if
                                 (type_from_receiver func.receiver = "") then true
                                          else false) fdecls) @ (symbols_from_structs
program.structs)
                         @ (symbols_from_interfaces program.interfaces)))
                        program.structs in


        let rec get_structs_from_symbols structs symbol_table =
                match structs with
                | [] -> []
```

```
                | h::t -> (let StructSymbol(_, struct_) = StringMap.find
                h.struct_name symbol_table in [struct_] @ get_structs_from_symbols t
                symbol_table)
        in

        let structs_ = get_structs_from_symbols program.structs symbol_table in

        {
                globals = program.globals;
                structs = structs_;
                interfaces = program.interfaces;
                functions = program.functions;
        }

and check_struct_fields struct_ =
         ignore (List.map check_void_decl struct_.members);

         List.fold_left (fun sym decl -> if
        (StringMap.mem (var_name_from_declaration decl)
        sym) then raise(Failure("Struct field: " ^
        (var_name_from_declaration (decl)) ^ " was redeclared")) else
                StringMap.add (var_name_from_declaration decl) decl sym)
        StringMap.empty struct_.members

and struct_implements_method struct_ symbol_table interface_method =
        let interface_method_name = var_name_from_direct_declarator
        interface_method.func_name in
        let fdecl = get_fdecl_for_receiver struct_.struct_name
        symbol_table interface_method_name in
```

```
        compare_functions symbol_table fdecl interface_method


and check_implements symbol_table struct_ =
        if (struct_.implements = "") then () else (
                let impl = struct_.implements in


                if (StringMap.mem impl symbol_table) then
                        let sym = StringMap.find impl symbol_table in
                        (match sym with
                        | InterfaceSymbol(_, interface) -> List.iter
                        (struct_implements_method struct_ symbol_table)
                        interface.funcs
                        | _ -> raise(Failure("Implementing a non-interface"))
                        )
                else raise(Failure("Implementing a non-existent interface"))


        )


and convert_constructor_to_fdecl constructor updated_body =
        {
                return_type = DeclSpecTypeSpecAny(PrimitiveType(Void));
                func_name =

DirectDeclarator(Var(Identifier(constructor.constructor_name)));
                body = updated_body;
                params = constructor.constructor_params;
                receiver = ("", "");
        }


and convert_destructor_to_fdecl destructor =
        {
```

```
                return_type = DeclSpecTypeSpecAny(PrimitiveType(Void));
                func_name =
                        DirectDeclarator(Var(Identifier(destructor.destructor_name)));
                body = destructor.destructor_body;
                params = [];
                receiver = ("", "");
        }

and isSuper stmt = match stmt with
        | Expr(Super(_)) -> true
        | _ -> false

and constructor_has_super constructor = match constructor.constructor_body with
        | CompoundStatement(d, s) -> (
                match s with
                | [] -> false
                | [singleton] -> isSuper singleton
                | h::t -> isSuper h)


and constructor_body_filtered_for_super body = match body with
        | CompoundStatement(d, s) -> (
                match s with
                | [] -> CompoundStatement(d, s)
                | [singleton] -> if (isSuper singleton) then
                        CompoundStatement(d, []) else CompoundStatement(d, s)
                | h::t -> if (isSuper h) then CompoundStatement(d, t) else
                        CompoundStatement(d, s)
        )
        | _ -> raise(Failure("No other constructor body"))
```

```
and getSuperExpr stmt = match stmt with
        | Expr(Super(e_list)) -> Expr(Super(e_list))
        | _ -> Expr(Noexpr)

and get_super_expr body = match body with
        | CompoundStatement(_, s) -> (
                match s with
                | [] -> Expr(Noexpr)
                | [singleton] -> getSuperExpr singleton
                | h::t -> getSuperExpr h)

and validate_super super_ symbol_table struct_ =
        let ancestor_constructor = get_ancestors_constructor symbol_table
        struct_ in

        if (ancestor_constructor.constructor_name =
                struct_.constructor.constructor_name) then
                        raise(Failure("Calling super when no parent constructor
                        is defined"))
        else
                match super_ with
                | Expr(Super(_)) -> let Expr(Super(elist)) = getSuperExpr super_ in
                validate_call_expr elist symbol_table
                     ancestor_constructor.constructor_params


and check_for_super_in_constructor symbols struct_ = match
struct_.constructor.constructor_body with
        | CompoundStatement(decls, stmts) -> (
```

```
                match (stmts) with
                | [] -> ()
                | [h] -> if (isSuper h) then validate_super h symbols
                struct_ else ()
                | h::t -> if (isSuper h) then validate_super h symbols
                struct_ else ()
        )
       | _ -> raise(Failure("Unexpected constructor body"))

and build_symbol_table program =
        let fdecls = program.functions @ stdlib_funcs in
        List.fold_left (fun m symbol -> if StringMap.mem
                (get_id_from_symbol symbol) m then
                                raise(Failure("redefining variable: " ^
                        get_id_from_symbol symbol)) else StringMap.add
                        (get_id_from_symbol symbol) symbol m) StringMap.empty
                          (symbols_from_decls program.globals
                         @ symbols_from_fdecls fdecls
                         @ (symbols_from_structs program.structs)
                         @ (symbols_from_interfaces program.interfaces))

and check_constructor_definition_in_struct program struct_ =
        ignore (List.iter (fun decl -> if (is_assignment_declaration decl) then
                raise(Failure("Cannot have assignment declaration in struct"))
        else ()) struct_.members);

        let symbol_table = build_symbol_table program in
        let constructor = struct_.constructor in
        let symbols =  List.fold_left (fun symbol_table sym -> if
        (StringMap.mem (get_id_from_symbol sym)
```

```
        symbol_table) then raise(Failure("Struct field: " ^
        (get_id_from_symbol sym) ^ " was redeclared")) else
                StringMap.add (get_id_from_symbol sym) sym symbol_table)
        symbol_table (symbols_from_decls struct_.members @
        symbols_from_func_params constructor.constructor_params) in

        if (constructor.constructor_name = "") then ()

        else
                ignore(check_for_super_in_constructor symbols struct_);
                let updated_body = constructor_body_filtered_for_super
                constructor.constructor_body in

                let func = convert_constructor_to_fdecl constructor
                updated_body
                in

                check_statement func
        symbols program updated_body

and check_destructor_definition program struct_ =
        let symbol_table = build_symbol_table program in
        let destructor = struct_.destructor in
        let symbols = List.fold_left (fun symbol_table sym -> if (StringMap.mem
        (get_id_from_symbol sym) symbol_table) then raise(Failure("Struct field: " ^
        (get_id_from_symbol sym) ^ "was redeclared")) else StringMap.add
        (get_id_from_symbol sym) sym symbol_table) symbol_table
        (symbols_from_decls struct_.members) in

        if (destructor.destructor_name = "") then ()
```

```
        else
                let func = convert_destructor_to_fdecl destructor in

                check_statement func symbols program destructor.destructor_body

and get_method_names struct_ = List.map (fun func -> var_name_from_direct_declarator func.func_name) struct_.methods

and apply_name_to_anon_def (prefix, count) adef = {
    anon_name =  prefix ^ "_" ^ (string_of_int count);
    anon_return_type = adef.anon_return_type;
    anon_params = adef.anon_params;
    anon_body = adef.anon_body;
}

and anon_defs_from_expr (prefix, count) expr = match expr with
          AnonFuncDef(anonDef) ->([(apply_name_to_anon_def (prefix, count) anonDef)], (count + 1))
   | Binop(e1, op, e2) ->
           let (defs1, count1) = (anon_defs_from_expr (prefix, count) e1) in
           let (defs2, count2) = (anon_defs_from_expr (prefix, count1) e2) in
           (defs1@defs2, count2)
   | AsnExpr(_, _, e) -> anon_defs_from_expr (prefix, count) e
   | Postfix(e1, _) -> (anon_defs_from_expr (prefix, count) e1)
   | Call(_, e, elist) ->
           let (defs1, count1) = (anon_defs_from_expr (prefix, count) e) in
           let (defs2, count2) = (anon_defs_from_expr_list (prefix, count1) elist) in
           (defs1@defs2, count2)
   | Make(_, elist) -> anon_defs_from_expr_list (prefix, count) elist
```

```
    | _ -> ([], count) (* Other expression types cannot possibly contain anonymous
function definitions *)


and anon_defs_from_expr_list (prefix, count) elist = match elist with

     [] -> ([], count)

   | [e] -> anon_defs_from_expr (prefix, count) e

   | h::t ->

           let (defs1, count1) = (anon_defs_from_expr (prefix, count) h) in

           let (defs2, count2) = (anon_defs_from_expr_list (prefix, count1) t) in

           (defs1@defs2, (count2))




and  anon_defs_from_declaration (prefix, count) decl = match decl with

         Declaration(declSpecs, initDecl) -> anon_defs_from_init_declarator (prefix,
count) initDecl


and anon_defs_from_declaration_list (prefix, count) declList = match declList with

     [] -> ([], count)

   | [d] -> anon_defs_from_declaration (prefix, count) d

   | h::t ->

           let (defs1, count1) = (anon_defs_from_declaration (prefix, count) h) in

            let (defs2, count2) = (anon_defs_from_declaration_list (prefix, count1) t)
in

           (defs1@defs2, count2)


and anon_defs_from_init_declarator (prefix, count) idecl = match idecl with

           InitDeclaratorAsn(_, _, e) -> anon_defs_from_expr (prefix, count) e

          | InitDeclList(initDeclList) -> anon_defs_from_init_declarator_list (prefix,
count) initDeclList

         | _ -> ([], count)
```

```
and anon_defs_from_init_declarator_list (prefix, count) ideclList = match ideclList
with

     [] -> ([], count)

   | [decl] -> anon_defs_from_init_declarator (prefix, count) decl

   | h::t ->

           let (defs1, count1) = (anon_defs_from_init_declarator (prefix, count) h) in

           let (defs2, count2) = (anon_defs_from_init_declarator_list (prefix, count1)
t) in

           (defs1@defs2, (count2))


and  anon_defs_from_statement (prefix, count) stmt = match stmt with

     Expr(e) -> anon_defs_from_expr (prefix, count) e

   | Return(e) -> anon_defs_from_expr (prefix, count) e

   | EmptyElse -> ([], 0)

   | If(e, s1, s2) ->

           let (defs1, count1) = (anon_defs_from_expr (prefix, count) e) in

           let (defs2, count2) = (anon_defs_from_statement (prefix, count1) s1) in

           let (defs3, count3) = (anon_defs_from_statement (prefix, count2) s2) in

           (defs1@defs2@defs3, count3)

   | For(e1, e2, e3, s) ->

           let (defs1, count1) = (anon_defs_from_expr (prefix, count) e1) in

           let (defs2, count2) = (anon_defs_from_expr (prefix, count1) e2) in

           let (defs3, count3) = (anon_defs_from_expr (prefix, count2) e3) in

           let (defs4, count4) = (anon_defs_from_statement (prefix, count3) s) in

           (defs1@defs2@defs3@defs4, count4)

   | While(e, s) ->

           let (defs1, count1) = (anon_defs_from_expr (prefix, count) e) in

           let (defs2, count2) = (anon_defs_from_statement (prefix, count1) s) in

           (defs1@defs2, count2)

   | CompoundStatement(declList, stmtList) ->

              let (defs1, count1) = (anon_defs_from_declaration_list (prefix, count)
declList) in
```

```
                let (defs2, count2) = (anon_defs_from_statement_list (prefix, count1)
stmtList) in
            (defs1@defs2, count2)

and anon_defs_from_statement_list (prefix, count) stmtList = match stmtList with
     [] -> ([], count)
   | [s] -> anon_defs_from_statement (prefix, count) s
   | h::t ->
            let (defs1, count1) = (anon_defs_from_statement (prefix, count) h) in
            let (defs2, count2) = (anon_defs_from_statement_list (prefix, count1) t) in
            (defs1@defs2, count2)



and anon_defs_from_func_decl (prefix, count) fdecl =
    let newPrefix =
        (match fdecl.func_name with
              DirectDeclarator(Var(Identifier(s))) -> "a_" ^ s
            | PointerDirDecl(_, Var(Identifier(s))) -> "a_" ^ s)
    in
    anon_defs_from_statement (newPrefix, 0) fdecl.body

and anon_defs_from_func_decl_list (prefix, count) fdlist = match fdlist with
      [] -> ([], count)
    | [x] -> anon_defs_from_func_decl (prefix, count) x
    | h::t ->
            let (defs1, count1) = (anon_defs_from_func_decl (prefix, count) h) in
            let (defs2, count2) = (anon_defs_from_func_decl_list (prefix, count1) t) in
            (defs1@defs2, count2)

and anon_defs_from_tprogram tprog =
```

```
          let (defs, _) = (anon_defs_from_func_decl_list ("_", 0) (List.rev
tprog.functions)) in

    List.rev defs


and anon_def_from_tsymbol tprogram tsym =

    match tsym with

        AnonFuncSymbol(s, _) ->

            let anonDefs = anon_defs_from_tprogram tprogram in

            let dummyDef = {

                anon_name = "PLACEHOLDER_ANON_DEF";

                anon_return_type = PrimitiveType(Void);

                anon_params = [];

                anon_body = CompoundStatement([], [])

            }

            in

            let (found, anonDef) = (List.fold_left

                (fun (isFound, foundDef) def ->

                    (match isFound with

                          true -> (isFound, foundDef)

                        | false ->

                            if (def.anon_name = s) then

                                (true, def)

                            else

                                (false, foundDef))) (false, dummyDef) anonDefs)

            in

            if (found = true) then

                anonDef

            else

                 let errorStr = "anon_def_from_tsymbol: Error - no anonDef with name "
^ s in

                raise(Failure(errorStr))
```

```
     | _ -> raise(Failure("Unexpected symbol type"))


(*and anon_defs_from_func_param_list tprogram *)
and compare_anon_defs_ignore_name a1 a2 =
   let b1 = {
       anon_name = "";
       anon_return_type = a1.anon_return_type;
       anon_params = a1.anon_params;
       anon_body = a1.anon_body
   }
   in
   let b2 = {
       anon_name = "";
       anon_return_type = a2.anon_return_type;
       anon_params = a2.anon_params;
       anon_body = a2.anon_body
   }
   in
   (b1 = b2)


and find_name_for_anon_def tprogram anonDef =
    let anonDefs = anon_defs_from_tprogram tprogram in
    let find_match (isFound, targetDef) def =
        if (isFound = true) then
            (isFound, targetDef) (* Leave alone *)
        else
            if ((compare_anon_defs_ignore_name targetDef def) = true) then
                (true, def)
            else
                (false, targetDef)
```

```
    in
    let (found, def) = List.fold_left find_match (false, anonDef) anonDefs in
    if (found = true) then
        def.anon_name
    else
            raise(Failure("find_name_for_anon_def: Error - could not find a matching
anonymous function definition"))



and find_func_containing_anon_def tprogram anonDef =
    let name = find_name_for_anon_def tprogram anonDef in
    let index_of_final_underscore = String.rindex name '_' in
    let fname = String.sub name 2 (index_of_final_underscore - 2) in
    let symtable = build_symbol_table tprogram in
    let interfaceSymbols = symbols_from_interfaces tprogram.interfaces in
    let interfaceMethodSymbols =
        let interfaces =
            List.map (fun (InterfaceSymbol(_, iface)) -> iface) interfaceSymbols
        in
        List.fold_left (fun accList iface ->
            accList@(symbols_from_fdecls iface.funcs)) [] interfaces
    in
       let updated_symbol_table = add_symbol_list_to_symtable interfaceMethodSymbols
symtable in
    let fsym = lookup_symbol_by_id updated_symbol_table (Identifier(fname)) in
    match fsym with
        FuncSymbol(_, fdecl) -> fdecl
         | _ -> raise(Failure("find_func_containing_anon_def: Error, incorrect symbol
type found"))


and find_symbol_containing_anon_def tprogram anonDef =
    let name = find_name_for_anon_def tprogram anonDef in
```

```
    let index_of_final_underscore = String.rindex name '_' in
    let fname = String.sub name 2 (index_of_final_underscore - 2) in
    let symtable = build_symbol_table tprogram in
    let interfaceSymbols = symbols_from_interfaces tprogram.interfaces in
    let interfaceMethodSymbols =
        let interfaces =
            List.map (fun (InterfaceSymbol(_, iface)) -> iface) interfaceSymbols
        in
        List.fold_left (fun accList iface ->
            accList@(symbols_from_fdecls iface.funcs)) [] interfaces
    in
       let updated_symbol_table = add_symbol_list_to_symtable interfaceMethodSymbols
symtable in
    lookup_symbol_by_id updated_symbol_table (Identifier(fname))



and find_struct_name_for_anon_def tprogram anonDef =
    let name = find_name_for_anon_def tprogram anonDef in
    "S" ^ name


and anon_defs_from_expr_list_no_recursion tprogram elist =
   List.fold_left (fun acc e ->
                        (match e with
                            AnonFuncDef(anonDef) ->
                                 let anonName = find_name_for_anon_def tprogram anonDef
in
                                let namedAnonDef = {
                                    anon_name = anonName;
                                    anon_return_type = anonDef.anon_return_type;
                                    anon_params = anonDef.anon_params;
                                    anon_body = anonDef.anon_body
```

```
                                }
                                in
                                acc@[namedAnonDef]
                            | _ -> acc)) [] elist


and expr_list_contains_anon_defs_no_recursion elist =
    let expr_contains_anon_def_at_this_level truthVal expr  =
        if (truthVal = true) then
            true
        else
            (match expr with
            AnonFuncDef(_) -> true
          | _ -> false)
    in
            List.fold_left expr_contains_anon_def_at_this_level false elist


and call_contains_anon_def call =
    let rec expr_contains_anon_def_at_this_level truthVal expr  =
        if (truthVal = true) then
            true
        else
            match expr with
            AnonFuncDef(_) -> true
          | _ -> false
    in
    match call with
        Call(_, _, elist) ->
            List.fold_left expr_contains_anon_def_at_this_level false elist
      | _ -> raise(Failure("call_contains_anon_def: Error - do not pass anything other
than a call expression to this function"))
```

```
   let rec expr_contains_anon_def symbols anonDef expr = match expr with
      |  AnonFuncDef(a) ->
            if (compare_anon_defs_ignore_name a anonDef) then
                (true, symbols)
            else
                (false, symbols)
      | Binop(e1, _, e2) ->
              let (found, newSyms) = (expr_contains_anon_def symbols anonDef e1) in
              if found then
                  (found, newSyms)
              else
                    let (found, newSyms) = (expr_contains_anon_def newSyms anonDef e2)
in
                  if found then
                      (true, newSyms)
                  else (false, symbols)
      | AsnExpr(e1, _, e2) ->
              let (found, newSyms) = (expr_contains_anon_def symbols anonDef e1) in
              if found then
                  (true, newSyms)
              else
                    let (found, newSyms) = (expr_contains_anon_def newSyms anonDef e2)
in
                  if found then
                      (true, newSyms)
                  else (false, symbols)
      | Literal(x) -> (false, symbols)
      | CompareExpr(e1, _, e2) ->
              let (found, newSyms) = (expr_contains_anon_def symbols anonDef e1) in
              if found then
                  (found, newSyms)
```

```
            else
                  let (found, newSyms) = (expr_contains_anon_def newSyms anonDef e2)
in
                if found then
                    (true, newSyms)
                else (false, symbols)
      | FloatLiteral(_) -> (false, symbols)
      | StringLiteral(_) -> (false, symbols)
      | Postfix(e, _) -> expr_contains_anon_def symbols anonDef e
      | Call(e1, e2, elist) ->
            let (found, newSyms) = (expr_contains_anon_def symbols anonDef e1) in
            if found then
                (true, newSyms)
            else
                  let (found, newSyms) = (expr_contains_anon_def newSyms anonDef e2)
in
                if found then
                    (true, newSyms)
                else
                        let (found, newSyms) = (expr_list_contains_anon_def newSyms
anonDef elist) in
                    if (found = true) then
                        (true, newSyms)
                    else
                        (false, symbols)
      | Make(_, elist) -> expr_list_contains_anon_def symbols anonDef elist
      | Clean(e) -> expr_contains_anon_def symbols anonDef e
      | Pointify(e) -> expr_contains_anon_def symbols anonDef e
      | Deref(e) -> expr_contains_anon_def symbols anonDef e
      | MemAccess(e, _) -> expr_contains_anon_def symbols anonDef e
      | Id(_) -> (false, symbols)
      | DeclExpr(decl) -> declaration_contains_anon_def symbols anonDef decl
```

```
      | Noexpr -> (false, symbols)

           | Unop(_, _) -> raise(Failure("expr_contains_anon_def: Error - unop not
supported"))

        | _ ->  raise(Failure("expr_contains_anon_def: Error - unexpected expression
type"))


   and expr_list_contains_anon_def symbols anonDef elist = match elist with

       [] -> (false, symbols)

      | [e] -> expr_contains_anon_def symbols anonDef e

      | h::t -> let (found, newSyms) = expr_contains_anon_def symbols anonDef h in

                if found then

                    (true, newSyms)

                else if

                     let (found, newSyms) = expr_list_contains_anon_def symbols anonDef
t in

                    found then (true, newSyms)

                else (false, symbols)


     and init_declarator_contains_anon_def symbols anonDef initDecl = match initDecl
with

        InitDeclaratorAsn(_, _, e) -> expr_contains_anon_def symbols anonDef e

       | InitDeclList(idlist) -> init_declarator_list_contains_anon_def symbols anonDef
idlist

      | InitDeclarator(_) -> (false, symbols)

          | _ -> raise(Failure("init_declarator_contains_anon_def: Error - unexpected
init_declarator type"))


     and init_declarator_list_contains_anon_def symbols anonDef initDeclList = match
initDeclList with

        [] -> (false, symbols)

      | [x] -> init_declarator_contains_anon_def symbols anonDef x

          | h::t -> let (found, newSyms) = init_declarator_contains_anon_def symbols
anonDef h in

                if found then
```

```
                    (true, newSyms)
                else
                        let (found, newSyms) = init_declarator_list_contains_anon_def
newSyms anonDef t in
                    if found then
                        (true, newSyms)
                    else
                        (false, symbols)

   and declaration_contains_anon_def symbols anonDef decl = match decl with
         Declaration(_, initDecl) -> init_declarator_contains_anon_def symbols anonDef
initDecl


    and declaration_list_contains_anon_def symbols anonDef declList = match declList
with
        [] -> (false, symbols)
      | [d] -> declaration_contains_anon_def symbols anonDef d
      | h::t ->
                let (found, newSyms) = declaration_contains_anon_def symbols anonDef h
in
              if found then
                  (true, newSyms)
              else
                      let (found, newSyms) = declaration_list_contains_anon_def symbols
anonDef t in
                  if found then
                      (true, newSyms)
                  else (false, symbols)

   and statement_contains_anon_def symbols anonDef stmt = match stmt with
        Expr(e) -> expr_contains_anon_def symbols anonDef e
      | Return(e) -> expr_contains_anon_def symbols anonDef e
```

```
      | If(e, s1, s2) -> let (found, newSyms) = expr_contains_anon_def symbols anonDef
e in
                        if found then
                            (true, newSyms)
                        else
                             let (found, newSyms) = statement_contains_anon_def newSyms
anonDef stmt in
                            if found then
                                (true, newSyms)
                            else (false, symbols)
      | CompoundStatement(declList, stmtList) ->
                             let (found, newSyms) = declaration_list_contains_anon_def
symbols anonDef declList in
                                                    if found then
                                                        (true, newSyms)
                                                    else
                                                                  let (found, newSyms) =
statement_list_contains_anon_def newSyms anonDef stmtList in
                                                        if found then
                                                            (true, newSyms)
                                                        else (false, symbols)


   and statement_list_contains_anon_def symbols anonDef stmtList = match stmtList with
        [] -> (false, symbols)
      | [s] -> statement_contains_anon_def symbols anonDef s
      | h::t ->
              let (found, newSyms) = statement_contains_anon_def symbols anonDef h in
              if found then
                  (true, newSyms)
              else
                       let (found, newSyms) = statement_list_contains_anon_def symbols
anonDef t in
                  if found then
```

```
                    (true, newSyms)
                else (false, symbols)

   and fdecl_contains_anon_def symbols anonDef fdecl =
       match fdecl.body with
            CompoundStatement(declList, stmtList) ->
               let psymbols = symbols_from_func_params fdecl.params in
               let bsymbols = symbols_from_decls declList in
                           let (found, newSyms) = declaration_list_contains_anon_def
(symbols@psymbols@bsymbols) anonDef declList in
               if (found = true) then
                  (true, newSyms)
               else
                              let (found, newSyms) = statement_list_contains_anon_def
(symbols@psymbols@bsymbols) anonDef stmtList in
                   if (found = true) then
                       (true, newSyms)
                   else (false, symbols)

   and fdecl_list_contains_anon_def symbols anonDef fdeclList = match fdeclList with
        [] -> (false, symbols)
      | [f] -> fdecl_contains_anon_def symbols anonDef f
      | h::t ->
              let (found, newSyms) = fdecl_contains_anon_def symbols anonDef h in
              if found then
                  (true, newSyms)
              else
                   let (found, newSyms) = fdecl_list_contains_anon_def symbols anonDef
t in
                  if found then
                      (true, newSyms)
                  else (false, symbols)
```

```
    and program_contains_anon_def anonDef program =
        let globals = symbols_from_decls program.globals in
        fdecl_list_contains_anon_def globals anonDef program.functions

and symbols_from_outside_scope_for_anon_def tprogram anonDef =

   let (found, symlist) = program_contains_anon_def anonDef tprogram in
   if (found = false) then
       raise(Failure("Error: program does not contain anonDef"))
   else
       symlist

and find_func_owning_anon_def tprogram anonDef =
    let dummyFuncDecl = {
        return_type = DeclSpecTypeSpecAny(PrimitiveType(Void));
        func_name = DirectDeclarator(Var(Identifier("")));
        receiver = ("", "");
        params = [];
        body = CompoundStatement([], [])
    }
    in
    (*let symbols = (build_symbol_table tprogram) in*)
    let (isFound, foundDecl) =
        List.fold_left (fun (isFound, foundDecl) fdecl ->
            match isFound with
                true ->
                    (*skip *)
                    (isFound, foundDecl)
              | false ->
```

```
                        let (isFound, _) = fdecl_contains_anon_def [] anonDef fdecl in

                        if (isFound = true) then

                            (true, fdecl)

                        else

                                                (false, foundDecl)) (false, dummyFuncDecl)
tprogram.functions

    in

    match isFound with

          true -> foundDecl

           | false -> raise(Failure("find_func_owning_anon_def: Error - could not find
function owning anonDef"))




and print_anon_def anonDef =

    Printf.printf "\n%s\n" (Astutil.string_of_anon_def anonDef)


and print_anon_defs = function

      [] -> ()

    | [x] -> print_anon_def x

    | h::t -> print_anon_def h; print_anon_defs t


let check_structs_satisfy_interfaces program =

        let symbol_table = build_symbol_table program in


        List.map (check_implements symbol_table) program.structs


let rec func_param_from_expr symbols expr =

    let te = type_from_expr symbols expr in

    ParamDeclWithType(DeclSpecTypeSpecAny(te))


and func_param_list_from_expr_list symbols expr_list = match expr_list with
```

```
      [] -> []
    | [e] -> [func_param_from_expr symbols e]
    | h::t -> [func_param_from_expr symbols h]@(func_param_list_from_expr_list symbols
t)

and funcs_with_receivers program =
    List.fold_left (fun accList fdecl ->
        let (rcv1, rcv2) = fdecl.receiver in
        if ((rcv1 <> "") ||  (rcv2 <> "")) then
            accList@[fdecl]
        else
            accList)[] program

and print_funcs_with_receivers program =
    let funcs_that_have_recvrs = funcs_with_receivers program in
    List.iter (fun f ->
        Printf.printf "%s\n\n" (Astutil.string_of_func f)) funcs_that_have_recvrs

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

       report_duplicate (fun a -> "duplicate for variable: " ^ a)  (sdecls);

       ignore (List.map check_void_decl program.globals);
```

```
              let fnames =    (List.map (fun func -> var_name_from_direct_declarator
func.func_name)
                                (List.filter (fun func -> if (type_from_receiver
                                                func.receiver = "") then true else false)
program.functions)) in
                                report_duplicate (fun a -> "duplicate functions:
                                        " ^ a) (fnames);


       let program = update_structs_in_program program
                       in ignore(List.map check_struct_fields
                       program.structs);


       ignore (List.map (check_constructor_definition_in_struct program)
        program.structs);


       ignore (List.map (check_destructor_definition program)
       program.structs);


       let struct_names = List.map (fun struct_ -> struct_.struct_name)
       program.structs in


       report_duplicate (fun a -> "duplicate structs: " ^ a) (struct_names);


       let check_duplicate_struct struct_ =
               report_duplicate (fun a -> "duplicate method: " ^ a)
       (get_method_names struct_) in


       List.map check_duplicate_struct (program.structs);


       let has_main = List.mem "main" fnames in
```

```
if has_main then () else raise(Failure("no function main declared"));

let is_printf_redefined = List.mem "printf" fnames in
if is_printf_redefined then raise(Failure("cannot redefine printf")) else
        ();

check_structs_satisfy_interfaces program;

let fdecls = program.functions @ stdlib_funcs in
(* Build map of function declarations *)
let functions_map = List.fold_left (fun m func -> StringMap.add
(var_name_from_direct_declarator func.func_name) func m) StringMap.empty
fdecls in

(* ****DEBUG ***** *)

 (*print_funcs_with_receivers program.functions;*)

(* **** END DEBUG ****)
let check_function func =

         let func_params = List.map var_name_from_func_param func.params
         in

         report_duplicate (fun a -> "duplicate function parameters: " ^
         a) func_params;

         param_list_has_void func.params (var_name_from_direct_declarator
         func.func_name);
```

```
                let local_decls = List.map var_name_from_declaration
                (get_decls_from_compound_stmt func.body) in

                report_duplicate (fun a -> "duplicate local variable: " ^ a)
                (local_decls);


               let symbol_table = List.fold_left (fun m symbol -> if StringMap.mem
                (get_id_from_symbol symbol) m then
                        raise(Failure("redefining variable: " ^
                        get_id_from_symbol symbol ^ " in function: " ^
                                (var_name_from_direct_declarator func.func_name))) else
StringMap.add
                        (get_id_from_symbol symbol) symbol m) StringMap.empty
                         (symbols_from_decls program.globals @
                         symbols_from_fdecls fdecls @
                         (symbols_from_func_params func.params) @ (symbols_from_decls
                         (get_decls_from_compound_stmt func.body))
                         @ (symbols_from_structs program.structs)
                         @ (symbols_from_interfaces program.interfaces)
                         @ ([symbol_from_receiver func.receiver]))

                in

                ignore(if (func.receiver <> ("", "")) then (if StringMap.mem (fst
                func.receiver) symbol_table then () else
                        raise(Failure("receiver: " ^ (fst func.receiver) ^ " is
                        not defined"))));

                List.iter (check_local_declaration symbol_table)
```

```
        (get_decls_from_compound_stmt func.body);

        List.iter (check_statement func symbol_table program )
        (get_stmts_from_compound_stmt func.body);

in List.iter check_function program.functions;
```

Ctree.ml

```
open Ast

module StringMap = Map.Make(String)

type cIdentifier = CIdentifier of string

type cPrimitive =
    Cvoid
  | Cchar
  | Cshort
  | Cint
  | Clong
  | Cfloat
  | Cdouble

type cProgram = {
        cstructs: cStruct list;
        cglobals: cDeclaration list;
        cfunctions: cFunc list;
}

and cStruct = {
  cstruct_name: string;
  cstruct_members: cSymbol list;
  cmethod_to_functions: cFunc StringMap.t;
}

and cFuncSignature = {
    func_return_type: cType;
```

```
    func_param_types: cType list;
}

and cFunc = {
  cfunc_name: string;
  cfunc_body: cStatement;
  cfunc_params: cFuncParam list;
  creturn_type: cType;
}

and cFuncDecl = {
  cfdecl_name: string;
  cfdecl_params: cType list;
  cfdecl_return_type: cType;
}

and cNonPointerType =
    CPrimitiveType of cPrimitive
  | CStruct of string

and cPointer =
    CPointer of cNonPointerType
  | CPointerPointer of cPointer

and cType =
    CType of cNonPointerType
  | CPointerType of cType * int
  | CFuncPointer of cFuncSignature

and cExpr =
    CBinop of cExpr * tOperator * cExpr
   | CAsnExpr of cExpr * tAssignmentOperator * cExpr
```

```
   | CLiteral of int
   | CFloatLiteral of float
   | CStringLiteral of string
   | CCastExpr of cType * cExpr
   | CPostfix of cExpr * tPostfixOperator
   | CCall of int * cExpr * cExpr * cExpr list (* The int field is a flag to
   indiciate it is a pointer dereference *)
   | CAlloc of cType * cExpr
   | CNeg of cExpr
   | CFree of cExpr
   | CDeref of cExpr
   | CArrayAccess of cExpr * cExpr
   | CCompareExpr of cExpr * tLogicalOperator * cExpr
   | CPointify of cExpr
   | CMemAccess of int * cExpr * cIdentifier (* The int field is a flag to
   indicate it is a pointer dereference *)
   | CId of cIdentifier
   | CDeclExpr of cDeclaration
   | CNoexpr
   | CNull

and cStatement =
    CExpr of cExpr
    | CEmptyElse
    | CReturn of cExpr
    | CCompoundStatement of cDeclaration list * cStatement list
    | CIf of cExpr * cStatement * cStatement
    | CFor of cExpr * cExpr * cExpr * cStatement
    | CWhile of cExpr * cStatement
    | CBreak

and cDirectDeclarator =
```

```
    CVar of cIdentifier

and cDeclarator =
    CDirectDeclarator of cDirectDeclarator

and cInitDeclarator =
    CInitDeclarator of cDeclarator
   |  CInitDeclaratorAsn of cDeclarator * tAssignmentOperator * cExpr

and cDeclarationSpecifiers =
    CDeclSpecTypeSpecAny of cType

and cFuncParam = cType * cIdentifier

and cDeclaration =
    CDeclaration of cDeclarationSpecifiers * cInitDeclarator

and cSymbol =
    CVarSymbol of string * cType
    | CFuncSymbol of string * cFunc
    | CStructSymbol of string * cStruct

let cStructName_from_tInterface name =
        String.concat "" ["_interface"; name]

let interface_field_name_in_struct interface_name struct_name =
        String.concat "" ["_"; (String.concat "_" [interface_name;struct_name])]

let cStructName_from_tStruct name =
        String.concat "" ["_struct"; name]

let virtual_table_name_from_tStruct name =
```

```
        String.concat "" ["_virtual";name]


let constructor_name_from_tStruct name =

        String.concat "_" ["_constructor";name]


let destructor_name_from_tStruct name =

        String.concat "_" ["_destructor";name]


let cType_from_tTypeSpec = function

    Void -> CType(CPrimitiveType(Cvoid))

  | Char -> CType(CPrimitiveType(Cchar))

  | Short -> CType(CPrimitiveType(Cshort))

  | Int -> CType(CPrimitiveType(Cint))

  | Long -> CType(CPrimitiveType(Clong))

  | Float -> CType(CPrimitiveType(Cfloat))

  | Double -> CType(CPrimitiveType(Cdouble))

  | Signed -> raise(Failure("cType_from_tTypeSpec: Error, Signed unsuported at the moment"))

  | Unsigned -> raise(Failure("cType_from_tTypeSpec: Error, Unsigned unsuported at the moment"))

  | String -> CPointerType(CType(CPrimitiveType(Cchar)), 1)

  (*| _ -> raise(Failure("cType_from_tTypeSpec: Error, unsupported tTypeSpec"))*)


let rec print_pointers n =

        if (n = 1) then "*" else

       ( String.concat "" ["*";(print_pointers (n-1))])


let rec sizeof_string tSymbol_table typ_ = match typ_ with

  | CType(CPrimitiveType(Cvoid)) -> "void"

  | CType(CPrimitiveType(Cchar)) -> "char"

  | CType(CPrimitiveType(Cint)) -> "int"

  | CType(CPrimitiveType(Clong)) -> "long"

  | CType(CPrimitiveType(Cfloat)) -> "float"

  | CType(CPrimitiveType(Cdouble)) -> "double"
```

```
  | CType(CStruct(t)) -> String.concat " " ["struct"; t]
  | CPointerType(base, n) -> String.concat " " [(sizeof_string tSymbol_table
  base); (print_pointers n)]

let rec cType_from_tType symbol_table = function
    PrimitiveType(typeSpec) -> cType_from_tTypeSpec typeSpec
  | PointerType(base_type, num) -> CPointerType(cType_from_tType symbol_table base_type,
  num)
  | ArrayType(array_type, ptr, e) -> (let t1 = Semant.type_of_array_type
  symbol_table (ArrayType(array_type, ptr, e)) in cType_from_tType symbol_table
  t1)
  | CustomType(s) -> (let sym = StringMap.find s symbol_table in
                                        match sym with
                                        | StructSymbol(name, _) ->
                                                    CType(CStruct(cStructName_from_tStruct
                                        name))
                                        | InterfaceSymbol(name, _) ->
CPointerType(CType(CStruct(cStructName_from_tInterface
                                        name)), 1))
  | AnonFuncType(t, tlist) ->
          let anonRetType = (cType_from_tType symbol_table t) in
          let anonParamTypes = List.map (fun x -> (cType_from_tType symbol_table x)) tlist in
          let captureParam = CPointerType(CType(CPrimitiveType(Cvoid)), 1) in
          CFuncPointer({
              func_return_type = anonRetType;
              func_param_types = anonParamTypes@[captureParam]
          })
  | _ -> raise(Failure("Haven't filled out yet"))


let id_exists_in_symtable symbols id =
```

```
  try
    StringMap.find (Astutil.string_of_identifier id) symbols;
    true
  with _ -> false


let id_exists_in_symlist symlist id =
  let check_sym_id_equal sym id =
    match sym with
      VarSymbol(name, _) -> if (name = (Astutil.string_of_identifier id)) then true else false
    | FuncSymbol(name, _) -> if (name = (Astutil.string_of_identifier id)) then true else false
     | AnonFuncSymbol(name, _) -> if ( name = (Astutil.string_of_identifier id)) then true else
false
  in
  let compare_symbol_with_id (id, (hasBeenFound, foundSymbol)) sym =
    match hasBeenFound with
      false -> if (check_sym_id_equal sym id) == true then (id, (true, sym))
               else (id, (hasBeenFound, foundSymbol))
    | true -> (id, (hasBeenFound, foundSymbol))
  in
    let (_, (isFound, foundSym)) = (List.fold_left compare_symbol_with_id (id, (false,
VarSymbol("ERROR_SYMBOL", PrimitiveType(Void)))) symlist) in isFound



let lookup_symbol_from_symlist_by_id symlist id =
  let check_sym_id_equal sym id =
    match sym with
      VarSymbol(name, _) -> name == (Astutil.string_of_identifier id)
    | FuncSymbol(name, _) -> name == (Astutil.string_of_identifier id)
    | AnonFuncSymbol(name, _) -> name == (Astutil.string_of_identifier id)
  in
  let compare_symbol_with_id (id, (hasBeenFound, foundSymbol)) sym =
    match hasBeenFound with
      false -> if (check_sym_id_equal sym id) == true then (id, (true, sym))
```

```
              else (id, (hasBeenFound, foundSymbol))

    | true -> (id, (hasBeenFound, foundSymbol))

  in

    match (List.fold_left compare_symbol_with_id (id, (false, VarSymbol("ERROR_SYMBOL",
PrimitiveType(Void)))) symlist) with

    (_, (true, foundSym)) -> foundSym

  | _ -> raise(Failure("lookup_symbol_from_symlist_by_id: Error, symbol not in table."))




let cDeclarationSpecifiers_from_tDeclarationSpecifiers symbol_table tDeclSpecs = function

        | DeclSpecTypeSpecAny(tType) ->

                        CDeclSpecTypeSpecAny(cType_from_tType symbol_table tType)


let cDeclaration_from_tFdecl symbol_table fdecl =

        let first_argument = [CPointerType(CType(CPrimitiveType(Cvoid)), 1)] in

        let cfunc_param_types = first_argument @ List.map (cType_from_tType symbol_table)

                                    (Semant.type_list_from_func_param_list fdecl.params) in

        let generate_void_star_param_types =

            let anonList =

                List.filter (fun p -> match p with

                    AnonFuncDecl(anonDecl) -> true

                  | _ -> false)fdecl.params

            in

            let returned_param_types =

                List.map (fun p -> match p with

                    AnonFuncDecl(anonDecl) ->

                        CPointerType(CType(CPrimitiveType(Cvoid)), 1)) anonList

            in

            returned_param_types

        in

        let extraParamTypes = generate_void_star_param_types in

        let cfunc_return_type =
```

```
                cType_from_tType symbol_table (Semant.type_from_declaration_specifiers

                fdecl.return_type) in

        let func_signature = {

                                func_return_type = cfunc_return_type;

                                func_param_types = cfunc_param_types@extraParamTypes;

                               } in


                    CVarSymbol((Semant.var_name_from_direct_declarator  fdecl.func_name),
CFuncPointer(func_signature))


(* The C Struct corresponding to the Cimple Interface consists of

 * 1) Function pointers instead of methods. The first argument is a void star *)


let cStruct_from_tInterface symbol_table interface =

        let cBodySymbol = [CVarSymbol("body",

        CPointerType(CType(CPrimitiveType(Cvoid)),

        1))] in (* This is the void * body that we apply to all the functions *)

        let cSymbols = List.map (cDeclaration_from_tFdecl symbol_table) interface.funcs in

        {

                cstruct_members = cBodySymbol @ cSymbols;

                cstruct_name = cStructName_from_tInterface interface.name;

                cmethod_to_functions = StringMap.empty

        }


let cSymbol_from_Implements implements =

        let cstruct_name = cStructName_from_tInterface implements in

        CVarSymbol(cstruct_name, CPointerType(CType(CStruct(cstruct_name)), 1))


let cFuncParam_from_tFuncParam symbol_table tFuncParam =

          (cType_from_tType  symbol_table  (Semant.type_from_func_param  tFuncParam),
(CIdentifier(Semant.var_name_from_func_param tFuncParam)))


let create_cfunc_param_for_receiver receiver =
```

```
        (CPointerType(CType(CPrimitiveType(Cvoid)), 1),
        CIdentifier("_body"))

 let create_initial_cast_decl receiver =
        let cstruct_name = cStructName_from_tStruct (fst receiver) in
        CDeclaration(CDeclSpecTypeSpecAny(CPointerType(CType(CStruct(cstruct_name)),
        1)), CInitDeclaratorAsn(CDirectDeclarator(CVar(CIdentifier(snd receiver)))
        , Asn, CCastExpr(CPointerType(CType(CStruct(cstruct_name)), 1),
        CId(CIdentifier("_body")))))

let number_of_anon_func_parameters_in_tFuncParamList plist =
    List.fold_left (fun acc f ->
                    (match f with
                       AnonFuncDecl(_) -> (acc + 1)
                     | _ -> acc)) 0 plist

let number_of_anon_func_parameters_in_tFuncDecl fdecl =
    number_of_anon_func_parameters_in_tFuncParamList fdecl.params

let cFunc_from_tFunc symbol_table tFunc =
    {
            creturn_type = cType_from_tType symbol_table
            (Semant.type_from_declaration_specifiers tFunc.return_type);

            cfunc_params = List.rev (List.map (cFuncParam_from_tFuncParam
            symbol_table) tFunc.params);

            cfunc_body = CCompoundStatement([], []);

            cfunc_name = Semant.var_name_from_direct_declarator
            tFunc.func_name;
    }
```

```
let rec cSymbol_from_sSymbol symbol_table sym = match sym with

    VarSymbol(s, t) -> CVarSymbol(s, (cType_from_tType symbol_table t))

  | FuncSymbol(s, fdecl) -> CFuncSymbol(s, (cFunc_from_tFunc symbol_table fdecl))

  | StructSymbol(s, strct) ->

          let (newStrct, _) = cStruct_from_tStruct symbol_table strct in

          CStructSymbol(s, newStrct)

  | _ -> raise(Failure("Not completed"))




 (*------------------- Function struct_members_from_anon_body-------------------------

 * This function returns a list of Ast.sSymbols representing the variables referenced within the
body of

 * an anonymous function that are declared outside of it's scope. This list will form the data

 * members of a special c struct that will be passed to a normal c function whenevever

 * an anonymous function in cimple is instantiated.

 *

 * Parameters:

         * symbols: A StringMap of symbols from outside the scope of the anonymous function def

         * psymbols: A symbol table of parameters to this anonymous function

         * members: A list of function parameters declared in the anon function definition

            * body: An Ast.tStatement (specifically a CompoundStatement) that is the body of the
anonymous function

 *------------------------------------------------------------------------------- *)


and struct_members_from_anon_body symbols psymbols members body =


  let rec print_member m = Printf.printf "%s\n" (Astutil.string_of_symbol m)


  and print_member_list mlist = match mlist with

        [] -> ()

      | [x] -> print_member x

      | h::t -> print_member h; print_member_list t
```

```
  in


  let symbol_is_capturable = function
        VarSymbol(_, _) -> true
      | _ -> false
  in


  let rec members_from_expr symbols psymbols members e = match e with
      Id(id) ->
          if (id_exists_in_symtable psymbols id) then []
          else if (id_exists_in_symlist members id) = true then
             []
          else if (id_exists_in_symtable symbols id) = true then
                  let sym = Semant.lookup_symbol_by_id symbols id in
                     if (symbol_is_capturable sym) then
                         [sym]
                     else
                         []
                (*else if (id_exists_in_symtable psymbols id) then []*)
                else (match id with
                  Identifier(s) ->
                      print_member_list members;
                      raise(Failure("members_from_expr: Error - undeclared symbol '" ^ s ^ "'")))
   |  Binop(e1, _, e2) -> let e1Members = members_from_expr symbols psymbols members e1 in
                            let e2Members = members_from_expr symbols psymbols (members@e1Members)
e2 in
                           e1Members@e2Members
   |  AsnExpr(e1, _, e2) -> let e1Members = members_from_expr symbols psymbols members e1 in
                                                let e2Members = members_from_expr symbols psymbols
(members@e1Members) e2 in
                            e1Members@e2Members
   |  Postfix(e1, _) -> let e1Members = members_from_expr symbols psymbols members e1 in
                            e1Members
```

```
   |  Call(_, e, elist) -> let eMembers = members_from_expr symbols psymbols members e in
                                        let elistMembers = members_from_expr_list symbols psymbols
(members@eMembers) elist in
                              eMembers@elistMembers
   | Make(_, elist) -> members_from_expr_list symbols psymbols members elist
   | Pointify(e) -> members_from_expr symbols psymbols members e
   | MemAccess(e, id2) -> let id1Members = members_from_expr symbols psymbols members e in
                                                 let id2Members = members_from_expr symbols psymbols
(members@id1Members) e in
                               id1Members@id2Members
   | AnonFuncDef(def) -> raise(Failure("members_from_expr: Error - nested anonymous functions not
supported"))
   | DeclExpr(decl) -> members_from_declaration symbols psymbols members decl
   | StringLiteral(_) -> []
   | _  -> []

   and members_from_expr_list symbols psymbols members elist = match elist with
     [] -> []
   | [x] -> members_from_expr symbols psymbols members x
   | h::t -> let hMembers = members_from_expr symbols psymbols members h in
             let tMembers = members_from_expr_list symbols psymbols (members@hMembers) t in
             hMembers@tMembers

   and members_from_init_declarator symbols psymbols members initDecl =
    match initDecl with
        InitDeclaratorAsn(_, _, e) -> members_from_expr symbols psymbols members e
     | InitDeclList(l) -> members_from_init_declarator_list symbols psymbols members l
     | _ -> []

   and members_from_init_declarator_list symbols psymbols members declList =
    match declList with
      (*[] -> members_from_expr symbols psymbols members Noexpr*)
      [] -> []
```

```
    | [x] -> members_from_init_declarator symbols psymbols members x
    | h::t -> let hmembers = members_from_init_declarator symbols psymbols members h in
              hmembers@(members_from_init_declarator_list symbols psymbols (members@hmembers) t)

  and members_from_declaration symbols psymbols members decl = match decl with
    Declaration(_, initDecl) ->
        members_from_init_declarator symbols psymbols members initDecl
    (*| _ -> [] [> Other types of declarations wouldn't reference variables from outside scope
<]*)

  and members_from_declaration_list symbols psymbols members declList = match declList with
     [] -> []
  | [x] -> (members_from_declaration symbols psymbols members x)
  | h::t -> let hmembers = members_from_declaration symbols psymbols members h in
            hmembers@(members_from_declaration_list symbols psymbols (members@hmembers) t)

  and members_from_statement_list symbols psymbols members stmtList = match stmtList with
  [] -> []
  | [x] -> (members_from_statement symbols psymbols members x)
  | h::t -> let hmembers = members_from_statement symbols psymbols members h in
            (hmembers)@(members_from_statement_list symbols psymbols (members@hmembers) t)

  and members_from_statement symbols psymbols members stmt = match stmt with
    CompoundStatement(decls, stmtList) ->
      let dmembers = (members_from_declaration_list symbols psymbols members decls) in
      dmembers@members_from_statement_list symbols psymbols (members@dmembers) stmtList
  | Expr(e) -> members_from_expr symbols psymbols members e
  | Return(e) -> members_from_expr symbols psymbols members e
  | If(e, s1, s2) -> let eMembers = members_from_expr symbols psymbols members e in
                     let s1Members = members_from_statement symbols psymbols (members@eMembers)
s1 in
                                      let s2Members = members_from_statement symbols psymbols
(members@eMembers@s1Members) s2 in
```

```
                    eMembers@s1Members@s2Members

   | For(e1, e2, e3, s) -> let e1Members = members_from_expr symbols psymbols members e1 in

                          let e2Members = members_from_expr symbols psymbols (members@e1Members)
e2 in

                                                  let e3Members = members_from_expr symbols psymbols
(members@e1Members@e2Members) e3 in

                                              let sMembers = members_from_statement symbols psymbols
(members@e1Members@e2Members@e3Members) s in

                          e1Members@e2Members@e3Members@sMembers

   | While(e, s) -> let eMembers = members_from_expr symbols psymbols members e in

                    let sMembers = members_from_statement symbols psymbols (members@eMembers) s
in

                  eMembers@sMembers

   | _ -> []

  in

  let mems = members_from_statement symbols psymbols members body in

  mems


(* --------------------------Function capture_struct_from_anon_def ------------------------
 * Returns a C struct to be used as a copy of the variables used within the body of an

 * anonymous function that were declared outside of its scope.

 *

 * Parameters:

         * program: an Ast.tProgram.

         * def: The Ast.tAnonFuncDef whose body we are looking through to find captured variables

 * ------------------------------------------------------------------------*)


and capture_struct_from_anon_def program def =

  let func = Semant.find_func_containing_anon_def program def in

  let receiverSymbols = [Semant.symbol_from_receiver func.receiver] in

  let interfaceSymbols = Semant.symbols_from_interfaces program.interfaces in

  let interfaceMethodSymbols =

      let interfaces =

          List.map (fun (InterfaceSymbol(_, iface)) -> iface) interfaceSymbols
```

```
      in
      List.fold_left (fun accList iface ->
          accList@(Semant.symbols_from_fdecls iface.funcs)) [] interfaces
  in
  (*(Astutil.print_symbol_table (Semant.symtable_from_symlist interfaceMethodSymbols));*)
  let extraSymbols = receiverSymbols@interfaceSymbols@interfaceMethodSymbols in
  let symlist = (Semant.symbols_from_outside_scope_for_anon_def program def)@extraSymbols in
  let symbols = Semant.symtable_from_symlist symlist in
  let builtinDecls = Semant.stdlib_funcs in
  let builtinSyms = Semant.symbols_from_fdecls builtinDecls in
  let rec symconvert m = cSymbol_from_sSymbol symbols m in
  let internal_anon_symbols = (fun stmt -> match stmt with
                CompoundStatement(declList, _) ->
                    Semant.symbols_from_decls declList) def.anon_body in
  let param_symbols = (Semant.symbols_from_func_params def.anon_params)@internal_anon_symbols in
  let param_symtable = (Semant.symtable_from_symlist param_symbols) in
  let updated_symbols = Semant.symtable_from_symlist (builtinSyms@symlist) in
    {
      cstruct_name = "S" ^ def.anon_name; (* 's' for 'struct' *)
          cstruct_members = (List.map symconvert (struct_members_from_anon_body updated_symbols
param_symtable [] def.anon_body));
      cmethod_to_functions = StringMap.empty
    }



and capture_struct_list_from_anon_def_list program defList = match defList with
      [] -> []
    | [x] -> [capture_struct_from_anon_def program x]
       | h::t -> [capture_struct_from_anon_def program h]@capture_struct_list_from_anon_def_list
program t


and cFunc_from_tMethod cStruct_Name tFuncName = String.concat "_" [cStruct_Name;tFuncName]
```

```
and cStruct_from_tStruct symbol_table tStruct =

        let symconvert m = cSymbol_from_sSymbol symbol_table m in

                    let defaultStructMemberSymbols = List.map symconvert (List.map
(Semant.symbol_from_declaration) tStruct.members) in


        (* If there is an interface then add a struct member corresponding to

         * the interface to our struct *)

        let cStructMemberSymbols = if (Semant.get_interface_for_struct

        tStruct.struct_name symbol_table <> "") then

                [cSymbol_from_Implements tStruct.implements] @

                defaultStructMemberSymbols else defaultStructMemberSymbols in


                    let (methods_to_cfunctions, cfuncs) = (List.fold_left (fun (sym, cfunc_list)
method_ ->

                                            (let tfunc_name =

                                                    Semant.var_name_from_direct_declarator

                                               method_.func_name in


                                           let initial_void_param =

                                                        create_cfunc_param_for_receiver

                                                        method_.receiver in


                                           let init_cast_decl =

                                                    create_initial_cast_decl

                                                    method_.receiver in



                                            let cfunc = {

                                                 creturn_type = (cType_from_tType

                                                 symbol_table

                                                 (Semant.type_from_declaration_specifiers

                                                 method_.return_type));
```

```
                                        cfunc_params =
                                                [initial_void_param] @ (List.map
                                        (cFuncParam_from_tFuncParam
                                        symbol_table) method_.params);

                                        cfunc_body =
                                                CCompoundStatement([init_cast_decl],
                                        []);

                                        cfunc_name = cFunc_from_tMethod
                                        (cStructName_from_tStruct
                                        tStruct.struct_name) tfunc_name;

                                    } in (StringMap.add tfunc_name cfunc
                                    sym, cfunc_list @ [cfunc])))
                                       (StringMap.empty, []) tStruct.methods) in

        ({
                cstruct_name = cStructName_from_tStruct tStruct.struct_name;
                cstruct_members = cStructMemberSymbols;
                cmethod_to_functions = methods_to_cfunctions;
        }, cfuncs)

let cDeclarationSpecifiers_from_tDeclarationSpecifiers symbol_table tDeclSpecs = function
        | DeclSpecTypeSpecAny(tType) ->
                        CDeclSpecTypeSpecAny(cType_from_tType symbol_table tType)

let cDeclaration_from_tFdecl symbol_table fdecl =
        let first_argument = [CPointerType(CType(CPrimitiveType(Cvoid)), 1)] in
        let cfunc_param_types = first_argument @ List.map (cType_from_tType symbol_table)
                                    (Semant.type_list_from_func_param_list fdecl.params) in
        let generate_void_star_param_types =
```

```
            let anonList =
                List.filter (fun p -> match p with
                    AnonFuncDecl(anonDecl) -> true
                  | _ -> false)fdecl.params
            in
            let returned_param_types =
                List.map (fun p -> match p with
                    AnonFuncDecl(anonDecl) ->
                        CPointerType(CType(CPrimitiveType(Cvoid)), 1)) anonList
            in
            returned_param_types
        in
        let extraParamTypes = generate_void_star_param_types in
        let cfunc_return_type =
                cType_from_tType symbol_table (Semant.type_from_declaration_specifiers
                fdecl.return_type) in
        let func_signature = {
                                func_return_type = cfunc_return_type;
                                func_param_types = cfunc_param_types@extraParamTypes;
                               } in

                     CVarSymbol((Semant.var_name_from_direct_declarator  fdecl.func_name),
CFuncPointer(func_signature))

(* The C Struct corresponding to the Cimple Interface consists of
 * 1) Function pointers instead of methods. The first argument is a void star *)

let cStruct_from_tInterface symbol_table interface =
        let cBodySymbol = [CVarSymbol("body",
        CPointerType(CType(CPrimitiveType(Cvoid)),
        1))] in (* This is the void * body that we apply to all the functions *)
        let bols = List.map (cDeclaration_from_tFdecl symbol_table) interface.funcs in
```

```
        {
                cstruct_members = cBodySymbol @ bols;
                cstruct_name = cStructName_from_tInterface interface.name;
                cmethod_to_functions = StringMap.empty
        }


let bol_from_Implements implements struct_name =
        let cstruct_name = cStructName_from_tInterface implements in
        let interface_field_name = interface_field_name_in_struct implements
        struct_name in
        CVarSymbol(interface_field_name, CType(CStruct(cstruct_name)))


let cFuncParam_from_tFuncParam symbol_table tFuncParam =
    match tFuncParam with
       AnonFuncDecl(_) ->
           let newName = "anon_" ^ (Semant.var_name_from_func_param tFuncParam) in
                        (cType_from_tType  symbol_table  (Semant.type_from_func_param  tFuncParam),
(CIdentifier(newName)))
     | _ ->
                     (cType_from_tType  symbol_table  (Semant.type_from_func_param  tFuncParam),
(CIdentifier(Semant.var_name_from_func_param tFuncParam)))


let create_cfunc_param_for_receiver receiver =
        (CPointerType(CType(CPrimitiveType(Cvoid)), 1),
        CIdentifier("_body"))



let create_initial_cast_decl receiver =
        let cstruct_name = cStructName_from_tStruct (fst receiver) in
        CDeclaration(CDeclSpecTypeSpecAny(CPointerType(CType(CStruct(cstruct_name)),
        1)), CInitDeclaratorAsn(CDirectDeclarator(CVar(CIdentifier(snd receiver)))
        , Asn, CCastExpr(CPointerType(CType(CStruct(cstruct_name)), 1),
        CId(CIdentifier("_body")))))
```

```
let cFunc_from_tFunc symbol_table tFunc =

    let generate_n_void_star_params n =

        let anonList =

            List.filter (fun p -> match p with

                AnonFuncDecl(anonDecl) -> true

              | _ -> false) tFunc.params

        in

        let returned_params =

            List.map (fun p -> match p with

                AnonFuncDecl(anonDecl) ->

                    (match anonDecl.anon_decl_name with

                        Identifier(s) ->

                            let paramName = "cap_anon_" ^ s in

                                                    (CPointerType(CType(CPrimitiveType(Cvoid)), 1),
CIdentifier(paramName)))) anonList

        in

        returned_params

    in

    let n = number_of_anon_func_parameters_in_tFuncDecl tFunc in

    let extraParams = generate_n_void_star_params n in

        {

                creturn_type = cType_from_tType symbol_table

                (Semant.type_from_declaration_specifiers tFunc.return_type);


                cfunc_params =List.rev (((extraParams)@List.map (cFuncParam_from_tFuncParam

                symbol_table) tFunc.params));


                cfunc_body = CCompoundStatement([], []);


                cfunc_name = Semant.var_name_from_direct_declarator tFunc.func_name;

        }
```

```
let cFunc_from_tMethod tStructName tFuncName =
        Semant.symbol_table_key_for_method tStructName tFuncName

let first_param_for_constructor struct_name =
        (CPointerType(CType(CStruct(cStructName_from_tStruct struct_name)), 2),
        CIdentifier("_this"))

let first_param_for_destructor struct_name =
        (CPointerType(CType(CStruct(cStructName_from_tStruct struct_name)), 2),
        CIdentifier("_this"))

let last_param_for_constructor =
        (CType(CPrimitiveType(Cint)), CIdentifier("_needs_malloc"))

let last_param_for_destructor =
        (CType(CPrimitiveType(Cint)), CIdentifier("_needs_free"))



and cFunction_from_tMethod object_type method_ tSymbol_table =
        match object_type  with
        | CustomType(name) -> ( let typ_symbol  =
                Semant.lookup_symbol_by_id tSymbol_table (Identifier(name)) in match
                typ_symbol with
                | StructSymbol(_, _) -> cFunc_from_tMethod
                (name) method_
                | InterfaceSymbol(_, _) -> method_)
        | PointerType(CustomType(name), 1) -> ( let typ_symbol  =
                Semant.lookup_symbol_by_id tSymbol_table (Identifier(name)) in match
                typ_symbol with
                | StructSymbol(_, _) -> cFunc_from_tMethod
                ( name) method_
```

```
                | InterfaceSymbol(_, _) -> method_)
        | _ -> raise(Failure("not done"))
(*
 * This function takes a cimple expression and returns the pair ((C expression,
 * statement[]), cDeclaration[]). The idea is that some cimple expression may generate multiple
 * assignment statements such as a make expression with a constructor.
 * Expressions should not create more declarations, only declarations create
 * more declarations.
 *)
let rec update_expr texpr tSymbol_table tprogram  = match texpr with
        | Neg(e) -> (let ((updated_e1, e1_stmts), decls) = update_expr e
                        tSymbol_table tprogram in ((CNeg(updated_e1), e1_stmts), decls))
        | Binop(e1, op, e2) -> (let ((updated_e1, e1_stmts), _) = update_expr e1
                                tSymbol_table tprogram in
                        let ((updated_e2, e2_stmts), _) = update_expr e2
                        tSymbol_table  tprogram in

                        ((CBinop(updated_e1, op, updated_e2), (e1_stmts@e2_stmts)), []))
        | CompareExpr(e1, op, e2) ->
                (
                let ((updated_e1, e1_stmts), _) = update_expr e1 tSymbol_table tprogram in
                let ((updated_e2, e2_stmts), _) = update_expr e2 tSymbol_table tprogram in
                ((CCompareExpr(updated_e1, op, updated_e2), (e1_stmts @ e2_stmts)), [])
                )
        | Clean(e) -> (let t1 = Semant.type_from_expr tSymbol_table e in
                        let ((updated_e, e_stmts), decls) = update_expr e tSymbol_table tprogram in
                        match t1 with
                        | PointerType(CustomType(s), _) -> ((CCall(0, CNoexpr,
                                CId(CIdentifier(destructor_name_from_tStruct s)),
[CCastExpr(CPointerType(CType(CStruct(cStructName_from_tStruct
                                        s)), 2), CPointify(updated_e))] @
```

```
                        [CLiteral(1)]), []), [])


                    | PointerType(_, _) -> ((CFree(updated_e), e_stmts),
                    decls))



        | AsnExpr(e1, op, e2)  ->
                (match e2 with
                            | Make(typ_, expr_list) -> cAllocExpr_from_tMakeExpr tSymbol_table
tprogram e1 e2
                        | _ -> (
                            let ((updated_e1, e1_stmts), _) = update_expr e1
                                    tSymbol_table tprogram in
                            let ((updated_e2, e2_stmts), _) = update_expr e2
                            tSymbol_table tprogram  in

                            let e1_type = Semant.type_from_expr tSymbol_table e1 in

                            let e2_type = Semant.type_from_expr tSymbol_table e2 in

                            (match (e1_type, e2_type) with

                            (* Check if we are assigning custom types. Since we are
                             * past semantic analysis the only possibilities are 1.
                             * we are assining a derived class to its ancestor or 2.
                             * we are assigning the same types. In those cases we
                             * need to cast *)

                            | (PointerType(CustomType(s), _),
                            PointerType(CustomType(t), _)) ->
                                (if (Semant.t1_inherits_t2 s t tSymbol_table) then
                                    ((CAsnExpr(updated_e1, op,
```

```
                                                CCastExpr((cType_from_tType tSymbol_table e1_type),
updated_e2)),

                                    (e1_stmts@e2_stmts)), [])


                                else

                                        ((CAsnExpr(updated_e1, op, updated_e2), e1_stmts

                                        @ e2_stmts), []))

                            | (CustomType(s), CustomType(t)) -> (if

                                    (Semant.t1_inherits_t2 s t tSymbol_table) then

                                    ((CAsnExpr(updated_e1, op,

                                    CCastExpr((cType_from_tType tSymbol_table

                                    e1_type), updated_e2)), (e1_stmts@e2_stmts)), [])


                                    else

                                        ((CAsnExpr(updated_e1, op, updated_e2), e1_stmts

                                        @ e2_stmts), []))


                            | _ -> ((CAsnExpr(updated_e1, op, updated_e2), e1_stmts

                            @ e2_stmts), [])))))


     | Call(expr, Id(Identifier(s)), expr_list) ->

                        let ret = cCallExpr_from_tCallExpr expr tSymbol_table tprogram s expr_list
in

                    ret

     | Super(_) -> ((CNoexpr, []), [])

     | ArrayAccess(e1, e2) -> ((CArrayAccess(fst(fst(update_expr e1 tSymbol_table

     tprogram)), fst(fst(update_expr e2 tSymbol_table tprogram))), []), [])

     | MemAccess(expr, Identifier(s)) -> (let typ_ = Semant.type_from_expr

     tSymbol_table expr in match (typ_) with

                | CustomType(name) -> ((CMemAccess(0, fst( fst (update_expr expr

                tSymbol_table tprogram )), CIdentifier(s)), []), [])

                | PointerType(CustomType(name), 1) -> ((CMemAccess(1, fst(fst

                (update_expr expr tSymbol_table tprogram )), CIdentifier(s)),
```

```
                []), [])
                | _ -> raise(Failure("Bad Mem Access")))
     | Id(Identifier(s)) -> ((CId(CIdentifier(s)), []), [])
     | Literal(d) -> ((CLiteral(d), []), [])
     | Make(typ_, expr_list) -> (let ctype = cType_from_tType tSymbol_table typ_
                 in match typ_ with
                | PrimitiveType(s) -> ((CAlloc(ctype,
                CId(CIdentifier((sizeof_string tSymbol_table ctype)))), []), [])
                | ArrayType(array_type, ptr, e) ->(let updated_e = fst(fst(update_expr
                                                e tSymbol_table tprogram)) in let pointer_type =
Semant.type_of_array_type
       tSymbol_table typ_ in let cpointer_type = cType_from_tType tSymbol_table
       pointer_type in (match cpointer_type with
       | CPointerType(base, num) ->let ctype_to_malloc = if (num = 1) then base
       else CPointerType(base, num-1) in (
       CAlloc(base, CBinop(updated_e, Mul,
       (CId(CIdentifier(sizeof_string tSymbol_table ctype_to_malloc)))))), []),
       []))

                | CustomType(s) -> ((CAlloc(ctype, CId(CIdentifier(sizeof_string tSymbol_table
               ctype))), []), []))
     | FloatLiteral(d) -> ((CFloatLiteral(d), []), [])
     | StringLiteral(s) -> ((CStringLiteral(s), []), [])
     | Postfix(e1, op) -> (let ((updated_e1, e1_stmts), _) = update_expr e1
                               tSymbol_table  tprogram in ((CPostfix(updated_e1, op), e1_stmts),
[]))
     | AnonFuncDef(anonDef) ->
             let anon_name = Semant.find_name_for_anon_def tprogram anonDef in
             let instanceName = "s" ^ anon_name in
             let structName = "S" ^ anon_name in
                     let decls = [CDeclaration(CDeclSpecTypeSpecAny(CType(CStruct(structName))),
CInitDeclarator(CDirectDeclarator(CVar(CIdentifier(instanceName)))))] in
             let assignments_from_capture_struct c =
```

```
                List.map (fun csym ->

                    (match csym with

                        CVarSymbol(s, _) ->

                                    CExpr(CAsnExpr(CMemAccess(0, CId(CIdentifier(instanceName)),
CIdentifier(s)), Asn, CId(CIdentifier(s))))

                                | _ -> raise(Failure("update_expr: Invalid CSymbol parameter"))))
c.cstruct_members

            in


            let captures = capture_struct_from_anon_def tprogram anonDef in


            let newAssignments = assignments_from_capture_struct captures in

            ((CId(CIdentifier(anon_name)), newAssignments), decls)

    | Noexpr -> ((CNoexpr, []), [])

    | Pointify(e) -> let ((updated_e, stmts), decls) = update_expr e

    tSymbol_table tprogram in ((CPointify(updated_e), stmts), decls)

    | Deref(e) -> let ((updated_e, stmts), decls) = update_expr e tSymbol_table

    tprogram in ((CDeref(updated_e), stmts), decls)

    | Nil -> ((CNull, []), [])

    | _ ->

            let expr_type = Astutil.string_of_expr texpr in

            raise(Failure("not finished for type " ^ expr_type))


and update_expr_list texpr_list tSymbol_table tprogram = match texpr_list with

    [] -> []

  | [e] -> [update_expr e tSymbol_table tprogram]

  | h::t -> [update_expr h tSymbol_table tprogram]@update_expr_list t tSymbol_table tprogram


and generate_stmts_for_parent_destructor symbol_table tprogram destructor

tStruct =


        let ancestor_destructor = Semant.get_ancestors_destructor symbol_table

        tStruct in
```

```
        let c_ancestor_destructor_name = destructor_name_from_tStruct
        ancestor_destructor.destructor_name in

        let first_arg = CCastExpr(CPointerType(cType_from_tType symbol_table
        (CustomType(ancestor_destructor.destructor_name)), 2),
        CId(CIdentifier("_this"))) in

        let last_arg = CLiteral(0) in

        [(CExpr(CCall(0, CNoexpr,
        CId(CIdentifier(c_ancestor_destructor_name)), [first_arg]@[last_arg])))]

and generate_stmts_for_super expr_list symbol_table tprogram constructor tStruct =
        let ancestor_constructor = Semant.get_ancestors_constructor
        symbol_table tStruct in

        let c_ancestor_constructor_name = constructor_name_from_tStruct
        ancestor_constructor.constructor_name in

        let first_arg = CCastExpr(CPointerType(cType_from_tType symbol_table
        (CustomType(ancestor_constructor.constructor_name)), 2), CId(CIdentifier("_this")))
        in

        let last_arg = CLiteral(0) in

        let cstParams = constructor.constructor_params in
        let funcParamSymbols = Semant.symbols_from_func_params cstParams in
        let anonParamSymbols = List.filter (fun sym ->
            (match sym with
                AnonFuncSymbol(_, _) -> true
              | _ -> false)) funcParamSymbols
```

```
in
let anonParamSymbolTable = Semant.symtable_from_symlist anonParamSymbols in
let is_in_symtable id =
    try
        StringMap.mem id anonParamSymbolTable
    with
        _ -> false
in
let fixedExprList =
    List.map (fun e ->
        (match e with
            Id(Identifier(id)) ->
                if ((is_in_symtable id) = true) then
                    Id(Identifier("anon_" ^ id))
                else
                    e)) expr_list
in
let capIds =
    List.fold_left (fun accList expr ->
        (match expr with
            Id(Identifier(id)) ->
                if ((is_in_symtable id) = true) then
                    accList@[CId(CIdentifier("cap_anon_" ^ id))]
                else
                    accList
          | _ -> raise(Failure("Invalid expr")))) []  expr_list
in
let fixedAnonParamSymbols =
    let fix_anon_name str =
            try
                let sub = String.sub str 0 5 in
                if (sub <> "anon_") then
```

```
                            "anon_" ^ str

                        else

                            str

                    with

                        _ -> "anon_" ^ str

            in

            List.map (fun sym ->

                (match sym with

                    AnonFuncSymbol(anonName, t) -> AnonFuncSymbol((fix_anon_name anonName), t)

                  | _ -> sym)) anonParamSymbols

        in

                     let  symbol_table  =  Semant.add_symbol_list_to_symbol_table  symbol_table
fixedAnonParamSymbols in

        let call_to_super_constructor_stmt = [(CExpr(CCall(0, CNoexpr, CId(CIdentifier(

                c_ancestor_constructor_name)), [first_arg]@(List.map2

                                (cExpr_from_tExpr_in_tCall symbol_table tprogram) fixedExprList

                                ancestor_constructor.constructor_params)@[last_arg]@capIds)))] in

                 let  StructSymbol(_,  ancestor_struct)  =  Semant.lookup_symbol_by_id  symbol_table
(Identifier(ancestor_constructor.constructor_name)) in


            let local_reassignment_of_members = List.fold_left (fun assignments

            member -> let member_id = Semant.var_name_from_declaration member

            in assignments @ [CExpr(CAsnExpr(CId(CIdentifier(member_id)), Asn, CMemAccess(1,

            CDeref(CId(CIdentifier("_this"))),

            (CIdentifier(member_id))))])]) [] ancestor_struct.members in


            call_to_super_constructor_stmt @ local_reassignment_of_members




and cAllocExpr_from_tMakeExpr tSymbol_table tprogram asn_expr tMakeExpr =

        let ((updated_e1, updated_stmts), _) = (update_expr asn_expr tSymbol_table tprogram ) in

        let Make(typ_, expr_list) = tMakeExpr in

        match typ_ with
```

```
        | CustomType(typ) -> (
                let StructSymbol(name, tStruct) = Semant.lookup_symbol_by_id
                tSymbol_table (Identifier(typ)) in

                if (tStruct.constructor.constructor_name <> "") then (
                        (* We have a constructor *)
                        let params = tStruct.constructor.constructor_params in

                                                                let more_params_filtered =
generate_extra_capture_func_params_from_expr_list tSymbol_table tprogram expr_list in
                        let updated_expr_list = (List.map2
                        (cExpr_from_tExpr_in_tCall tSymbol_table tprogram) expr_list params) in

                           let anonParams = Semant.anon_defs_from_expr_list_no_recursion tprogram
expr_list in
                        let update_anon_def_expr_list anonList =
                            List.fold_left (fun ((e, slist), dlist) def ->
                                        let ((_, _slist), _dlist) = update_expr (AnonFuncDef(def))
tSymbol_table tprogram in
                                        ((Noexpr, slist@_slist), dlist@_dlist)) ((Noexpr, []), [])
anonList
                        in
                                             let ((updated_expr, updated_slist), updated_dlist) =
update_anon_def_expr_list anonParams in

                       ((CCall(0, CNoexpr,
                        CId(CIdentifier(constructor_name_from_tStruct
                        tStruct.struct_name)),
                        [CCastExpr(CPointerType(CType(CStruct(cStructName_from_tStruct
                        tStruct.struct_name)), 2), CPointify(updated_e1))] @
                          updated_expr_list @ [CLiteral(1)]@more_params_filtered), updated_slist),
updated_dlist)
                ) else (
                       ((CCall(0, CNoexpr,
                        CId(CIdentifier(constructor_name_from_tStruct
```

```
                        tStruct.struct_name)),
                        [CCastExpr(CPointerType(CType(CStruct(cStructName_from_tStruct
                        tStruct.struct_name)), 2), CPointify(updated_e1))] @
                        [CLiteral(1)]), []), [])
                ))
        | (ArrayType(array_type, ptr, e)) -> (let updated_e = fst(fst(update_expr
                                            e tSymbol_table tprogram)) in let pointer_type =
Semant.type_of_array_type
        tSymbol_table typ_ in let cpointer_type = cType_from_tType tSymbol_table
        pointer_type in (match cpointer_type with
        | CPointerType(base, num) ->let ctype_to_malloc = if (num = 1) then base
        else CPointerType(base, num-1) in ((CAsnExpr(updated_e1, Asn,
        CAlloc(base, CBinop(updated_e, Mul,
        (CId(CIdentifier(sizeof_string tSymbol_table ctype_to_malloc)))))), []),
        [])))



and cExpr_from_tExpr_in_tCall tSymbol_table  tprogram  tExpr tFuncParam =
      let expr_type = Semant.type_from_expr tSymbol_table tExpr in
      let param_type = Semant.type_from_func_param tFuncParam in
        match (expr_type, param_type) with
            |  (CustomType(a), CustomType(b)) ->
                    if (Semant.is_interface tSymbol_table (Identifier(b))) then
                                CPointify(CMemAccess(0, fst( fst (update_expr tExpr tSymbol_table
tprogram )),
                            CIdentifier(interface_field_name_in_struct b a)))
                    else ( if (Semant.t1_inherits_t2 a b tSymbol_table) then
                        CCastExpr(CType(CStruct(cStructName_from_tStruct b)),
                                        fst (fst (update_expr tExpr tSymbol_table  tprogram )))
                    else fst (fst (update_expr tExpr tSymbol_table  tprogram )))
            | (PointerType(CustomType(a), 1),
            CustomType(b)) ->  CPointify(CMemAccess(1, fst( fst (update_expr tExpr
            tSymbol_table  tprogram )),
```

```
            CIdentifier(interface_field_name_in_struct b a)))
            | _ -> fst (fst (update_expr tExpr tSymbol_table tprogram )
            )


and generate_extra_capture_func_params_from_expr_list tSym tprogram expr_list =
        let anonParams = Semant.anon_defs_from_expr_list_no_recursion tprogram expr_list in
        let capture_struct_instance_name_from_anon_def def =
            let capStruct = capture_struct_from_anon_def tprogram def in
                            let subname = String.sub capStruct.cstruct_name 1 ((String.length
capStruct.cstruct_name) - 1) in
            let structname = "s" ^ subname in
            structname
        in
        let capture_params_from_anon_def_list defList =
            List.fold_left (fun elist def ->
                    elist@[CPointify(CId(CIdentifier((capture_struct_instance_name_from_anon_def
def))))]
            ) [CNoexpr] defList
        in
        let more_params = capture_params_from_anon_def_list anonParams in
        let remove_noexpr_from_list elist =
            List.filter (fun e ->
                match e with
                    CNoexpr -> false
                  | _ -> true) elist
        in
        let more_params_filtered = remove_noexpr_from_list more_params in
        more_params_filtered


and cCallExpr_from_tCallExpr expr tSym  tprogram func_name expr_list =
    match expr with
        | Noexpr -> let sym = StringMap.find func_name tSym in
                    (match sym with
```

```
                        FuncSymbol(_, fdecl) ->

                             let hasAnonParams = Semant.expr_list_contains_anon_defs_no_recursion
expr_list in

                            if (hasAnonParams = true) then

                                let update_anon_def_expr_list anonList =

                                    List.fold_left (fun ((e, slist), dlist) def ->

                                                            let ((_, _slist), _dlist) = update_expr
(AnonFuncDef(def)) tSym tprogram in

                                                ((Noexpr, slist@_slist), dlist@_dlist)) ((Noexpr, []),
[]) anonList

                                in

                                                                        let more_params_filtered =
generate_extra_capture_func_params_from_expr_list tSym tprogram expr_list in

                                     let anonParams = Semant.anon_defs_from_expr_list_no_recursion
tprogram expr_list in

                                              let ((updated_expr, updated_slist), updated_dlist) =
update_anon_def_expr_list anonParams in

                                let paramExpressions = List.rev (List.map2

                                         (cExpr_from_tExpr_in_tCall tSym  tprogram ) expr_list
fdecl.params) in



                                let ret =

                                                  ((CCall(0, CNoexpr, CId(CIdentifier(func_name)),
paramExpressions@more_params_filtered) , updated_slist), updated_dlist)

                                in

                                ret;

                            else

                                if (func_name = "printf") then

                                    let expr_list = (List.rev expr_list) in

                                                                        let replacementParamList =
Semant.func_param_list_from_expr_list tSym expr_list in

                                    ((CCall(0, CNoexpr, CId(CIdentifier(func_name)), (List.map2

                                         (cExpr_from_tExpr_in_tCall tSym  tprogram ) expr_list
replacementParamList)), []), [])

                                else

                                    let ret =

                                    ((CCall(0, CNoexpr, CId(CIdentifier(func_name)), (List.map2
```

```
                                        (cExpr_from_tExpr_in_tCall tSym  tprogram ) expr_list
fdecl.params)), []), [])

                                    in

                                    ret

                        | AnonFuncSymbol(anonName, AnonFuncType(_, tlist)) ->

                                let rec funcParam_from_tType t = (match t with

                                    _ -> ParamDeclWithType(DeclSpecTypeSpecAny(t)))


                                  and funcParamList_from_tTypeList tlist = (match tlist with

                                      [] -> []

                                    | [x] -> [funcParam_from_tType x]

                                                                        | h::t -> [funcParam_from_tType
h]@(funcParamList_from_tTypeList t))

                                in

                                let fParamList = funcParamList_from_tTypeList tlist in

                                    let fParamExprList = (List.map2 (cExpr_from_tExpr_in_tCall tSym
tprogram ) expr_list fParamList) in

                                let extraParamName = "cap_anon_" ^ func_name in

                                let extraParamExpr = CId(CIdentifier(extraParamName)) in

                                        ((CCall(1, CNoexpr, CId(CIdentifier("anon_" ^ func_name)),
fParamExprList@[extraParamExpr]), []), []))


        | _ -> let expr_type = Semant.type_from_expr tSym expr in (match expr_type with

                | CustomType(a) -> let fdecl = Semant.get_fdecl_for_receiver a

                tSym func_name in


                        if (Semant.is_interface tSym (Identifier(a))) then

                                let updated_expr = (fst (fst (update_expr expr tSym tprogram )))

                                in

                                let cexpr_list = [CMemAccess(1,

                                (updated_expr), CIdentifier("body"))] @

                                (List.map2 (cExpr_from_tExpr_in_tCall tSym tprogram )

                                expr_list fdecl.params) in
```

```
                                ((CCall(1, (fst (fst (update_expr expr
                                tSym tprogram))),
                                CId(CIdentifier(func_name)), cexpr_list), []),
                                [])
                            else
                                 let hasAnonParams = Semant.expr_list_contains_anon_defs_no_recursion
expr_list in
                                if (hasAnonParams = true) then
                                    let update_anon_def_expr_list anonList =
                                        List.fold_left (fun ((e, slist), dlist) def ->
                                                                let ((_, _slist), _dlist) = update_expr
(AnonFuncDef(def)) tSym tprogram in
                                                ((Noexpr, slist@_slist), dlist@_dlist)) ((Noexpr, []),
[]) anonList
                                    in
                                                                                        let  extra_params  =
generate_extra_capture_func_params_from_expr_list tSym tprogram expr_list in
                                         let anonParams = Semant.anon_defs_from_expr_list_no_recursion
tprogram expr_list in
                                              let ((updated_expr, updated_slist), updated_dlist) =
update_anon_def_expr_list anonParams in
                                    let first_arg =
                                            CCastExpr(CPointerType(CType(CPrimitiveType(Cvoid)),
                                            1), CPointify(fst(fst (update_expr expr
                                            tSym tprogram
                                            )))) in
                                    ((CCall(1, CMemAccess(0, fst (fst( (update_expr expr
                                    tSym tprogram))), CIdentifier("_virtual")),
                                    CId(CIdentifier(
                                    func_name)), [first_arg] @ (List.map2
                                    (cExpr_from_tExpr_in_tCall tSym  tprogram ) expr_list
                                    fdecl.params)@extra_params), updated_slist), updated_dlist)
                                else
                                    let first_arg =
                                            CCastExpr(CPointerType(CType(CPrimitiveType(Cvoid)),
```

```
                                        1), CPointify(fst(fst (update_expr expr
                                        tSym tprogram
                                        )))) in
                                    ((CCall(1, CMemAccess(0, fst (fst( (update_expr expr
                                    tSym tprogram))), CIdentifier("_virtual")),
                                    CId(CIdentifier(
                                    func_name)), [first_arg] @ (List.map2
                                    (cExpr_from_tExpr_in_tCall tSym  tprogram ) expr_list
                                    fdecl.params)),[]), [])
                | PointerType(CustomType(a), 1) ->
                                    let hasAnonParams = Semant.expr_list_contains_anon_defs_no_recursion
expr_list in
                            if (hasAnonParams = true) then
                                let fdecl =
                                Semant.get_fdecl_for_receiver a tSym func_name in
                                        let first_arg =
                                                CCastExpr(CPointerType(CType(CPrimitiveType(Cvoid)),
                                                1), fst (fst (update_expr expr tSym
                                                tprogram
                                                ))) in
                                let update_anon_def_expr_list anonList =
                                    List.fold_left (fun ((e, slist), dlist) def ->
                                            let ((_, _slist), _dlist) = update_expr (AnonFuncDef(def))
tSym tprogram in
                                            ((Noexpr, slist@_slist), dlist@_dlist)) ((Noexpr, []), [])
anonList
                                in
                                  let extra_params = generate_extra_capture_func_params_from_expr_list
tSym tprogram expr_list in
                                let anonParams = Semant.anon_defs_from_expr_list_no_recursion tprogram
expr_list in
                                            let ((updated_expr, updated_slist), updated_dlist) =
update_anon_def_expr_list anonParams in
                                ((CCall(1, CMemAccess(1,
                                fst (fst(update_expr expr tSym tprogram)),
```

```
                        CIdentifier("_virtual")),
                    CId(CIdentifier(
                        func_name)), [first_arg] @ (List.map2
                        (cExpr_from_tExpr_in_tCall tSym  tprogram ) expr_list
                        fdecl.params@extra_params)), updated_slist), updated_dlist)
                else
                    let fdecl =
                    Semant.get_fdecl_for_receiver a tSym func_name in
                            let first_arg =
                                    CCastExpr(CPointerType(CType(CPrimitiveType(Cvoid)),
                                    1), fst (fst (update_expr expr tSym
                                    tprogram
                                    ))) in

                                    ((CCall(1, CMemAccess(1,
                                    fst (fst(update_expr expr tSym tprogram)),
                                    CIdentifier("_virtual")),
                                   CId(CIdentifier(
                                    func_name)), [first_arg] @ (List.map2
                                    (cExpr_from_tExpr_in_tCall tSym  tprogram ) expr_list
                                    fdecl.params)), []), [])
            | _ -> raise(Failure("No other functions can call methods")))

let generate_virtual_table_assignments isPointer tStruct tSymbol_table id=
        let fdecls = Semant.get_unique_method_names_for_struct tSymbol_table tStruct in

         List.map (fun tmethod_name ->

               let inter_fdecl = Semant.get_fdecl_for_receiver
                                    tStruct.struct_name tSymbol_table
                                    tmethod_name in
```

```
                let cFunc_name = cFunc_from_tMethod (fst(inter_fdecl.receiver)) tmethod_name in


                if (isPointer >= 0) then
                                        CExpr(CAsnExpr(CMemAccess(isPointer,CId(CIdentifier(id)),
CIdentifier(tmethod_name)),
                      Asn,
                     (CId(CIdentifier(cFunc_name)))))
                else
                                        CExpr(CAsnExpr(CMemAccess(1,CDeref(CId(CIdentifier(id))),
CIdentifier(tmethod_name)),
                      Asn,
                     (CId(CIdentifier(cFunc_name))))))
          fdecls




let c_init_decl_from_string str =
       CInitDeclarator(CDirectDeclarator(CVar(CIdentifier(str))))


let c_init_decl_from_string_asn str op cExpr =
       CInitDeclaratorAsn(CDirectDeclarator(CVar(CIdentifier(str))), op, cExpr)




let generate_decls_and_stmts_from_id tSymbol_table tprogram id decl typ_ =
       match decl with
       | Declaration(_, InitDeclList([InitDeclarator(_)])) ->
                      let ctype = cType_from_tType tSymbol_table typ_ in
                      let cinit_decl = c_init_decl_from_string id in
                      ([CDeclaration(CDeclSpecTypeSpecAny(ctype),
                      cinit_decl)], [])


       | Declaration(_, InitDeclList([InitDeclaratorAsn(_, op, expr)])) ->
                      let ((updated_expr, extra_stmts), _) = update_expr expr
```

```
                        tSymbol_table  tprogram in

                        let ctype = cType_from_tType tSymbol_table typ_ in
                        let cinit_decl = c_init_decl_from_string_asn id op
                        updated_expr in

                        ([CDeclaration(CDeclSpecTypeSpecAny(ctype),
                        cinit_decl)], extra_stmts)


let update_decl_for_non_custom_type id decl tSymbol_table  tprogram =
        let sym = Semant.lookup_symbol_by_id tSymbol_table (Identifier(id)) in (match
        sym with
        | VarSymbol(id, type_) -> (match decl with
                        | Declaration(_, InitDeclList([InitDeclarator(DirectDeclarator(_))])) ->
                                generate_decls_and_stmts_from_id tSymbol_table tprogram
                                id decl type_
                        | Declaration(_, InitDeclarator(DirectDeclarator(_))) ->
                                generate_decls_and_stmts_from_id tSymbol_table tprogram
                                id decl type_
                        | Declaration(_, InitDeclaratorAsn(declrt, _, _)) ->
                                generate_decls_and_stmts_from_id tSymbol_table tprogram
                                id decl type_
                        | _ ->
                                generate_decls_and_stmts_from_id tSymbol_table tprogram
                                id decl type_)
        | FuncSymbol(_, _) -> raise(Failure("update_decl: FuncSymbol not supported"))
        | AnonFuncSymbol(_, _) -> raise(Failure("update_decl: AnonFuncSymbol not supported"))
        )

let declare_virtual_table_stack tSymbol_table tStruct id =
        let virtual_table_name = virtual_table_name_from_tStruct
```

```
        tStruct.struct_name in


        let virtual_table_id = String.concat "" ["_";id;virtual_table_name] in

        let virtual_table_init_decl = c_init_decl_from_string (String.concat ""
        ["_"; id; (virtual_table_name)]) in

                let virtual_table_assignments = generate_virtual_table_assignments 0 tStruct
tSymbol_table
        virtual_table_id in

        let assign_virtual_table_back_to_id = CExpr(CAsnExpr(CMemAccess(0,
        CId(CIdentifier(id)),
        CIdentifier("_virtual")), Asn,
        CPointify(CId(CIdentifier(virtual_table_id))))) in

        ([CDeclaration(CDeclSpecTypeSpecAny(CType(CStruct(virtual_table_name))),
        virtual_table_init_decl)], virtual_table_assignments @
        [assign_virtual_table_back_to_id])

let update_interface_decl_for_struct id struct_name tSymbol_table =
        let interface = Semant.get_interface tSymbol_table
        (Semant.get_interface_for_struct struct_name tSymbol_table) in

        let cstruct_name_for_interface = cStructName_from_tInterface
        interface.name in

        let cinit_decl = c_init_decl_from_string (String.concat ""
        ["_";id;(cStructName_from_tInterface interface.name)]) in

        CDeclaration(CDeclSpecTypeSpecAny(CType(CStruct(cstruct_name_for_interface))),
```

```
                                                cinit_decl)

let interface_decl_and_assignments_for_struct isPointer struct_ tSymbol_table id

=

        let custom_type = struct_.struct_name in


        let implements = Semant.get_interface_for_struct custom_type tSymbol_table in


        if (implements <> "") then (
                    let interface = Semant.get_interface tSymbol_table
                                    (Semant.get_interface_for_struct
                                     custom_type tSymbol_table) in


                    let interface_decl = update_interface_decl_for_struct
                         id custom_type tSymbol_table in


                    let cstruct_for_interface = cStructName_from_tInterface
                                          implements in


                    let access_id = String.concat "" ["_";id;cstruct_for_interface] in


                    let fdecls = (List.map (fun fdecl -> Semant.var_name_from_direct_declarator
                                                fdecl.func_name) interface.funcs) in
                    let interface_assignments  = List.map (fun tmethod_name ->
                                    let inter_fdecl =  Semant.get_fdecl_for_receiver
                                                custom_type tSymbol_table
                                                tmethod_name in


                                    let cFunc_name = cFunc_from_tMethod (fst
                                         (inter_fdecl.receiver)) tmethod_name in
```

```
                                CExpr(CAsnExpr(CMemAccess(0,CId(CIdentifier(access_id)),
                                    CIdentifier(tmethod_name)),
                                            Asn,
                                            (CId(CIdentifier(cFunc_name)))))
                                    ) fdecls in


                    let reference_implementer_asn =
                              let struct_expr = if (isPointer = 0) then
                                      CPointify(CId(CIdentifier(id))) else (if
                                              (isPointer > 0) then
                                              (CId(CIdentifier(id)))
                                      else
                                              (CDeref(CId(CIdentifier(id))))) in


                                      CExpr(CAsnExpr(CMemAccess(0,
                                      CId(CIdentifier(access_id)),
                                      CIdentifier("body")), Asn,
                                      CCastExpr(CPointerType(CType(CPrimitiveType(Cvoid)),
                                      1), struct_expr))) in


                                let interface_field_name = interface_field_name_in_struct implements
custom_type in


                    let cStruct_mem_access_expr = if (isPointer >= 0) then CMemAccess(isPointer,
                                              CId(CIdentifier(id)),
                                              CIdentifier(interface_field_name))
                    else (CMemAccess(1, CDeref(CId(CIdentifier(id))),
                    CIdentifier(interface_field_name))) in


                    let implementer_add_interface_assn =
                                                      CExpr(CAsnExpr(cStruct_mem_access_expr,
```

```
                                                Asn,
                                                CId(CIdentifier(access_id)))) in

                        let assignments = interface_assignments @
                                                [reference_implementer_asn] @
                                                [implementer_add_interface_assn]
                                                in

                        ([interface_decl],
                                        assignments)

                ) else ([], [])

let update_decl_for_custom_type id decl custom_type tSymbol_table  tprogram =
        let sym = Semant.lookup_symbol_by_id tSymbol_table
        (Identifier(custom_type)) in

        match sym with
        | StructSymbol(s, struct_) ->

                        let cstruct_type = CustomType(custom_type) in

                        let (cstruct_decl, stmts) = generate_decls_and_stmts_from_id
                        tSymbol_table  tprogram id decl cstruct_type in

                        let (virtual_table_decl, assignments) =
                                declare_virtual_table_stack tSymbol_table
                                struct_ id in

                        let (decls, assigns) =
                                interface_decl_and_assignments_for_struct 0
                                struct_ tSymbol_table id in
```

```
                        (cstruct_decl@decls@virtual_table_decl,
                        stmts@assigns@assignments)



let update_decl tSymbol_table  tprogram decl  =
                let id = Semant.var_name_from_declaration decl in
                   let tType = Semant.type_from_identifier tSymbol_table (Identifier(id))
                   in

                   match (tType) with
                   | PrimitiveType(t) -> update_decl_for_non_custom_type id decl
                   tSymbol_table tprogram
                   | CustomType(t) -> update_decl_for_custom_type id decl t
                   tSymbol_table tprogram
                   | _ -> generate_decls_and_stmts_from_id tSymbol_table  tprogram id decl tType



let rec update_statement tstmt tSymbol_table  tprogram  =  match tstmt with
        | CompoundStatement(decls, stmts) ->
                let updated_symbol_table = Semant.add_to_symbol_table tSymbol_table decls in
                let (new_decls, new_stmts) =
                    List.fold_left (fun decl_stmt_acc decl ->
                        let (n_decls, n_stmts) = update_decl
                        updated_symbol_table tprogram decl in
                          ((fst (decl_stmt_acc)) @ n_decls, (snd (decl_stmt_acc) @ n_stmts))) ([],
[]) decls in

                let more_new_stmts =
                    List.fold_left (fun stmt_acc stmt ->
                                                        let ((updated_stmt, additional_stmts),
additional_decls) =
                                                update_statement stmt
```

```
                                                updated_symbol_table  tprogram in

                                              stmt_acc @ additional_stmts @ [updated_stmt]) [] stmts
in

                let more_new_decls =

                    List.fold_left (fun decl_acc stmt ->

                                                                  let ((updated_stmt, additional_stmts),
additional_decls) =

                                                               update_statement stmt updated_symbol_table
tprogram in

                                              additional_decls@decl_acc ) [] stmts in


                  ((CCompoundStatement(new_decls@more_new_decls, new_stmts @ more_new_stmts), []),
[])


        | EmptyElse -> ((CEmptyElse, []), [])

        | Return(e) -> let ((updated_e, stmts), newDecls) = update_expr e tSymbol_table tprogram

         in ((CReturn(updated_e), stmts), newDecls)

        | If(e, stmt1, stmt2) ->

                let ((updated_expr, stmts), decls) = update_expr e tSymbol_table  tprogram  in

                     let ((updated_stmt1, additional_stmts), additional_decls) = update_statement
stmt1 tSymbol_table  tprogram in

                   let ((updated_stmt2, additional_stmts2), additional_decls2) = update_statement
stmt2 tSymbol_table  tprogram  in

          ((CIf(updated_expr, updated_stmt1, updated_stmt2), additional_stmts@additional_stmts2),
[])

            | For(e1, e2, e3, stmt) -> let ((updated_e1, stmts_e1), decls_e1) = update_expr e1
tSymbol_table  tprogram in

                                              let ((updated_e2, stmts_e2), decls_e2) = update_expr e2
tSymbol_table  tprogram in

                                              let ((updated_e3, stmts_e3), decls_e3) = update_expr e3
tSymbol_table  tprogram in

                                                 let ((updated_stmt, additional_stmts), decls_stmt) =
update_statement stmt  tSymbol_table  tprogram in

                                                                                  let accumulated_stmts =
(stmts_e1@stmts_e2@stmts_e3@additional_stmts) in

                                                                                  let accumulated_decls =
(decls_e1@decls_e2@decls_e3@decls_stmt) in

                                             ((CFor(updated_e1, updated_e2, updated_e3, updated_stmt),
accumulated_stmts), accumulated_decls)
```

```
        | While(e1, stmt) ->

                  let ((updated_e1, stmts_e1), decls_e1) = update_expr e1 tSymbol_table  tprogram
in

                 let ((updated_stmt, additional_stmts), additional_decls) = update_statement stmt
tSymbol_table tprogram in

                               ((CWhile(updated_e1,  updated_stmt),  stmts_e1  @  additional_stmts),
(decls_e1@additional_decls))

        | Break -> ((CBreak, []), [])

        | Expr(e) -> let ((updated_e, stmts), decls) = update_expr e tSymbol_table  tprogram in

                     ((CExpr(updated_e), stmts), decls)


let cFunc_from_tDestructor symbol_table tprogram destructor tStruct =

        let cdestructor_name = destructor_name_from_tStruct

        tStruct.struct_name in


        let first_param = first_param_for_destructor tStruct.struct_name in


        let last_param = last_param_for_destructor in


        let augmented_decls = List.fold_left (fun cdecls tdecl ->

                    let tdecl_id = (Semant.var_name_from_declaration tdecl) in

                    let tdecl_type = (Semant.type_from_declaration tdecl) in


                    let (cdecl, _)  = generate_decls_and_stmts_from_id symbol_table tprogram

                    tdecl_id tdecl tdecl_type in cdecls @ cdecl

        ) [] tStruct.members in


        {

                creturn_type = CType(CPrimitiveType(Cvoid));


                cfunc_params = [first_param] @ [last_param];


                cfunc_body = CCompoundStatement(augmented_decls,
```

```
                []);


                cfunc_name = cdestructor_name;
        }


let cFunc_from_tConstructor symbol_table  tprogram constructor tStruct =
        let cconstructor_name = constructor_name_from_tStruct
        tStruct.struct_name
        in


        let first_param = first_param_for_constructor tStruct.struct_name in


        let last_param = last_param_for_constructor in


        let augmented_decls = List.fold_left (fun cdecls tdecl ->
                    let tdecl_id = (Semant.var_name_from_declaration tdecl) in
                    let tdecl_type = (Semant.type_from_declaration tdecl) in


                    let (cdecl, _)  = generate_decls_and_stmts_from_id symbol_table tprogram
                    tdecl_id tdecl tdecl_type in cdecls @ cdecl
        ) [] tStruct.members in


        let generate_void_star_params =
            let anonList =
                List.filter (fun p -> match p with
                    AnonFuncDecl(anonDecl) -> true
                  | _ -> false) constructor.constructor_params
            in
            let returned_params =
                List.map (fun p -> match p with
                    AnonFuncDecl(anonDecl) ->
                        (match anonDecl.anon_decl_name with
```

```
                            Identifier(s) ->

                                let paramName = "cap_anon_" ^ s in

                                                    (CPointerType(CType(CPrimitiveType(Cvoid)), 1),
CIdentifier(paramName)))) anonList

            in

            returned_params

        in

        let extraParams = generate_void_star_params in


        {

                creturn_type = CType(CPrimitiveType(Cvoid));


                cfunc_params = [first_param] @ (List.map

                                                (cFuncParam_from_tFuncParam

                                                symbol_table)

                                                constructor.constructor_params)

                @ [last_param]@extraParams;


                cfunc_body = CCompoundStatement(augmented_decls,

                []);


                cfunc_name = cconstructor_name;


        }




let virtual_table_struct_for_tStruct symbol_table tStruct =

        let all_methods_for_struct = (List.map(Semant.get_fdecl_for_receiver

        tStruct.struct_name symbol_table)

        (Semant.get_unique_method_names_for_struct symbol_table tStruct)) in


        let methodMemberSymbols = List.map (cDeclaration_from_tFdecl
```

```
        symbol_table) all_methods_for_struct in

        let virt_table_name = virtual_table_name_from_tStruct
        tStruct.struct_name in

        {
                cstruct_name = virt_table_name;
                cstruct_members = methodMemberSymbols;
                cmethod_to_functions = StringMap.empty;
        }



let cStruct_from_tStruct symbol_table tprogram tStruct =
        let fieldSymbols = List.map (cSymbol_from_sSymbol
        symbol_table) (List.map (Semant.symbol_from_declaration)
        tStruct.members) in



        let virtual_table_name = virtual_table_name_from_tStruct
        tStruct.struct_name in

        let virtual_table_symbol =
                CVarSymbol("_virtual",
                CPointerType(CType(CStruct(virtual_table_name)), 1)) in

        let defaultStructMemberSymbols = [virtual_table_symbol] @ fieldSymbols in

        (* If there is an interface then add a struct member corresponding to
         * the interface to our struct *)
        let cStructMemberSymbols = if (Semant.get_interface_for_struct
        tStruct.struct_name symbol_table <> "") then
                [bol_from_Implements (Semant.get_interface_for_struct
```

```
                tStruct.struct_name symbol_table) tStruct.struct_name] @
                defaultStructMemberSymbols else defaultStructMemberSymbols in

                    let (methods_to_cfunctions, cfuncs) = (List.fold_left (fun (sym, cfunc_list)
method_ ->
                                        (let tfunc_name =
                                                Semant.var_name_from_direct_declarator
                                          method_.func_name in

                                        let initial_void_param =
                                                    create_cfunc_param_for_receiver
                                                    method_.receiver in

                                        let init_cast_decl =
                                                create_initial_cast_decl
                                                method_.receiver in

                                        let generate_void_star_params =
                                            let anonList =
                                                List.filter (fun p -> match p with
                                                    AnonFuncDecl(anonDecl) -> true
                                                  | _ -> false) method_.params
                                            in
                                            let returned_params =
                                                List.map (fun p -> match p with
                                                    AnonFuncDecl(anonDecl) ->
                                                        (match anonDecl.anon_decl_name with
                                                            Identifier(s) ->
                                                                let paramName = "cap_anon_" ^
s in

(CPointerType(CType(CPrimitiveType(Cvoid)), 1), CIdentifier(paramName)))) anonList
                                            in
```

```
                                    returned_params
                                in
                                let extraParams = generate_void_star_params in



                                let cfunc = {
                                    creturn_type = (cType_from_tType
                                    symbol_table
                                    (Semant.type_from_declaration_specifiers
                                    method_.return_type));

                                    cfunc_params =
                                            [initial_void_param] @ (List.map
                                    (cFuncParam_from_tFuncParam
                                    symbol_table) method_.params)@extraParams;

                                    cfunc_body =
                                            CCompoundStatement([init_cast_decl],
                                    []);

                                    cfunc_name = cFunc_from_tMethod
                                    tStruct.struct_name tfunc_name;

                                } in (StringMap.add tfunc_name cfunc
                                sym, cfunc_list @ [cfunc])))
                                (StringMap.empty, []) tStruct.methods) in

let cFunc_for_constructor =
        cFunc_from_tConstructor symbol_table tprogram tStruct.constructor tStruct in

let cFunc_for_destructor = cFunc_from_tDestructor symbol_table tprogram
tStruct.destructor tStruct in
```

```
         ({
                cstruct_name = cStructName_from_tStruct tStruct.struct_name;
                cstruct_members = cStructMemberSymbols;
                cmethod_to_functions = methods_to_cfunctions;
        }, cfuncs, (tStruct, cFunc_for_constructor, cFunc_for_destructor))



let update_cFunc tSymbol_table tprogram cFunc tFunc  =
        let updated_symbol_table = List.fold_left (fun m symbol -> StringMap.add
                    (Semant.get_id_from_symbol symbol) symbol m) tSymbol_table
((Semant.symbols_from_func_params
        tFunc.params) @ ([Semant.symbol_from_receiver tFunc.receiver])) in

        let CCompoundStatement(decls, stmts) = cFunc.cfunc_body in
              let CCompoundStatement(updated_decls, updated_stmts) = fst (fst (update_statement
tFunc.body updated_symbol_table tprogram )) in
        {
                cfunc_name = cFunc.cfunc_name;
                creturn_type = cFunc.creturn_type;
                cfunc_body = CCompoundStatement(decls @ updated_decls,
                updated_stmts);
                cfunc_params = cFunc.cfunc_params;
        }


let update_cFunc_from_anonDef tSymbol_table tprogram cFunc anonDef =
        let funcCaller = Semant.find_func_owning_anon_def tprogram anonDef in
        let rcvr = funcCaller.receiver in
        let rcvrSymbol = Semant.symbol_from_receiver rcvr in
        let globals = Semant.symbols_from_decls tprogram.globals in
        let builtinDecls = Semant.stdlib_funcs in
        let builtinSyms = Semant.symbols_from_fdecls builtinDecls in
        let localDecls = Semant.get_decls_from_compound_stmt anonDef.anon_body in
```

```
        let interfaceSymbols = Semant.symbols_from_interfaces tprogram.interfaces in
        let interfaceMethodSymbols =
            let interfaces =
                List.map (fun (InterfaceSymbol(_, iface)) -> iface) interfaceSymbols
            in
            List.fold_left (fun accList iface ->
                accList@(Semant.symbols_from_fdecls iface.funcs)) [] interfaces
        in


        let localSyms = Semant.symbols_from_decls localDecls in
        let paramSyms = Semant.symbols_from_func_params anonDef.anon_params in
                            let exceptSyms = Semant.symtable_from_symlist
(globals@builtinSyms@[rcvrSymbol]@interfaceMethodSymbols@paramSyms@localSyms) in
        let id_exists_in_symtable table id =
            try
                (fun x -> true)(StringMap.find id table)
            with _ ->
                false
        in


        let rec fix_expr locals instance_name expr = match expr with
            CBinop(e1, op, e2) ->
                    let fe1 = fix_expr locals instance_name e1 in
                    let fe2 = fix_expr locals instance_name e2 in
                    CBinop(fe1, op, fe2)
           | CAsnExpr(e1, aop, e2) ->
                    let fe1 = fix_expr locals instance_name e1 in
                    let fe2 = fix_expr locals instance_name e2 in
                    CAsnExpr(fe1, aop, fe2)
           | CCastExpr(t, e) ->
                   let fe = fix_expr locals instance_name e in
                    CCastExpr(t, fe)
```

```
| CPostfix(e, pop) ->
        let fe = fix_expr locals instance_name e in
        CPostfix(fe, pop)
| CCall(i, e1, e2, elist) ->
        let fe1 = fix_expr locals instance_name e1 in
        let fe2 = fix_expr locals instance_name e2 in
        let felist = fix_expr_list locals instance_name elist in
        CCall(i, fe1, fe2, felist)
| CDeref(e) ->
        let fe = fix_expr locals instance_name e in
        CDeref(fe)
| CCompareExpr(e1, lop, e2) ->
        let fe1 = fix_expr locals instance_name e1 in
        let fe2 = fix_expr locals instance_name e2 in
        CCompareExpr(fe1, lop, fe2)
| CPointify(e) ->
        let fe = fix_expr locals instance_name e in
        CPointify(fe)
| CMemAccess(i, e, id) ->
        let fe = fix_expr locals instance_name e in
        let fixedExpr = fix_expr locals instance_name (CId(id)) in
        (match fixedExpr with
             CId(fid) ->
                CMemAccess(i, fe, fid)
           | CMemAccess(_, _, _) -> fixedExpr)
| CId(CIdentifier(s)) ->
        if (id_exists_in_symtable tSymbol_table s) then
             expr
        else if (id_exists_in_symtable exceptSyms s) then
             expr
        else
            CMemAccess(1, CId(CIdentifier(instance_name)), CIdentifier(s))
```

```
            | CDeclExpr(CDeclaration(declSpecs, initDecl)) ->
                    let fInitDecl = fix_init_declarator locals instance_name initDecl in
                    CDeclExpr(CDeclaration(declSpecs, fInitDecl))
            | _ -> expr

        and fix_expr_list locals instance_name elist = match elist with
             [] -> []
           | [e] -> [fix_expr locals instance_name e]
           | h::t -> [fix_expr locals instance_name h]@(fix_expr_list locals instance_name t)

        and fix_init_declarator locals instance_name initDecl = match initDecl with
           | CInitDeclaratorAsn(dd, aop, e) ->
                            let CInitDeclarator(fdd) = fix_init_declarator locals instance_name
(CInitDeclarator(dd)) in
                  let fe = fix_expr locals instance_name e in
                  CInitDeclaratorAsn(fdd, aop, fe)
           | _ -> initDecl

        and fix_declaration locals instance_name decl = match decl with
             CDeclaration(declSpecs, initDecl) ->
                 let fidecl = fix_init_declarator locals instance_name initDecl in
                 CDeclaration(declSpecs, fidecl)

        and fix_declaration_list locals instance_name declList = match declList with
             [] -> []
           | [d] -> [fix_declaration locals instance_name d]
                | h::t -> [fix_declaration locals instance_name h]@(fix_declaration_list locals
instance_name t)

        and fix_statement locals instance_name stmt = match stmt with
             CExpr(e) ->
                 let fe = fix_expr locals instance_name e in
                 CExpr(fe)
```

```
          | CReturn(e) ->
                let fe = fix_expr locals instance_name e in
                CReturn(fe)
          | CCompoundStatement(declList, stmtList) ->
                let fdl = fix_declaration_list locals instance_name declList in
                let fsl = fix_statement_list locals instance_name stmtList in
                CCompoundStatement(fdl, fsl)
          | CIf(e, s1, s2) ->
                let fe = fix_expr locals instance_name e in
                let fs1 = fix_statement locals instance_name s1 in
                let fs2 = fix_statement locals instance_name s2 in
                CIf(fe, fs1, fs2)
          | CFor(e1, e2, e3, s) ->
                let fe1 = fix_expr locals instance_name e1 in
                let fe2 = fix_expr locals instance_name e2 in
                let fe3 = fix_expr locals instance_name e3 in
                let fs = fix_statement locals instance_name s in
                CFor(fe1, fe2, fe3, fs)
          | CWhile(e, s) ->
                let fe = fix_expr locals instance_name e in
                let fs = fix_statement locals instance_name s in
                CWhile(fe, fs)
          | _ -> stmt


       and fix_statement_list locals instance_name stmtList = match stmtList with
            [] -> []
          | [s] -> [fix_statement locals instance_name s]
                  | h::t -> [fix_statement locals instance_name h]@(fix_statement_list locals
instance_name t)


        in
        let updated_symbol_list =
```

```
                            ((Semant.symbols_from_func_params anonDef.anon_params) @
(Semant.symbols_from_outside_scope_for_anon_def tprogram anonDef)@[rcvrSymbol] )

        in

        let updated_symbol_table =

            (List.fold_left (fun m symbol ->

                StringMap.add (Semant.get_id_from_symbol symbol) symbol m)

            tSymbol_table (updated_symbol_list)) in

        let anon_name = Semant.find_name_for_anon_def tprogram anonDef in

        let instanceName = "s" ^ anon_name in

        let structName = "S" ^ anon_name in

                                                                    let newDecls =
[CDeclaration(CDeclSpecTypeSpecAny(CPointerType(CType(CStruct(structName)), 1)),

                    CInitDeclaratorAsn(CDirectDeclarator(CVar(CIdentifier(instanceName))), Asn,
CCastExpr(CPointerType(CType(CStruct(structName)), 1), CId(CIdentifier("capture_struct"))))))]

        in




        let CompoundStatement(decls, _) = anonDef.anon_body in

        let locals = Semant.symbols_from_decls decls in

        let CCompoundStatement(decls, stmts) = cFunc.cfunc_body in

               let cmpstmt = fst (fst (update_statement anonDef.anon_body updated_symbol_table
tprogram )) in

                    let CCompoundStatement(updated_decls, updated_stmts) = fix_statement
(locals@[rcvrSymbol]) instanceName cmpstmt in

        {

                cfunc_name = cFunc.cfunc_name;

                creturn_type = cFunc.creturn_type;

                cfunc_body = CCompoundStatement(newDecls@updated_decls @decls,

                updated_stmts);

                cfunc_params = cFunc.cfunc_params;

        }


let update_cDestructor tSymbol_table tprogram cFunc tStruct =

        let updated_symbol_table = List.fold_left (fun m symbol -> StringMap.add
```

```
        (Semant.get_id_from_symbol symbol) symbol m) tSymbol_table ((Semant.symbols_from_decls
        (Semant.get_decls_from_compound_stmt
        tStruct.constructor.constructor_body)) @ (Semant.symbols_from_decls
        tStruct.members) @ (Semant.symbols_from_func_params
        tStruct.constructor.constructor_params))  in




        let ancestor_destructor = Semant.get_ancestors_destructor tSymbol_table
        tStruct in


        let parent_destructor_call =
        if (ancestor_destructor.destructor_name <> tStruct.struct_name &&
        ancestor_destructor.destructor_name <> "") then
              generate_stmts_for_parent_destructor tSymbol_table tprogram
              tStruct.destructor tStruct else [] in


        let head_assignments = List.fold_left (fun assignments tdecl ->
                    let tdecl_id = (Semant.var_name_from_declaration tdecl) in


                    let asn_expr = CExpr(CAsnExpr(CId(CIdentifier(tdecl_id)),
                    Asn, CMemAccess(1,
                      CDeref(CId(CIdentifier("_this"))), CIdentifier(tdecl_id)))) in assignments @
[asn_expr] ) [] tStruct.members in


        let CCompoundStatement(original_decls, stmts) = cFunc.cfunc_body in


        let CCompoundStatement(updated_decls, updated_stmts) = fst (
                        fst (update_statement tStruct.destructor.destructor_body
                        updated_symbol_table tprogram)) in


        let free_this = CIf(CId(CIdentifier("_needs_free")),
```

```
        CCompoundStatement([], [CExpr(CFree(CMemAccess(1,
        CDeref(CId(CIdentifier("_this"))),
                  CIdentifier("_virtual")))))]@[CExpr(CFree(CDeref(CId(CIdentifier("_this"))))))]),
CEmptyElse) in

        {
                cfunc_name = cFunc.cfunc_name;
                creturn_type = cFunc.creturn_type;
                cfunc_body = CCompoundStatement(original_decls @ updated_decls,
                head_assignments @ parent_destructor_call @ updated_stmts @ [free_this]);
                cfunc_params = cFunc.cfunc_params;
        }



let update_cConstructor tSymbol_table tprogram cFunc tStruct =
        let updated_symbol_table = List.fold_left (fun m symbol -> StringMap.add
        (Semant.get_id_from_symbol symbol) symbol m) tSymbol_table ((Semant.symbols_from_decls
        (Semant.get_decls_from_compound_stmt
        tStruct.constructor.constructor_body)) @ (Semant.symbols_from_decls
        tStruct.members) @ (Semant.symbols_from_func_params
        tStruct.constructor.constructor_params))  in

        let ctype = cType_from_tType tSymbol_table
        (CustomType(tStruct.struct_name)) in

        let virtual_table_name = virtual_table_name_from_tStruct
        tStruct.struct_name in



        let virtual_table_type =
                CType(CStruct(virtual_table_name)) in
```

```
        let alloc_virtual_table =
                CExpr(CAsnExpr(CMemAccess(1, CDeref(CId(CIdentifier("_this"))),
                CIdentifier("_virtual")),
        Asn, CAlloc(virtual_table_type, CId(CIdentifier(sizeof_string tSymbol_table
        virtual_table_type))))) in


                  let alloc_this = CIf(CId(CIdentifier("_needs_malloc")),
CExpr(CAsnExpr(CDeref(CId(CIdentifier("_this"))),
                    Asn, CAlloc(ctype, CId(CIdentifier((sizeof_string tSymbol_table
                    ctype)))))), CExpr(CNoexpr)) in


        let virt_table_assignments = generate_virtual_table_assignments 1 tStruct tSymbol_table
        "(*_this)->_virtual" in


        let (interface_decls, interface_assignments) =
                interface_decl_and_assignments_for_struct (-1) tStruct
                tSymbol_table "_this" in


        let tail_assignments = List.fold_left (fun assignments tdecl ->
                    let tdecl_id = (Semant.var_name_from_declaration tdecl) in


                    let asn_expr = CExpr(CAsnExpr(CMemAccess(1,
                    CDeref(CId(CIdentifier("_this"))), CIdentifier(tdecl_id)),
                              Asn, CId(CIdentifier(tdecl_id)))) in assignments @ [asn_expr] ) []
tStruct.members in


        let stmts_for_super =
            if (Semant.constructor_has_super tStruct.constructor) then
                                        (let Expr(Super(expr_list)) = Semant.get_super_expr
tStruct.constructor.constructor_body in
                                generate_stmts_for_super expr_list updated_symbol_table tprogram
tStruct.constructor tStruct)
        else
            []
```

```
        in


        let CCompoundStatement(original_decls, stmts) = cFunc.cfunc_body in


        let CCompoundStatement(updated_decls, updated_stmts) = fst (
                        fst (update_statement tStruct.constructor.constructor_body
                        updated_symbol_table tprogram)) in
        {
                cfunc_name = cFunc.cfunc_name;
                creturn_type = cFunc.creturn_type;
                cfunc_body = CCompoundStatement(original_decls @ interface_decls
                @ updated_decls,
                [alloc_this] @ [alloc_virtual_table] @ stmts_for_super @ updated_stmts @ stmts @
                tail_assignments@virt_table_assignments @ interface_assignments);
                cfunc_params = cFunc.cfunc_params;
        }


let rec cFunc_from_anonDef symbol_table tprogram anonDef =
    let rec convert_anon_params symbol_table params =
        (match params with
                                [] -> [(CPointerType(CType(CPrimitiveType(Cvoid)), 1),
CIdentifier("capture_struct"))]
                            | [p] -> [cFuncParam_from_tFuncParam symbol_table
p]@[(CPointerType(CType(CPrimitiveType(Cvoid)), 1), CIdentifier("capture_struct"))]
          | h::t -> let htype = (cFuncParam_from_tFuncParam symbol_table h) in
                    let ttype = (convert_anon_params symbol_table t) in
                    [htype]@ttype)
    in
    {
    cfunc_name = anonDef.anon_name;
    cfunc_body = CCompoundStatement([], []);
    cfunc_params = (convert_anon_params symbol_table anonDef.anon_params);
    creturn_type = cType_from_tType symbol_table anonDef.anon_return_type }
```

```
and cFunc_list_from_anonDef_list symbol_table tprogram adlist =
    match adlist with
    [] -> []
  | [x] -> [cFunc_from_anonDef symbol_table tprogram x]
  | h::t -> let hfuncs = [(cFunc_from_anonDef symbol_table tprogram h)] in
            let tfuncs = (cFunc_list_from_anonDef_list symbol_table tprogram t) in
            hfuncs@tfuncs



let cProgram_from_tProgram program =
        let updated_program = Semant.update_structs_in_program program in

        let tSymbol_table = Semant.build_symbol_table updated_program in
            let cstructs_and_functions = List.map (cStruct_from_tStruct tSymbol_table program )
updated_program.structs in

        let cstructs = List.map (fun (structs, _, _) -> structs) cstructs_and_functions in

        let virt_table_structs = List.map (
                virtual_table_struct_for_tStruct tSymbol_table)
        updated_program.structs in

        let cfuncs_methods = List.concat (List.map (fun(_, methods, _) ->
                methods) cstructs_and_functions) in

        let cconstructors = List.map (fun(_, _, constructor) -> constructor)
        (List.filter (fun(_, _, (_, const, _)) -> if (const.cfunc_name = "") then false
        else true) cstructs_and_functions) in

        (*let cdestructors = List.map (fun(_, _, constructor) -> constructor)*)
        (*(List.filter (fun(_, _, (_, _, destr)) -> if (destr.cfunc_name = "") then false*)
```

```
        (*else true) cstructs_and_functions) in*)


        let cglobals = List.fold_left (fun acc (decls, _) -> acc @

        decls) [] (List.map (update_decl tSymbol_table updated_program)

        updated_program.globals) in




        let cStructs = virt_table_structs @ (List.map (cStruct_from_tInterface

        tSymbol_table) program.interfaces) @ cstructs in


        let tAnonDefs = Semant.anon_defs_from_tprogram program in

           let cFuncsTranslatedFromAnonDefs = cFunc_list_from_anonDef_list tSymbol_table program
tAnonDefs in

        let capture_structs = capture_struct_list_from_anon_def_list program tAnonDefs in

        (* The function bodies have not been filled out yet. Just the parameters

         * and return types *)

           let cDeclaredMethodsAndFuncs = cfuncs_methods @ (List.rev (List.map (cFunc_from_tFunc
tSymbol_table)

        (List.filter (fun fdecl ->

                if (fdecl.receiver = ("", "")) then true else

                false) program.functions))) in


        let cUpdatedDeclaredMethodsAndFuncs = List.fold_left (fun acc cFunc ->

                let sym_ = StringMap.find

        cFunc.cfunc_name tSymbol_table in (match sym_ with

                                        | FuncSymbol(_, fdecl) -> acc @ [update_cFunc tSymbol_table
program

                                          cFunc fdecl]

                                    | _ -> raise(Failure("error")))

                        ) [] cDeclaredMethodsAndFuncs in


        let cConstructors = List.fold_left (fun acc (tStruct, cConst, _) ->

                acc @ [update_cConstructor tSymbol_table program cConst tStruct]) []
```

```
            cconstructors in


        let cDestructors = List.fold_left (fun acc (tStruct, _, cDestr) ->

            acc @ [update_cDestructor tSymbol_table program cDestr tStruct]) []

            cconstructors in




        let anon_def_for_function fn =

            List.find (fun af ->

                if (af.anon_name = fn.cfunc_name) then

                    true

                else

                    false) tAnonDefs

        in


        let cUpdatedFuncsTranslatedFromAnonDefs =

            List.map (fun f ->

                let anonDef = anon_def_for_function f in

                                    update_cFunc_from_anonDef tSymbol_table program f anonDef)
cFuncsTranslatedFromAnonDefs

        in

                let cFuncs = cConstructors @ cDestructors @ cUpdatedDeclaredMethodsAndFuncs @
cUpdatedFuncsTranslatedFromAnonDefs

        in

        {

                cstructs = cStructs@capture_structs;

                cglobals = cglobals;

                cfunctions = cFuncs;

        }
```

Ccodegen.ml

```
open Ast

module StringMap = Map.Make(String)

type cIdentifier = CIdentifier of string

type cPrimitive =
    Cvoid
  | Cchar
  | Cshort
  | Cint
  | Clong
  | Cfloat
  | Cdouble

type cProgram = {
        cstructs: cStruct list;
        cglobals: cDeclaration list;
        cfunctions: cFunc list;
}

and cStruct = {
  cstruct_name: string;
  cstruct_members: cSymbol list;
  cmethod_to_functions: cFunc StringMap.t;
}

and cFuncSignature = {
    func_return_type: cType;
    func_param_types: cType list;
```

```
}

and cFunc = {
  cfunc_name: string;
  cfunc_body: cStatement;
  cfunc_params: cFuncParam list;
  creturn_type: cType;
}

and cFuncDecl = {
  cfdecl_name: string;
  cfdecl_params: cType list;
  cfdecl_return_type: cType;
}

and cNonPointerType =
    CPrimitiveType of cPrimitive
  | CStruct of string

and cPointer =
    CPointer of cNonPointerType
  | CPointerPointer of cPointer

and cType =
    CType of cNonPointerType
  | CPointerType of cType * int
  | CFuncPointer of cFuncSignature

and cExpr =
    CBinop of cExpr * tOperator * cExpr
   | CAsnExpr of cExpr * tAssignmentOperator * cExpr
   | CLiteral of int
```

```
    | CFloatLiteral of float
    | CStringLiteral of string
    | CCastExpr of cType * cExpr
    | CPostfix of cExpr * tPostfixOperator
    | CCall of int * cExpr * cExpr * cExpr list (* The int field is a flag to
    indiciate it is a pointer dereference *)
    | CAlloc of cType * cExpr
    | CNeg of cExpr
    | CFree of cExpr
    | CDeref of cExpr
    | CArrayAccess of cExpr * cExpr
    | CCompareExpr of cExpr * tLogicalOperator * cExpr
    | CPointify of cExpr
    | CMemAccess of int * cExpr * cIdentifier (* The int field is a flag to
    indicate it is a pointer dereference *)
    | CId of cIdentifier
    | CDeclExpr of cDeclaration
    | CNoexpr
    | CNull

and cStatement =
    CExpr of cExpr
    | CEmptyElse
    | CReturn of cExpr
    | CCompoundStatement of cDeclaration list * cStatement list
    | CIf of cExpr * cStatement * cStatement
    | CFor of cExpr * cExpr * cExpr * cStatement
    | CWhile of cExpr * cStatement
    | CBreak

and cDirectDeclarator =
    CVar of cIdentifier
```

```
and cDeclarator =
    CDirectDeclarator of cDirectDeclarator

and cInitDeclarator =
    CInitDeclarator of cDeclarator
   |  CInitDeclaratorAsn of cDeclarator * tAssignmentOperator * cExpr

and cDeclarationSpecifiers =
    CDeclSpecTypeSpecAny of cType

and cFuncParam = cType * cIdentifier

and cDeclaration =
    CDeclaration of cDeclarationSpecifiers * cInitDeclarator

and cSymbol =
    CVarSymbol of string * cType
    | CFuncSymbol of string * cFunc
    | CStructSymbol of string * cStruct

let cStructName_from_tInterface name =
        String.concat "" ["_interface"; name]

let interface_field_name_in_struct interface_name struct_name =
        String.concat "" ["_"; (String.concat "_" [interface_name;struct_name])]

let cStructName_from_tStruct name =
        String.concat "" ["_struct"; name]

let virtual_table_name_from_tStruct name =
        String.concat "" ["_virtual";name]
```

```
let constructor_name_from_tStruct name =

        String.concat "_" ["_constructor";name]


let destructor_name_from_tStruct name =

        String.concat "_" ["_destructor";name]


let cType_from_tTypeSpec = function

    Void -> CType(CPrimitiveType(Cvoid))

  | Char -> CType(CPrimitiveType(Cchar))

  | Short -> CType(CPrimitiveType(Cshort))

  | Int -> CType(CPrimitiveType(Cint))

  | Long -> CType(CPrimitiveType(Clong))

  | Float -> CType(CPrimitiveType(Cfloat))

  | Double -> CType(CPrimitiveType(Cdouble))

  | Signed -> raise(Failure("cType_from_tTypeSpec: Error, Signed unsuported at the moment"))

  | Unsigned -> raise(Failure("cType_from_tTypeSpec: Error, Unsigned unsuported at the moment"))

  | String -> CPointerType(CType(CPrimitiveType(Cchar)), 1)

  (*| _ -> raise(Failure("cType_from_tTypeSpec: Error, unsupported tTypeSpec"))*)


let rec print_pointers n =

        if (n = 1) then "*" else

       ( String.concat "" ["*";(print_pointers (n-1))])


let rec sizeof_string tSymbol_table typ_ = match typ_ with

  | CType(CPrimitiveType(Cvoid)) -> "void"

  | CType(CPrimitiveType(Cchar)) -> "char"

  | CType(CPrimitiveType(Cint)) -> "int"

  | CType(CPrimitiveType(Clong)) -> "long"

  | CType(CPrimitiveType(Cfloat)) -> "float"

  | CType(CPrimitiveType(Cdouble)) -> "double"

  | CType(CStruct(t)) -> String.concat " " ["struct"; t]
```

```
  | CPointerType(base, n) -> String.concat " " [(sizeof_string tSymbol_table
  base); (print_pointers n)]

let rec cType_from_tType symbol_table = function
    PrimitiveType(typeSpec) -> cType_from_tTypeSpec typeSpec
  | PointerType(base_type, num) -> CPointerType(cType_from_tType symbol_table base_type,
  num)
  | ArrayType(array_type, ptr, e) -> (let t1 = Semant.type_of_array_type
  symbol_table (ArrayType(array_type, ptr, e)) in cType_from_tType symbol_table
  t1)
  | CustomType(s) -> (let sym = StringMap.find s symbol_table in
                                        match sym with
                                        | StructSymbol(name, _) ->
                                                      CType(CStruct(cStructName_from_tStruct
                                        name))
                                        | InterfaceSymbol(name, _) ->
CPointerType(CType(CStruct(cStructName_from_tInterface
                                        name)), 1))
  | AnonFuncType(t, tlist) ->
          let anonRetType = (cType_from_tType symbol_table t) in
          let anonParamTypes = List.map (fun x -> (cType_from_tType symbol_table x)) tlist in
          let captureParam = CPointerType(CType(CPrimitiveType(Cvoid)), 1) in
          CFuncPointer({
              func_return_type = anonRetType;
              func_param_types = anonParamTypes@[captureParam]
          })
  | _ -> raise(Failure("Haven't filled out yet"))



let id_exists_in_symtable symbols id =
  try
```

```
    StringMap.find (Astutil.string_of_identifier id) symbols;
    true
  with _ -> false


let id_exists_in_symlist symlist id =
  let check_sym_id_equal sym id =
    match sym with
      VarSymbol(name, _) -> if (name = (Astutil.string_of_identifier id)) then true else false
    | FuncSymbol(name, _) -> if (name = (Astutil.string_of_identifier id)) then true else false
     | AnonFuncSymbol(name, _) -> if ( name = (Astutil.string_of_identifier id)) then true else
false
  in
  let compare_symbol_with_id (id, (hasBeenFound, foundSymbol)) sym =
    match hasBeenFound with
      false -> if (check_sym_id_equal sym id) == true then (id, (true, sym))
               else (id, (hasBeenFound, foundSymbol))
    | true -> (id, (hasBeenFound, foundSymbol))
  in
    let (_, (isFound, foundSym)) = (List.fold_left compare_symbol_with_id (id, (false,
VarSymbol("ERROR_SYMBOL", PrimitiveType(Void)))) symlist) in isFound



let lookup_symbol_from_symlist_by_id symlist id =
  let check_sym_id_equal sym id =
    match sym with
      VarSymbol(name, _) -> name == (Astutil.string_of_identifier id)
    | FuncSymbol(name, _) -> name == (Astutil.string_of_identifier id)
    | AnonFuncSymbol(name, _) -> name == (Astutil.string_of_identifier id)
  in
  let compare_symbol_with_id (id, (hasBeenFound, foundSymbol)) sym =
    match hasBeenFound with
      false -> if (check_sym_id_equal sym id) == true then (id, (true, sym))
               else (id, (hasBeenFound, foundSymbol))
```

```
    | true -> (id, (hasBeenFound, foundSymbol))

  in

    match (List.fold_left compare_symbol_with_id (id, (false, VarSymbol("ERROR_SYMBOL",
PrimitiveType(Void)))) symlist) with

    (_, (true, foundSym)) -> foundSym

  | _ -> raise(Failure("lookup_symbol_from_symlist_by_id: Error, symbol not in table."))




let cDeclarationSpecifiers_from_tDeclarationSpecifiers symbol_table tDeclSpecs = function

        | DeclSpecTypeSpecAny(tType) ->

                        CDeclSpecTypeSpecAny(cType_from_tType symbol_table tType)


let cDeclaration_from_tFdecl symbol_table fdecl =

        let first_argument = [CPointerType(CType(CPrimitiveType(Cvoid)), 1)] in

        let cfunc_param_types = first_argument @ List.map (cType_from_tType symbol_table)

                                    (Semant.type_list_from_func_param_list fdecl.params) in

        let generate_void_star_param_types =

            let anonList =

                List.filter (fun p -> match p with

                    AnonFuncDecl(anonDecl) -> true

                  | _ -> false)fdecl.params

            in

            let returned_param_types =

                List.map (fun p -> match p with

                    AnonFuncDecl(anonDecl) ->

                        CPointerType(CType(CPrimitiveType(Cvoid)), 1)) anonList

            in

            returned_param_types

        in

        let extraParamTypes = generate_void_star_param_types in

        let cfunc_return_type =

                cType_from_tType symbol_table (Semant.type_from_declaration_specifiers
```

```
                fdecl.return_type) in
        let func_signature = {
                                func_return_type = cfunc_return_type;
                                func_param_types = cfunc_param_types@extraParamTypes;
                               } in

                    CVarSymbol((Semant.var_name_from_direct_declarator  fdecl.func_name),
CFuncPointer(func_signature))

(* The C Struct corresponding to the Cimple Interface consists of
 * 1) Function pointers instead of methods. The first argument is a void star *)

let cStruct_from_tInterface symbol_table interface =
        let cBodySymbol = [CVarSymbol("body",
        CPointerType(CType(CPrimitiveType(Cvoid)),
        1))] in (* This is the void * body that we apply to all the functions *)
        let cSymbols = List.map (cDeclaration_from_tFdecl symbol_table) interface.funcs in
        {
                cstruct_members = cBodySymbol @ cSymbols;
                cstruct_name = cStructName_from_tInterface interface.name;
                cmethod_to_functions = StringMap.empty
        }

let cSymbol_from_Implements implements =
        let cstruct_name = cStructName_from_tInterface implements in
        CVarSymbol(cstruct_name, CPointerType(CType(CStruct(cstruct_name)), 1))

let cFuncParam_from_tFuncParam symbol_table tFuncParam =
          (cType_from_tType  symbol_table  (Semant.type_from_func_param  tFuncParam),
(CIdentifier(Semant.var_name_from_func_param tFuncParam)))

let create_cfunc_param_for_receiver receiver =
        (CPointerType(CType(CPrimitiveType(Cvoid)), 1),
```

```
        CIdentifier("_body"))


 let create_initial_cast_decl receiver =
        let cstruct_name = cStructName_from_tStruct (fst receiver) in
        CDeclaration(CDeclSpecTypeSpecAny(CPointerType(CType(CStruct(cstruct_name)),
        1)), CInitDeclaratorAsn(CDirectDeclarator(CVar(CIdentifier(snd receiver)))
        , Asn, CCastExpr(CPointerType(CType(CStruct(cstruct_name)), 1),
        CId(CIdentifier("_body")))))


let number_of_anon_func_parameters_in_tFuncParamList plist =
    List.fold_left (fun acc f ->
                     (match f with
                        AnonFuncDecl(_) -> (acc + 1)
                      | _ -> acc)) 0 plist


let number_of_anon_func_parameters_in_tFuncDecl fdecl =
    number_of_anon_func_parameters_in_tFuncParamList fdecl.params


let cFunc_from_tFunc symbol_table tFunc =
    {
            creturn_type = cType_from_tType symbol_table
            (Semant.type_from_declaration_specifiers tFunc.return_type);


            cfunc_params = List.rev (List.map (cFuncParam_from_tFuncParam
            symbol_table) tFunc.params);


            cfunc_body = CCompoundStatement([], []);


            cfunc_name = Semant.var_name_from_direct_declarator
            tFunc.func_name;
    }
```

```
let rec cSymbol_from_sSymbol symbol_table sym = match sym with

    VarSymbol(s, t) -> CVarSymbol(s, (cType_from_tType symbol_table t))

  | FuncSymbol(s, fdecl) -> CFuncSymbol(s, (cFunc_from_tFunc symbol_table fdecl))

  | StructSymbol(s, strct) ->

          let (newStrct, _) = cStruct_from_tStruct symbol_table strct in

          CStructSymbol(s, newStrct)

  | _ -> raise(Failure("Not completed"))




 (*------------------- Function struct_members_from_anon_body-------------------------

 * This function returns a list of Ast.sSymbols representing the variables referenced within the
body of

 * an anonymous function that are declared outside of it's scope. This list will form the data

 * members of a special c struct that will be passed to a normal c function whenevever

 * an anonymous function in cimple is instantiated.

 *

 * Parameters:

         * symbols: A StringMap of symbols from outside the scope of the anonymous function def

         * psymbols: A symbol table of parameters to this anonymous function

         * members: A list of function parameters declared in the anon function definition

            * body: An Ast.tStatement (specifically a CompoundStatement) that is the body of the
anonymous function

 *------------------------------------------------------------------------------- *)



and struct_members_from_anon_body symbols psymbols members body =


  let rec print_member m = Printf.printf "%s\n" (Astutil.string_of_symbol m)


  and print_member_list mlist = match mlist with

        [] -> ()

      | [x] -> print_member x

      | h::t -> print_member h; print_member_list t

  in
```

```
  let symbol_is_capturable = function
        VarSymbol(_, _) -> true
      | _ -> false
  in


  let rec members_from_expr symbols psymbols members e = match e with
      Id(id) ->
          if (id_exists_in_symtable psymbols id) then []
          else if (id_exists_in_symlist members id) = true then
             []
          else if (id_exists_in_symtable symbols id) = true then
                  let sym = Semant.lookup_symbol_by_id symbols id in
                     if (symbol_is_capturable sym) then
                         [sym]
                     else
                         []
                (*else if (id_exists_in_symtable psymbols id) then []*)
                else (match id with
                  Identifier(s) ->
                      print_member_list members;
                      raise(Failure("members_from_expr: Error - undeclared symbol '" ^ s ^ "'")))
   |  Binop(e1, _, e2) -> let e1Members = members_from_expr symbols psymbols members e1 in
                            let e2Members = members_from_expr symbols psymbols (members@e1Members)
e2 in
                           e1Members@e2Members
   |  AsnExpr(e1, _, e2) -> let e1Members = members_from_expr symbols psymbols members e1 in
                                                 let e2Members = members_from_expr symbols psymbols
(members@e1Members) e2 in
                            e1Members@e2Members
   |  Postfix(e1, _) -> let e1Members = members_from_expr symbols psymbols members e1 in
                            e1Members
   |  Call(_, e, elist) -> let eMembers = members_from_expr symbols psymbols members e in
```

```
                                        let elistMembers = members_from_expr_list symbols psymbols
(members@eMembers) elist in

                              eMembers@elistMembers

   | Make(_, elist) -> members_from_expr_list symbols psymbols members elist

   | Pointify(e) -> members_from_expr symbols psymbols members e

   | MemAccess(e, id2) -> let id1Members = members_from_expr symbols psymbols members e in

                                                 let id2Members = members_from_expr symbols psymbols
(members@id1Members) e in

                               id1Members@id2Members

   | AnonFuncDef(def) -> raise(Failure("members_from_expr: Error - nested anonymous functions not
supported"))

   | DeclExpr(decl) -> members_from_declaration symbols psymbols members decl

   | StringLiteral(_) -> []

   | _  -> []


   and members_from_expr_list symbols psymbols members elist = match elist with

     [] -> []

   | [x] -> members_from_expr symbols psymbols members x

   | h::t -> let hMembers = members_from_expr symbols psymbols members h in

             let tMembers = members_from_expr_list symbols psymbols (members@hMembers) t in

             hMembers@tMembers


   and members_from_init_declarator symbols psymbols members initDecl =

    match initDecl with

        InitDeclaratorAsn(_, _, e) -> members_from_expr symbols psymbols members e

     | InitDeclList(l) -> members_from_init_declarator_list symbols psymbols members l

     | _ -> []


   and members_from_init_declarator_list symbols psymbols members declList =

    match declList with

      (*[] -> members_from_expr symbols psymbols members Noexpr*)

      [] -> []

    | [x] -> members_from_init_declarator symbols psymbols members x
```

```
    | h::t -> let hmembers = members_from_init_declarator symbols psymbols members h in
              hmembers@(members_from_init_declarator_list symbols psymbols (members@hmembers) t)

   and members_from_declaration symbols psymbols members decl = match decl with
     Declaration(_, initDecl) ->
         members_from_init_declarator symbols psymbols members initDecl
    (*| _ -> [] [> Other types of declarations wouldn't reference variables from outside scope
<]*)

   and members_from_declaration_list symbols psymbols members declList = match declList with
      [] -> []
   | [x] -> (members_from_declaration symbols psymbols members x)
   | h::t -> let hmembers = members_from_declaration symbols psymbols members h in
             hmembers@(members_from_declaration_list symbols psymbols (members@hmembers) t)

   and members_from_statement_list symbols psymbols members stmtList = match stmtList with
   [] -> []
   | [x] -> (members_from_statement symbols psymbols members x)
   | h::t -> let hmembers = members_from_statement symbols psymbols members h in
             (hmembers)@(members_from_statement_list symbols psymbols (members@hmembers) t)

   and members_from_statement symbols psymbols members stmt = match stmt with
     CompoundStatement(decls, stmtList) ->
       let dmembers = (members_from_declaration_list symbols psymbols members decls) in
       dmembers@members_from_statement_list symbols psymbols (members@dmembers) stmtList
   | Expr(e) -> members_from_expr symbols psymbols members e
   | Return(e) -> members_from_expr symbols psymbols members e
   | If(e, s1, s2) -> let eMembers = members_from_expr symbols psymbols members e in
                        let s1Members = members_from_statement symbols psymbols (members@eMembers)
s1 in
                                          let s2Members = members_from_statement symbols psymbols
(members@eMembers@s1Members) s2 in
                       eMembers@s1Members@s2Members
```

```
   | For(e1, e2, e3, s) -> let e1Members = members_from_expr symbols psymbols members e1 in
                           let e2Members = members_from_expr symbols psymbols (members@e1Members)
e2 in
                                                   let e3Members = members_from_expr symbols psymbols
(members@e1Members@e2Members) e3 in
                                                 let sMembers = members_from_statement symbols psymbols
(members@e1Members@e2Members@e3Members) s in
                           e1Members@e2Members@e3Members@sMembers
   | While(e, s) -> let eMembers = members_from_expr symbols psymbols members e in
                      let sMembers = members_from_statement symbols psymbols (members@eMembers) s
in
                    eMembers@sMembers
   | _ -> []
  in
  let mems = members_from_statement symbols psymbols members body in
  mems


(* --------------------------Function capture_struct_from_anon_def ------------------------
 * Returns a C struct to be used as a copy of the variables used within the body of an
 * anonymous function that were declared outside of its scope.
 *
 * Parameters:
         * program: an Ast.tProgram.
         * def: The Ast.tAnonFuncDef whose body we are looking through to find captured variables
 * ------------------------------------------------------------------------*)


and capture_struct_from_anon_def program def =
  let func = Semant.find_func_containing_anon_def program def in
  let receiverSymbols = [Semant.symbol_from_receiver func.receiver] in
  let interfaceSymbols = Semant.symbols_from_interfaces program.interfaces in
  let interfaceMethodSymbols =
      let interfaces =
          List.map (fun (InterfaceSymbol(_, iface)) -> iface) interfaceSymbols
      in
```

```
      List.fold_left (fun accList iface ->

          accList@(Semant.symbols_from_fdecls iface.funcs)) [] interfaces

  in

  (*(Astutil.print_symbol_table (Semant.symtable_from_symlist interfaceMethodSymbols));*)

  let extraSymbols = receiverSymbols@interfaceSymbols@interfaceMethodSymbols in

  let symlist = (Semant.symbols_from_outside_scope_for_anon_def program def)@extraSymbols in

  let symbols = Semant.symtable_from_symlist symlist in

  let builtinDecls = Semant.stdlib_funcs in

  let builtinSyms = Semant.symbols_from_fdecls builtinDecls in

  let rec symconvert m = cSymbol_from_sSymbol symbols m in

  let internal_anon_symbols = (fun stmt -> match stmt with

                CompoundStatement(declList, _) ->

                    Semant.symbols_from_decls declList) def.anon_body in

  let param_symbols = (Semant.symbols_from_func_params def.anon_params)@internal_anon_symbols in

  let param_symtable = (Semant.symtable_from_symlist param_symbols) in

  let updated_symbols = Semant.symtable_from_symlist (builtinSyms@symlist) in

    {

      cstruct_name = "S" ^ def.anon_name; (* 's' for 'struct' *)

          cstruct_members = (List.map symconvert (struct_members_from_anon_body updated_symbols
param_symtable [] def.anon_body));

      cmethod_to_functions = StringMap.empty

    }




and capture_struct_list_from_anon_def_list program defList = match defList with

      [] -> []

    | [x] -> [capture_struct_from_anon_def program x]

       | h::t -> [capture_struct_from_anon_def program h]@capture_struct_list_from_anon_def_list
program t



and cFunc_from_tMethod cStruct_Name tFuncName = String.concat "_" [cStruct_Name;tFuncName]



and cStruct_from_tStruct symbol_table tStruct =
```

```
        let symconvert m = cSymbol_from_sSymbol symbol_table m in

                    let defaultStructMemberSymbols = List.map symconvert (List.map
(Semant.symbol_from_declaration) tStruct.members) in


        (* If there is an interface then add a struct member corresponding to

         * the interface to our struct *)

        let cStructMemberSymbols = if (Semant.get_interface_for_struct

        tStruct.struct_name symbol_table <> "") then

                [cSymbol_from_Implements tStruct.implements] @

                defaultStructMemberSymbols else defaultStructMemberSymbols in


                    let (methods_to_cfunctions, cfuncs) = (List.fold_left (fun (sym, cfunc_list)
method_ ->

                                            (let tfunc_name =

                                                    Semant.var_name_from_direct_declarator

                                               method_.func_name in


                                           let initial_void_param =

                                                        create_cfunc_param_for_receiver

                                                        method_.receiver in


                                           let init_cast_decl =

                                                    create_initial_cast_decl

                                                    method_.receiver in



                                            let cfunc = {

                                                 creturn_type = (cType_from_tType

                                                 symbol_table

                                                 (Semant.type_from_declaration_specifiers

                                                 method_.return_type));


                                                 cfunc_params =
```

```
                                                [initial_void_param] @ (List.map
                                            (cFuncParam_from_tFuncParam
                                            symbol_table) method_.params);

                                            cfunc_body =
                                                    CCompoundStatement([init_cast_decl],
                                            []);

                                            cfunc_name = cFunc_from_tMethod
                                            (cStructName_from_tStruct
                                            tStruct.struct_name) tfunc_name;

                                        } in (StringMap.add tfunc_name cfunc
                                        sym, cfunc_list @ [cfunc])))
                                           (StringMap.empty, []) tStruct.methods) in

        ({
                cstruct_name = cStructName_from_tStruct tStruct.struct_name;
                cstruct_members = cStructMemberSymbols;
                cmethod_to_functions = methods_to_cfunctions;
        }, cfuncs)

let cDeclarationSpecifiers_from_tDeclarationSpecifiers symbol_table tDeclSpecs = function
        | DeclSpecTypeSpecAny(tType) ->
                        CDeclSpecTypeSpecAny(cType_from_tType symbol_table tType)

let cDeclaration_from_tFdecl symbol_table fdecl =
        let first_argument = [CPointerType(CType(CPrimitiveType(Cvoid)), 1)] in
        let cfunc_param_types = first_argument @ List.map (cType_from_tType symbol_table)
                                    (Semant.type_list_from_func_param_list fdecl.params) in
        let generate_void_star_param_types =
            let anonList =
```

```
            List.filter (fun p -> match p with
                AnonFuncDecl(anonDecl) -> true
              | _ -> false)fdecl.params
        in
        let returned_param_types =
            List.map (fun p -> match p with
                AnonFuncDecl(anonDecl) ->
                    CPointerType(CType(CPrimitiveType(Cvoid)), 1)) anonList
        in
        returned_param_types
    in
    let extraParamTypes = generate_void_star_param_types in
    let cfunc_return_type =
            cType_from_tType symbol_table (Semant.type_from_declaration_specifiers
            fdecl.return_type) in
    let func_signature = {
                            func_return_type = cfunc_return_type;
                            func_param_types = cfunc_param_types@extraParamTypes;
                           } in

                 CVarSymbol((Semant.var_name_from_direct_declarator  fdecl.func_name),
CFuncPointer(func_signature))

(* The C Struct corresponding to the Cimple Interface consists of
 * 1) Function pointers instead of methods. The first argument is a void star *)

let cStruct_from_tInterface symbol_table interface =
    let cBodySymbol = [CVarSymbol("body",
    CPointerType(CType(CPrimitiveType(Cvoid)),
    1))] in (* This is the void * body that we apply to all the functions *)
    let bols = List.map (cDeclaration_from_tFdecl symbol_table) interface.funcs in
    {
```

```
                cstruct_members = cBodySymbol @ bols;

                cstruct_name = cStructName_from_tInterface interface.name;

                cmethod_to_functions = StringMap.empty

        }


let bol_from_Implements implements struct_name =

        let cstruct_name = cStructName_from_tInterface implements in

        let interface_field_name = interface_field_name_in_struct implements

        struct_name in

        CVarSymbol(interface_field_name, CType(CStruct(cstruct_name)))


let cFuncParam_from_tFuncParam symbol_table tFuncParam =

    match tFuncParam with

       AnonFuncDecl(_) ->

           let newName = "anon_" ^ (Semant.var_name_from_func_param tFuncParam) in

                        (cType_from_tType  symbol_table  (Semant.type_from_func_param  tFuncParam),
(CIdentifier(newName)))

     | _ ->

                    (cType_from_tType  symbol_table  (Semant.type_from_func_param  tFuncParam),
(CIdentifier(Semant.var_name_from_func_param tFuncParam)))


let create_cfunc_param_for_receiver receiver =

        (CPointerType(CType(CPrimitiveType(Cvoid)), 1),

        CIdentifier("_body"))




let create_initial_cast_decl receiver =

        let cstruct_name = cStructName_from_tStruct (fst receiver) in

        CDeclaration(CDeclSpecTypeSpecAny(CPointerType(CType(CStruct(cstruct_name)),

        1)), CInitDeclaratorAsn(CDirectDeclarator(CVar(CIdentifier(snd receiver)))

        , Asn, CCastExpr(CPointerType(CType(CStruct(cstruct_name)), 1),

        CId(CIdentifier("_body")))))
```

```
let cFunc_from_tFunc symbol_table tFunc =

    let generate_n_void_star_params n =

        let anonList =

            List.filter (fun p -> match p with

                AnonFuncDecl(anonDecl) -> true

              | _ -> false) tFunc.params

        in

        let returned_params =

            List.map (fun p -> match p with

                AnonFuncDecl(anonDecl) ->

                    (match anonDecl.anon_decl_name with

                        Identifier(s) ->

                            let paramName = "cap_anon_" ^ s in

                                                    (CPointerType(CType(CPrimitiveType(Cvoid)), 1),
CIdentifier(paramName)))) anonList

        in

        returned_params

    in

    let n = number_of_anon_func_parameters_in_tFuncDecl tFunc in

    let extraParams = generate_n_void_star_params n in

        {

                creturn_type = cType_from_tType symbol_table

                (Semant.type_from_declaration_specifiers tFunc.return_type);


                cfunc_params =List.rev (((extraParams)@List.map (cFuncParam_from_tFuncParam

                symbol_table) tFunc.params));


                cfunc_body = CCompoundStatement([], []);


                cfunc_name = Semant.var_name_from_direct_declarator tFunc.func_name;

        }
```

```
let cFunc_from_tMethod tStructName tFuncName =
        Semant.symbol_table_key_for_method tStructName tFuncName

let first_param_for_constructor struct_name =
        (CPointerType(CType(CStruct(cStructName_from_tStruct struct_name)), 2),
        CIdentifier("_this"))

let first_param_for_destructor struct_name =
        (CPointerType(CType(CStruct(cStructName_from_tStruct struct_name)), 2),
        CIdentifier("_this"))

let last_param_for_constructor =
        (CType(CPrimitiveType(Cint)), CIdentifier("_needs_malloc"))

let last_param_for_destructor =
        (CType(CPrimitiveType(Cint)), CIdentifier("_needs_free"))



and cFunction_from_tMethod object_type method_ tSymbol_table =
        match object_type  with
        | CustomType(name) -> ( let typ_symbol  =
                Semant.lookup_symbol_by_id tSymbol_table (Identifier(name)) in match
                typ_symbol with
                | StructSymbol(_, _) -> cFunc_from_tMethod
                (name) method_
                | InterfaceSymbol(_, _) -> method_)
        | PointerType(CustomType(name), 1) -> ( let typ_symbol  =
                Semant.lookup_symbol_by_id tSymbol_table (Identifier(name)) in match
                typ_symbol with
                | StructSymbol(_, _) -> cFunc_from_tMethod
                ( name) method_
                | InterfaceSymbol(_, _) -> method_)
```

```
        | _ -> raise(Failure("not done"))
(*
 * This function takes a cimple expression and returns the pair ((C expression,
 * statement[]), cDeclaration[]). The idea is that some cimple expression may generate multiple
 * assignment statements such as a make expression with a constructor.
 * Expressions should not create more declarations, only declarations create
 * more declarations.
 *)
let rec update_expr texpr tSymbol_table tprogram  = match texpr with
        | Neg(e) -> (let ((updated_e1, e1_stmts), decls) = update_expr e
                        tSymbol_table tprogram in ((CNeg(updated_e1), e1_stmts), decls))
        | Binop(e1, op, e2) -> (let ((updated_e1, e1_stmts), _) = update_expr e1
                                tSymbol_table tprogram in
                        let ((updated_e2, e2_stmts), _) = update_expr e2
                        tSymbol_table  tprogram in


                        ((CBinop(updated_e1, op, updated_e2), (e1_stmts@e2_stmts)), []))
        | CompareExpr(e1, op, e2) ->
                (
                let ((updated_e1, e1_stmts), _) = update_expr e1 tSymbol_table tprogram in
                let ((updated_e2, e2_stmts), _) = update_expr e2 tSymbol_table tprogram in
                ((CCompareExpr(updated_e1, op, updated_e2), (e1_stmts @ e2_stmts)), [])
                )
        | Clean(e) -> (let t1 = Semant.type_from_expr tSymbol_table e in
                        let ((updated_e, e_stmts), decls) = update_expr e tSymbol_table tprogram in
                        match t1 with
                        | PointerType(CustomType(s), _) -> ((CCall(0, CNoexpr,
                                        CId(CIdentifier(destructor_name_from_tStruct s)),
[CCastExpr(CPointerType(CType(CStruct(cStructName_from_tStruct
                                        s)), 2), CPointify(updated_e))] @
                          [CLiteral(1)]), []), [])
```

```
                    | PointerType(_, _) -> ((CFree(updated_e), e_stmts),
                    decls))



        | AsnExpr(e1, op, e2)  ->
                (match e2 with
                           | Make(typ_, expr_list) -> cAllocExpr_from_tMakeExpr tSymbol_table
tprogram e1 e2
                      | _ -> (
                          let ((updated_e1, e1_stmts), _) = update_expr e1
                                  tSymbol_table tprogram in
                          let ((updated_e2, e2_stmts), _) = update_expr e2
                          tSymbol_table tprogram  in

                          let e1_type = Semant.type_from_expr tSymbol_table e1 in

                          let e2_type = Semant.type_from_expr tSymbol_table e2 in

                          (match (e1_type, e2_type) with

                          (* Check if we are assigning custom types. Since we are
                           * past semantic analysis the only possibilities are 1.
                           * we are assining a derived class to its ancestor or 2.
                           * we are assigning the same types. In those cases we
                           * need to cast *)

                          | (PointerType(CustomType(s), _),
                          PointerType(CustomType(t), _)) ->
                              (if (Semant.t1_inherits_t2 s t tSymbol_table) then
                                  ((CAsnExpr(updated_e1, op,
                                             CCastExpr((cType_from_tType tSymbol_table e1_type),
updated_e2)),
```

```
                                        (e1_stmts@e2_stmts)), [])


                                    else

                                            ((CAsnExpr(updated_e1, op, updated_e2), e1_stmts

                                            @ e2_stmts), []))

                                | (CustomType(s), CustomType(t)) -> (if

                                        (Semant.t1_inherits_t2 s t tSymbol_table) then

                                        ((CAsnExpr(updated_e1, op,

                                        CCastExpr((cType_from_tType tSymbol_table

                                        e1_type), updated_e2)), (e1_stmts@e2_stmts)), [])


                                        else

                                                ((CAsnExpr(updated_e1, op, updated_e2), e1_stmts

                                                @ e2_stmts), []))


                                | _ -> ((CAsnExpr(updated_e1, op, updated_e2), e1_stmts

                                @ e2_stmts), [])))))


     | Call(expr, Id(Identifier(s)), expr_list) ->

                           let ret = cCallExpr_from_tCallExpr expr tSymbol_table tprogram s expr_list
in

                        ret

     | Super(_) -> ((CNoexpr, []), [])

     | ArrayAccess(e1, e2) -> ((CArrayAccess(fst(fst(update_expr e1 tSymbol_table

     tprogram)), fst(fst(update_expr e2 tSymbol_table tprogram))), []), [])

     | MemAccess(expr, Identifier(s)) -> (let typ_ = Semant.type_from_expr

     tSymbol_table expr in match (typ_) with

                | CustomType(name) -> ((CMemAccess(0, fst( fst (update_expr expr

                tSymbol_table tprogram )), CIdentifier(s)), []), [])

                | PointerType(CustomType(name), 1) -> ((CMemAccess(1, fst(fst

                (update_expr expr tSymbol_table tprogram )), CIdentifier(s)),

                []), [])
```

```
                | _ -> raise(Failure("Bad Mem Access")))
     | Id(Identifier(s)) -> ((CId(CIdentifier(s)), []), [])
     | Literal(d) -> ((CLiteral(d), []), [])
     | Make(typ_, expr_list) -> (let ctype = cType_from_tType tSymbol_table typ_
                 in match typ_ with
                | PrimitiveType(s) -> ((CAlloc(ctype,
                CId(CIdentifier((sizeof_string tSymbol_table ctype)))), []), [])
                | ArrayType(array_type, ptr, e) ->(let updated_e = fst(fst(update_expr
                                                e tSymbol_table tprogram)) in let pointer_type =
Semant.type_of_array_type
       tSymbol_table typ_ in let cpointer_type = cType_from_tType tSymbol_table
       pointer_type in (match cpointer_type with
       | CPointerType(base, num) ->let ctype_to_malloc = if (num = 1) then base
       else CPointerType(base, num-1) in (
       CAlloc(base, CBinop(updated_e, Mul,
       (CId(CIdentifier(sizeof_string tSymbol_table ctype_to_malloc)))))), []),
       []))


                | CustomType(s) -> ((CAlloc(ctype, CId(CIdentifier(sizeof_string tSymbol_table
               ctype))), []), []))
     | FloatLiteral(d) -> ((CFloatLiteral(d), []), [])
     | StringLiteral(s) -> ((CStringLiteral(s), []), [])
     | Postfix(e1, op) -> (let ((updated_e1, e1_stmts), _) = update_expr e1
                               tSymbol_table  tprogram in ((CPostfix(updated_e1, op), e1_stmts),
[]))
     | AnonFuncDef(anonDef) ->
            let anon_name = Semant.find_name_for_anon_def tprogram anonDef in
            let instanceName = "s" ^ anon_name in
            let structName = "S" ^ anon_name in
                    let decls = [CDeclaration(CDeclSpecTypeSpecAny(CType(CStruct(structName))),
CInitDeclarator(CDirectDeclarator(CVar(CIdentifier(instanceName)))))] in
            let assignments_from_capture_struct c =
                List.map (fun csym ->
```

```
                    (match csym with

                        CVarSymbol(s, _) ->

                                    CExpr(CAsnExpr(CMemAccess(0, CId(CIdentifier(instanceName)),
CIdentifier(s)), Asn, CId(CIdentifier(s))))

                                | _ -> raise(Failure("update_expr: Invalid CSymbol parameter"))))
c.cstruct_members

             in


             let captures = capture_struct_from_anon_def tprogram anonDef in


             let newAssignments = assignments_from_capture_struct captures in

             ((CId(CIdentifier(anon_name)), newAssignments), decls)

     | Noexpr -> ((CNoexpr, []), [])

     | Pointify(e) -> let ((updated_e, stmts), decls) = update_expr e

     tSymbol_table tprogram in ((CPointify(updated_e), stmts), decls)

     | Deref(e) -> let ((updated_e, stmts), decls) = update_expr e tSymbol_table

     tprogram in ((CDeref(updated_e), stmts), decls)

     | Nil -> ((CNull, []), [])

     | _ ->

             let expr_type = Astutil.string_of_expr texpr in

             raise(Failure("not finished for type " ^ expr_type))


and update_expr_list texpr_list tSymbol_table tprogram = match texpr_list with

    [] -> []

  | [e] -> [update_expr e tSymbol_table tprogram]

  | h::t -> [update_expr h tSymbol_table tprogram]@update_expr_list t tSymbol_table tprogram


and generate_stmts_for_parent_destructor symbol_table tprogram destructor

tStruct =


        let ancestor_destructor = Semant.get_ancestors_destructor symbol_table

        tStruct in
```

```
        let c_ancestor_destructor_name = destructor_name_from_tStruct
        ancestor_destructor.destructor_name in

        let first_arg = CCastExpr(CPointerType(cType_from_tType symbol_table
        (CustomType(ancestor_destructor.destructor_name)), 2),
        CId(CIdentifier("_this"))) in

        let last_arg = CLiteral(0) in

        [(CExpr(CCall(0, CNoexpr,
        CId(CIdentifier(c_ancestor_destructor_name)), [first_arg]@[last_arg])))]

and generate_stmts_for_super expr_list symbol_table tprogram constructor tStruct =
        let ancestor_constructor = Semant.get_ancestors_constructor
        symbol_table tStruct in

        let c_ancestor_constructor_name = constructor_name_from_tStruct
        ancestor_constructor.constructor_name in

        let first_arg = CCastExpr(CPointerType(cType_from_tType symbol_table
        (CustomType(ancestor_constructor.constructor_name)), 2), CId(CIdentifier("_this")))
        in

        let last_arg = CLiteral(0) in

        let cstParams = constructor.constructor_params in
        let funcParamSymbols = Semant.symbols_from_func_params cstParams in
        let anonParamSymbols = List.filter (fun sym ->
            (match sym with
                AnonFuncSymbol(_, _) -> true
              | _ -> false)) funcParamSymbols
        in
```

```
let anonParamSymbolTable = Semant.symtable_from_symlist anonParamSymbols in
let is_in_symtable id =
    try
        StringMap.mem id anonParamSymbolTable
    with
        _ -> false
in
let fixedExprList =
    List.map (fun e ->
        (match e with
            Id(Identifier(id)) ->
                if ((is_in_symtable id) = true) then
                    Id(Identifier("anon_" ^ id))
                else
                    e)) expr_list
in
let capIds =
    List.fold_left (fun accList expr ->
        (match expr with
            Id(Identifier(id)) ->
                if ((is_in_symtable id) = true) then
                    accList@[CId(CIdentifier("cap_anon_" ^ id))]
                else
                    accList
          | _ -> raise(Failure("Invalid expr")))) []  expr_list
in
let fixedAnonParamSymbols =
    let fix_anon_name str =
            try
                let sub = String.sub str 0 5 in
                if (sub <> "anon_") then
                    "anon_" ^ str
```

```
                        else

                            str

                    with

                        _ -> "anon_" ^ str

            in

            List.map (fun sym ->

                (match sym with

                    AnonFuncSymbol(anonName, t) -> AnonFuncSymbol((fix_anon_name anonName), t)

                  | _ -> sym)) anonParamSymbols

        in

                    let  symbol_table  =  Semant.add_symbol_list_to_symbol_table  symbol_table
fixedAnonParamSymbols in

        let call_to_super_constructor_stmt = [(CExpr(CCall(0, CNoexpr, CId(CIdentifier(

                c_ancestor_constructor_name)), [first_arg]@(List.map2

                                (cExpr_from_tExpr_in_tCall symbol_table tprogram) fixedExprList

                                ancestor_constructor.constructor_params)@[last_arg]@capIds)))] in

                let  StructSymbol(_,  ancestor_struct)  =  Semant.lookup_symbol_by_id  symbol_table
(Identifier(ancestor_constructor.constructor_name)) in


            let local_reassignment_of_members = List.fold_left (fun assignments

            member -> let member_id = Semant.var_name_from_declaration member

            in assignments @ [CExpr(CAsnExpr(CId(CIdentifier(member_id)), Asn, CMemAccess(1,

            CDeref(CId(CIdentifier("_this"))),

            (CIdentifier(member_id))))))]) [] ancestor_struct.members in


            call_to_super_constructor_stmt @ local_reassignment_of_members




and cAllocExpr_from_tMakeExpr tSymbol_table tprogram asn_expr tMakeExpr =

        let ((updated_e1, updated_stmts), _) = (update_expr asn_expr tSymbol_table tprogram ) in

        let Make(typ_, expr_list) = tMakeExpr in

        match typ_ with

        | CustomType(typ) -> (
```

```
                let StructSymbol(name, tStruct) = Semant.lookup_symbol_by_id

                tSymbol_table (Identifier(typ)) in


                if (tStruct.constructor.constructor_name <> "") then (

                        (* We have a constructor *)

                        let params = tStruct.constructor.constructor_params in


                                                                let more_params_filtered =
generate_extra_capture_func_params_from_expr_list tSymbol_table tprogram expr_list in

                        let updated_expr_list = (List.map2

                        (cExpr_from_tExpr_in_tCall tSymbol_table tprogram) expr_list params) in


                           let anonParams = Semant.anon_defs_from_expr_list_no_recursion tprogram
expr_list in

                        let update_anon_def_expr_list anonList =

                            List.fold_left (fun ((e, slist), dlist) def ->

                                       let ((_, _slist), _dlist) = update_expr (AnonFuncDef(def))
tSymbol_table tprogram in

                                       ((Noexpr, slist@_slist), dlist@_dlist)) ((Noexpr, []), [])
anonList

                        in

                                            let ((updated_expr, updated_slist), updated_dlist) =
update_anon_def_expr_list anonParams in


                       ((CCall(0, CNoexpr,

                        CId(CIdentifier(constructor_name_from_tStruct

                        tStruct.struct_name)),

                        [CCastExpr(CPointerType(CType(CStruct(cStructName_from_tStruct

                        tStruct.struct_name)), 2), CPointify(updated_e1))] @

                          updated_expr_list @ [CLiteral(1)]@more_params_filtered), updated_slist),
updated_dlist)

                ) else (

                       ((CCall(0, CNoexpr,

                        CId(CIdentifier(constructor_name_from_tStruct

                        tStruct.struct_name)),
```

```
                        [CCastExpr(CPointerType(CType(CStruct(cStructName_from_tStruct
                        tStruct.struct_name)), 2), CPointify(updated_e1))] @
                        [CLiteral(1)]), []), [])
                ))
        | (ArrayType(array_type, ptr, e)) -> (let updated_e = fst(fst(update_expr
                                            e tSymbol_table tprogram)) in let pointer_type =
Semant.type_of_array_type
        tSymbol_table typ_ in let cpointer_type = cType_from_tType tSymbol_table
        pointer_type in (match cpointer_type with
        | CPointerType(base, num) ->let ctype_to_malloc = if (num = 1) then base
        else CPointerType(base, num-1) in ((CAsnExpr(updated_e1, Asn,
        CAlloc(base, CBinop(updated_e, Mul,
        (CId(CIdentifier(sizeof_string tSymbol_table ctype_to_malloc)))))), []),
        [])))



and cExpr_from_tExpr_in_tCall tSymbol_table  tprogram  tExpr tFuncParam =
      let expr_type = Semant.type_from_expr tSymbol_table tExpr in
      let param_type = Semant.type_from_func_param tFuncParam in
        match (expr_type, param_type) with
            |  (CustomType(a), CustomType(b)) ->
                    if (Semant.is_interface tSymbol_table (Identifier(b))) then
                                CPointify(CMemAccess(0, fst( fst (update_expr tExpr tSymbol_table
tprogram )),
                            CIdentifier(interface_field_name_in_struct b a)))
                    else ( if (Semant.t1_inherits_t2 a b tSymbol_table) then
                        CCastExpr(CType(CStruct(cStructName_from_tStruct b)),
                                        fst (fst (update_expr tExpr tSymbol_table  tprogram )))
                    else fst (fst (update_expr tExpr tSymbol_table  tprogram )))
            | (PointerType(CustomType(a), 1),
            CustomType(b)) ->  CPointify(CMemAccess(1, fst( fst (update_expr tExpr
            tSymbol_table  tprogram )),
            CIdentifier(interface_field_name_in_struct b a)))
```

```
            | _ -> fst (fst (update_expr tExpr tSymbol_table tprogram )

            )


and generate_extra_capture_func_params_from_expr_list tSym tprogram expr_list =

        let anonParams = Semant.anon_defs_from_expr_list_no_recursion tprogram expr_list in

        let capture_struct_instance_name_from_anon_def def =

            let capStruct = capture_struct_from_anon_def tprogram def in

                            let  subname  =  String.sub  capStruct.cstruct_name  1  ((String.length
capStruct.cstruct_name) - 1) in

            let structname = "s" ^ subname in

            structname

        in

        let capture_params_from_anon_def_list defList =

            List.fold_left (fun elist def ->

                    elist@[CPointify(CId(CIdentifier((capture_struct_instance_name_from_anon_def
def))))]

            ) [CNoexpr] defList

        in

        let more_params = capture_params_from_anon_def_list anonParams in

        let remove_noexpr_from_list elist =

            List.filter (fun e ->

                match e with

                    CNoexpr -> false

                  | _ -> true) elist

        in

        let more_params_filtered = remove_noexpr_from_list more_params in

        more_params_filtered


and cCallExpr_from_tCallExpr expr tSym  tprogram func_name expr_list =

    match expr with

        | Noexpr -> let sym = StringMap.find func_name tSym in

                    (match sym with

                        FuncSymbol(_, fdecl) ->
```

```
                            let hasAnonParams = Semant.expr_list_contains_anon_defs_no_recursion
expr_list in

                        if (hasAnonParams = true) then

                            let update_anon_def_expr_list anonList =

                                List.fold_left (fun ((e, slist), dlist) def ->

                                                    let ((_, _slist), _dlist) = update_expr
(AnonFuncDef(def)) tSym tprogram in

                                            ((Noexpr, slist@_slist), dlist@_dlist)) ((Noexpr, []),
[]) anonList

                            in

                                                                let more_params_filtered =
generate_extra_capture_func_params_from_expr_list tSym tprogram expr_list in

                                let anonParams = Semant.anon_defs_from_expr_list_no_recursion
tprogram expr_list in

                                    let ((updated_expr, updated_slist), updated_dlist) =
update_anon_def_expr_list anonParams in

                            let paramExpressions = List.rev (List.map2

                                    (cExpr_from_tExpr_in_tCall tSym  tprogram ) expr_list
fdecl.params) in



                            let ret =

                                        ((CCall(0, CNoexpr, CId(CIdentifier(func_name)),
paramExpressions@more_params_filtered) , updated_slist), updated_dlist)

                            in

                            ret;

                        else

                            if (func_name = "printf") then

                                let expr_list = (List.rev expr_list) in

                                                                let replacementParamList =
Semant.func_param_list_from_expr_list tSym expr_list in

                                ((CCall(0, CNoexpr, CId(CIdentifier(func_name)), (List.map2

                                        (cExpr_from_tExpr_in_tCall tSym  tprogram ) expr_list
replacementParamList)), []), [])

                            else

                                let ret =

                                ((CCall(0, CNoexpr, CId(CIdentifier(func_name)), (List.map2
```

```
                                        (cExpr_from_tExpr_in_tCall tSym  tprogram ) expr_list
fdecl.params)), []), [])

                                    in

                                    ret

                        | AnonFuncSymbol(anonName, AnonFuncType(_, tlist)) ->

                                let rec funcParam_from_tType t = (match t with

                                    _ -> ParamDeclWithType(DeclSpecTypeSpecAny(t)))


                                  and funcParamList_from_tTypeList tlist = (match tlist with

                                      [] -> []

                                    | [x] -> [funcParam_from_tType x]

                                                                        | h::t -> [funcParam_from_tType
h]@(funcParamList_from_tTypeList t))

                                in

                                let fParamList = funcParamList_from_tTypeList tlist in

                                    let fParamExprList = (List.map2 (cExpr_from_tExpr_in_tCall tSym
tprogram ) expr_list fParamList) in

                                let extraParamName = "cap_anon_" ^ func_name in

                                let extraParamExpr = CId(CIdentifier(extraParamName)) in

                                        ((CCall(1, CNoexpr, CId(CIdentifier("anon_" ^ func_name)),
fParamExprList@[extraParamExpr]), []), []))


        | _ -> let expr_type = Semant.type_from_expr tSym expr in (match expr_type with

                | CustomType(a) -> let fdecl = Semant.get_fdecl_for_receiver a

                tSym func_name in


                        if (Semant.is_interface tSym (Identifier(a))) then

                                let updated_expr = (fst (fst (update_expr expr tSym tprogram )))

                                in

                                let cexpr_list = [CMemAccess(1,

                                (updated_expr), CIdentifier("body"))] @

                                (List.map2 (cExpr_from_tExpr_in_tCall tSym tprogram )

                                expr_list fdecl.params) in
```

```
                                ((CCall(1, (fst (fst (update_expr expr
                                tSym tprogram))),
                                CId(CIdentifier(func_name)), cexpr_list), []),
                                [])
                            else
                                 let hasAnonParams = Semant.expr_list_contains_anon_defs_no_recursion
expr_list in
                                if (hasAnonParams = true) then
                                    let update_anon_def_expr_list anonList =
                                        List.fold_left (fun ((e, slist), dlist) def ->
                                                                let ((_, _slist), _dlist) = update_expr
(AnonFuncDef(def)) tSym tprogram in
                                                ((Noexpr, slist@_slist), dlist@_dlist)) ((Noexpr, []),
[]) anonList
                                    in
                                                                                        let  extra_params  =
generate_extra_capture_func_params_from_expr_list tSym tprogram expr_list in
                                         let anonParams = Semant.anon_defs_from_expr_list_no_recursion
tprogram expr_list in
                                              let ((updated_expr, updated_slist), updated_dlist) =
update_anon_def_expr_list anonParams in
                                    let first_arg =
                                            CCastExpr(CPointerType(CType(CPrimitiveType(Cvoid)),
                                            1), CPointify(fst(fst (update_expr expr
                                            tSym tprogram
                                            )))) in
                                    ((CCall(1, CMemAccess(0, fst (fst( (update_expr expr
                                    tSym tprogram))), CIdentifier("_virtual")),
                                    CId(CIdentifier(
                                    func_name)), [first_arg] @ (List.map2
                                    (cExpr_from_tExpr_in_tCall tSym  tprogram ) expr_list
                                    fdecl.params)@extra_params), updated_slist), updated_dlist)
                                else
                                    let first_arg =
                                            CCastExpr(CPointerType(CType(CPrimitiveType(Cvoid)),
```

```
                                        1), CPointify(fst(fst (update_expr expr
                                        tSym tprogram
                                        )))) in
                                    ((CCall(1, CMemAccess(0, fst (fst( (update_expr expr
                                    tSym tprogram))), CIdentifier("_virtual")),
                                    CId(CIdentifier(
                                    func_name)), [first_arg] @ (List.map2
                                    (cExpr_from_tExpr_in_tCall tSym  tprogram ) expr_list
                                    fdecl.params)),[]), [])
                | PointerType(CustomType(a), 1) ->
                                    let hasAnonParams = Semant.expr_list_contains_anon_defs_no_recursion
expr_list in
                            if (hasAnonParams = true) then
                                let fdecl =
                                Semant.get_fdecl_for_receiver a tSym func_name in
                                        let first_arg =
                                                CCastExpr(CPointerType(CType(CPrimitiveType(Cvoid)),
                                                1), fst (fst (update_expr expr tSym
                                                tprogram
                                                ))) in
                                let update_anon_def_expr_list anonList =
                                    List.fold_left (fun ((e, slist), dlist) def ->
                                            let ((_, _slist), _dlist) = update_expr (AnonFuncDef(def))
tSym tprogram in
                                            ((Noexpr, slist@_slist), dlist@_dlist)) ((Noexpr, []), [])
anonList
                                in
                                  let extra_params = generate_extra_capture_func_params_from_expr_list
tSym tprogram expr_list in
                                let anonParams = Semant.anon_defs_from_expr_list_no_recursion tprogram
expr_list in
                                            let ((updated_expr, updated_slist), updated_dlist) =
update_anon_def_expr_list anonParams in
                                ((CCall(1, CMemAccess(1,
                                fst (fst(update_expr expr tSym tprogram)),
```

```
                        CIdentifier("_virtual")),
                    CId(CIdentifier(
                        func_name)), [first_arg] @ (List.map2
                        (cExpr_from_tExpr_in_tCall tSym  tprogram ) expr_list
                        fdecl.params@extra_params)), updated_slist), updated_dlist)
                else
                    let fdecl =
                    Semant.get_fdecl_for_receiver a tSym func_name in
                            let first_arg =
                                    CCastExpr(CPointerType(CType(CPrimitiveType(Cvoid)),
                                    1), fst (fst (update_expr expr tSym
                                    tprogram
                                    ))) in

                                    ((CCall(1, CMemAccess(1,
                                    fst (fst(update_expr expr tSym tprogram)),
                                    CIdentifier("_virtual")),
                                   CId(CIdentifier(
                                    func_name)), [first_arg] @ (List.map2
                                    (cExpr_from_tExpr_in_tCall tSym  tprogram ) expr_list
                                    fdecl.params)), []), [])
            | _ -> raise(Failure("No other functions can call methods")))

let generate_virtual_table_assignments isPointer tStruct tSymbol_table id=
        let fdecls = Semant.get_unique_method_names_for_struct tSymbol_table tStruct in

         List.map (fun tmethod_name ->

               let inter_fdecl = Semant.get_fdecl_for_receiver
                                    tStruct.struct_name tSymbol_table
                                    tmethod_name in
```

```
                let cFunc_name = cFunc_from_tMethod (fst(inter_fdecl.receiver)) tmethod_name in


                if (isPointer >= 0) then
                                        CExpr(CAsnExpr(CMemAccess(isPointer,CId(CIdentifier(id)),
CIdentifier(tmethod_name)),
                      Asn,
                     (CId(CIdentifier(cFunc_name)))))
                else
                                        CExpr(CAsnExpr(CMemAccess(1,CDeref(CId(CIdentifier(id))),
CIdentifier(tmethod_name)),
                      Asn,
                     (CId(CIdentifier(cFunc_name))))))
          fdecls




let c_init_decl_from_string str =
       CInitDeclarator(CDirectDeclarator(CVar(CIdentifier(str))))


let c_init_decl_from_string_asn str op cExpr =
       CInitDeclaratorAsn(CDirectDeclarator(CVar(CIdentifier(str))), op, cExpr)




let generate_decls_and_stmts_from_id tSymbol_table tprogram id decl typ_ =
       match decl with
       | Declaration(_, InitDeclList([InitDeclarator(_)])) ->
                      let ctype = cType_from_tType tSymbol_table typ_ in
                      let cinit_decl = c_init_decl_from_string id in
                      ([CDeclaration(CDeclSpecTypeSpecAny(ctype),
                      cinit_decl)], [])


       | Declaration(_, InitDeclList([InitDeclaratorAsn(_, op, expr)])) ->
                      let ((updated_expr, extra_stmts), _) = update_expr expr
```

```
                        tSymbol_table  tprogram in

                        let ctype = cType_from_tType tSymbol_table typ_ in
                        let cinit_decl = c_init_decl_from_string_asn id op
                        updated_expr in

                        ([CDeclaration(CDeclSpecTypeSpecAny(ctype),
                        cinit_decl)], extra_stmts)


let update_decl_for_non_custom_type id decl tSymbol_table  tprogram =
        let sym = Semant.lookup_symbol_by_id tSymbol_table (Identifier(id)) in (match
        sym with
        | VarSymbol(id, type_) -> (match decl with
                        | Declaration(_, InitDeclList([InitDeclarator(DirectDeclarator(_))])) ->
                                generate_decls_and_stmts_from_id tSymbol_table tprogram
                                id decl type_
                        | Declaration(_, InitDeclarator(DirectDeclarator(_))) ->
                                generate_decls_and_stmts_from_id tSymbol_table tprogram
                                id decl type_
                        | Declaration(_, InitDeclaratorAsn(declrt, _, _)) ->
                                generate_decls_and_stmts_from_id tSymbol_table tprogram
                                id decl type_
                        | _ ->
                                generate_decls_and_stmts_from_id tSymbol_table tprogram
                                id decl type_)
        | FuncSymbol(_, _) -> raise(Failure("update_decl: FuncSymbol not supported"))
        | AnonFuncSymbol(_, _) -> raise(Failure("update_decl: AnonFuncSymbol not supported"))
        )

let declare_virtual_table_stack tSymbol_table tStruct id =
        let virtual_table_name = virtual_table_name_from_tStruct
```

```
        tStruct.struct_name in


        let virtual_table_id = String.concat "" ["_";id;virtual_table_name] in

        let virtual_table_init_decl = c_init_decl_from_string (String.concat ""
        ["_"; id; (virtual_table_name)]) in

                let virtual_table_assignments = generate_virtual_table_assignments 0 tStruct
tSymbol_table
        virtual_table_id in

        let assign_virtual_table_back_to_id = CExpr(CAsnExpr(CMemAccess(0,
        CId(CIdentifier(id)),
        CIdentifier("_virtual")), Asn,
        CPointify(CId(CIdentifier(virtual_table_id))))) in

        ([CDeclaration(CDeclSpecTypeSpecAny(CType(CStruct(virtual_table_name))),
        virtual_table_init_decl)], virtual_table_assignments @
        [assign_virtual_table_back_to_id])

let update_interface_decl_for_struct id struct_name tSymbol_table =
        let interface = Semant.get_interface tSymbol_table
        (Semant.get_interface_for_struct struct_name tSymbol_table) in

        let cstruct_name_for_interface = cStructName_from_tInterface
        interface.name in

        let cinit_decl = c_init_decl_from_string (String.concat ""
        ["_";id;(cStructName_from_tInterface interface.name)]) in

        CDeclaration(CDeclSpecTypeSpecAny(CType(CStruct(cstruct_name_for_interface))),
```

```
                                                cinit_decl)

let interface_decl_and_assignments_for_struct isPointer struct_ tSymbol_table id

=

        let custom_type = struct_.struct_name in


        let implements = Semant.get_interface_for_struct custom_type tSymbol_table in


        if (implements <> "") then (
                    let interface = Semant.get_interface tSymbol_table
                                    (Semant.get_interface_for_struct
                                     custom_type tSymbol_table) in


                    let interface_decl = update_interface_decl_for_struct
                         id custom_type tSymbol_table in


                    let cstruct_for_interface = cStructName_from_tInterface
                                          implements in


                    let access_id = String.concat "" ["_";id;cstruct_for_interface] in


                    let fdecls = (List.map (fun fdecl -> Semant.var_name_from_direct_declarator
                                                fdecl.func_name) interface.funcs) in
                    let interface_assignments  = List.map (fun tmethod_name ->
                                    let inter_fdecl =  Semant.get_fdecl_for_receiver
                                                custom_type tSymbol_table
                                                tmethod_name in


                                    let cFunc_name = cFunc_from_tMethod (fst
                                         (inter_fdecl.receiver)) tmethod_name in
```

```
                                CExpr(CAsnExpr(CMemAccess(0,CId(CIdentifier(access_id)),
                                    CIdentifier(tmethod_name)),
                                            Asn,
                                            (CId(CIdentifier(cFunc_name)))))
                                    ) fdecls in


                    let reference_implementer_asn =
                              let struct_expr = if (isPointer = 0) then
                                      CPointify(CId(CIdentifier(id))) else (if
                                              (isPointer > 0) then
                                              (CId(CIdentifier(id)))
                                      else
                                              (CDeref(CId(CIdentifier(id))))) in


                                      CExpr(CAsnExpr(CMemAccess(0,
                                      CId(CIdentifier(access_id)),
                                      CIdentifier("body")), Asn,
                                      CCastExpr(CPointerType(CType(CPrimitiveType(Cvoid)),
                                      1), struct_expr))) in


                                let interface_field_name = interface_field_name_in_struct implements
custom_type in


                    let cStruct_mem_access_expr = if (isPointer >= 0) then CMemAccess(isPointer,
                                              CId(CIdentifier(id)),
                                              CIdentifier(interface_field_name))
                    else (CMemAccess(1, CDeref(CId(CIdentifier(id))),
                    CIdentifier(interface_field_name))) in


                    let implementer_add_interface_assn =
                                                      CExpr(CAsnExpr(cStruct_mem_access_expr,
```

```
                                                Asn,
                                                CId(CIdentifier(access_id)))) in

                        let assignments = interface_assignments @
                                                [reference_implementer_asn] @
                                                [implementer_add_interface_assn]
                                                in

                        ([interface_decl],
                                        assignments)

                ) else ([], [])

let update_decl_for_custom_type id decl custom_type tSymbol_table  tprogram =
        let sym = Semant.lookup_symbol_by_id tSymbol_table
        (Identifier(custom_type)) in

        match sym with
        | StructSymbol(s, struct_) ->

                        let cstruct_type = CustomType(custom_type) in

                        let (cstruct_decl, stmts) = generate_decls_and_stmts_from_id
                        tSymbol_table  tprogram id decl cstruct_type in

                        let (virtual_table_decl, assignments) =
                                declare_virtual_table_stack tSymbol_table
                                struct_ id in

                        let (decls, assigns) =
                                interface_decl_and_assignments_for_struct 0
                                struct_ tSymbol_table id in
```

```
                        (cstruct_decl@decls@virtual_table_decl,
                        stmts@assigns@assignments)



let update_decl tSymbol_table  tprogram decl  =
                let id = Semant.var_name_from_declaration decl in
                   let tType = Semant.type_from_identifier tSymbol_table (Identifier(id))
                   in

                   match (tType) with
                   | PrimitiveType(t) -> update_decl_for_non_custom_type id decl
                   tSymbol_table tprogram
                   | CustomType(t) -> update_decl_for_custom_type id decl t
                   tSymbol_table tprogram
                   | _ -> generate_decls_and_stmts_from_id tSymbol_table  tprogram id decl tType



let rec update_statement tstmt tSymbol_table  tprogram  =  match tstmt with
        | CompoundStatement(decls, stmts) ->
                let updated_symbol_table = Semant.add_to_symbol_table tSymbol_table decls in
                let (new_decls, new_stmts) =
                    List.fold_left (fun decl_stmt_acc decl ->
                        let (n_decls, n_stmts) = update_decl
                        updated_symbol_table tprogram decl in
                          ((fst (decl_stmt_acc)) @ n_decls, (snd (decl_stmt_acc) @ n_stmts))) ([],
[]) decls in

                let more_new_stmts =
                    List.fold_left (fun stmt_acc stmt ->
                                                        let ((updated_stmt, additional_stmts),
additional_decls) =
                                                update_statement stmt
```

```
                                                updated_symbol_table  tprogram in

                                              stmt_acc @ additional_stmts @ [updated_stmt]) [] stmts
in

                let more_new_decls =

                    List.fold_left (fun decl_acc stmt ->

                                                                  let ((updated_stmt, additional_stmts),
additional_decls) =

                                                               update_statement stmt updated_symbol_table
tprogram in

                                              additional_decls@decl_acc ) [] stmts in


                  ((CCompoundStatement(new_decls@more_new_decls, new_stmts @ more_new_stmts), []),
[])


        | EmptyElse -> ((CEmptyElse, []), [])

        | Return(e) -> let ((updated_e, stmts), newDecls) = update_expr e tSymbol_table tprogram

         in ((CReturn(updated_e), stmts), newDecls)

        | If(e, stmt1, stmt2) ->

                let ((updated_expr, stmts), decls) = update_expr e tSymbol_table  tprogram  in

                     let ((updated_stmt1, additional_stmts), additional_decls) = update_statement
stmt1 tSymbol_table  tprogram in

                   let ((updated_stmt2, additional_stmts2), additional_decls2) = update_statement
stmt2 tSymbol_table  tprogram  in

          ((CIf(updated_expr, updated_stmt1, updated_stmt2), additional_stmts@additional_stmts2),
[])

            | For(e1, e2, e3, stmt) -> let ((updated_e1, stmts_e1), decls_e1) = update_expr e1
tSymbol_table  tprogram in

                                              let ((updated_e2, stmts_e2), decls_e2) = update_expr e2
tSymbol_table  tprogram in

                                              let ((updated_e3, stmts_e3), decls_e3) = update_expr e3
tSymbol_table  tprogram in

                                                 let ((updated_stmt, additional_stmts), decls_stmt) =
update_statement stmt  tSymbol_table  tprogram in

                                                                                  let accumulated_stmts =
(stmts_e1@stmts_e2@stmts_e3@additional_stmts) in

                                                                                  let accumulated_decls =
(decls_e1@decls_e2@decls_e3@decls_stmt) in

                                             ((CFor(updated_e1, updated_e2, updated_e3, updated_stmt),
accumulated_stmts), accumulated_decls)
```

```
        | While(e1, stmt) ->

                  let ((updated_e1, stmts_e1), decls_e1) = update_expr e1 tSymbol_table  tprogram
in

                 let ((updated_stmt, additional_stmts), additional_decls) = update_statement stmt
tSymbol_table tprogram in

                               ((CWhile(updated_e1,  updated_stmt),  stmts_e1  @  additional_stmts),
(decls_e1@additional_decls))

        | Break -> ((CBreak, []), [])

        | Expr(e) -> let ((updated_e, stmts), decls) = update_expr e tSymbol_table  tprogram in

                     ((CExpr(updated_e), stmts), decls)


let cFunc_from_tDestructor symbol_table tprogram destructor tStruct =

        let cdestructor_name = destructor_name_from_tStruct

        tStruct.struct_name in


        let first_param = first_param_for_destructor tStruct.struct_name in


        let last_param = last_param_for_destructor in


        let augmented_decls = List.fold_left (fun cdecls tdecl ->

                    let tdecl_id = (Semant.var_name_from_declaration tdecl) in

                    let tdecl_type = (Semant.type_from_declaration tdecl) in


                    let (cdecl, _)  = generate_decls_and_stmts_from_id symbol_table tprogram

                    tdecl_id tdecl tdecl_type in cdecls @ cdecl

        ) [] tStruct.members in


        {

                creturn_type = CType(CPrimitiveType(Cvoid));


                cfunc_params = [first_param] @ [last_param];


                cfunc_body = CCompoundStatement(augmented_decls,
```

```
                []);


                cfunc_name = cdestructor_name;
        }


let cFunc_from_tConstructor symbol_table  tprogram constructor tStruct =
        let cconstructor_name = constructor_name_from_tStruct
        tStruct.struct_name
        in


        let first_param = first_param_for_constructor tStruct.struct_name in


        let last_param = last_param_for_constructor in


        let augmented_decls = List.fold_left (fun cdecls tdecl ->
                    let tdecl_id = (Semant.var_name_from_declaration tdecl) in
                    let tdecl_type = (Semant.type_from_declaration tdecl) in


                    let (cdecl, _)  = generate_decls_and_stmts_from_id symbol_table tprogram
                    tdecl_id tdecl tdecl_type in cdecls @ cdecl
        ) [] tStruct.members in


        let generate_void_star_params =
            let anonList =
                List.filter (fun p -> match p with
                    AnonFuncDecl(anonDecl) -> true
                  | _ -> false) constructor.constructor_params
            in
            let returned_params =
                List.map (fun p -> match p with
                    AnonFuncDecl(anonDecl) ->
                        (match anonDecl.anon_decl_name with
```

```
                            Identifier(s) ->

                                let paramName = "cap_anon_" ^ s in

                                                    (CPointerType(CType(CPrimitiveType(Cvoid)), 1),
CIdentifier(paramName)))) anonList

            in

            returned_params

        in

        let extraParams = generate_void_star_params in


        {

                creturn_type = CType(CPrimitiveType(Cvoid));


                cfunc_params = [first_param] @ (List.map

                                                (cFuncParam_from_tFuncParam

                                                symbol_table)

                                                constructor.constructor_params)

                @ [last_param]@extraParams;


                cfunc_body = CCompoundStatement(augmented_decls,

                []);


                cfunc_name = cconstructor_name;


        }




let virtual_table_struct_for_tStruct symbol_table tStruct =

        let all_methods_for_struct = (List.map(Semant.get_fdecl_for_receiver

        tStruct.struct_name symbol_table)

        (Semant.get_unique_method_names_for_struct symbol_table tStruct)) in


        let methodMemberSymbols = List.map (cDeclaration_from_tFdecl
```

```
        symbol_table) all_methods_for_struct in

        let virt_table_name = virtual_table_name_from_tStruct
        tStruct.struct_name in

        {
                cstruct_name = virt_table_name;
                cstruct_members = methodMemberSymbols;
                cmethod_to_functions = StringMap.empty;
        }



let cStruct_from_tStruct symbol_table tprogram tStruct =
        let fieldSymbols = List.map (cSymbol_from_sSymbol
        symbol_table) (List.map (Semant.symbol_from_declaration)
        tStruct.members) in



        let virtual_table_name = virtual_table_name_from_tStruct
        tStruct.struct_name in

        let virtual_table_symbol =
                CVarSymbol("_virtual",
                CPointerType(CType(CStruct(virtual_table_name)), 1)) in

        let defaultStructMemberSymbols = [virtual_table_symbol] @ fieldSymbols in

        (* If there is an interface then add a struct member corresponding to
         * the interface to our struct *)
        let cStructMemberSymbols = if (Semant.get_interface_for_struct
        tStruct.struct_name symbol_table <> "") then
                [bol_from_Implements (Semant.get_interface_for_struct
```

```
                tStruct.struct_name symbol_table) tStruct.struct_name] @
                defaultStructMemberSymbols else defaultStructMemberSymbols in

                    let (methods_to_cfunctions, cfuncs) = (List.fold_left (fun (sym, cfunc_list)
method_ ->
                                        (let tfunc_name =
                                                Semant.var_name_from_direct_declarator
                                          method_.func_name in

                                        let initial_void_param =
                                                    create_cfunc_param_for_receiver
                                                    method_.receiver in

                                        let init_cast_decl =
                                                create_initial_cast_decl
                                                method_.receiver in

                                        let generate_void_star_params =
                                            let anonList =
                                                List.filter (fun p -> match p with
                                                    AnonFuncDecl(anonDecl) -> true
                                                  | _ -> false) method_.params
                                            in
                                            let returned_params =
                                                List.map (fun p -> match p with
                                                    AnonFuncDecl(anonDecl) ->
                                                        (match anonDecl.anon_decl_name with
                                                            Identifier(s) ->
                                                                let paramName = "cap_anon_" ^
s in

(CPointerType(CType(CPrimitiveType(Cvoid)), 1), CIdentifier(paramName)))) anonList
                                            in
```

```
                                    returned_params
                                in
                                let extraParams = generate_void_star_params in



                                let cfunc = {
                                    creturn_type = (cType_from_tType
                                    symbol_table
                                    (Semant.type_from_declaration_specifiers
                                    method_.return_type));

                                    cfunc_params =
                                            [initial_void_param] @ (List.map
                                    (cFuncParam_from_tFuncParam
                                    symbol_table) method_.params)@extraParams;

                                    cfunc_body =
                                            CCompoundStatement([init_cast_decl],
                                    []);

                                    cfunc_name = cFunc_from_tMethod
                                    tStruct.struct_name tfunc_name;

                                } in (StringMap.add tfunc_name cfunc
                                sym, cfunc_list @ [cfunc])))
                                (StringMap.empty, []) tStruct.methods) in

let cFunc_for_constructor =
        cFunc_from_tConstructor symbol_table tprogram tStruct.constructor tStruct in

let cFunc_for_destructor = cFunc_from_tDestructor symbol_table tprogram
tStruct.destructor tStruct in
```

```
         ({
                cstruct_name = cStructName_from_tStruct tStruct.struct_name;
                cstruct_members = cStructMemberSymbols;
                cmethod_to_functions = methods_to_cfunctions;
        }, cfuncs, (tStruct, cFunc_for_constructor, cFunc_for_destructor))



let update_cFunc tSymbol_table tprogram cFunc tFunc  =
        let updated_symbol_table = List.fold_left (fun m symbol -> StringMap.add
                    (Semant.get_id_from_symbol symbol) symbol m) tSymbol_table
((Semant.symbols_from_func_params
        tFunc.params) @ ([Semant.symbol_from_receiver tFunc.receiver])) in

        let CCompoundStatement(decls, stmts) = cFunc.cfunc_body in
              let CCompoundStatement(updated_decls, updated_stmts) = fst (fst (update_statement
tFunc.body updated_symbol_table tprogram )) in
        {
                cfunc_name = cFunc.cfunc_name;
                creturn_type = cFunc.creturn_type;
                cfunc_body = CCompoundStatement(decls @ updated_decls,
                updated_stmts);
                cfunc_params = cFunc.cfunc_params;
        }


let update_cFunc_from_anonDef tSymbol_table tprogram cFunc anonDef =
        let funcCaller = Semant.find_func_owning_anon_def tprogram anonDef in
        let rcvr = funcCaller.receiver in
        let rcvrSymbol = Semant.symbol_from_receiver rcvr in
        let globals = Semant.symbols_from_decls tprogram.globals in
        let builtinDecls = Semant.stdlib_funcs in
        let builtinSyms = Semant.symbols_from_fdecls builtinDecls in
        let localDecls = Semant.get_decls_from_compound_stmt anonDef.anon_body in
```

```
        let interfaceSymbols = Semant.symbols_from_interfaces tprogram.interfaces in
        let interfaceMethodSymbols =
            let interfaces =
                List.map (fun (InterfaceSymbol(_, iface)) -> iface) interfaceSymbols
            in
            List.fold_left (fun accList iface ->
                accList@(Semant.symbols_from_fdecls iface.funcs)) [] interfaces
        in


        let localSyms = Semant.symbols_from_decls localDecls in
        let paramSyms = Semant.symbols_from_func_params anonDef.anon_params in
                            let exceptSyms = Semant.symtable_from_symlist
(globals@builtinSyms@[rcvrSymbol]@interfaceMethodSymbols@paramSyms@localSyms) in
        let id_exists_in_symtable table id =
            try
                (fun x -> true)(StringMap.find id table)
            with _ ->
                false
        in


        let rec fix_expr locals instance_name expr = match expr with
            CBinop(e1, op, e2) ->
                    let fe1 = fix_expr locals instance_name e1 in
                    let fe2 = fix_expr locals instance_name e2 in
                    CBinop(fe1, op, fe2)
           | CAsnExpr(e1, aop, e2) ->
                    let fe1 = fix_expr locals instance_name e1 in
                    let fe2 = fix_expr locals instance_name e2 in
                    CAsnExpr(fe1, aop, fe2)
           | CCastExpr(t, e) ->
                   let fe = fix_expr locals instance_name e in
                    CCastExpr(t, fe)
```

```
| CPostfix(e, pop) ->
        let fe = fix_expr locals instance_name e in
        CPostfix(fe, pop)
| CCall(i, e1, e2, elist) ->
        let fe1 = fix_expr locals instance_name e1 in
        let fe2 = fix_expr locals instance_name e2 in
        let felist = fix_expr_list locals instance_name elist in
        CCall(i, fe1, fe2, felist)
| CDeref(e) ->
        let fe = fix_expr locals instance_name e in
        CDeref(fe)
| CCompareExpr(e1, lop, e2) ->
        let fe1 = fix_expr locals instance_name e1 in
        let fe2 = fix_expr locals instance_name e2 in
        CCompareExpr(fe1, lop, fe2)
| CPointify(e) ->
        let fe = fix_expr locals instance_name e in
        CPointify(fe)
| CMemAccess(i, e, id) ->
        let fe = fix_expr locals instance_name e in
        let fixedExpr = fix_expr locals instance_name (CId(id)) in
        (match fixedExpr with
             CId(fid) ->
                CMemAccess(i, fe, fid)
           | CMemAccess(_, _, _) -> fixedExpr)
| CId(CIdentifier(s)) ->
        if (id_exists_in_symtable tSymbol_table s) then
             expr
        else if (id_exists_in_symtable exceptSyms s) then
             expr
        else
            CMemAccess(1, CId(CIdentifier(instance_name)), CIdentifier(s))
```

```
            | CDeclExpr(CDeclaration(declSpecs, initDecl)) ->
                    let fInitDecl = fix_init_declarator locals instance_name initDecl in
                    CDeclExpr(CDeclaration(declSpecs, fInitDecl))
            | _ -> expr

        and fix_expr_list locals instance_name elist = match elist with
             [] -> []
           | [e] -> [fix_expr locals instance_name e]
           | h::t -> [fix_expr locals instance_name h]@(fix_expr_list locals instance_name t)

        and fix_init_declarator locals instance_name initDecl = match initDecl with
           | CInitDeclaratorAsn(dd, aop, e) ->
                            let CInitDeclarator(fdd) = fix_init_declarator locals instance_name
(CInitDeclarator(dd)) in
                  let fe = fix_expr locals instance_name e in
                  CInitDeclaratorAsn(fdd, aop, fe)
           | _ -> initDecl

        and fix_declaration locals instance_name decl = match decl with
             CDeclaration(declSpecs, initDecl) ->
                 let fidecl = fix_init_declarator locals instance_name initDecl in
                 CDeclaration(declSpecs, fidecl)

        and fix_declaration_list locals instance_name declList = match declList with
             [] -> []
           | [d] -> [fix_declaration locals instance_name d]
                | h::t -> [fix_declaration locals instance_name h]@(fix_declaration_list locals
instance_name t)

        and fix_statement locals instance_name stmt = match stmt with
             CExpr(e) ->
                 let fe = fix_expr locals instance_name e in
                 CExpr(fe)
```

```
          | CReturn(e) ->
                let fe = fix_expr locals instance_name e in
                CReturn(fe)
          | CCompoundStatement(declList, stmtList) ->
                let fdl = fix_declaration_list locals instance_name declList in
                let fsl = fix_statement_list locals instance_name stmtList in
                CCompoundStatement(fdl, fsl)
          | CIf(e, s1, s2) ->
                let fe = fix_expr locals instance_name e in
                let fs1 = fix_statement locals instance_name s1 in
                let fs2 = fix_statement locals instance_name s2 in
                CIf(fe, fs1, fs2)
          | CFor(e1, e2, e3, s) ->
                let fe1 = fix_expr locals instance_name e1 in
                let fe2 = fix_expr locals instance_name e2 in
                let fe3 = fix_expr locals instance_name e3 in
                let fs = fix_statement locals instance_name s in
                CFor(fe1, fe2, fe3, fs)
          | CWhile(e, s) ->
                let fe = fix_expr locals instance_name e in
                let fs = fix_statement locals instance_name s in
                CWhile(fe, fs)
          | _ -> stmt


       and fix_statement_list locals instance_name stmtList = match stmtList with
            [] -> []
          | [s] -> [fix_statement locals instance_name s]
                  | h::t -> [fix_statement locals instance_name h]@(fix_statement_list locals
instance_name t)


        in
        let updated_symbol_list =
```

```
                            ((Semant.symbols_from_func_params anonDef.anon_params) @
(Semant.symbols_from_outside_scope_for_anon_def tprogram anonDef)@[rcvrSymbol] )

        in

        let updated_symbol_table =

            (List.fold_left (fun m symbol ->

                StringMap.add (Semant.get_id_from_symbol symbol) symbol m)

            tSymbol_table (updated_symbol_list)) in

        let anon_name = Semant.find_name_for_anon_def tprogram anonDef in

        let instanceName = "s" ^ anon_name in

        let structName = "S" ^ anon_name in

                                                                    let newDecls =
[CDeclaration(CDeclSpecTypeSpecAny(CPointerType(CType(CStruct(structName)), 1)),

                    CInitDeclaratorAsn(CDirectDeclarator(CVar(CIdentifier(instanceName))), Asn,
CCastExpr(CPointerType(CType(CStruct(structName)), 1), CId(CIdentifier("capture_struct"))))))]

        in




        let CompoundStatement(decls, _) = anonDef.anon_body in

        let locals = Semant.symbols_from_decls decls in

        let CCompoundStatement(decls, stmts) = cFunc.cfunc_body in

               let cmpstmt = fst (fst (update_statement anonDef.anon_body updated_symbol_table
tprogram )) in

                    let CCompoundStatement(updated_decls, updated_stmts) = fix_statement
(locals@[rcvrSymbol]) instanceName cmpstmt in

        {

                cfunc_name = cFunc.cfunc_name;

                creturn_type = cFunc.creturn_type;

                cfunc_body = CCompoundStatement(newDecls@updated_decls @decls,

                updated_stmts);

                cfunc_params = cFunc.cfunc_params;

        }


let update_cDestructor tSymbol_table tprogram cFunc tStruct =

        let updated_symbol_table = List.fold_left (fun m symbol -> StringMap.add
```

```
        (Semant.get_id_from_symbol symbol) symbol m) tSymbol_table ((Semant.symbols_from_decls
        (Semant.get_decls_from_compound_stmt
        tStruct.constructor.constructor_body)) @ (Semant.symbols_from_decls
        tStruct.members) @ (Semant.symbols_from_func_params
        tStruct.constructor.constructor_params))  in




        let ancestor_destructor = Semant.get_ancestors_destructor tSymbol_table
        tStruct in


        let parent_destructor_call =
        if (ancestor_destructor.destructor_name <> tStruct.struct_name &&
        ancestor_destructor.destructor_name <> "") then
              generate_stmts_for_parent_destructor tSymbol_table tprogram
              tStruct.destructor tStruct else [] in


        let head_assignments = List.fold_left (fun assignments tdecl ->
                    let tdecl_id = (Semant.var_name_from_declaration tdecl) in


                    let asn_expr = CExpr(CAsnExpr(CId(CIdentifier(tdecl_id)),
                    Asn, CMemAccess(1,
                      CDeref(CId(CIdentifier("_this"))), CIdentifier(tdecl_id)))) in assignments @
[asn_expr] ) [] tStruct.members in


        let CCompoundStatement(original_decls, stmts) = cFunc.cfunc_body in


        let CCompoundStatement(updated_decls, updated_stmts) = fst (
                        fst (update_statement tStruct.destructor.destructor_body
                        updated_symbol_table tprogram)) in


        let free_this = CIf(CId(CIdentifier("_needs_free")),
```

```
        CCompoundStatement([], [CExpr(CFree(CMemAccess(1,
        CDeref(CId(CIdentifier("_this"))),
                  CIdentifier("_virtual")))))]@[CExpr(CFree(CDeref(CId(CIdentifier("_this"))))))]),
CEmptyElse) in

        {
                cfunc_name = cFunc.cfunc_name;
                creturn_type = cFunc.creturn_type;
                cfunc_body = CCompoundStatement(original_decls @ updated_decls,
                head_assignments @ parent_destructor_call @ updated_stmts @ [free_this]);
                cfunc_params = cFunc.cfunc_params;
        }



let update_cConstructor tSymbol_table tprogram cFunc tStruct =
        let updated_symbol_table = List.fold_left (fun m symbol -> StringMap.add
        (Semant.get_id_from_symbol symbol) symbol m) tSymbol_table ((Semant.symbols_from_decls
        (Semant.get_decls_from_compound_stmt
        tStruct.constructor.constructor_body)) @ (Semant.symbols_from_decls
        tStruct.members) @ (Semant.symbols_from_func_params
        tStruct.constructor.constructor_params))  in

        let ctype = cType_from_tType tSymbol_table
        (CustomType(tStruct.struct_name)) in

        let virtual_table_name = virtual_table_name_from_tStruct
        tStruct.struct_name in



        let virtual_table_type =
                CType(CStruct(virtual_table_name)) in
```

```
        let alloc_virtual_table =
                CExpr(CAsnExpr(CMemAccess(1, CDeref(CId(CIdentifier("_this"))),
                CIdentifier("_virtual")),
        Asn, CAlloc(virtual_table_type, CId(CIdentifier(sizeof_string tSymbol_table
        virtual_table_type))))) in


                  let alloc_this = CIf(CId(CIdentifier("_needs_malloc")),
CExpr(CAsnExpr(CDeref(CId(CIdentifier("_this"))),
                    Asn, CAlloc(ctype, CId(CIdentifier((sizeof_string tSymbol_table
                    ctype)))))), CExpr(CNoexpr)) in


        let virt_table_assignments = generate_virtual_table_assignments 1 tStruct tSymbol_table
        "(*_this)->_virtual" in


        let (interface_decls, interface_assignments) =
                interface_decl_and_assignments_for_struct (-1) tStruct
                tSymbol_table "_this" in


        let tail_assignments = List.fold_left (fun assignments tdecl ->
                    let tdecl_id = (Semant.var_name_from_declaration tdecl) in


                    let asn_expr = CExpr(CAsnExpr(CMemAccess(1,
                    CDeref(CId(CIdentifier("_this"))), CIdentifier(tdecl_id)),
                              Asn, CId(CIdentifier(tdecl_id)))) in assignments @ [asn_expr] ) []
tStruct.members in


        let stmts_for_super =
            if (Semant.constructor_has_super tStruct.constructor) then
                                        (let Expr(Super(expr_list)) = Semant.get_super_expr
tStruct.constructor.constructor_body in
                                generate_stmts_for_super expr_list updated_symbol_table tprogram
tStruct.constructor tStruct)
        else
            []
```

```
        in


        let CCompoundStatement(original_decls, stmts) = cFunc.cfunc_body in


        let CCompoundStatement(updated_decls, updated_stmts) = fst (
                        fst (update_statement tStruct.constructor.constructor_body
                        updated_symbol_table tprogram)) in
        {
                cfunc_name = cFunc.cfunc_name;
                creturn_type = cFunc.creturn_type;
                cfunc_body = CCompoundStatement(original_decls @ interface_decls
                @ updated_decls,
                [alloc_this] @ [alloc_virtual_table] @ stmts_for_super @ updated_stmts @ stmts @
                tail_assignments@virt_table_assignments @ interface_assignments);
                cfunc_params = cFunc.cfunc_params;
        }


let rec cFunc_from_anonDef symbol_table tprogram anonDef =
    let rec convert_anon_params symbol_table params =
        (match params with
                                [] -> [(CPointerType(CType(CPrimitiveType(Cvoid)), 1),
CIdentifier("capture_struct"))]
                            | [p] -> [cFuncParam_from_tFuncParam symbol_table
p]@[(CPointerType(CType(CPrimitiveType(Cvoid)), 1), CIdentifier("capture_struct"))]
          | h::t -> let htype = (cFuncParam_from_tFuncParam symbol_table h) in
                    let ttype = (convert_anon_params symbol_table t) in
                    [htype]@ttype)
    in
    {
    cfunc_name = anonDef.anon_name;
    cfunc_body = CCompoundStatement([], []);
    cfunc_params = (convert_anon_params symbol_table anonDef.anon_params);
    creturn_type = cType_from_tType symbol_table anonDef.anon_return_type }
```

```
and cFunc_list_from_anonDef_list symbol_table tprogram adlist =
    match adlist with
    [] -> []
  | [x] -> [cFunc_from_anonDef symbol_table tprogram x]
  | h::t -> let hfuncs = [(cFunc_from_anonDef symbol_table tprogram h)] in
            let tfuncs = (cFunc_list_from_anonDef_list symbol_table tprogram t) in
            hfuncs@tfuncs



let cProgram_from_tProgram program =
        let updated_program = Semant.update_structs_in_program program in

        let tSymbol_table = Semant.build_symbol_table updated_program in
            let cstructs_and_functions = List.map (cStruct_from_tStruct tSymbol_table program )
updated_program.structs in

        let cstructs = List.map (fun (structs, _, _) -> structs) cstructs_and_functions in

        let virt_table_structs = List.map (
                virtual_table_struct_for_tStruct tSymbol_table)
        updated_program.structs in

        let cfuncs_methods = List.concat (List.map (fun(_, methods, _) ->
                methods) cstructs_and_functions) in

        let cconstructors = List.map (fun(_, _, constructor) -> constructor)
        (List.filter (fun(_, _, (_, const, _)) -> if (const.cfunc_name = "") then false
        else true) cstructs_and_functions) in

        (*let cdestructors = List.map (fun(_, _, constructor) -> constructor)*)
        (*(List.filter (fun(_, _, (_, _, destr)) -> if (destr.cfunc_name = "") then false*)
```

```
        (*else true) cstructs_and_functions) in*)


        let cglobals = List.fold_left (fun acc (decls, _) -> acc @

        decls) [] (List.map (update_decl tSymbol_table updated_program)

        updated_program.globals) in




        let cStructs = virt_table_structs @ (List.map (cStruct_from_tInterface

        tSymbol_table) program.interfaces) @ cstructs in


        let tAnonDefs = Semant.anon_defs_from_tprogram program in

           let cFuncsTranslatedFromAnonDefs = cFunc_list_from_anonDef_list tSymbol_table program
tAnonDefs in

        let capture_structs = capture_struct_list_from_anon_def_list program tAnonDefs in

        (* The function bodies have not been filled out yet. Just the parameters

         * and return types *)

           let cDeclaredMethodsAndFuncs = cfuncs_methods @ (List.rev (List.map (cFunc_from_tFunc
tSymbol_table)

        (List.filter (fun fdecl ->

                if (fdecl.receiver = ("", "")) then true else

                false) program.functions))) in


        let cUpdatedDeclaredMethodsAndFuncs = List.fold_left (fun acc cFunc ->

                let sym_ = StringMap.find

        cFunc.cfunc_name tSymbol_table in (match sym_ with

                                        | FuncSymbol(_, fdecl) -> acc @ [update_cFunc tSymbol_table
program

                                          cFunc fdecl]

                                    | _ -> raise(Failure("error")))

                        ) [] cDeclaredMethodsAndFuncs in


        let cConstructors = List.fold_left (fun acc (tStruct, cConst, _) ->

                acc @ [update_cConstructor tSymbol_table program cConst tStruct]) []
```

```
            cconstructors in


        let cDestructors = List.fold_left (fun acc (tStruct, _, cDestr) ->

            acc @ [update_cDestructor tSymbol_table program cDestr tStruct]) []

            cconstructors in




        let anon_def_for_function fn =

            List.find (fun af ->

                if (af.anon_name = fn.cfunc_name) then

                    true

                else

                    false) tAnonDefs

        in


        let cUpdatedFuncsTranslatedFromAnonDefs =

            List.map (fun f ->

                let anonDef = anon_def_for_function f in

                                    update_cFunc_from_anonDef tSymbol_table program f anonDef)
cFuncsTranslatedFromAnonDefs

        in

                let cFuncs = cConstructors @ cDestructors @ cUpdatedDeclaredMethodsAndFuncs @
cUpdatedFuncsTranslatedFromAnonDefs

        in

        {

                cstructs = cStructs@capture_structs;

                cglobals = cglobals;

                cfunctions = cFuncs;

        }
```

Astutil.ml

```
open Ast

module StringMap = Map.Make(String)

let string_of_op = function
        Add -> "PLUS"
      | Sub -> "MINUS"
      | Mul -> "TIMES"
      | Div -> "DIVIDE"
      | Mod -> "MOD"
      | And -> "AND"
      | Or -> "OR"
      | BitAnd -> "BITWISE_AND"
      | BitOr -> "BITWISE_OR"
      | Xor -> "XOR"
      | Not -> "NOT"
      | Lsh -> "LSHIFT"
      | Rsh -> "RSHIFT"

let string_of_postfix_op = function
      | PostPlusPlus -> "++"
      | PostMinusMinus -> "--"
      | PostEmptyOp -> ""

let string_of_assignment_op = function
        Asn -> "EQUALS"
      | MulAsn -> "TIMES_EQUALS"
      | DivAsn -> "DIVIDE_EQUALS"
```

```
      | ModAsn -> "MOD_EQUALS"
      | AddAsn -> "ADD_EQUALS"
      | SubAsn -> "MINUS_EQUALS"
      | LshAsn -> "LSHIFT_EQUALS"
      | RshAsn -> "RSHIFT_EQUALS"
      | AndAsn -> "AND_EQUALS"
      | XorAsn -> "XOR_EQUALS"
      | OrAsn -> "OR_EQUALS"

let string_of_logical_op = function
        Eql -> "EQUALS"
      | NotEql -> "NOT_EQUALS"
      | Less -> "LESS_THAN"
      | LessEql -> "LESS_THAN_EQUALS"
      | Greater -> "GREATER_THAN"
      | GreaterEql -> "GREATER_THAN_EQUALS"
      | LogicalAnd -> "LOGICAL AND"
      | LogicalOr -> "LOGICAL OR"

let string_of_type_qualifier = function
        Const -> "const"
      | Volatile -> "volatile"

let string_of_type_spec = function
        Void -> "void"
      | Char -> "char"
      | Short -> "short"
      | Int -> "int"
      | Long -> "long"
      | Float -> "float"
```

```
      | Double -> "double"
      | Signed -> "signed"
      | Unsigned -> "unsigned"
      | String -> "string"

let rec string_of_ptr = function
      PtrType(x, y) -> "Pointer(" ^ string_of_ptr x ^
      string_of_ptr y ^ ")"
      | Pointer -> "Pointer"
      | NoPointer -> ""

let rec string_of_type = function
      PrimitiveType(t) -> string_of_type_spec t
    | PointerType(t, d) -> string_of_type t ^ " depth: " ^ string_of_int d
    | ArrayType(t, ptr, expr) -> "[" ^ string_of_type t ^ string_of_ptr ptr ^"]"
    | CustomType(t) -> t
    | NilType -> "NilType"
    | AnonFuncType(t, types) -> "AnonFuncType(" ^ string_of_type t ^ "," ^
    (String.concat "," (List.map string_of_type types))

let string_of_storage_class_spec = function
        Auto -> "auto"
      | Register -> "register"
      | Static -> "static"
      | Extern -> "extern"
      | Typedef -> "typedef"

let string_of_identifier = function
        Identifier(s) -> s
```

```
let rec string_of_declaration_specifiers = function

          DeclSpecTypeSpec(tspec) -> "DeclSpecTypeSpec(" ^ string_of_type_spec tspec ^
")"

             | DeclSpecTypeSpecInitList(t, idspecs) -> "DeclSpecTypeSpecInitList(" ^
string_of_type t ^ ", " ^ string_of_declaration_specifiers idspecs ^ ")"

      | DeclSpecTypeSpecAny(t) -> string_of_type t


let string_of_type_spec_indicator = function

        TypeSpec(tspec) -> "TypeSpec(" ^ string_of_type_spec tspec ^ ")"

               | TypeSpecWithDeclSpec(tspec, declSpec) -> "TypeSpecWithDeclSpec(" ^
string_of_type_spec tspec ^ ", " ^ string_of_declaration_specifiers declSpec ^ ")"




let string_of_unary_op = function

        PlusPlus -> "PlusPlus"


let string_of_variable = function

        Var(id) -> "Var(" ^ string_of_identifier id ^ ")"


let string_of_declarator = function

    DirectDeclarator(v) -> string_of_variable v

   | PointerDirDecl(ptr, decl) -> string_of_ptr ptr ^ "("

   ^ string_of_variable decl ^ ")"


let string_of_receiver receiver =

        match (receiver) with

        ("", "") -> ""

        | (d, u) -> "RECEIVER("^d^","^u^") "


let rec string_of_expr = function

   Literal(x) -> "Int(" ^ string_of_int x ^ ")"

  | FloatLiteral(x) -> "Float(" ^ string_of_float x ^ ")"
```

```
  | StringLiteral(s) -> "String(" ^ s ^ ")"
  | Id (x) -> "Identifier(" ^ string_of_identifier x ^ ")"
  | Deref(e) -> "Deref(" ^ string_of_expr e ^ ")"
  | Pointify(e) -> "Pointify(" ^ string_of_expr e ^ ")"
  | Neg(e) -> "-" ^ string_of_expr e
  | Postfix(e1, op) -> "Postfix(" ^ string_of_expr e1 ^ "," ^
  (string_of_postfix_op op) ^ ")"
  | CompareExpr(e1, op, e2) -> "Compare(" ^ string_of_expr e1 ^ "," ^
  string_of_logical_op op ^ "," ^ string_of_expr e2 ^ ")"
  | ArrayAccess(e1, e2) -> "ArrayAccess" ^ string_of_expr e1 ^ "[" ^
  string_of_expr e2 ^ "]"
  | Noexpr -> ""
  | Nil -> "NIL"
  | Clean(expr) -> "Clean(" ^ string_of_expr expr ^ ")"
  | AsnExpr(e1, asnOp, e) -> string_of_assignment_op asnOp ^ "(" ^
  string_of_expr e1 ^  ", " ^ string_of_expr e ^ ")"
  | Super(expr_list) -> "Super(" ^ string_of_expr_list expr_list ^ ")"
    | Binop(e1, op, e2) -> string_of_op op ^ "(" ^ string_of_expr e1 ^ ", " ^
string_of_expr e2 ^ ")"
  | Unop(e, unOp) -> string_of_unary_op unOp ^ "(" ^ string_of_expr e ^ ")"
  | Call(e, e2, exprList) -> "Call(" ^ "Receiver(" ^ string_of_expr e ^")"  ^
    "FunctionName: " ^ (string_of_expr e2) ^ " Params: " ^ (string_of_expr_list
exprList) ^ ")"
  | Make(typ_, exprList) -> "Make(" ^ string_of_type typ_ ^ string_of_expr_list
  exprList ^ ")"
  | MemAccess(e, Identifier(t)) -> "Access(" ^ "Var(" ^ string_of_expr e ^ ")" ^ ","
  ^ t ^")"
    | AnonFuncDef(anonDef) -> "AnonFuncDef(ReturnType: " ^ (string_of_type
anonDef.anon_return_type) ^ ", Params: " ^ (string_of_func_param_list
anonDef.anon_params) ^ ", Body: " ^ (string_of_statement anonDef.anon_body) ^ ")"
  | DeclExpr(d) -> "DeclExpr(" ^ string_of_declaration d ^ ")"
and string_of_func_param_list = function
```

```
    [] -> ""
  | [x] -> string_of_func_param x
  | h::t -> (string_of_func_param h) ^ ", " ^ (string_of_func_param_list t)


and string_of_expr_list = function
    [] -> ""
  | [e] -> string_of_expr e
  | h::t -> string_of_expr h ^ string_of_expr_list t


  and  string_of_init_declarator = function
   InitDeclarator(x) -> string_of_declarator x
  | InitDeclList([]) -> ""
  | InitDeclList(h::t) -> let string_of_init_decl_list str
  initdecl =  str ^ (string_of_init_declarator initdecl) in
  string_of_init_declarator h ^ "," ^  (List.fold_left string_of_init_decl_list
  "" t)
  | InitDeclaratorAsn(decl, asnop, expr) -> string_of_assignment_op asnop ^ "("
  ^ string_of_declarator decl ^ " " ^ string_of_expr expr


and string_of_anon_func_decl d = "AnonFuncDecl(Name: " ^ (string_of_identifier
d.anon_decl_name) ^ ", ReturnType: " ^ (string_of_type d.anon_decl_return_type) ^ ",
Params: " ^ (string_of_func_param_list d.anon_decl_params) ^ ")"


and string_of_anon_def d = "AnonDef(AnonName: " ^ d.anon_name ^ ", AnonReturnType: " ^
string_of_type d.anon_return_type ^ ", AnonParams: " ^ string_of_func_param_list
d.anon_params ^ ", AnonBody: " ^ string_of_statement d.anon_body ^ ")"


and string_of_declaration = function Declaration(x, y) -> "(" ^
string_of_declaration_specifiers x ^ " " ^
  string_of_init_declarator y ^ ")"


and string_of_declaration_list = function
   [] -> ""
```

```
  | h :: t -> string_of_declaration h ^ ", " ^ (string_of_declaration_list t)


and string_of_statement = function

 Expr(e) -> "Statement(" ^ string_of_expr e ^ ")"

 | Return(e) -> "RETURN(" ^ (string_of_expr e) ^")"

 | If(e, s1, s2) -> "IF " ^ (string_of_expr e) ^" " ^ (string_of_statement s1)^ "

 " ^ (string_of_statement s2)

 | EmptyElse -> ""

 | Break -> "BREAK"

 | For(e1, e2, e3, s) -> "FOR " ^ (string_of_expr e1) ^ " " ^ (string_of_expr

 e2) ^ " " ^ (string_of_expr e3) ^ " " ^ (string_of_statement s)

 | While(e, s) -> "WHILE " ^ (string_of_expr e) ^ " " ^ (string_of_statement s)

  | CompoundStatement(dl, sl) -> "CompoundStatement(Declarations: "    ^
string_of_declaration_list dl ^ " " ^"StatementList: " ^ String.concat ", " (List.map
string_of_statement sl) ^ ")"



and string_of_statement_list = function

        [] -> ""

       | h :: t -> string_of_statement h ^ ", " ^ (string_of_statement_list t)


and string_of_func_param = function

        FuncParamsDeclared(decl_specs, declarator) ->

                        "PARAM(" ^ string_of_declaration_specifiers

                        decl_specs ^ " " ^

                        string_of_declarator declarator ^ ") "

        | ParamDeclWithType(decl_specs) -> "PARAM(" ^

        string_of_declaration_specifiers decl_specs ^ ") "

        | AnonFuncDecl(afd) -> string_of_anon_func_decl afd
```

```
let string_of_constructor constructor = "Constructor(" ^ constructor.constructor_name
^ "Body:

        " ^ string_of_statement (constructor.constructor_body) ^ ")"


let string_of_func fdecl = "FuncDecl(Name: " ^

      string_of_declarator fdecl.func_name ^ " ReturnType: " ^

      string_of_receiver fdecl.receiver ^ string_of_declaration_specifiers

      fdecl.return_type ^ " Parameters: " ^

      String.concat ", " (List.map

      string_of_func_param fdecl.params) ^ " Body: " ^ string_of_statement

      fdecl.body ^ ") "


let string_of_struct struct_decl = "Struct(" ^

        string_of_declaration_list struct_decl.members ^ ", " ^

        struct_decl.struct_name ^ ", " ^ (string_of_constructor

        struct_decl.constructor) ^ "," ^  struct_decl.extends ^ ", " ^

        struct_decl.implements ^ ")"


let string_of_list_objs f list_objs = String.concat ", " (List.map f list_objs)


let string_of_interface interface = "INTERFACE(" ^ interface.name ^

(string_of_list_objs string_of_func interface.funcs) ^ ")"


let string_of_program program =

        string_of_declaration_list program.globals  ^ (string_of_list_objs

        string_of_interface program.interfaces) ^ (string_of_list_objs

        string_of_struct program.structs) ^ (string_of_list_objs string_of_func

        program.functions)


let string_of_symbol = function
```

```
    VarSymbol(s, t) -> "Variable_Symbol(Name: " ^ s ^ ", Type: " ^ string_of_type t ^
")"

 | FuncSymbol(s, fdecl) ->

             "Function_Symbol(Name: " ^ s ^ ")"


 | StructSymbol(s, strct) -> "Struct_Symbol" (* Finish me! *)

 | InterfaceSymbol(s, ti) -> "Interface_Symbol" (* Finish me! *)

  | AnonFuncSymbol(s, t) -> "AnonymousFunction_Symbol(Name: " ^ s ^ ", Type: " ^
string_of_type t ^ ")"


let string_of_symbol_simple = function

   VarSymbol(s, t) -> s

 | FuncSymbol(s, fdecl) -> s

 | StructSymbol(s, strct) -> s

 | InterfaceSymbol(s, ti) -> s

 | AnonFuncSymbol(s, t) -> s


let rec string_of_symbol_list l = match l with

    [] -> ""

  | [x] -> string_of_symbol x

  | h::t -> string_of_symbol h ^ string_of_symbol_list t




let rec string_of_func_decl_list dlist = match dlist with

    [] -> "\n"

  | [x] -> (string_of_func x) ^ "\n\n"

  | h::t -> (string_of_func h) ^ "\n\n" ^ (string_of_func_decl_list t) ^ "\n\n"




(*let apply_name_to_anon_def (prefix, count) adef = {*)
```

```
    (*anon_name =  prefix ^ "_" ^ (string_of_int count);*)

    (*anon_return_type = adef.anon_return_type;*)

    (*anon_params = adef.anon_params;*)

    (*anon_body = adef.anon_body;*)

(*}*)


(*let rec anon_defs_from_expr (prefix, count) expr = match expr with*)

        (*AnonFuncDef(anonDef) ->([(apply_name_to_anon_def (prefix, count) anonDef)],
(count + 1))*)

   (*| Binop(e1, op, e2) -> *)

           (*let (defs1, count1) = (anon_defs_from_expr (prefix, count) e1) in*)

           (*let (defs2, count2) = (anon_defs_from_expr (prefix, count1) e2) in*)

           (*(defs1@defs2, count2)*)

   (*| AsnExpr(_, _, e) -> anon_defs_from_expr (prefix, count) e*)

   (*| Postfix(e1, _) -> (anon_defs_from_expr (prefix, count) e1)*)

   (*| Call(_, e, elist) -> *)

           (*let (defs1, count1) = (anon_defs_from_expr (prefix, count) e) in*)

             (*let (defs2, count2) = (anon_defs_from_expr_list (prefix, count1) elist)
in*)

           (*(defs1@defs2, count2);*)

    (*| _ -> ([], count) [> Other expression types cannot possibly contain anonymous
function definitions <] *)


(*and anon_defs_from_expr_list (prefix, count) elist = match elist with  *)

     (*[] -> ([], count)*)

   (*| [e] -> anon_defs_from_expr (prefix, count) e*)

   (*| h::t ->*)

           (*let (defs1, count1) = (anon_defs_from_expr (prefix, count) h) in*)

           (*let (defs2, count2) = (anon_defs_from_expr_list (prefix, count1) t) in*)

           (*(defs1@defs2, (count2))*)
```

```
(*let rec anon_defs_from_declaration (prefix, count) decl = match decl with*)

        (*Declaration(declSpecs, initDecl) -> anon_defs_from_init_declarator (prefix,
count) initDecl*)



(*and anon_defs_from_declaration_list (prefix, count) declList = match declList with*)

     (*[] -> ([], count)*)

   (*| [d] -> anon_defs_from_declaration (prefix, count) d*)

   (*| h::t ->*)

           (*let (defs1, count1) = (anon_defs_from_declaration (prefix, count) h) in*)

             (*let (defs2, count2) = (anon_defs_from_declaration_list (prefix, count1)
t) in*)

           (*(defs1@defs2, count2)*)



(*and anon_defs_from_init_declarator (prefix, count) idecl = match idecl with*)

           (*InitDeclaratorAsn(_, _, e) -> anon_defs_from_expr (prefix, count) e*)

                (*| InitDeclList(initDeclList) -> anon_defs_from_init_declarator_list
(prefix, count) initDeclList*)

         (*| _ -> ([], count)*)



(*and anon_defs_from_init_declarator_list (prefix, count) ideclList = match ideclList
with*)

     (*[] -> ([], count)*)

   (*| [decl] -> anon_defs_from_init_declarator (prefix, count) decl*)

   (*| h::t -> *)

            (*let (defs1, count1) = (anon_defs_from_init_declarator (prefix, count) h)
in*)

                (*let (defs2, count2) = (anon_defs_from_init_declarator_list (prefix,
count1) t) in*)

           (*(defs1@defs2, (count2))*)



(*let rec anon_defs_from_statement (prefix, count) stmt = match stmt with*)

     (*Expr(e) -> anon_defs_from_expr (prefix, count) e*)

   (*| Return(e) -> anon_defs_from_expr (prefix, count) e*)
```

```
   (*| If(e, s1, s2) -> *)
           (*let (defs1, count1) = (anon_defs_from_expr (prefix, count) e) in*)
           (*let (defs2, count2) = (anon_defs_from_statement (prefix, count1) s1) in*)
           (*let (defs3, count3) = (anon_defs_from_statement (prefix, count2) s2) in*)
           (*(defs1@defs2@defs3, count3)*)
   (*| For(e1, e2, e3, s) -> *)
           (*let (defs1, count1) = (anon_defs_from_expr (prefix, count) e1) in*)
           (*let (defs2, count2) = (anon_defs_from_expr (prefix, count1) e2) in*)
           (*let (defs3, count3) = (anon_defs_from_expr (prefix, count2) e3) in*)
           (*let (defs4, count4) = (anon_defs_from_statement (prefix, count3) s) in*)
           (*(defs1@defs2@defs3@defs4, count4)        *)
   (*| While(e, s) -> *)
           (*let (defs1, count1) = (anon_defs_from_expr (prefix, count) e) in*)
           (*let (defs2, count2) = (anon_defs_from_statement (prefix, count1) s) in*)
           (*(defs1@defs2, count2)*)
   (*| CompoundStatement(declList, stmtList) ->*)
           (*let newSymbols = Semant.symbols_from_fdecl *)
              (*let (defs1, count1) = (anon_defs_from_declaration_list (prefix, count)
declList) in*)
               (*let (defs2, count2) = (anon_defs_from_statement_list (prefix, count1)
stmtList) in*)
           (*(defs1@defs2, count2) *)


(*and anon_defs_from_statement_list (prefix, count) stmtList = match stmtList with*)
     (*[] -> ([], count)*)
   (*| [s] -> anon_defs_from_statement (prefix, count) s*)
   (*| h::t ->*)
           (*let (defs1, count1) = (anon_defs_from_statement (prefix, count) h) in*)
            (*let (defs2, count2) = (anon_defs_from_statement_list (prefix, count1) t)
in*)
           (*(defs1@defs2, count2) *)
```

```
(*let rec anon_defs_from_func_decl (prefix, count) fdecl = *)
    (*let newPrefix = *)
        (*(match fdecl.func_name with*)
              (*DirectDeclarator(Var(Identifier(s))) -> "a_" ^ s*)
            (*| PointerDirDecl(_, Var(Identifier(s))) -> "a_" ^ s)*)
    (*in*)
    (*anon_defs_from_statement (newPrefix, 0) fdecl.body*)


(*and anon_defs_from_func_decl_list (prefix, count) fdlist = match fdlist with*)
      (*[] -> ([], count)*)
    (*| [x] -> anon_defs_from_func_decl (prefix, count) x*)
    (*| h::t ->*)
           (*let (defs1, count1) = (anon_defs_from_func_decl (prefix, count) h) in*)
            (*let (defs2, count2) = (anon_defs_from_func_decl_list (prefix, count1) t)
in*)
           (*(defs1@defs2, count2)*)


(*let anon_defs_from_tprogram tprog =*)
          (*let (defs, _) = (anon_defs_from_func_decl_list ("_", 0) (List.rev
tprog.functions)) in*)
    (*List.rev defs*)


(*let rec print_anon_def anonDef = *)
    (*Printf.printf "\n%s\n" (string_of_anon_def anonDef)*)


(*and print_anon_defs = function*)
      (*[] -> ()*)
    (*| [x] -> print_anon_def x*)
    (*| h::t -> print_anon_def h; print_anon_defs t*)
```

```
let print_symbol_table symtable =

   (Printf.printf "SYMBOL_TABLE:-------------\n\n");

    let l  = StringMap.bindings symtable in

         List.iter (fun (name, sym) -> Printf.printf "%s\n" name) l
```