better call **SOL**

SHAPE ORIENTED LANGUAGE FINAL REPORT

Aditya Narayanamoorthy	`an2753`	*Language Guru*
Erik Dyer	`ead2174`	*System Architect*
Gergana Alteva	`gla2112`	*Program Manager*
Kunal Baweja	`kb2896`	*Tester*

December 19, 2017

Contents

Chapter 1

Introduction

SOL (Shape Oriented Language), is a domain specific programming language that allows programmers to create 2D animations with ease, through an object-oriented approach. Engineers, programmers, scientists and designers, through SOL, have the ability to define and create objects, known as Shapes, and dictate their appearance and movements on the screen. SOLś simplicity saves developers the trouble of learning complicated third-party animation tools, without sacrificing control over behavior of objects. It compiles into LLVM IR bytecode, making it adaptable across different architectures. The produced LLVM IR bytecode can be translated further into assembly code and linked statically against a predefined library before compiling down into an executable for a specific architecture using the LLVM compiler and GCC compilers respectively. The predefined library used by SOL for graphic rendering has been built on top of SDL2, that abstracts away the lower level details for drawing and animating objects.

1.1 Background

SOL takes its inspiration from the ease of programming in *object-oriented paradigm* and the complexity of existing libraries/solutions for rendering graphics. It attempts to combine both of them and provide a language which allows developers to organize graphics into a collection of *Shapes* (much like objects) which can be easily defined, created and interacted with to produce powerful images and/or animations at a fast paced development.

SOL is commonly used to model various types of scientific data, but it can also be applicable in other domains, such as:

1. Drawing engineering models
2. Data visualization
3. Bored college students making funny memes
4. Entertaining animations

Chapter 2

Language Tutorial

This chapter provides a simple language tutorial to guide programmers towards creating their first SOL program! It has been divided into three parts:

1. **Environment setup** - guide to setting up the SOL compiler and development environment
2. **Language Quick Tour** - a bried descriptioin of basic components of language.
3. **Sample Program** - create and rotate a line about its midpoint

2.1 Environment setup

SOL compiler has been developed and tested on Ubuntu 15.04 and Ubuntu 16.04 environments, and capable of supporting others as well. It can work on multiple architecture as the LLVM IR bytecode can further be compiled into architecture specific executables. The environment setup tutorial assumes you have the latest version of SOL compile source code downloaded from our repository.

2.1.1 Building SOL compiler (Ubuntu 15.04 or later version)

1. SOL compiler is written in OCaml. Download the source code from our github repository:
   `https://github.com/bawejakunal/sol`
2. Download and run the bash script below, which installs the latest compatible version of OCaml compiler and opam for *-nix OS environments.
   `https://raw.githubusercontent.com/ocaml/ocaml-ci-scripts/master/.travis-ocaml.sh`
3. Configure the opam environment for importing Ocaml packages
   ```
   eval `opam config env`
   ```
4. Install LLVM compiler toolchain and ocaml bindings, for the SOL compiler
   ```
   ./install-llvm.sh
   ```
5. Download and Install SDL2 and SDL2_gfx libraries which are used by our predefined static library for graphics rendering.
   ```
   wget http://www.ferzkopp.net/Software/SDL2_gfx/SDL2_gfx-1.0.3.tar.gz
   ```
   ```
   ./install-sdl-gfx.sh
   ```

6. Build the SOL compiler as `sol.native` and the static graphic libarary as `predefined.o`
   `make all`

2.2 SOL Quick Tour

This section provides a quick tour on the data types, data structures and the shape oriented programming paradigm provided by SOL, with examples. Towards the end we walk through the steps to generate an executable SOL program. For a detailed explanation of language syntax, semantics and built-int functions refer to chapter 3.

2.2.1 Primitive data types

All variables of primitive data types are declared starting with their type followed by an identification. SOL supports following primitives:

1. `int` - integers
2. `float` - double precision floating point number
3. `char` - single character from `ASCII` character set
4. `string` - a sequence of one or more `ASCII` characters

Example:

```
func main() {
    int x;  /* declare primitives */
    float y;
    char c;
    string s;

    x = 5;  /* assign values to primitives */
    y = 6.5;
    c = 'a';
    s = "better call SOL";
}
```

2.2.2 Arrays and Shapes

SOL supports two complex data structures:

Array

An array is a fixed size sequence of primitives. Individual elements can be accessed within array by specifying indices as integers in range `0` to `1 less than array length`.
Example:

```
func main() {
    int [3]a;       /* declare array of 3 integers */
    a = [1, 2, 3]; /* init array with 3 elements*/
    a[1] = 4;       /* update second element value to 4 */
    a[2] = a[0];    /* last element equal to first element */
}
```

Shape

A collection of one or more variables of *primitives, arrays and/or shapes* which come together to describe a *shape*. These variables are called *member variables* of a shape. A shape definition can also contain function definitions, referred as *member functions*.
Example:

```
class Line {
    string name;  /* identify a line by name */
    int [2]start; /* first end point of a line */
    int [2]end;   /* other end point of a line */

    /* compulsory constructor for a shape */
    construct(int [2]s, int[2]e) {
        /* constructor can be empty definition */
        start = s;
        end = e;
    }

    /* compulsory draw member function.*/
    draw() {
        /* accepts no arguments */
        /* can be empty definition */
    }
}

func main() {
    Line l; /* declare a variable of shape Line */
    l = shape Line([1,1], [2,2]); /* instantiate a Line object */
}
```

2.2.3 Operators

SOL supports arithmetic operations, relational comparisons and logical operations. For all binary operators described in this section, the operands are specified to the left and right, respectively, of the operator and *both operands must be expressions of same data types*.
All logical operators accept *boolean logic expressions*,represented as *integer expression* in SOL. Non-zero expressions correspond to `true` and a `zero` corresponds to `false`.

1. *Binary arithmetic operators*: `+, -, *, / %`
2. *Relational operators (binary)*: `==, |=, >, >=, <, <=`
3. *Logical operators*: `&&, ||` *(operands must be of type int)*
4. *Logical not*: `!` *(unary and right associative)*
5. *Unary negation*: `-` *(unary and right associative)*

All expressions are evaluated left to right. Please refer to section 3.5.2 for exact order of preference and associativity rules for each operator.

Example:

```
func main() {
    int x;
    int y;
    int c;
    float f;
    float g;

    x = 2;
    y = 3;
    f = 3.0;
    g = 6.0;

    y = x + y;  /*5: integer addition */
    x = y - x;  /*3: subtraction */

    g = g * f;  /* 18: floating point multiplication*/
    f = g / f;  /* 6: floating point division */
    y = y % 2;  /*1: modulo operation */

    c = g == f; /*0: EQUALITY is false */
    c = g != f; /*1: NOT EQUAL is true */
    c = g > f;  /*1: g GREATER THAN f */
    c = y >= x; /*1: y GREATER THAN OR EQUAL x is true */
    c = y < x;  /*0: y LESS THAN x is false */
    c = 5 <= 5; /*1: 5 LESS THAN OR EQUALS 5 is true */

    c = 5 && 0; /*0: LOGICAL AND of true and false */
    c = 2 || 0; /*1: LOGICAL OR of true and false */

    c = !c;    /*0: LOGICAL NOT of true(1) */
    f = -f;    /*-6: unary negation of arithmetic expression */
}
```

2.2.4 Control flow statements

SOL program statements are executed in order, with the entry point being the main function. However, sometimes developers need to execute only a branch of source code (jump through some statements) or execute a portion of code repeatedly. SOL provides two control flow statements if and while for conditional branching and looping through a portion of code, respectively.

if-statement

An *if statement block* allows to execute or skip a code branch based on a *logical expression*. Example:

```
func main() {
  if (1 > 2) {
    /* condition if false; skip this code block */
    consolePrint("NOT PRINTED");
  }
  consolePrint("Hello World");
}
```

while-statement

A while statement block allows to execute a portion of source code repeatedly until a logical

```
func main() {
    int i;
    i = 1;
    while (i <= 5) {
        consolePrint("Hello");  /* this loop prints Hello 5 times*/
        i = i + 1;    /* loop terminates when i exceeds 5 */
    }
}
```

2.2.5 Functions in SOL

In SOL functions can be defined as a way to abstract away and re-use a code lock, with a named representation. These are useful if a particular piece of code needs to be executed at multiple places in the program. Functions in SOL can be defined as stand alone functions or as *member functions* of a *shape* definition. Function definitions begin with the `func` keyword and they optionally accept arguments as input values and return a result, for which the result type needs to be indicated in the function definition. A function may not return any result (void type) in which case no return type needs to be mentioned during function definition. Please refer section 3.6.4 for a detailed explanation of function definition syntax.

Example:

```
/* define a function that accepts two integers and returns their
   sum */
```

```
func int add(int x, int y) {
    /* sum of two integers */
    return x + y;
}

func main() {
    int sum;
    sum = add(2, 3);
    if (sum == 5) {
        consolePrint("CORRECT");
    }
}
```

Functions in SOL also support recursion. SOL also provides a number a of type conversion functions and built-in functions for displaying and animating shapes on screen and printing text on screen or console. Please refer to section 3.8 for detailed information, syntax on these functions.

2.3 Rotate a Line around midpoint (Sample SOL program)

The following program shows a simple example to define a line, create it and rotate it around its midpoint.

1. As SOL treats all animation as an interaction of *shapes*, we first define a straight line as a *shape* in SOL, define its `draw` function.

```
shape Line {
    int [2]start;
    int [3]mid;
    int [2]end;

    construct(int [2]s, int [2]e) {
        start = s;
        end = e;
        mid[0] = (s[0] + e[0]) / 2;
        mid[1] = (s[1] + e[1]) / 2;
    }

  draw() {
    /* draw red line */
    drawCurve(start, mid, end, 100, [200, 0, 0]);
  }
}
```

2. The next step is to create a line object and instantiate it within `main` function, which is the entry point to a SOL program.

```
func main() {
  Line l; /* declare a line shape */

  /* create a line with given end points */
  l = shape Line([10,10], [100, 100]);

  /* rotate line here */
}
```

Chapter 3

Language Reference Manual

3.1 Introduction

SOL is a simple language that allows programmers to create 2D animations with ease. Programmers will have the ability to define and create objects, known as shapes, and dictate where they appear, and how they move. As a lightweight object-oriented language, SOL allows for unlimited design opportunities and eases the burden of animation. In addition, SOLs simplicity saves programmers the trouble of learning complicated third-party animation tools, without sacrificing control over behavior of objects.

3.2 Conventions

The following conventions are followed throughout this SOL Reference Manual.

1. `literal` - Fixed space font for literals such as commands, functions, keywords, and programming language structures.
2. *variable* - The variables for SOL progamming language and words or concept being defined are denoted in italics.

The following conventions are applied while drawing and animating objects, using internal functions (see Section 3.8):

1. The origin of the drawing canvas is on the top left of the screen.
2. The positive X-axis goes from left to right.
3. The positive Y-axis goes from top to bottom.
4. Positive angles specify rotation in a clockwise direction.
5. Coordinates are specified as integer arrays of size 2, consisting of an X-coordinate followed by a Y-coordinate.
6. Colors are specified as integer arrays of size 3, consisting of Red, Green and Blue values in the range 0 - 255, where `[0, 0, 0]` is black and `[255, 255, 255]` is white.

3.3 Lexical Conventions

This section describes the complete lexical conventions followed for a syntactically correct SOL program, forming various parts of the language.

3.3.1 *Comments*

Comments in SOL start with character sequence `/*` and end at character sequence `*/`. They may extend over multiple lines and all characters following `/*` are ignored until an ending `*/` is encountered.

3.3.2 *Identifiers*

In SOL, an identifier is a sequence of characters from the set of english alphabets, arabic numerals and underscore (_). The first character of an identifier should always be a lower case english alphabet. Identifiers are case sensitive. Identifiers cannot be any of the reserved keywords mentioned in section 3.3.3.

3.3.3 *Keywords*

Keywords in SOL include data types, built-in functions, and control statements, and may not be used as identifiers as they are reserved.

int	if	main	shape
float	while	setFramerate	parent
char	func	getFramerate	extends
string	construct	print	
	return	consolePrint	
		intToString	
		floatToString	
		charToString	
		render	
		wait	
		drawPoint	
		drawCurve	
		translate	
		rotate	

3.3.4 *Integer Constants*

A sequence of one or more digits representing a number in base-10, optionally preceded by a unary negation operator (`-`), to represent negative integers.
Eg: `1234`

3.3.5 *Float Constants*

Similar to an integer, a float has an *integer*, a decimal point (`.`), and a fractional part. Both the integer and fractional part are a sequence of one or more digits. A negative float is represented

by a preceding unary negation operator (`-`).
Eg: `0.55 10.2`

3.3.6 *Character Constants*

An ASCII character within single quotation marks.
Eg: `'x' 'a'`

3.3.7 *Escape Sequences*

The following are special characters represented by escape sequences.

Name	**Escape**
newline	\n
tab	\t
backslash	\\
single quote	\'
double quote	\"
ASCII NUL character	\0

3.3.8 *String constants*

A SOL *string* is a sequence of zero or more *characters* within double quotation marks.
Eg: `"cat"`

3.3.9 *Operators*

SOL has mainly four categories of operators defined below:

Assignment Operator

The right associative *assignment operator* is denoted by the (`=`) symbol having a variable identifier to its left and a valid expression on its right. The *assignment operator* assigns the evaluated value of expression on the right to the variable on left.

Unary Negation Operator

The right associative unary negation operator (`-`) can be used to negate the value of an arithmetic expression.

Arithmetic Operators

The following table describes `binary arithmetic operators` supported in SOL which operate on two `arithmetic expressions` specified before and after the operator respectively. The said expressions must both be of type `int` or `float`. Please refer to section 3.5.2 for precedence and associativity rules.

Operator	Definition
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo

Comparison Operators

The comparison operators are left associative binary operators for comparing values of operands defined as expressions. Please refer to section 3.5.2 for precedence and associativity rules.

Operator	Definition
==	Equality
!=	Not Equals
<	Less than
>	Greater than
<=	Less than or equals
>=	Greater than or equals

Logical Operators

The logical operators evaluate boolean expressions and return an integer as result - with 0 as False and 1 as True. Please refer to section 3.5.2 for precedence and associativity rules.

Operator	Definition
&&	AND
||	OR
!	NOT

3.3.10 *Punctuators*

The following symbols are used for semantic organization in SOL:

Punctuator	Usage
{}	Used to denote a block of code. Must be present as a pair.
()	Specifies conditions for statements before the subsequent code, or denotes the arguments of a function. Must be present as a pair.
[]	Indicates an array. Must be present as a pair.
;	Signals the end of a line of code.
,	Used to separate arguments for a function, or elements in an array definition.

3.4 Identifier Scope

3.4.1 *Block Scope*

Identifier scope is a specific area of code wherein an identifier exists. A scope of an identifier is from its declaration until the end of the code block within which it is declared.

3.4.2 *File Scope*

Any identifier (such as a variable or a function) that is defined outside a code block has file scope i.e. it exists throughout the file.
If an identifier with file scope has the same name as an identifier with block scope, the block-scope identifier gets precedence.

3.5 Expressions and Operators

3.5.1 *Typecasting*

A typecast is the conversion of an expression from one type to another. SOL supports explicit casting of *int* to *float*, *float* to *int*, and *int*, *float* and *char* to *string*. To cast an expression to a different type, place the desired type in parentheses in front of the expression.
Eg: `(int) myFloat /* Returns the integer value of myFloat */`

3.5.2 *Precedence and Associativity*

SOL expressions are evaluated with the following rules:

1. Expressions are evaluated from left to right, operators are left associative, unless stated otherwise.
2. Expressions within parenthesis take highest precedence and evaluated prior to substituting in outer expression.
3. The unary negation operator (`-`) and logical not operator (`!`) are placed at the second level of precedence, above the binary, comparison and logical operators. It groups right to left as described in section 3.3.9.
4. The third level of precedence is taken by multiplication (`*`), division (`/`) and modulo (`%`) operations.
5. Addition (`+`) and subtraction (`-`) operations are at the fourth level of precedence.
6. At the fifth level of precedence are the comparison operators: `<, >, <=, >=`.
7. At sixth level of precedence are the equality comparison operators: `==` and `!=`.
8. The logical operators, OR (`||`) and AND (`&&`) take up the next level of precedence.
9. At the final level of precedence, the right associative assignment operator (`=`) is placed, which ensures that the expression to its right is evaluated before assignment to left variable identifier.

3.5.3 *Dot Accessor*

To access members of a declared `shape` (further described in section 3.11), use the dot accessor '.'.
Eg: `shape_object.point1 /* This accesses the variable point1 within the object shape_object */`

3.6 Declaring Identifers

Declarations determine how an identifier should be interpreted by the compiler. A declaration should include the identifier type and the given name.

3.6.1 *Type Specifiers*

SOL provides four type specifiers for data types:

- *int* - integer number
- *float* - floating point number
- *char* - a single character
- *string* - string (ordered sequence of characters)

3.6.2 *Declaring Variables*

An identifier, also referred to as a *variable* is declared by specifying the `sprimitive type` or name of `Shape`, followed by a valid identifier, as specified in section 3.3.2. Variables can be declared only at the beginning of a function or at the top of the source files, as global variables, which are accessible within all subsequent function or shape definitions.

3.6.3 *Array Declarators*

An array may be formed from any of the primitive types and `shape`s, but each array may only contain one type of primitive or `shape`. At declaration, the type specifier and the size of the array must be indicated. The array size need not be specified for strings, which are character arrays. SOL supports `fixed size arrays`, declared at compile time i.e. a program can not allocate dynamically sized arrays at runtime. Arrays are most commonly used in SOL to specify coordinates with two integers or drawing colors in RGB format with a three element array.
Eg: `int[2] coor; /* Array of two integers */`

3.6.4 *Function Declarators and Definition*

Functions are declared with the keyword: `func`. This is followed by the *return type* of the function. If no return type is specified, then the function automatically does not return any value. Functions are given a name (a valid *identifier*) followed by function formal arguments. These arguments are a comma-separated list of variable declarations within parentheses. Primitives are passed into functions by value, and objects and arrays are passed by reference. This function declaration is then followed by the function definition, within curly braces; functions must always be defined immediately after they are declared.
Function can also be defined within *shape* definitions in which case it is referred as *member function* of the class. (see section 3.11)

Example:

```
func example(int a, int b){
    /* a function named example that takes
       two arguments, both of type int */
}
```

3.6.5 *Constructor Declarators*

Constructors are declared with the keyword: `construct`. Constructor definitions are similar to a function definition with three additional constraints:

1. Constructors are defined inside the class definition
2. A construct is defined with `construct` keyword, followed by optional formal arguments, within parenthesis as a comma-separated list of variable declarations, similar to function definitions
3. Constructors do not have a return type specified

Example:

```
shape Point {
    int [2]coordinate;
    construct (int x, int y) {
        /* constructor definition */
        coordinate[0] = x;
        coordinate[1] = y;
    }
}
```

Please see section 3.11 for defining shapes in SOL and creating shape instances.

3.6.6 *Definitions*

A definition of a primitive type variable includes a value, assigned by the assignment operator '`=`'. For defining arrays, `rvalue` is the sequence of array literals within square brackets. *Shapes* are objects which are initialized by calling the `construct`, with optional parameters (see section 3.11). In SOL programs, all variables *must* be *declared* before assigning values.
Example:

```
char y;         /* declarations */
float z;
int [3]w;       /* array declaration */
string s;
Triangle t;

y = 'b';        /* definitions */
z = 3.4;
w = [5, 2, 0];
s = "cats";
t = shape Triangle();    /* a triangle object */
```

3.7 Statements

A statement in SOL refers to a complete instruction for a SOL program. All statements are executed in order of sequence. The four types of statements are described in detail below:

3.7.1 *Expression Statement*

Expression statements are those statements that get evaluated and produce a result. This can be as simple as an assignment or a function call.
Eg: `x = 5; /* assign 5 to identifier x */`

3.7.2 *If Statement*

An *if* statement is a conditional statement, that is specified with the `if` keyword followed by an *expression* specified within a pair of parenthesis; further followed by a block of code within curly braces. The code specified within the `if` block executes if the expression evaluates to a non-zero *integer*.
Example:

```
int x;
x = 1;
if (x == 1) {
    /* This code gets executed */
}
```

3.7.3 *While Statement*

A *while* statement specifies the looping construct in SOL. It starts with the `while` keyword, followed by an expression specified within a pair of parenthesis; this is followed by a block of code within curly braces which is executed repeatedly as long as the condition in parentheses is valid. This condition is re-evaluated before each iteration and the code within `while` block executes if the condition evaluates to a non-zero *integer*.
Example:

```
int x;
x = 5;
while (x > 0) {
    /* This code gets executed 5 times */
    x = x - 1;
}
```

3.7.4 *Return statement*

Stops execution of a function and returns to where the function was called originally in the code. Potentially returns a value; this value must conform with the return type specified in the function declaration. If no return type was specified, a *return* statement without any value specified is

syntactically valid (but not compulsory).
Example:

```
func int sum(int x, int y) {
    /* return sum of two integers */
    return x + y;
}
```

3.8 Internal Functions

SOL specifies a set of required/internal functions that must be defined for specific tasks such as drawing, rendering or as an entry point to the program, described below.

3.8.1 *main*

Every SOL program must contain a `main` function as this is the entrypoint of the program. The `main` function may, declare and define variables or shape objects or call other functions written in the program. The `main` function does not take inputs as SOL programs do not depend on user input.
Example:

```
func main() {
    /* Entry point for SOL programs */
    int x;  /* variable declaration */
    x = 1;  /* assign value */
    consolePrint("Hello World"); /* call function */
}
```

Arguments: None

3.8.2 *setFramerate*

Call `setFramerate` to specify frames per second to render on screen. The frame rate is specified as a *positive integer argument* and returns `0` for success and `-1` to indicate failure. By default, frame rate is set to 30 frames per second for a SOL program.
Arguments: `rate (int)`
Return: `0 for success, -1 for failure`

3.8.3 *getFramerate*

Call `getFramerate` to get the current number of frames rendered per second as *integer*.
Arguments: `None`
Return: `frames per second (int)`

3.8.4 *consolePrint*

Prints a string to the console. Com]monly used to print error messages.
Arguments: `text (string)`

3.8.5 Type Conversion Functions

SOL provides following type conversion functions for converting expressions of a given type to expression of another type.

intToString

Accepts an expression (`src`) of type `int` as the argument and returns the `string` representation of evaluated result.
Argument: `src (int)`
Return: value of type `string`

floatToString

Accepts an expression (`src`) of type `float` as the argument and returns the `string` representation of evaluated result.
Argument: `src (float)`
Return: value of type `string`

charToString

Accepts an expression (`src`) of type `char` as the argument and returns the `string` representation of evaluated result.
Argument: `src (char)`
Return: value of type `string`

3.9 Drawing Functions

The following set of functions are also a category of internal/required functions, which describe the drawing aspects for `shape` objects defined in a SOL program.

3.9.1 *drawPoint*

Draws a point at a specified coordinate in the specified color.
Arguments: `pt (int[2]), color (int[3])`

3.9.2 *drawCurve*

`drawCurve` is one of the basic internal functions used to draw a Bézier curve. SOL defines all possible shapes as a collection of Bézier curves. The function arguments in order are, the *three control points* for the curve, a *step size* to define smoothness of curve, and the *color* of curve in RGB format.
Arguments: `pt1 (int[2]), pt2 (int[2]), pt3 (int[2]), steps(int), color (int[3])`

3.9.3 *print*

Displays horizontal text on the render screen at the coordinates specified by the user, in specified color.
Arguments: `pt (int[2]), text (string), color (int[3])`

3.9.4 *draw*

For every `shape` definition `draw` is a required function that must be defined by the programmer. The `draw` function does not accept any input arguments and called internally to display the object on screen.

3.10 Animation Functions

The following functions are used to animate the objects drawn in a SOL program.

3.10.1 *translate*

Displaces a `shape` by specifying a two-element array of integers, where the first element is the number of pixels along the horizontal axis and the second element along the vertical axis, over a specified time period in seconds.
Arguments: `displace (int[2]), time (int)`

3.10.2 *rotate*

Rotate a `shape` around an axis point by a specified number of degrees over a time period in seconds.
Arguments: `axis (int[2]), angle (float), time (float)`

3.10.3 *render*

Specify the set of motions to be animated. This code-block can be defined for shapes that need to move or can be left undefined for non-moving shapes. Within this function, various `rotate` and `translate` calls can be made to move the shape. This should be specified in the `main` function.
Arguments: None

3.10.4 *wait*

Pauses animation for a specified amount of time (in seconds). To be called in the `render` function.
Arguments: `time (float)`

3.11 Classes

SOL follows an object-oriented paradigm for defining objects (drawn `shape`s) which can be further animated using the animation functions described in Section 3.10.

3.11.1 *shape*

Similar to a class in C++; a *shape* defines a particular 2-D shape as part of the drawing on screen. The name of a *shape* must always start with an uppercase english alphabet.

Shape definition

A *shape* definition starts with the `shape` keyword, followed by the *shape name*,(eg: `Triangle`) and the definition within curly braces ({}) code block. Shape definitions may optionally contain *member variables*.

Every *shape* must define a *constructor* using `construct` keyword and `draw` function. The `construct` definition can optionally have formal arguments as input parameters. The `draw` function does not accept any arguments and its definition can have multiple `drawPoint`, `drawCurve` and `print` function calls to describe on screen display of the object.

It is possible to define *functions* in a shape definition. The member functions are defined with the same rules as specified in section 3.6.4.
When *member variables* are accessed within a member function, it is implied that the member variables belong to the current object that calls the function. If a *member variable* or *global variable* name is same as that of a local variable or *formal argument* in function definition, then the *local variable* or *formal argument* overshadows the other conflicting variable.

Example:

```
shape Triangle {
    int[2] a; /* Corners of a triangle */
    int[2] b;
    int[2] c;

    int[2] p;  /* mid points of lines*/
    int[2] q;
    int[2] r;

    construct (int [2]x, int [2]y, int [2]z) {
        a = x;
        b = y;
        c = z;

        findCenter(p, a, b);
        findCenter(q, b, c);
        findCenter(r, a, c)
    }

    /* internal draw function definition */
    draw() {
        /* Draw triangle lines with bezier curves */
        drawcurve(a, p, b, 100, [255,0,0]); /*red*/
```

```
        drawcurve(b, q, c, 100, [0,255,0]); /*green*/
        drawcurve(c, r, a, 100, [0,0,255]); /*blue*/
    }

    /* write result in pre-allocated array res */
    func findCenter(int[2]m, int[2]x, int[2]y){
        m[0] = (x[0] + y[0]) / 2;
        m[1] = (x[1] + y[1]) / 2;
    }
}
```

Creating Shape Instances

Actual instances for a shape definition can be created, which represent the actual shapes rendered on the screen.
To instantiate an object for a shape, we first declare a variable of defined *shape* (say `Triangle`) and then instantiate by calling the constructor.
Example:

```
func main() {
    /* declare variable of shape Triangle */
    Triangle t;

    /* instantiate a triangle */
    t = shape Triangle([100,100], [200,100], [150,200]);
}
```

3.11.2 *Inheritance*

SOL allows single class inheritance for shapes i.e given a shape, such as `Line`, one may create a sub-shape of `Line`, called `LineBottom`, and inherit all of its fields from the parent `shape`, `Line`, using the keyword `extends`.
Example:

```
shape Line {
    int[2] a;
    int[2] b;

    construct (int[2] a_init, int[2] b_init) {
        int i;
        i = 0;
        /* copy values */
        while (i < 2) {
            a[i] = a_init[i];
            b[i] = b_init[i];
            i = i + 1;
        }
```

```
    }

    func findCentre(int[2] res, 2int[2] x, int[2] y) {
        /* write result to res */
        int i;
        i = 0;
        while(i < 2) {
            res[i] = (a[i] + b[i]) / 2;
            i = i + 1;
        }
    }

    func draw() {
        drawcurve(a, findCentre(a, b), b, [0, 0, 0]);
    }
}

/* Subclass of Line */
shape LineBottom extends Line {
    int[2] c;
    int[2] d;

    construct (int[2] a_init, int[2] b_init, int[2] c_init) {
        parent(a_init, b_init);
        c = c_init;
        d = b;
    }

    func draw() {
        parent();
        drawcurve(c, findCentre(c, d), d, [0, 0, 0]);
    }
}
```

***parent* (keyword)**

The parent `shape`'s functions can be accessed by the function call `parent()`. This invokes the implementation of the current member function defined in the parent `shape`. In constructors, the `parent()` calls the constructor for the parent `shape`.

Appendices

Appendix A

SOL Compiler

Code listing for compiler code. Author names are mentioned as first comment line of each code listing.

A.1 scanner.mll

```
(* Ocamllex scanner for SOL *)

{ open Parser }

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

```
| "!"       { NOT }
| "if"      { IF }
| "while"   { WHILE }
| "return" { RETURN }
| "int"     { INT }
| "float"   { FLOAT }
| "char"    { CHAR }
| "string"  { STRING }
| "func"    { FUNC }
| "shape"   { SHAPE }
| "construct"  { CONSTRUCT }
| "draw"    { DRAW }
| '.'       { DOT }
(*| "parent"  { PARENT }
| "extends"  { EXTENDS }
| "main"   { MAIN }  (* Consider moving out when main needs to be a
    reserved keyword *)
| "consolePrint"  { CONSOLEPRINT }
| "length"  { LENGTH }
| "setFramerate"  { SETFRAMERATE }
| "translate"  { TRANSLATE }
| "rotate"  { ROTATE }
| "render"  { RENDER }
| "wait"  { WAIT }*)
| ['0'-'9']+'.'['0'-'9']+ as lxm { FLOAT_LITERAL(float_of_string
   lxm) }
| ['0'-'9']+ as lxm { INT_LITERAL(int_of_string lxm) }
| '''[^ '\\' ''' '"']?''' as lxm { CHAR_LITERAL(lxm.[1]) }
| ''''\\'[''' '"' '\\' 't' 'n']''' as lxm { CHAR_LITERAL(lxm.[1]) }
| '"' (('\\'[''' '"' '\\' 't' 'n'])+ | [^ '\\' ''' '"']+)* '"' as
   lxm
  { let str = String.sub (lxm) 1 ((String.length lxm) - 2) in
        let unescaped_str = Scanf.unescaped str in
        STRING_LITERAL(unescaped_str) }
| ['A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { SHAPE_ID(lxm) }
| ['a'-'z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped
   char)) }

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

A.2 parser.mly

```
/* Ocamlyacc parser for SOL */

%{
open Ast
%}

%token SEMI LPAREN RPAREN LBRACE RBRACE LSQUARE RSQUARE COMMA
%token PLUS MINUS TIMES DIVIDE MODULO ASSIGN NOT DOT
%token EQ NEQ LT LEQ GT GEQ AND OR
%token RETURN IF WHILE INT FLOAT CHAR STRING FUNC
%token SHAPE CONSTRUCT DRAW /*PARENT EXTENDS MAIN CONSOLEPRINT
   LENGTH SETFRAMERATE */
/*%token DRAWCURVE DRAWPOINT PRINT
%token TRANSLATE ROTATE RENDER WAIT*/
%token <int> INT_LITERAL
%token <float> FLOAT_LITERAL
%token <char> CHAR_LITERAL
%token <string> STRING_LITERAL
%token <string> ID
%token <string> SHAPE_ID
%token EOF

%right ASSIGN
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE MODULO
%right NOT NEG /* Have to add in parentheses */
%left DOT
%left LPAREN RPAREN LSQUARE RSQUARE

%start program
%type <Ast.program> program

%%

program:
  decls EOF { $1 }

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

```
 | decls sdecl { let (v, s, f) = $1 in v, ($2 :: s), f }

fdecl:
   FUNC ID LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list
      RBRACE /* Handling case for empty return type */
     { { ftype = Void;
   fname = $2;
   formals = $4;
   locals = List.rev $7;
   body = List.rev $8 } }

 | FUNC typ ID LPAREN formals_opt RPAREN LBRACE vdecl_list
    stmt_list RBRACE
     { { ftype = $2;
         fname = $3;
         formals = $5;
         locals = List.rev $8;
         body = List.rev $9 } }

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

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

typ:
    INT { Int }
  | FLOAT { Float }
  | CHAR { Char }
  | STRING { String }
  | SHAPE_ID { Shape($1) }

/*formal_typ:
    typ {$1}
  | formal_typ LSQUARE RSQUARE { Array(0, $1) }*/
/* Removing because we do not need variable length arrays as
   function formal parameters */

local_typ:
    typ {$1}
  | local_typ LSQUARE INT_LITERAL RSQUARE { Array ($3, $1)}
  /* Not adding in Void here*/

vdecl_list:
    /* nothing */    { [] }
```

```
  | vdecl_list vdecl { $2 :: $1 }

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

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

stmt:
    expr SEMI { Expr $1 }
  | RETURN SEMI { Return Noexpr }
  /*| vdecl { VDecl($1, Noexpr) }
  | local_typ ID ASSIGN expr SEMI { VDecl(($1, $2), $4) }*/
  | RETURN expr SEMI { Return $2 }
  | LBRACE stmt_list RBRACE { Block(List.rev $2) }
  | IF LPAREN expr RPAREN stmt { If($3, $5) }
  | WHILE LPAREN expr RPAREN stmt { While($3, $5) }

/*expr_opt:*/
    /* nothing */ /*{ Noexpr }
  | expr          { $1 }*/
/* Removed because only usage was for FOR statements */

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

expr:
    INT_LITERAL          { Int_literal($1) }
  | FLOAT_LITERAL          { Float_literal($1) }
  | CHAR_LITERAL          { Char_literal($1) }
  | STRING_LITERAL          { String_literal($1) }
  | LSQUARE array_expr RSQUARE        { Array_literal(List.length
     $2, List.rev $2) }
  | expr PLUS   expr { Binop($1, Add,   $3) }
  | expr MINUS  expr { Binop($1, Sub,   $3) }
  | expr TIMES  expr { Binop($1, Mult,  $3) }
  | expr DIVIDE expr { Binop($1, Div,   $3) }
  | expr MODULO expr { Binop($1, Mod,   $3) }
  | expr EQ     expr { Binop($1, Equal, $3) }
  | expr NEQ    expr { Binop($1, Neq,   $3) }
  | expr LT     expr { Binop($1, Less,  $3) }
  | expr LEQ    expr { Binop($1, Leq,   $3) }
  | expr GT     expr { Binop($1, Greater, $3) }
  | expr GEQ    expr { Binop($1, Geq,   $3) }
  | expr AND    expr { Binop($1, And,   $3) }
```

```
  | expr OR     expr { Binop($1, Or,    $3) }
  | MINUS expr %prec NEG { Unop(Neg, $2) }
  | NOT expr         { Unop(Not, $2) }
  | lvalue ASSIGN expr   { Assign($1, $3) }
  | ID LPAREN actuals_opt RPAREN { Call($1, $3) }
  | SHAPE SHAPE_ID LPAREN actuals_opt RPAREN { Inst_shape($2, $4) }
  | ID DOT ID LPAREN actuals_opt RPAREN { Shape_fn($1, $3, $5) }
  | LPAREN expr RPAREN { $2 }
  | lvalue  { Lval($1) }
  /* TODO: Include expression for typecasting */

lvalue:
    ID    { Id($1) }
  | ID LSQUARE expr RSQUARE     { Access($1, $3) } /*Access a
     specific element of an array*/
  | ID DOT lvalue   { Shape_var($1, $3) }

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

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

sdecl:
    SHAPE SHAPE_ID LBRACE vdecl_list cdecl ddecl shape_fdecl_list
       RBRACE
      { { sname = $2;
      pname = None;
      member_vs = List.rev $4;
      construct = $5; (* NOTE: Make this optional later *)
      draw = $6;
      member_fs = $7;
      }
    }

cdecl:
   CONSTRUCT LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list
      RBRACE
     { { ftype = Void;
   fname = "constructor";
   formals = $3;
   locals = List.rev $6;
   body = List.rev $7 }
   }
```

```
ddecl:
   DRAW LPAREN RPAREN LBRACE vdecl_list stmt_list RBRACE
     { { ftype = Void;
   fname = "draw";
   formals = [];
   locals = List.rev $5;
   body = List.rev $6 }
   }

shape_fdecl_list:
    /* nothing */ { [] }
  | fdecl_list    { List.rev $1 }

fdecl_list:
    fdecl                   { [$1] }
  | fdecl_list fdecl { $2 :: $1 }
```

A.3 ast.ml

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

type unary_op = Not | Neg

type typ =
          Int
        | Float
        | Char
        | String
        | Void (* For internal use *)
        | Array of int * typ (*first expr is the size of the array
          *)
        | Shape of string
and
   expr =
          Int_literal of int
        | Float_literal of float
        | Char_literal of char
        | String_literal of string
        | Array_literal of int * expr list
        | Binop of expr * op * expr
        | Unop of unary_op * expr
        | Noexpr
        | Assign of lvalue * expr
        | Call of string * expr list
        | Lval of lvalue
```

```
        | Inst_shape of string * expr list
        | Shape_fn of string * string * expr list
and
        lvalue =
          Id of string
        | Access of string * expr
        | Shape_var of string * lvalue

type bind = typ * string

type stmt =
          Block of stmt list
        | Expr of expr
        (* | VDecl of bind * expr *)
        | Return of expr
        | If of expr * stmt
        | While of expr * stmt

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

type shape_dec = {
        sname            :       string;
        pname            :       string option; (*parent name*)
        member_vs        :       bind list;
        construct        :       func_dec;
        draw             :       func_dec;
        member_fs        :       func_dec list;
}

type program = bind list * shape_dec list * func_dec list

(* Pretty-printing functions *)

let string_of_op = function
    Add -> "+"
  | Sub -> "-"
  | Mult -> "*"
  | Div -> "/"
  | Mod -> "%"
  | Equal -> "=="
  | Neq -> "!="
```

```
  | Less -> "<"
  | Leq -> "<="
  | Greater -> ">"
  | Geq -> ">="
  | And -> "&&"
  | Or -> "||"

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

let rec string_of_expr = function
    Int_literal(l) -> string_of_int l
  | Float_literal(l) -> string_of_float l
  | Char_literal(l) -> Char.escaped l
  | String_literal(l) -> l
  | Array_literal(len, l) -> string_of_int len ^ ": [" ^ String.
     concat ", " (List.map string_of_expr l) ^ "]"
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^
        string_of_expr e2
  | Unop(o, e) -> string_of_uop o ^ string_of_expr e
  | Assign(l, e) -> (string_of_lvalue l) ^ " = " ^ string_of_expr e
  | Call(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ "
        )"
  | Inst_shape(s, el) -> "shape " ^ s ^ "(" ^ String.concat ", " (
     List.map string_of_expr el) ^ ")"
  | Shape_fn(s, f, el) ->
      s ^ "." ^ f ^ "(" ^ String.concat ", " (List.map
        string_of_expr el) ^ ")"
  | Noexpr -> ""
  | Lval(l) -> string_of_lvalue l

 and

string_of_lvalue = function
  Id(s) -> s
| Access(id, idx) -> id ^ "[" ^ string_of_expr idx ^ "]"
| Shape_var(s, v) -> s ^ "." ^ (string_of_lvalue v)

and string_of_typ = function
    Int -> "int"
  | Float -> "float"
  | Char -> "char"
  | Void -> "void"
  | String -> "string"
```

```
  | Array(l,t) -> string_of_typ t ^ " [" ^ string_of_int l ^ "]"
  | Shape(s) -> "Shape " ^ s

let rec string_of_stmt = function
    Block(stmts) ->
      "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "
        }\n"
  | Expr(expr) -> string_of_expr expr ^ ";\n";
  (* | VDecl(id, expr) -> string_of_typ (fst id) ^ " " ^ snd id ^
     ": " ^ string_of_expr expr *)
  | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n";
  | If(e, s) -> "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt
     s
  | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^
     string_of_stmt s

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

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

let string_of_sdecl sdecl =
  "Shape " ^ sdecl.sname ^ "(" ^ String.concat ", " (List.map snd
     sdecl.construct.formals) ^
  ")\n Member Variables: " ^ String.concat "" (List.map
     string_of_vdecl sdecl.member_vs) ^
  "\n Draw: " ^ string_of_fdecl sdecl.draw ^
  "\n Member functions: " ^ String.concat "" (List.map
     string_of_fdecl sdecl.member_fs)

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

A.4 semant.ml

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

open Ast
```

```
open Sast

module StringMap = Map.Make(String)

type symbol_table = {
  parent: symbol_table option;
  mutable
  variables: bind list
}

type translation_environment = {
  scope: symbol_table;
  functions: Ast.func_dec StringMap.t;
}

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

let find_local (scope: symbol_table) name =
  try
    let _ = List.find (fun (_, s) -> s = name) scope.variables in
      raise(Failure("Local variable already declared with name " ^
         name))
  with Not_found -> ()

(* Semantic checking of a program. Returns void if successful,
   throws an exception if something is wrong.

   Check each global variable, then check each function *)

let check (globals, shapes, functions) =

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

  (* Raise an exception if a given binding is to a void type *)
```

```
let check_not_void exceptf = function
    (Void, n) -> raise (Failure (exceptf n))
  | _ -> ()
in

(* Raise an exception of the given rvalue type cannot be assigned
     to
    the given lvalue type *)
let check_assign lvaluet rvaluet err =
  let types = (lvaluet, rvaluet) in match types with
      (Array(l1, t1), Array(l2, t2)) -> if t1 == t2 && l1 == l2
         then lvaluet else raise err
    | (Shape(l_s), Shape(r_s)) -> if l_s = r_s then lvaluet else
       raise err
    | _ -> if lvaluet == rvaluet then lvaluet else raise err
in

(**** Checking Global Variables ****)

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

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

(**** Checking Shapes ****)

report_duplicate (fun n -> "duplicate shape " ^ n)
  (List.map (fun sd -> sd.sname) shapes);

let shape_decls = List.fold_left (fun m sd -> StringMap.add sd.
   sname sd m)
                        StringMap.empty shapes
in

let shape_decl s = try StringMap.find s shape_decls
     with Not_found -> raise (Failure ("unrecognized shape " ^ s)
        )
in

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

if List.mem "consolePrint" (List.map (fun fd -> fd.fname)
   functions)
then raise (Failure ("function consolePrint may not be defined"))
    else ();
```

```
if List.mem "setFramerate" (List.map (fun fd -> fd.fname)
   functions)
then raise (Failure ("function setFramerate may not be defined"))
    else ();

if List.mem "length" (List.map (fun fd -> fd.fname) functions)
then raise (Failure ("function length may not be defined")) else
   ();

report_duplicate (fun n -> "duplicate function " ^ n)
  (List.map (fun fd -> fd.fname) functions);

(* Function declaration for a named function *)
let built_in_decls =  StringMap.add "consolePrint"
   { ftype = Void; fname = "consolePrint"; formals = [(String, "x
     ")];
     locals = []; body = [] } (StringMap.add "intToFloat"
   { ftype = Float; fname = "intToFloat"; formals = [(Int, "x")];
     locals = []; body = [] } (StringMap.add "floatToInt"
   { ftype = Int; fname = "floatToInt"; formals = [(Float, "x")];
     locals = []; body = [] } (StringMap.add "intToString"
   { ftype = String; fname = "intToString"; formals = [(Int, "x")
     ];
     locals = []; body = [] } (StringMap.add "floatToString"
   { ftype = String; fname = "floatToString"; formals = [(Float,
     "x")];
     locals = []; body = [] } (StringMap.add "charToString"
   { ftype = String; fname = "charToString"; formals = [(Char, "x
     ")];
     locals = []; body = [] } (StringMap.add "setFramerate"
   { ftype = Void; fname = "setFramerate"; formals = [(Int, "x")
     ];
     locals = []; body = [] } (StringMap.add "getFramerate"
   { ftype = Int; fname = "getFramerate"; formals = [];
     locals = []; body = [] } (StringMap.add "drawCurve"
   { ftype = Void; fname = "drawCurve"; formals =
       [(Array(2, Int), "x"); (Array(2, Int), "y"); (Array(2, Int
          ), "z"); (Int, "stepsize"); (Array(3, Int), "rgb")];
     locals = []; body = [] } (StringMap.add "drawPoint"
   { ftype = Void; fname = "drawPoint"; formals = [(Array(2, Int)
     , "x"); (Array(3, Int), "rgb")];
     locals = []; body = [] } (StringMap.singleton "print"
   { ftype = Void; fname = "print"; formals = [(Array(2, Int), "x
     "); (String, "text"); (Array(3, Int), "rgb")];
     locals = []; body = [] })))))))))))
in
```

```
let function_decls = List.fold_left (fun m fd -> StringMap.add fd
   .fname fd m)
                         built_in_decls functions
in

let function_decl s s_map = try StringMap.find s s_map
     with Not_found -> raise (Failure ("unrecognized function " ^
        s))
in

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

let check_function g_env func =

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

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

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

  report_duplicate (fun n -> "duplicate local " ^ n ^ " in " ^
     func.fname)
    (List.map snd func.locals);

  (* Type of each variable (global, formal, or local *)
  (* let symbols = List.fold_left (fun m (t, n) -> StringMap.add
     n t m)
      StringMap.empty (globals @ func.formals @ func.locals )
  in

  let type_of_identifier s =
    try StringMap.find s symbols
    with Not_found -> raise (Failure ("undeclared identifier " ^
       s))
  in *)
  let map_op tup = match tup with
      (Add, Int) -> IAdd
    | (Sub, Int) -> ISub
    | (Mult, Int) -> IMult
    | (Div, Int) -> IDiv
    | (Equal, Int) -> IEqual
```

```
  | (Neq, Int) -> INeq
  | (Less, Int) -> ILess
  | (Leq, Int) -> ILeq
  | (Greater, Int) -> IGreater
  | (Geq, Int) -> IGeq
  | (And, Int) -> IAnd
  | (Or, Int) -> IOr
  | (Mod, Int) -> IMod
  | (Add, Float) -> FAdd
  | (Sub, Float) -> FSub
  | (Mult, Float) -> FMult
  | (Div, Float) -> FDiv
  | (Equal, Float) -> FEqual
  | (Neq, Float) -> FNeq
  | (Less, Float) -> FLess
  | (Leq, Float) -> FLeq
  | (Greater, Float) -> FGreater
  | (Geq, Float) -> FGeq
  | (Mod, Float) -> FMod
  | (_, _) -> raise(Failure("Invalid operation " ^ (
    string_of_op (fst tup)) ^ " for type " ^ (string_of_typ (
    snd tup)))) in

(* Return the type of an expression or throw an exception *)
let rec expr env = function
          Int_literal i -> SInt_literal(i), Int
  | Float_literal f -> SFloat_literal(f), Float
  | Char_literal c -> SChar_literal(c), Char
  | String_literal s -> SString_literal(s), String
  | Array_literal(l, s) as a -> let prim_type = List.fold_left
    (fun t1 e -> let t2 = snd (expr env e) in
      if t1 == t2 then t1
      else raise (Failure("Elements of differing types found in
          array " ^ string_of_expr (a) ^ ": " ^
        string_of_typ t1 ^ ", " ^ string_of_typ t2)))
    (snd (expr env (List.hd (s)))) (List.tl s) in
    (if l == List.length s then
      let s_s = List.map (fun e -> expr env e) s in
      SArray_literal(l, s_s), Array(l, prim_type)
    else raise(Failure("Something wrong with auto-assigning
       length to array literal " ^ string_of_expr a)))
  | Binop(e1, op, e2) as e ->
      let ta = expr env e1 and tb = expr env e2
      in let _, t1 = ta and _, t2 = tb in
            (match op with
        Add | Sub | Mult | Div | Mod when t1 = Int && t2 = Int
           -> SBinop(ta, map_op (op, Int), tb), Int
```

```
        | Add | Sub | Mult | Div | Mod when t1 = Float && t2 =
          Float -> SBinop(ta, map_op (op, Float), tb), Float
            | Equal | Neq when t1 = t2 && t1 = Int -> SBinop(ta
              , map_op (op, Int), tb), Int
        | Equal | Neq when t1 = t2 && t1 = Float -> SBinop(ta,
          map_op (op, Float), tb), Int
            | Less | Leq | Greater | Geq when t1 = Int && t2 =
              Int -> SBinop(ta, map_op (op, Int), tb), Int
        | Less | Leq | Greater | Geq when t1 = Float && t2 =
          Float -> SBinop(ta, map_op (op, Float), tb), Int
            | And | Or when t1 = Int && t2 = Int -> SBinop(ta,
              map_op (op, Int), tb), Int
        | _ -> raise (Failure ("illegal binary operator " ^
                 string_of_typ t1 ^ " " ^ string_of_op op ^ "
                   " ^
                 string_of_typ t2 ^ " in " ^ string_of_expr e)
                   )
            )
    | Unop(op, e) as ex ->
        let t1 = expr env e
        in let _, t = t1 in
       (match op with
         Neg when t = Int -> SUnop(INeg, t1), Int
  | Neg when t = Float -> SUnop(FNeg, t1), Float
       | Not when t = Int -> SUnop(INot, t1), Int
  | _ -> raise (Failure ("illegal unary operator " ^ string_of_uop
     op ^
                      string_of_typ t ^ " in " ^
                         string_of_expr ex))
      )
    | Noexpr -> SNoexpr, Void
    | Assign(lval, e) as ex ->
        let (slval, lt) = lval_expr env lval and (rexpr, rt) =
           expr env e in
      ignore(check_assign lt rt (Failure ("illegal assignment " ^
          string_of_typ lt ^
          " = " ^ string_of_typ rt ^ " in " ^
          string_of_expr ex)));
      SAssign(slval, (rexpr, rt)), lt
    | Call(fname, actuals) as call -> let fd = function_decl
       fname env.functions in
        ignore(if (fname = "drawCurve" || fname = "drawPoint"||
           fname = "print") then
          if func.fname = "draw"
          then ()
          else raise(Failure("drawCurve/drawPoint/print can only
             be called within a draw()!"))
```

```
        else ()
        );
        if List.length actuals != List.length fd.formals then
         raise (Failure ("expecting " ^ string_of_int
           (List.length fd.formals) ^ " arguments in " ^
              string_of_expr call))
        else (* TODO: Add special case for checking type of
           actual array vs formal array *)
         List.iter2 (fun (ft, _) e -> let _, et = expr env e in
            ignore (check_assign ft et
              (Failure ("illegal actual argument found " ^
                 string_of_typ et ^
              " expected " ^ string_of_typ ft ^ " in " ^
                 string_of_expr e))))
           fd.formals actuals;
         let sactuals = List.map (fun a -> expr env a) actuals in
         let s_fd = {sfname = fd.fname; styp = fd.ftype; sformals
             = fd.formals; slocals = fd.locals;
           sbody = []} in
           (* Not converting the body to a list of stmt_details,
              to prevent recursive conversions,
           and also because this detail is not needed when making
               a function call *)
        SCall(s_fd, sactuals), fd.ftype
    | Shape_fn(s, fname, actuals) as call -> (try
        let (t, _) = find_variable env.scope s in
        match t with
          Shape(sname) -> let sd = shape_decl sname in
            let fd = try List.find (fun member_fd -> fname =
               member_fd.fname) sd.member_fs
              with Not_found -> raise(Failure("Member function "
                 ^ fname ^ " not found in shape declaration " ^
                 sname)) in
            if List.length actuals != List.length fd.formals then
              raise (Failure ("expecting " ^ string_of_int
               (List.length fd.formals) ^ " arguments in " ^
                  string_of_expr call))
            else (* TODO: Add special case for checking type of
               actual array vs formal array *)
             List.iter2 (fun (ft, _) e -> let _, et = expr env e
                in
                ignore (check_assign ft et
                  (Failure ("illegal actual argument found " ^
                     string_of_typ et ^
                  " expected " ^ string_of_typ ft ^ " in " ^
                     string_of_expr e))))
               fd.formals actuals;
```

```
            let sactuals = List.map (fun a -> expr env a)
               actuals in
            let s_fd = {sfname = fd.fname; styp = fd.ftype;
               sformals = fd.formals; slocals = fd.locals;
              sbody = []} in
              (* Not converting the body to a list of
                 stmt_details, to prevent recursive conversions,
              and also because this detail is not needed when
                 making a function call *)
            SShape_fn(s, t, s_fd, sactuals), fd.ftype
         | _ -> raise(Failure("Member function access " ^ fname
            ^ " for a non-shape variable " ^ s))
       with Not_found -> raise(Failure("Undeclared identifier "
          ^ s)))
  | Lval l -> let (slval_det, ltyp) = (lval_expr env l) in
    SLval(slval_det), ltyp
  | Inst_shape (sname, actuals) ->
  (* Check if the shape exists *)
    let sd = shape_decl sname in
     if List.length actuals != List.length sd.construct.formals
        then
       raise (Failure ("expecting " ^ string_of_int
         (List.length sd.construct.formals) ^ " arguments in "
           ^ string_of_sdecl sd))
     else (* TODO: Add special case for checking type of actual
        array vs formal array *)
       List.iter2 (fun (ft, _) e -> let _, et = expr env e in
          ignore (check_assign ft et
            (Failure ("illegal actual argument found " ^
               string_of_typ et ^
            " expected " ^ string_of_typ ft ^ " in " ^
               string_of_expr e))))
         sd.construct.formals actuals;
       let sactuals = List.map (fun a -> expr env a) actuals in
       let s_sd = {ssname = sd.sname; spname = sd.pname;
          smember_vs = sd.member_vs; sconstruct = {sfname = "
          Construct";
         styp = Void; sformals = []; slocals = []; sbody = []};
             sdraw = {sfname = "Draw";
         styp = Void; sformals = []; slocals = []; sbody = []};
             smember_fs = []} in
         (* Not converting the shape completely, to prevent
            recursive conversions,
         and also because this detail is not needed when making
             a shape instantiation *)
     SInst_shape(s_sd, sactuals), Shape(sname)
```

```
and lval_expr env = function
    Id s -> (try
      let (t, _) = find_variable env.scope s in
      ((SId(s), t), t)
      with Not_found -> raise(Failure("Undeclared identifier "
        ^ s)))
  | Access(id, idx) -> (try
      let (t, _) = find_variable env.scope id
      and (idx', t_ix) = expr env idx in
      let eval_type = function
        Array(_, a_t) -> if t_ix == Int
        (* Note: Cannot check if index is within array bounds
           because the value cannot be evaluated at this stage
           *)
          then a_t
          else raise (Failure("Improper array element access:
             ID " ^ id ^ ", index " ^
            string_of_expr idx))
      | _ -> raise (Failure(id ^ "is not an array type"))
      in ((SAccess(id, (idx', t_ix)), t), eval_type t)
      with Not_found -> raise(Failure("Undeclared identifier "
        ^ id)))
  | Shape_var(s, v) -> try
        let (t, _) = find_variable env.scope s in
        match t with
          Shape(sname) -> let sd = shape_decl sname in
            let shape_scope = {parent = Some(env.scope);
               variables = env.scope.variables @ sd.member_vs}
               in
            let shape_env = {env with scope = shape_scope} in
            let (v_slval, val_typ) = (lval_expr shape_env v) in
            ((SShape_var(s, v_slval), t), val_typ)
            (* (match v_slval with
              SId(v_n), _ -> let (v_t, _) = try List.find (fun
                 (_, n) -> n = v_n) sd.member_vs
                with Not_found -> raise(Failure("Member
                   variable " ^ v_n ^ " not found in shape
                   declaration " ^ sname)) in
              ((SShape_var(s, v_slval), t), val_typ)
            | SAccess(id, _), _ -> let _ = try List.find (fun (
               _, n) -> n = id) sd.member_vs
                with Not_found -> raise(Failure("Member
                   variable " ^ id ^ " not found in shape
                   declaration " ^ sname)) in
                ignore(print_string (string_of_typ val_typ));
              ((SShape_var(s, v_slval), t), val_typ)
```

```
              | SShape_var(member_s, _), _ -> let _ = try List.
                find (fun (_, n) -> n = member_s) sd.member_vs
                 with Not_found -> raise(Failure("Member
                   variable " ^ member_s ^ " not found in shape
                    declaration " ^ sname)) in
               ((SShape_var(s, v_slval), t), val_typ)
              ) *)
        | _ -> raise(Failure("Attempted member variable access
           for a non-shape variable " ^ s))
      with Not_found -> raise(Failure("Undeclared identifier "
         ^ s))

and check_bool_expr env e = (let (e', t) = (expr env e) in if t
    != Int (* This is not supposed to be recursive! *)
 then raise (Failure ("expected Int expression (that evaluates
    to 0 or 1) in " ^ string_of_expr e))
 else (e', t))

(* Verify a statement or throw an exception *)
and stmt env = function
    Block sl -> let rec check_block env = function
       [Return _ as s] -> [stmt env s]
     | Return _ :: _ -> raise (Failure "nothing may follow a
       return")
     (* | Block sl :: ss -> (check_block env sl) @ check_block
        env ss *) (* What were you thinking, Edwards? *)
     | s :: ss -> stmt env s :: check_block env ss
     | [] -> []
    in let scope' = {parent = Some(env.scope); variables = []}
    in let env' = {env with scope = scope'}
    in let sl = check_block env' sl in
    scope'.variables <- List.rev scope'.variables;
    SBlock(sl)
  | Expr e -> SExpr(expr env e)
  (* | VDecl(b, e) -> let _ = find_local env.scope (snd b) in
      env.scope.variables <- b :: env.scope.variables;
      (* Check that the expression type is compatible with the
         type of the variable
        EXCEPT when the expression is a Noexpr
      *)
      let lt = fst b in
      let e' = expr env e in
      let rt = snd (e') in let _ = (match rt with
      | Void -> lt
      | _ -> check_assign lt rt "Assign" (Failure ("illegal
         assignment " ^ string_of_typ lt ^
          " = " ^ string_of_typ rt ^ " in " ^
```

```
                string_of_expr e))) in
           SVDecl(b, e') *)
     | Return e -> let e', t = expr env e in if t = func.ftype
        then SReturn((e', t)) else
        raise (Failure ("return gives " ^ string_of_typ t ^ "
           expected " ^
                        string_of_typ func.ftype ^ " in " ^
                           string_of_expr e))

     | If(p, b1) -> let e' = check_bool_expr env p in SIf(e', stmt
         env b1)
     | While(p, s) -> let e' = check_bool_expr env p in SWhile(e',
         stmt env s)
   in

   let l_scope = {parent = Some(g_env.scope); variables = func.
      formals @ func.locals} in
   let l_env = {g_env with scope = l_scope} in

   {sfname = func.fname; styp = func.ftype; sformals = func.
      formals; slocals = func.locals;
     sbody = let sbl = stmt l_env (Block func.body) in
     match sbl with
         SBlock(sl) -> sl
       | _ -> raise(Failure("This isn't supposed to happen!"))}


 in

 let check_shape g_env shape =

   List.iter (check_not_void (fun n -> "illegal void member
      variable " ^ n ^
     " in " ^ shape.sname)) shape.member_vs;

   report_duplicate (fun n -> "duplicate member variable " ^ n ^ "
       in " ^ shape.sname)
     (List.map snd shape.member_vs);

   report_duplicate (fun n -> "duplicate member function " ^ n)
     (List.map (fun fd -> fd.fname) shape.member_fs);

   let function_decls = List.fold_left (fun m fd -> StringMap.add
      fd.fname fd m)
                        g_env.functions shape.member_fs
   in
```

```
    let s_scope = {parent = Some(g_env.scope); variables = g_env.
       scope.variables @ shape.member_vs} in
    let s_env = {scope = s_scope; functions = function_decls} in
    {ssname = shape.sname; spname = None; smember_vs = shape.
       member_vs;
      sconstruct = (let s_construct = check_function s_env shape.
         construct in
        let s_construct = {s_construct with sfname = shape.sname ^
           "__construct"} in
        try( let last_s_construct = List.hd (List.rev s_construct.
           sbody) in (match last_s_construct with
            SReturn(_) -> raise(Failure("Constructor cannot have
               return statement for shape " ^ shape.sname))
          | _ -> s_construct)) with Failure "hd" -> s_construct);
      sdraw = (let s_draw = check_function s_env shape.draw in
        let s_draw = {s_draw with sfname = shape.sname ^ "__draw"}
           in
        try( let last_s_draw = List.hd (List.rev s_draw.sbody) in (
           match last_s_draw with
            SReturn(_) -> raise(Failure("Draw function cannot have
               return statement for shape " ^ shape.sname))
          | _ -> s_draw)) with Failure "hd" -> s_draw);
      smember_fs = (List.map (function f -> let s_f =
         check_function s_env f in
        let s_f = {s_f with sfname = shape.sname ^ "__" ^ s_f.
           sfname} in
        match s_f.styp with
      | Void -> s_f
      | _ -> try(let last_s = List.hd (List.rev s_f.sbody) in (
         match last_s with
        | SReturn(_) -> s_f
        | _ -> raise(Failure("Function must have return statement
           of type " ^ string_of_typ s_f.styp))))
        with Failure "hd" -> s_f
      ) shape.member_fs)}

  in

  (* Check each individual function *)

  let g_scope = {parent = None; variables = globals} in
  let g_env = {scope = g_scope; functions = function_decls} in
  (globals,
    List.map (check_shape g_env) shapes,
    List.map (function f -> let s_f = check_function g_env f in
       match s_f.styp with
    | Void -> s_f
```

```
    | _ -> let last_s = List.hd (List.rev s_f.sbody) in (match
      last_s with
      | SReturn(_) -> s_f
      | _ -> raise(Failure("Function must have return statement of
        type " ^ string_of_typ s_f.styp)))
    ) functions)
```

A.5 sast.ml

```
open Ast

type sop = IAdd | ISub | IMult | IDiv | IEqual | INeq | ILess |
   ILeq | IGreater | IGeq | IAnd | IOr | IMod |
           FAdd | FSub | FMult | FDiv | FEqual | FNeq | FLess |
              FLeq | FGreater | FGeq | FMod
(* I = integer, F = floats, may add strings *)

type sunary_op = INot | INeg | FNeg

type sexpr_detail =
      SInt_literal of int
    | SFloat_literal of float
    | SChar_literal of char
    | SString_literal of string
    | SArray_literal of int * sexpr list
    | SBinop of sexpr * sop * sexpr
    | SUnop of sunary_op * sexpr
    | SNoexpr
    | SAssign of slvalue * sexpr
    | SCall of sfunc_dec * sexpr list
    | SLval of slvalue
    | SInst_shape of sshape_dec * sexpr list
    | SShape_fn of string * typ * sfunc_dec * sexpr list

and sexpr = sexpr_detail * typ

and slvalue_detail =
      SId of string          (* VDecl ? of bind * expr *)
    | SAccess of string * sexpr
    | SShape_var of string * slvalue

and slvalue = slvalue_detail * typ

and stmt_detail =
      SBlock of stmt_detail list
    | SExpr of sexpr
```

```
    (* | SVDecl of bind * sexpr *)
    | SReturn of sexpr
    | SIf of sexpr * stmt_detail
    | SWhile of sexpr * stmt_detail

and sfunc_dec = {
    sfname    :    string;
    styp    :     typ;
    sformals    :    bind list;
    slocals    :    bind list;
    sbody    :    stmt_detail list;
}

and sshape_dec = {
  ssname   : string;
  spname   :   string option; (*parent name*)
  smember_vs : bind list;
  sconstruct : sfunc_dec;
  sdraw    : sfunc_dec;
  smember_fs : sfunc_dec list;
}

type sprogram = bind list * sshape_dec list * sfunc_dec list

(* Pretty-printing functions *)

let string_of_sop = function
   IAdd -> "+"
 | ISub -> "-"
 | IMult -> "*"
 | IDiv -> "/"
 | IMod -> "%"
 | IEqual -> "=="
 | INeq -> "!="
 | ILess -> "<"
 | ILeq -> "<="
 | IGreater -> ">"
 | IGeq -> ">="
 | IAnd -> "&&"
 | IOr -> "||"
 | FAdd -> "+"
 | FSub -> "-"
 | FMult -> "*"
 | FDiv -> "/"
 | FMod -> "%"
 | FEqual -> "=="
 | FNeq -> "!="
```

```
 | FLess -> "<"
 | FLeq -> "<="
 | FGreater -> ">"
 | FGeq -> ">="

let string_of_suop = function
   INeg -> "-"
 | INot -> "!"
 | FNeg -> "-"

let rec string_of_sexpr (s: sexpr) = match fst s with
   SInt_literal(l) -> string_of_int l
 | SFloat_literal(l) -> string_of_float l
 | SChar_literal(l) -> Char.escaped l
 | SString_literal(l) -> l
 | SArray_literal(len, l) -> string_of_int len ^ ": [" ^ String.
    concat ", " (List.map string_of_sexpr l) ^ "]"
 | SBinop(e1, o, e2) ->
     string_of_sexpr e1 ^ " " ^ string_of_sop o ^ " " ^
        string_of_sexpr e2
 | SUnop(o, e) -> string_of_suop o ^ string_of_sexpr e
 | SAssign(l, e) -> (string_of_slvalue l) ^ " = " ^ string_of_sexpr
     e
 | SCall(f, el) ->
     string_of_sfdecl f ^ "(" ^ String.concat ", " (List.map
        string_of_sexpr el) ^ ")"
 | SInst_shape(s, el) -> "shape " ^ s.ssname ^ "(" ^ String.concat
    ", " (List.map string_of_sexpr el) ^ ")"
 | SShape_fn(s, styp, f, el) ->
     s ^ "(" ^ (string_of_typ styp) ^ ")." ^ string_of_sfdecl f ^ "
        (" ^ String.concat ", " (List.map string_of_sexpr el) ^ ")"
 | SNoexpr -> ""
 | SLval(l) -> string_of_slvalue l

and string_of_slvalue = function
  SId(s), _ -> s
| SAccess(id, idx), _ -> id ^ "[" ^ string_of_sexpr idx ^ "]"
| SShape_var(s, v), _ -> s ^ "." ^ (string_of_slvalue v)

and string_of_sstmt = function
   SBlock(stmts) ->
     "{\n" ^ String.concat "" (List.map string_of_sstmt stmts) ^ "
        }\n"
 | SExpr(expr) -> string_of_sexpr expr ^ ";\n";
 (* | SVDecl(id, expr) -> string_of_typ (fst id) ^ " " ^ snd id ^
    ": " ^ string_of_sexpr expr *)
 | SReturn(expr) -> "return " ^ string_of_sexpr expr ^ ";\n";
```

```
| SIf(e, s) -> "if (" ^ string_of_sexpr e ^ ")\n" ^
   string_of_sstmt s
| SWhile(e, s) -> "while (" ^ string_of_sexpr e ^ ") " ^
   string_of_sstmt s

and string_of_svdecl (t, id) = string_of_typ t ^ " " ^ id ^ ";\n"

and string_of_sfdecl fdecl =
 string_of_typ fdecl.styp ^ " " ^
 fdecl.sfname ^ "(" ^ String.concat ", " (List.map snd fdecl.
    sformals) ^
 ")\n{\n" ^
 String.concat "" (List.map string_of_svdecl fdecl.slocals) ^
 String.concat "" (List.map string_of_sstmt fdecl.sbody) ^
 "}\n"

 let string_of_ssdecl sdecl =
  "Shape " ^ sdecl.ssname ^ "(" ^ String.concat ", " (List.map snd
     sdecl.sconstruct.sformals) ^
  ")\n Member Variables: " ^ String.concat "" (List.map
     string_of_svdecl sdecl.smember_vs) ^
  "\n Draw: " ^ string_of_sfdecl sdecl.sdraw ^
  "\n Member functions: " ^ String.concat "" (List.map
     string_of_sfdecl sdecl.smember_fs)

let string_of_sprogram (vars, shapes, funcs) =
 String.concat "" (List.map string_of_svdecl vars) ^ "\n" ^
 String.concat "\n" (List.map string_of_ssdecl shapes) ^ "\n" ^
 String.concat "\n" (List.map string_of_sfdecl funcs)
```

A.6 codegen.ml

```
(* Code generation: translate takes a semantically checked AST and
produces LLVM IR

LLVM tutorial: Make sure to read the OCaml version of the tutorial

http://llvm.org/docs/tutorial/index.html

Detailed documentation on the OCaml LLVM library:

http://llvm.moe/
http://llvm.moe/ocaml/

*)
```

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

module StringMap = Map.Make(String)

(* Define helper function to find index of an element in a list *)
let rec index_of cmp lst idx = match lst with
| [] -> raise(Failure("Element not found!"))
| hd::tl -> if (cmp hd) then idx else index_of cmp tl (idx + 1)

(* Define helper function to create list of integers in a range *)
let range a b =
  let rec aux a b =
    if a > b then [] else a :: aux (a+1) b  in
  if a > b then List.rev (aux b a) else aux a b;;

let translate (globals, shapes, functions) =
  (* ignore(print_string(string_of_int(inst_count) ^ "\n")); *)
  let context = L.global_context () in
  let the_module = L.create_module context "SOL"
  and i32_t  = L.i32_type   context
  and f32_t  = L.double_type context
  and i8_t   = L.i8_type    context
  and void_t = L.void_type  context in

  (* Create map of shape name to its definition, for convenience *)
  let shape_defs = List.fold_left
    (fun m sshape -> StringMap.add sshape.S.ssname sshape m)
    StringMap.empty shapes in
  let shape_def s = (try StringMap.find s shape_defs with Not_found
     -> raise(Failure("shape_def Not_found! " ^ s))) in

  let named_shape_types = List.fold_left
    (fun m ssdecl -> let name = ssdecl.S.ssname in StringMap.add
       name (L.named_struct_type context name) m)
    StringMap.empty shapes in
  let shape_type s = (try StringMap.find s named_shape_types with
     Not_found -> raise(Failure("shape_type Not_found!"))) in

  let rec ltype_of_typ = function
      A.Int -> i32_t
    | A.Float -> f32_t
    | A.Char -> i8_t
    | A.String -> L.pointer_type i8_t
    | A.Void -> void_t
    | A.Array(l, t) -> L.array_type (ltype_of_typ t) l
```

```
    | A.Shape(s) -> shape_type s
    in

  (* Declare each global variable; remember its value in a map *)
  let global_vars =
    let global_var m (t, n) =
      let init = L.const_int (ltype_of_typ t) 0
      in StringMap.add n (L.define_global n init the_module) m in
    List.fold_left global_var StringMap.empty globals in

  let global_vars = StringMap.add "_Running"
    (L.define_global "_Running" (L.const_int (L.i1_type context) 0)
       the_module) global_vars in

  (* Instantiate global constants used for printing/comparisons,
     once *)
  let string_format_str = L.define_global "fmt" (L.const_stringz
     context "%s\n") the_module in
  let int_format_str = L.define_global "int_fmt" (L.const_stringz
     context "%d") the_module in
  let float_format_str = L.define_global "flt_fmt" (L.const_stringz
      context "%f") the_module in
  let char_format_str = L.define_global "char_fmt" (L.const_stringz
      context "%c") the_module in

  (* Declare printf(), which the consolePrint built-in function
     will call *)
  let printf_t = L.var_arg_function_type i32_t [| L.pointer_type
     i8_t |] in
  let printf_func = L.declare_function "printf" printf_t the_module
      in

  (* Declare the built-in startSDL(), which initializes the SDL
     environment *)
  let startSDL_t = L.var_arg_function_type i32_t [| |] in
  let startSDL_func = L.declare_function "startSDL" startSDL_t
     the_module in

  (* Declare the built-in onRenderStartSDL(), which runs before
     every render loop in SDL *)
  let onRenderStartSDL_t = L.var_arg_function_type void_t [| |] in
  let onRenderStartSDL_func = L.declare_function "onRenderStartSDL"
      onRenderStartSDL_t the_module in

  (* Declare the built-in onRenderFinishSDL(), which runs after
     every render loop in SDL *)
  let onRenderFinishSDL_t = L.var_arg_function_type void_t [| |] in
```

```
let onRenderFinishSDL_func = L.declare_function "
   onRenderFinishSDL" onRenderFinishSDL_t the_module in

(* Declare the built-in stopSDL(), which cleans up and exits the
   SDL environment *)
let stopSDL_t = L.var_arg_function_type i32_t [| |] in
let stopSDL_func = L.declare_function "stopSDL" stopSDL_t
   the_module in

(* (* Declare the built-in intToFloat() function *)
let intToFloat_t = L.function_type f32_t [|i32_t|] in
let intToFloat_func = L.declare_function "intToFloat"
   intToFloat_t the_module in

(* Declare the built-in floatToInt() function *)
let floatToInt_t = L.function_type i32_t [|f32_t|] in
let floatToInt_func = L.declare_function "floatToInt"
   floatToInt_t the_module in *)

(* Declare the built-in sprintf() function, used by the *ToString
    functions *)
let sprintf_t = L.var_arg_function_type i32_t [| L.pointer_type
   i8_t; L.pointer_type i8_t |] in
let sprintf_func = L.declare_function "sprintf" sprintf_t
   the_module in

(* Declare the built-in drawCurve(), which draws a curve in SDL
   *)
let drawCurve_t = L.function_type i32_t
  [|  L.pointer_type (L.array_type i32_t 2) ;  L.pointer_type (L.
     array_type i32_t 2) ;
      L.pointer_type (L.array_type i32_t 2) ; i32_t ;  L.
         pointer_type (L.array_type i32_t 3) |] in
let drawCurve_func = L.declare_function "drawCurve" drawCurve_t
   the_module in

(* Declare the built-in drawPoint(), which draws a point in SDL
   *)
let drawPoint_t = L.function_type i32_t
  [| L.pointer_type (L.array_type i32_t 2) ; L.pointer_type (L.
     array_type i32_t 3) |] in
let drawPoint_func = L.declare_function "drawPoint" drawPoint_t
   the_module in

(* Declare the built-in drawPoint(), which draws a point in SDL
   *)
let print_t = L.function_type i32_t
```

```
    [| L.pointer_type (L.array_type i32_t 2) ; L.pointer_type i8_t
       ; L.pointer_type (L.array_type i32_t 3) |] in
  let print_func = L.declare_function "print" print_t the_module in

  (* Declare the built-in setFramerate() function *)
  let setFramerate_t = L.function_type void_t [| i32_t |] in
  let setFramerate_func = L.declare_function "setFramerate"
     setFramerate_t the_module in

  (* Declare the built-in getFramerate() function *)
  let getFramerate_t = L.function_type i32_t [| |] in
  let getFramerate_func = L.declare_function "getFramerate"
     getFramerate_t the_module in

  (* Define each function (arguments and return type) so we can
     call it *)
  let function_decls =
    let function_decl m sfdecl =
      let name = sfdecl.S.sfname
      and formal_types =
        Array.of_list (List.map
          (fun (t,_) -> let ltyp = ltype_of_typ t in
            match t with
              A.Array(_) -> L.pointer_type (ltyp)
            | _ -> ltyp)
        sfdecl.S.sformals)
      in let ftype = (match name with
          "main" -> L.function_type i32_t formal_types
        | _ -> L.function_type (ltype_of_typ sfdecl.S.styp)
           formal_types) in
      StringMap.add name (L.define_function name ftype the_module,
         sfdecl) m in
    List.fold_left function_decl StringMap.empty functions in

  (* Add in member functions for each shape *)
  let function_decls =
    let shape_function_decl m ssdecl =
    let sname = ssdecl.S.ssname in
      let m = List.fold_left (fun m smember_f ->
        let f_name = smember_f.S.sfname
        and formal_types =
          Array.of_list (L.pointer_type (shape_type sname) ::
            List.map (fun (t,_) -> let ltyp = ltype_of_typ t in
              match t with
                A.Array(_) -> L.pointer_type (ltyp)
              | _ -> ltyp) smember_f.S.sformals)
```

```
        in let ftype = L.function_type (ltype_of_typ smember_f.S.
           styp) formal_types in
        StringMap.add f_name (L.define_function f_name ftype
           the_module, smember_f) m)
      m ssdecl.S.smember_fs in
      (* Add in each constructor and draw as well *)
      let construct_name = ssdecl.S.sconstruct.S.sfname and
      formal_types = Array.of_list (List.map (fun (t,_) -> let ltyp
          = ltype_of_typ t in
            match t with
              A.Array(_) -> L.pointer_type (ltyp)
            | _ -> ltyp)  ssdecl.S.sconstruct.S.sformals) in
      let ftype = L.function_type (L.pointer_type (shape_type sname
         )) formal_types in
      let m = StringMap.add construct_name (L.define_function
         construct_name ftype the_module, ssdecl.S.sconstruct) m in
      let draw_name = ssdecl.S.sdraw.S.sfname and
      formal_types = [| L.pointer_type (shape_type sname) |]
        in let ftype = L.function_type (void_t) formal_types in
      StringMap.add draw_name (L.define_function draw_name ftype
         the_module, ssdecl.S.sdraw) m in
    List.fold_left shape_function_decl function_decls shapes in

  (* Fill in the body of each shape type *)
  let shape_decl ssdecl =
    let name = ssdecl.S.ssname in
    let s_type = shape_type name in
    let lmember_vs = List.rev (List.fold_left (fun l (t, _) -> (
       ltype_of_typ t) :: l ) [] ssdecl.S.smember_vs) in
    let lmember_fs = List.rev (List.fold_left (fun l smember_f ->
      let formal_types =
        Array.of_list (List.map (fun (t,_) -> ltype_of_typ t)
           smember_f.S.sformals) in
      let ftype = L.function_type (ltype_of_typ smember_f.S.styp)
         formal_types in
       (L.pointer_type ftype) :: l ) [] ssdecl.S.smember_fs) in
    (L.struct_set_body s_type (Array.of_list(lmember_vs @
       lmember_fs)) false) in

  ignore(List.iter shape_decl shapes);

  (* Fill in the body of the given function *)
  let build_function_body sfdecl member_vars =
    (* ignore(print_string (sfdecl.S.sfname ^ "\n")); *)
    let (the_function, _) = (try StringMap.find sfdecl.S.sfname
       function_decls with Not_found -> raise(Failure("
       build_function_body Not_found!"))) in
```

```
    let builder = L.builder_at_end context (L.entry_block
       the_function) in

    (* SPECIAL CASE: For the main(), add in a call to the
       initalization of the SDL window *)
    let _ = match sfdecl.S.sfname with
        "main" -> ignore(L.build_call startSDL_func [|  |] "
          startSDL" builder)
      | _ -> () in
    (* TODO: Consider storing the returned value somewhere, return
       that as an error *)

    let const_zero = L.const_int i32_t 0 in

    (* Construct the function's "locals": formal arguments and
       locally
       declared variables.  Allocate each on the stack, initialize
          their
       value, if appropriate, and remember their values in the "
          locals" map *)
    let local_vars =
      let add_formal m (t, n) p =
      (* Special case: for member variables, add the first formal (
         pointer to the instance) directly into the map *)
      if (String.length n > 2 && (String.sub n 0 2) = "__")
      then (StringMap.add n p m)
      else(
        L.set_value_name n p;
          let local =
            (match t with
            (* For arrays, use the pointer directly *)
              A.Array(_) -> p
            | _ -> let l = L.build_alloca (ltype_of_typ t) n
                builder in
                 ignore (L.build_store p l builder); l) in
                 StringMap.add n local m
      ) in

      let add_local m (t, n) =
        (* Special case: for constructors, don't re-add the local
           into the map *)
        if (String.length n > 2 && (String.sub n 0 2) = "__")
        then m
        else (
                let local_var = L.build_alloca (ltype_of_typ t) n
                   builder        in
          StringMap.add n local_var m
```

```
      ) in

    let formals = (List.fold_left2 add_formal StringMap.empty
       sfdecl.S.sformals
        (Array.to_list (L.params the_function)) )
    (* (* The only case where a mismatch occurs is for shape-
       member functions, when the first argument is the shape
    - in this case, ignore the first argument *)
    with Invalid_argument("List.fold_left2") -> List.fold_left2
       add_formal StringMap.empty sfdecl.S.sformals
        (List.tl (Array.to_list (L.params the_function))) *) in
    List.fold_left add_local formals sfdecl.S.slocals in

  (* Return the value for a variable or formal argument *)
  let lookup n = try StringMap.find n local_vars
                 with Not_found -> (try StringMap.find n
                    member_vars
                   with Not_found -> (try StringMap.find n
                      global_vars with Not_found -> raise(Failure
                      ("lookup Not_found!"))))
  in

  (* Construct code for an expression; return its value *)
  let rec expr builder loadval = function
           S.SInt_literal(i), _ -> L.const_int i32_t i
    | S.SFloat_literal(f), _ -> L.const_float f32_t f
    | S.SChar_literal(c), _ -> L.const_int i8_t (Char.code c)
    | S.SString_literal(s), _ -> L.build_global_stringptr s "tmp"
        builder
    | S.SNoexpr, _ -> const_zero
    | S.SArray_literal(_, s), (A.Array(_, prim_typ) as t) ->
        let const_array = L.const_array (ltype_of_typ prim_typ) (
           Array.of_list (List.map (fun e -> expr builder true e)
            s)) in
        if loadval then const_array
        else (let arr_ref = L.build_alloca (ltype_of_typ t) "
           arr_ptr" builder in
          ignore(L.build_store const_array arr_ref builder);
             arr_ref)
    | S.SArray_literal(_, _), _ -> raise(Failure("Invalid Array
       literal being created!"))
    | S.SBinop (e1, op, e2), _ ->
          let e1' = expr builder true e1
          and e2' = expr builder true e2 in
      (match op with
        S.IAnd -> L.build_and
          (L.build_icmp L.Icmp.Ne e1' const_zero "tmp" builder)
```

```
          (L.build_icmp L.Icmp.Ne e2' const_zero "tmp" builder)
          "tmp" builder
    | S.IOr -> L.build_or
        (L.build_icmp L.Icmp.Ne e1' const_zero "tmp" builder)
        (L.build_icmp L.Icmp.Ne e2' const_zero"tmp" builder)
        "tmp" builder
    | _ ->  (match op with
          S.IAdd    -> L.build_add
        | S.ISub    -> L.build_sub
        | S.IMult   -> L.build_mul
        | S.IDiv    -> L.build_sdiv
        | S.IMod    -> L.build_srem
        | S.IEqual  -> L.build_icmp L.Icmp.Eq
        | S.INeq    -> L.build_icmp L.Icmp.Ne
        | S.ILess   -> L.build_icmp L.Icmp.Slt
        | S.ILeq    -> L.build_icmp L.Icmp.Sle
        | S.IGreater-> L.build_icmp L.Icmp.Sgt
        | S.IGeq    -> L.build_icmp L.Icmp.Sge
        | S.FAdd    -> L.build_fadd
        | S.FSub    -> L.build_fsub
        | S.FMult   -> L.build_fmul
        | S.FDiv    -> L.build_fdiv
        | S.FMod    -> L.build_frem
        | S.FEqual  -> L.build_fcmp L.Fcmp.Oeq
        | S.FNeq    -> L.build_fcmp L.Fcmp.One
        | S.FLess   -> L.build_fcmp L.Fcmp.Olt
        | S.FLeq    -> L.build_fcmp L.Fcmp.Ole
        | S.FGreater-> L.build_fcmp L.Fcmp.Ogt
        | S.FGeq    -> L.build_fcmp L.Fcmp.Oge
        | _ -> raise(Failure("Found some binary operator that isn
          't handled!"))
        ) e1' e2' "tmp" builder
    )

  | S.SUnop(op, e), _ ->
          let e' = expr builder true e in
            (match op with
              S.INeg    -> L.build_neg e' "tmp" builder
    | S.INot    -> L.build_icmp L.Icmp.Eq e' const_zero "tmp"
      builder
    | S.FNeg    -> L.build_fneg e' "tmp" builder)
  | S.SAssign (lval, s_e), _ -> let e' = expr builder true s_e
     in
                     ignore (L.build_store e' (lval_expr
                       builder lval) builder); e'
  (* L.build_call consolePrint_func [| (expr builder e) |] "
     consolePrint" builder *)
```

```
(* | A.Call ("intToFloat", [e]) ->
L.build_call intToFloat_func [| (expr builder e) |] "
   intToFloat" builder
| A.Call ("floatToInt", [e]) ->
L.build_call floatToInt_func [| (expr builder e) |] "
   floatToInt" builder
| A.Call ("intToString", [e]) ->
L.build_call intToString_func [| (expr builder e) |] "
   intToString" builder
| A.Call ("floatToString", [e]) ->
L.build_call floatToString_func [| (expr builder e) |] "
   floatToString" builder
| A.Call ("charToString", [e]) ->
L.build_call charToString_func [| (expr builder e) |] "
   charToString" builder
| A.Call ("length", [e]) ->
L.build_call length_func [| (expr builder e) |] "length"
   builder
| A.Call ("setFramerate", [e]) ->
L.build_call setFramerate_func [| (expr builder e) |] "
   setFramerate" builder *)
| S.SCall (s_f, act), _ -> let f_name = s_f.S.sfname in
let actuals = List.rev (List.map
  (fun (s_e, t) ->
    (* Send a pointer to array types instead of the actual
       array *)
    match t with
      A.Array(_) -> expr builder false (s_e, t)
    | _ -> expr builder true (s_e, t))
  (List.rev act)) in (* Why reverse twice? *)

(match f_name with
    "consolePrint" -> let fmt_str_ptr =
        L.build_in_bounds_gep string_format_str [| const_zero
            ; const_zero |] "tmp" builder in
      L.build_call printf_func (Array.of_list (fmt_str_ptr ::
          actuals)) "printf" builder
  | "intToString" -> let result = L.build_array_alloca i8_t (
     L.const_int i32_t 12) "intToString" builder in
      let int_fmt_ptr =
        L.build_in_bounds_gep int_format_str [| const_zero ;
           const_zero |] "tmp" builder in
      ignore(L.build_call sprintf_func (Array.of_list (result
          :: int_fmt_ptr :: actuals)) "intToStringResult"
         builder);
      result
```

```
        | "floatToString" -> let result = L.build_array_alloca i8_t
          (L.const_int i32_t 20) "floatToString" builder in
          let flt_fmt_ptr =
            L.build_in_bounds_gep float_format_str [| const_zero
               ; const_zero |] "tmp" builder in
          ignore(L.build_call sprintf_func (Array.of_list (result
              :: flt_fmt_ptr :: actuals)) "floatToStringResult"
             builder);
          result
        | "charToString" -> let result = L.build_array_alloca i8_t
          (L.const_int i32_t 2) "charToString" builder in
          let char_fmt_ptr =
            L.build_in_bounds_gep char_format_str [| const_zero ;
                const_zero |] "tmp" builder in
          ignore(L.build_call sprintf_func (Array.of_list (result
              :: char_fmt_ptr :: actuals)) "charToStringResult"
             builder);
          result
        | "drawCurve" -> L.build_call drawCurve_func (Array.of_list
          (actuals)) "drawCurve_result" builder
        | "drawPoint" -> L.build_call drawPoint_func (Array.of_list
          (actuals)) "drawPoint_result" builder
        | "print" -> L.build_call print_func (Array.of_list(actuals
          )) "print_result" builder
        | "setFramerate" -> L.build_call setFramerate_func (Array.
          of_list(actuals)) "" builder
        | "getFramerate" -> L.build_call getFramerate_func (Array.
          of_list(actuals)) "getFramerate_result" builder
        | _ -> let (fdef, fdecl), actuals =
          (try StringMap.find f_name function_decls, actuals
          with Not_found ->
          (* In this case, another member function is being
             called from inside a member function *)
          let (sname, inst_name) = (match List.hd sfdecl.S.
             sformals with
              (A.Shape(s), n) -> (s, n)
            | _ -> (* In this case, the member function is being
                called from the constructor *)
                 match List.hd sfdecl.S.slocals with
                     (A.Shape(s), n) -> (s, n)
                   | _ -> raise(Failure("SCall Not_found"))
          )
          in
          let new_f_name = sname ^ "__" ^ f_name in
          (StringMap.find new_f_name function_decls, (lookup
             inst_name) :: actuals)) in
              let result = (match fdecl.S.styp with A.Void -> ""
```

```
                                                  | _ -> f_name ^ "
                                                     _result") in
          L.build_call fdef (Array.of_list actuals) result builder)
      | S.SShape_fn(s, styp, s_f, act), _ -> let obj = lookup s in
          let f_name = (match styp with
              A.Shape(sname) -> sname
            | _ -> raise(Failure("Non-shape type object in member
               function call!"))) ^ "__" ^ s_f.S.sfname in
          let actuals = List.rev (List.map
            (fun (s_e, t) ->
              (* Send a pointer to array types instead of the
                 actual array *)
              match t with
                A.Array(_) -> expr builder false (s_e, t)
              | _ -> expr builder true (s_e, t))
              (List.rev act)) in
          let (fdef, fdecl) = (try StringMap.find f_name
             function_decls with Not_found -> raise(Failure("
             SShape_fn Not_found!"))) in
          let result = (match fdecl.S.styp with A.Void -> ""
                                                  | _ -> f_name ^ "
                                                     _result") in
          L.build_call fdef (Array.of_list (obj :: actuals)) result
              builder
      | S.SLval(l), _ -> let lval = lval_expr builder l in
          if loadval then L.build_load lval "tmp" builder
          else lval
      | S.SInst_shape(_, sactuals), A.Shape(sname) ->
          let actuals =
          List.rev (List.map (fun (s_e, t) -> let ll_expr = expr
             builder true (s_e, t) in
            (* Send a pointer to array types instead of the actual
               array *)
            match t with
              A.Array(_) -> let copy = L.build_alloca (ltype_of_typ
                  t) "arr_copy" builder in
                ignore(L.build_store ll_expr copy builder); copy
            | _ -> ll_expr)
            (List.rev sactuals)) in
          (* Call the constructor *)
          let (constr, _) = (try StringMap.find (sname ^ "
             __construct") function_decls with Not_found -> raise(
             Failure("SInst_shape Not_found!"))) in
          let new_inst = L.build_call constr (Array.of_list actuals
             ) (sname ^ "_inst_ptr") builder in
          L.build_load new_inst (sname ^ "_inst") builder
```

```
    | S.SInst_shape(_, _), _ -> raise(Failure("Cannot instantiate
      a shape of non-shape type!"))


  and lval_expr builder = function
    S.SId(s), _ -> lookup s
  | S.SAccess(id, idx), _(* el_typ *) ->
      (* ignore(print_string "access"); *)
      let arr = lookup id in
      let idx' = expr builder true idx in
      (* let arr_len = L.array_length (ltype_of_typ el_typ) in
      if (idx' < const_zero || idx' >= (L.const_int i32_t arr_len
        ))
        then raise(Failure("Attempted access out of array bounds
          "))
        (* TODO: figure out how to check for access out of array
          bounds *)
        else  *)L.build_gep arr [| const_zero ; idx' |] "tmp"
          builder
      (*let id' = lookup id
      and idx' = expr builder idx in
      if idx' < (expr builder (A.Int_literal 0)) || idx' > id
        '.(1) then raise(Failure("Attempted access out of array
        bounds"))
      else L.const_int i32_t idx'*)
  | S.SShape_var(s, v), s_t ->
      let rec resolve_shape_var obj var obj_type =
        (* Find index of variable in the shape definition *)
        match obj_type with
            A.Shape(sname) -> let sdef = shape_def sname in
            (match var with
                S.SId(v_n), _ -> let index = index_of (fun (_,
                  member_var) -> v_n = member_var) sdef.S.
                  smember_vs 0 in
                 L.build_struct_gep obj index "tmp" builder
              | S.SAccess(v_n, idx), _ -> let index = index_of (
                fun (_, member_var) -> v_n = member_var) sdef.S.
                smember_vs 0 in
                  let arr = L.build_struct_gep obj index "tmp"
                    builder in
                  let idx' = expr builder true idx in
                  L.build_gep arr [| const_zero ; idx' |] "tmp"
                    builder
              | S.SShape_var(member_n, member_v), member_t ->
                  let index = index_of (fun (_, member_var) ->
                    member_n = member_var) sdef.S.smember_vs 0
                    in
```

```
                    let id = L.build_struct_gep obj index "tmp"
                       builder in
                    resolve_shape_var id member_v member_t
              )
            | _ -> raise(Failure("Cannot access a shape variable of
                a non-shape type object!"))
        in
        resolve_shape_var (lookup s) v s_t


    in

    (* Invoke "f builder" if the current block doesn't already
       have a terminal (e.g., a branch). *)
    let add_terminal builder f =
      match L.block_terminator (L.insertion_block builder) with
              Some _ -> ()
      | None -> (* ignore(print_string "Found no return statement
         !"); *)ignore (f builder) in

    (* Build the code for the given statement; return the builder
       for
       the statement's successor *)
    let rec stmt builder = function
              S.SBlock sl -> List.fold_left stmt builder sl
      | S.SExpr e -> ignore (expr builder true e); builder
      (* | S.SVDecl ((t, n), e) -> let var = L.build_alloca (
         ltype_of_typ t) n builder in
          let e' = expr builder e in
          ignore(L.build_store e' var builder); builder *)
      | S.SReturn e -> ignore (match sfdecl.S.styp with
                A.Void -> L.build_ret_void builder
              | _ -> L.build_ret (expr builder true e) builder);
                builder
      | S.SIf (predicate, then_stmt) ->
          let pred' = expr builder true predicate in
          let llty_str = L.string_of_lltype (L.type_of pred') in (*
              TODO: Find a less hack-y way to do this! *)
          let bool_val =
            (match llty_str with
                "i32" -> (L.build_icmp L.Icmp.Ne pred' const_zero "
                   tmp" builder)
              | "i1" -> pred'
              | _ -> raise(Failure("Type of predicate is wrong!")))
                 in
```

```
                let merge_bb = L.append_block context "merge"
                   the_function in

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

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

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

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

          let pred_builder = L.builder_at_end context pred_bb in
          let pred' = expr pred_builder true predicate in
          let llty_str = L.string_of_lltype (L.type_of pred') in (*
              TODO: Find a less hack-y way to do this! *)
          let bool_val =
            (match llty_str with
                "i32" -> (L.build_icmp L.Icmp.Ne pred' const_zero "
                   tmp" pred_builder)
              | "i1" -> pred'
              | _ -> raise(Failure("Type of predicate is wrong!")))
                  in

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

    in

    (* Build the code for each statement in the function *)
    let builder = stmt builder (S.SBlock sfdecl.S.sbody) in
```

```
      (* SPECIAL CASE: For the main(), add in a call to the main
         rendering of the SDL window, return its result *)
      let _ = match sfdecl.S.sfname with
          "main" ->
          (* Find all shape objects within scope *)
          let final_objs = List.rev (List.fold_left (fun lst (t, n)
             -> match t with
              A.Shape(sname) -> let inst = lookup n in
                let rec find_child_objs p_lst p_inst p_sname =
                  let v_lst = (p_sname, p_inst) :: p_lst in
                  (* Look through the shape's member variables, to
                     see if it has any other shape members *)
                  let sdef = shape_def p_sname in
                  List.fold_left (fun l (v_t, v_n) -> match v_t with
                      A.Shape(v_sname) -> (* Find reference to
                         variable shape *)
                        let index = index_of (fun (_, member_var) ->
                           v_n = member_var) sdef.S.smember_vs 0 in
                        let v_inst = L.build_struct_gep p_inst index
                           "tmp" builder in
                        (v_sname, v_inst) :: (find_child_objs l
                           v_inst v_sname)
                    | _ -> l) v_lst sdef.S.smember_vs
                in find_child_objs lst inst sname
            | _ -> lst)
          [] (List.rev sfdecl.S.slocals)) in

          (* Create a loop while program is running *)
          let pred_bb = L.append_block context "while" the_function
             in
          ignore (L.build_br pred_bb builder);

          let body_bb = L.append_block context "while_body"
             the_function in
          let builder_body = L.builder_at_end context body_bb in
          (* Build call to onRenderStartSDL() *)
          ignore(L.build_call onRenderStartSDL_func [| |] ""
             builder_body);

          (* Build a call to each shape's draw function *)
          List.iter (fun (n, o) ->
            let (draw_fn, _) = StringMap.find (n ^ "__draw")
               function_decls in
            ignore(L.build_call draw_fn [| o |] "" builder_body) )
          final_objs;

          (* Build call to onRenderFinishSDL() *)
```

```
      ignore(L.build_call onRenderFinishSDL_func [|  |] ""
        builder_body);
      ignore(L.build_br pred_bb builder_body);

      let pred_builder = L.builder_at_end context pred_bb in
      let merge_bb = L.append_block context "merge" the_function
        in
      ignore (L.build_cond_br
        (L.build_load (StringMap.find "_Running" global_vars) "
          _Running_val" pred_builder)
      body_bb merge_bb pred_builder);
      let builder = L.builder_at_end context merge_bb in

      let stopSDL_ret = L.build_alloca i32_t "stopSDL_ret"
        builder in
        ignore(L.build_store (L.build_call stopSDL_func [|  |] "
          stopSDL_ret" builder) stopSDL_ret builder);
        ignore(L.build_ret (L.build_load stopSDL_ret "stopSDL_ret
          " builder) builder)
    | _ -> () in

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

let build_object_function_body sfdecl sdecl =
  let sname = sdecl.S.ssname in
  let stype = shape_type sname in
  let (the_function, _) = (try StringMap.find sfdecl.S.sfname
     function_decls with Not_found -> raise(Failure("
     build_object_function_body Not_found!"))) in
  let builder = L.builder_at_end context (L.entry_block
     the_function) in
  let construct_name = sname ^ "__construct" in
  let inst_name = "__" ^ sname ^ "_inst" in

  let shape_inst =
    if sfdecl.S.sfname = construct_name
    (* SPECIAL CASE: For the construct(), add creation of an
       object of the required type *)
    then L.build_alloca stype inst_name builder
```

```
      (* In all other cases, return the first argument of the
         function *)
      else let inst = Array.get (L.params the_function) 0 in ignore
        (L.set_value_name inst_name inst); inst
  in

  let sfdecl =
    if sfdecl.S.sfname = construct_name
    then {sfdecl with S.styp = A.Shape(sname); S.slocals = (A.
       Shape(sname), inst_name) :: sfdecl.S.slocals}
    else {sfdecl with S.sformals = (A.Shape(sname), inst_name) ::
        sfdecl.S.sformals}
  in

  (* Create pointers to all member variables *)
  let member_vars = List.fold_left
    (fun m ((_, n), i) -> let member_val = L.build_struct_gep
       shape_inst i n builder in
      StringMap.add n member_val m)
    StringMap.empty (List.mapi (fun i v -> (v, i)) sdecl.S.
       smember_vs) in
  let member_vars = if sfdecl.S.sfname = construct_name
    then StringMap.add inst_name shape_inst member_vars
    else member_vars
  in

  (* Build rest of the function body *)
  build_function_body sfdecl member_vars;

  (* SPECIAL CASE: For the construct(), return the instantiated
     object *)
  if sfdecl.S.sfname = construct_name
  then let bbs = L.basic_blocks the_function in
    let builder = L.builder_at_end context (Array.get bbs ((Array
       .length bbs) - 1)) in
    (* build_function_body would have inserted a void return
       statement at the end; remove this *)
    match L.block_terminator (L.insertion_block builder) with
      Some ins -> (L.delete_instruction ins);
        ignore(L.build_ret shape_inst builder)
    | None -> ignore(L.build_ret shape_inst builder)
  else ()

in

List.iter (fun f -> build_function_body f StringMap.empty)
   functions;
```

```
  List.iter (fun s ->
    build_object_function_body s.S.sconstruct s;
    build_object_function_body s.S.sdraw s;
    List.iter (fun f -> build_object_function_body f s) s.S.
       smember_fs;)
  shapes;
  the_module
```

A.7 sol.ml

```
(* Code generation: translate takes a semantically checked AST and
produces LLVM IR

LLVM tutorial: Make sure to read the OCaml version of the tutorial

http://llvm.org/docs/tutorial/index.html

Detailed documentation on the OCaml LLVM library:

http://llvm.moe/
http://llvm.moe/ocaml/

*)

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

module StringMap = Map.Make(String)

(* Define helper function to find index of an element in a list *)
let rec index_of cmp lst idx = match lst with
| [] -> raise(Failure("Element not found!"))
| hd::tl -> if (cmp hd) then idx else index_of cmp tl (idx + 1)

(* Define helper function to create list of integers in a range *)
let range a b =
  let rec aux a b =
    if a > b then [] else a :: aux (a+1) b  in
  if a > b then List.rev (aux b a) else aux a b;;

let translate (globals, shapes, functions) =
  (* ignore(print_string(string_of_int(inst_count) ^ "\n")); *)
  let context = L.global_context () in
  let the_module = L.create_module context "SOL"
  and i32_t  = L.i32_type   context
```

```
and f32_t  = L.double_type context
and i8_t   = L.i8_type    context
and void_t = L.void_type  context in

(* Create map of shape name to its definition, for convenience *)
let shape_defs = List.fold_left
  (fun m sshape -> StringMap.add sshape.S.ssname sshape m)
  StringMap.empty shapes in
let shape_def s = (try StringMap.find s shape_defs with Not_found
    -> raise(Failure("shape_def Not_found! " ^ s))) in

let named_shape_types = List.fold_left
  (fun m ssdecl -> let name = ssdecl.S.ssname in StringMap.add
     name (L.named_struct_type context name) m)
  StringMap.empty shapes in
let shape_type s = (try StringMap.find s named_shape_types with
   Not_found -> raise(Failure("shape_type Not_found!"))) in

let rec ltype_of_typ = function
    A.Int -> i32_t
  | A.Float -> f32_t
  | A.Char -> i8_t
  | A.String -> L.pointer_type i8_t
  | A.Void -> void_t
  | A.Array(l, t) -> L.array_type (ltype_of_typ t) l
  | A.Shape(s) -> shape_type s
  in

(* Declare each global variable; remember its value in a map *)
let global_vars =
  let global_var m (t, n) =
    let init = L.const_int (ltype_of_typ t) 0
    in StringMap.add n (L.define_global n init the_module) m in
  List.fold_left global_var StringMap.empty globals in

let global_vars = StringMap.add "_Running"
  (L.define_global "_Running" (L.const_int (L.i1_type context) 0)
     the_module) global_vars in

(* Instantiate global constants used for printing/comparisons,
   once *)
let string_format_str = L.define_global "fmt" (L.const_stringz
   context "%s\n") the_module in
let int_format_str = L.define_global "int_fmt" (L.const_stringz
   context "%d") the_module in
let float_format_str = L.define_global "flt_fmt" (L.const_stringz
    context "%f") the_module in
```

```
let char_format_str = L.define_global "char_fmt" (L.const_stringz
    context "%c") the_module in

(* Declare printf(), which the consolePrint built-in function
   will call *)
let printf_t = L.var_arg_function_type i32_t [| L.pointer_type
   i8_t |] in
let printf_func = L.declare_function "printf" printf_t the_module
    in

(* Declare the built-in startSDL(), which initializes the SDL
   environment *)
let startSDL_t = L.var_arg_function_type i32_t [| |] in
let startSDL_func = L.declare_function "startSDL" startSDL_t
   the_module in

(* Declare the built-in onRenderStartSDL(), which runs before
   every render loop in SDL *)
let onRenderStartSDL_t = L.var_arg_function_type void_t [| |] in
let onRenderStartSDL_func = L.declare_function "onRenderStartSDL"
    onRenderStartSDL_t the_module in

(* Declare the built-in onRenderFinishSDL(), which runs after
   every render loop in SDL *)
let onRenderFinishSDL_t = L.var_arg_function_type void_t [| |] in
let onRenderFinishSDL_func = L.declare_function "
   onRenderFinishSDL" onRenderFinishSDL_t the_module in

(* Declare the built-in stopSDL(), which cleans up and exits the
   SDL environment *)
let stopSDL_t = L.var_arg_function_type i32_t [| |] in
let stopSDL_func = L.declare_function "stopSDL" stopSDL_t
   the_module in

(* (* Declare the built-in intToFloat() function *)
let intToFloat_t = L.function_type f32_t [|i32_t|] in
let intToFloat_func = L.declare_function "intToFloat"
   intToFloat_t the_module in

(* Declare the built-in floatToInt() function *)
let floatToInt_t = L.function_type i32_t [|f32_t|] in
let floatToInt_func = L.declare_function "floatToInt"
   floatToInt_t the_module in *)

(* Declare the built-in sprintf() function, used by the *ToString
    functions *)
```

```
let sprintf_t = L.var_arg_function_type i32_t [| L.pointer_type
   i8_t; L.pointer_type i8_t |] in
let sprintf_func = L.declare_function "sprintf" sprintf_t
   the_module in

(* Declare the built-in drawCurve(), which draws a curve in SDL
   *)
let drawCurve_t = L.function_type i32_t
  [|  L.pointer_type (L.array_type i32_t 2) ;  L.pointer_type (L.
     array_type i32_t 2) ;
      L.pointer_type (L.array_type i32_t 2) ; i32_t ;  L.
         pointer_type (L.array_type i32_t 3) |] in
let drawCurve_func = L.declare_function "drawCurve" drawCurve_t
   the_module in

(* Declare the built-in drawPoint(), which draws a point in SDL
   *)
let drawPoint_t = L.function_type i32_t
  [| L.pointer_type (L.array_type i32_t 2) ; L.pointer_type (L.
     array_type i32_t 3) |] in
let drawPoint_func = L.declare_function "drawPoint" drawPoint_t
   the_module in

(* Declare the built-in drawPoint(), which draws a point in SDL
   *)
let print_t = L.function_type i32_t
  [| L.pointer_type (L.array_type i32_t 2) ; L.pointer_type i8_t
     ; L.pointer_type (L.array_type i32_t 3) |] in
let print_func = L.declare_function "print" print_t the_module in

(* Declare the built-in setFramerate() function *)
let setFramerate_t = L.function_type void_t [| i32_t |] in
let setFramerate_func = L.declare_function "setFramerate"
   setFramerate_t the_module in

(* Declare the built-in getFramerate() function *)
let getFramerate_t = L.function_type i32_t [| |] in
let getFramerate_func = L.declare_function "getFramerate"
   getFramerate_t the_module in

(* Define each function (arguments and return type) so we can
   call it *)
let function_decls =
  let function_decl m sfdecl =
    let name = sfdecl.S.sfname
    and formal_types =
      Array.of_list (List.map
```

```
        (fun (t,_) -> let ltyp = ltype_of_typ t in
          match t with
            A.Array(_) -> L.pointer_type (ltyp)
          | _ -> ltyp)
      sfdecl.S.sformals)
    in let ftype = (match name with
        "main" -> L.function_type i32_t formal_types
      | _ -> L.function_type (ltype_of_typ sfdecl.S.styp)
         formal_types) in
    StringMap.add name (L.define_function name ftype the_module,
       sfdecl) m in
  List.fold_left function_decl StringMap.empty functions in

(* Add in member functions for each shape *)
let function_decls =
  let shape_function_decl m ssdecl =
  let sname = ssdecl.S.ssname in
    let m = List.fold_left (fun m smember_f ->
      let f_name = smember_f.S.sfname
      and formal_types =
        Array.of_list (L.pointer_type (shape_type sname) ::
          List.map (fun (t,_) -> let ltyp = ltype_of_typ t in
            match t with
              A.Array(_) -> L.pointer_type (ltyp)
            | _ -> ltyp) smember_f.S.sformals)
      in let ftype = L.function_type (ltype_of_typ smember_f.S.
         styp) formal_types in
      StringMap.add f_name (L.define_function f_name ftype
         the_module, smember_f) m)
    m ssdecl.S.smember_fs in
    (* Add in each constructor and draw as well *)
    let construct_name = ssdecl.S.sconstruct.S.sfname and
    formal_types = Array.of_list (List.map (fun (t,_) -> let ltyp
        = ltype_of_typ t in
          match t with
            A.Array(_) -> L.pointer_type (ltyp)
          | _ -> ltyp)  ssdecl.S.sconstruct.S.sformals) in
    let ftype = L.function_type (L.pointer_type (shape_type sname
       )) formal_types in
    let m = StringMap.add construct_name (L.define_function
       construct_name ftype the_module, ssdecl.S.sconstruct) m in
    let draw_name = ssdecl.S.sdraw.S.sfname and
    formal_types = [| L.pointer_type (shape_type sname) |]
      in let ftype = L.function_type (void_t) formal_types in
    StringMap.add draw_name (L.define_function draw_name ftype
       the_module, ssdecl.S.sdraw) m in
  List.fold_left shape_function_decl function_decls shapes in
```

```
(* Fill in the body of each shape type *)
let shape_decl ssdecl =
  let name = ssdecl.S.ssname in
  let s_type = shape_type name in
  let lmember_vs = List.rev (List.fold_left (fun l (t, _) -> (
     ltype_of_typ t) :: l ) [] ssdecl.S.smember_vs) in
  let lmember_fs = List.rev (List.fold_left (fun l smember_f ->
    let formal_types =
      Array.of_list (List.map (fun (t,_) -> ltype_of_typ t)
         smember_f.S.sformals) in
    let ftype = L.function_type (ltype_of_typ smember_f.S.styp)
       formal_types in
      (L.pointer_type ftype) :: l ) [] ssdecl.S.smember_fs) in
  (L.struct_set_body s_type (Array.of_list(lmember_vs @
     lmember_fs)) false) in

ignore(List.iter shape_decl shapes);

(* Fill in the body of the given function *)
let build_function_body sfdecl member_vars =
  (* ignore(print_string (sfdecl.S.sfname ^ "\n")); *)
  let (the_function, _) = (try StringMap.find sfdecl.S.sfname
     function_decls with Not_found -> raise(Failure("
     build_function_body Not_found!"))) in
  let builder = L.builder_at_end context (L.entry_block
     the_function) in

  (* SPECIAL CASE: For the main(), add in a call to the
     initalization of the SDL window *)
  let _ = match sfdecl.S.sfname with
      "main" -> ignore(L.build_call startSDL_func [|  |] "
         startSDL" builder)
    | _ -> () in
  (* TODO: Consider storing the returned value somewhere, return
     that as an error *)

  let const_zero = L.const_int i32_t 0 in

  (* Construct the function's "locals": formal arguments and
     locally
     declared variables.  Allocate each on the stack, initialize
        their
     value, if appropriate, and remember their values in the "
        locals" map *)
  let local_vars =
    let add_formal m (t, n) p =
```

```
    (* Special case: for member variables, add the first formal (
       pointer to the instance) directly into the map *)
    if (String.length n > 2 && (String.sub n 0 2) = "__")
    then (StringMap.add n p m)
    else(
      L.set_value_name n p;
        let local =
          (match t with
          (* For arrays, use the pointer directly *)
            A.Array(_) -> p
          | _ -> let l = L.build_alloca (ltype_of_typ t) n
             builder in
              ignore (L.build_store p l builder); l) in
              StringMap.add n local m
    ) in

    let add_local m (t, n) =
      (* Special case: for constructors, don't re-add the local
         into the map *)
      if (String.length n > 2 && (String.sub n 0 2) = "__")
      then m
      else (
              let local_var = L.build_alloca (ltype_of_typ t) n
                builder        in
        StringMap.add n local_var m
      ) in

    let formals = (List.fold_left2 add_formal StringMap.empty
       sfdecl.S.sformals
        (Array.to_list (L.params the_function)) )
    (* (* The only case where a mismatch occurs is for shape-
       member functions, when the first argument is the shape
    - in this case, ignore the first argument *)
    with Invalid_argument("List.fold_left2") -> List.fold_left2
       add_formal StringMap.empty sfdecl.S.sformals
        (List.tl (Array.to_list (L.params the_function))) *) in
    List.fold_left add_local formals sfdecl.S.slocals in

  (* Return the value for a variable or formal argument *)
  let lookup n = try StringMap.find n local_vars
                 with Not_found -> (try StringMap.find n
                    member_vars
                   with Not_found -> (try StringMap.find n
                      global_vars with Not_found -> raise(Failure
                      ("lookup Not_found!"))))
  in
```

```
(* Construct code for an expression; return its value *)
let rec expr builder loadval = function
          S.SInt_literal(i), _ -> L.const_int i32_t i
  | S.SFloat_literal(f), _ -> L.const_float f32_t f
  | S.SChar_literal(c), _ -> L.const_int i8_t (Char.code c)
  | S.SString_literal(s), _ -> L.build_global_stringptr s "tmp"
     builder
  | S.SNoexpr, _ -> const_zero
  | S.SArray_literal(_, s), (A.Array(_, prim_typ) as t) ->
      let const_array = L.const_array (ltype_of_typ prim_typ) (
         Array.of_list (List.map (fun e -> expr builder true e)
          s)) in
      if loadval then const_array
      else (let arr_ref = L.build_alloca (ltype_of_typ t) "
         arr_ptr" builder in
        ignore(L.build_store const_array arr_ref builder);
          arr_ref)
  | S.SArray_literal(_, _), _ -> raise(Failure("Invalid Array
    literal being created!"))
  | S.SBinop (e1, op, e2), _ ->
        let e1' = expr builder true e1
        and e2' = expr builder true e2 in
    (match op with
      S.IAnd -> L.build_and
        (L.build_icmp L.Icmp.Ne e1' const_zero "tmp" builder)
        (L.build_icmp L.Icmp.Ne e2' const_zero "tmp" builder)
        "tmp" builder
    | S.IOr -> L.build_or
      (L.build_icmp L.Icmp.Ne e1' const_zero "tmp" builder)
      (L.build_icmp L.Icmp.Ne e2' const_zero"tmp" builder)
      "tmp" builder
    | _ ->  (match op with
        S.IAdd    -> L.build_add
      | S.ISub    -> L.build_sub
      | S.IMult   -> L.build_mul
      | S.IDiv    -> L.build_sdiv
      | S.IMod    -> L.build_srem
      | S.IEqual  -> L.build_icmp L.Icmp.Eq
      | S.INeq    -> L.build_icmp L.Icmp.Ne
      | S.ILess   -> L.build_icmp L.Icmp.Slt
      | S.ILeq    -> L.build_icmp L.Icmp.Sle
      | S.IGreater-> L.build_icmp L.Icmp.Sgt
      | S.IGeq    -> L.build_icmp L.Icmp.Sge
      | S.FAdd    -> L.build_fadd
      | S.FSub    -> L.build_fsub
      | S.FMult   -> L.build_fmul
      | S.FDiv    -> L.build_fdiv
```

```
        | S.FMod     -> L.build_frem
        | S.FEqual   -> L.build_fcmp L.Fcmp.Oeq
        | S.FNeq     -> L.build_fcmp L.Fcmp.One
        | S.FLess    -> L.build_fcmp L.Fcmp.Olt
        | S.FLeq     -> L.build_fcmp L.Fcmp.Ole
        | S.FGreater-> L.build_fcmp L.Fcmp.Ogt
        | S.FGeq     -> L.build_fcmp L.Fcmp.Oge
        | _ -> raise(Failure("Found some binary operator that isn
          't handled!"))
        ) e1' e2' "tmp" builder
      )

    | S.SUnop(op, e), _ ->
          let e' = expr builder true e in
            (match op with
              S.INeg     -> L.build_neg e' "tmp" builder
      | S.INot     -> L.build_icmp L.Icmp.Eq e' const_zero "tmp"
        builder
      | S.FNeg     -> L.build_fneg e' "tmp" builder)
    | S.SAssign (lval, s_e), _ -> let e' = expr builder true s_e
      in
                          ignore (L.build_store e' (lval_expr
                            builder lval) builder); e'
    (* L.build_call consolePrint_func [| (expr builder e) |] "
       consolePrint" builder *)
    (* | A.Call ("intToFloat", [e]) ->
    L.build_call intToFloat_func [| (expr builder e) |] "
       intToFloat" builder
    | A.Call ("floatToInt", [e]) ->
    L.build_call floatToInt_func [| (expr builder e) |] "
       floatToInt" builder
    | A.Call ("intToString", [e]) ->
    L.build_call intToString_func [| (expr builder e) |] "
       intToString" builder
    | A.Call ("floatToString", [e]) ->
    L.build_call floatToString_func [| (expr builder e) |] "
       floatToString" builder
    | A.Call ("charToString", [e]) ->
    L.build_call charToString_func [| (expr builder e) |] "
       charToString" builder
    | A.Call ("length", [e]) ->
    L.build_call length_func [| (expr builder e) |] "length"
       builder
    | A.Call ("setFramerate", [e]) ->
    L.build_call setFramerate_func [| (expr builder e) |] "
       setFramerate" builder *)
    | S.SCall (s_f, act), _ -> let f_name = s_f.S.sfname in
```

```
let actuals = List.rev (List.map
  (fun (s_e, t) ->
    (* Send a pointer to array types instead of the actual
       array *)
    match t with
      A.Array(_) -> expr builder false (s_e, t)
    | _ -> expr builder true (s_e, t))
  (List.rev act)) in (* Why reverse twice? *)

(match f_name with
    "consolePrint" -> let fmt_str_ptr =
        L.build_in_bounds_gep string_format_str [| const_zero
            ; const_zero |] "tmp" builder in
      L.build_call printf_func (Array.of_list (fmt_str_ptr ::
          actuals)) "printf" builder
  | "intToString" -> let result = L.build_array_alloca i8_t (
     L.const_int i32_t 12) "intToString" builder in
      let int_fmt_ptr =
        L.build_in_bounds_gep int_format_str [| const_zero ;
           const_zero |] "tmp" builder in
      ignore(L.build_call sprintf_func (Array.of_list (result
          :: int_fmt_ptr :: actuals)) "intToStringResult"
         builder);
      result
  | "floatToString" -> let result = L.build_array_alloca i8_t
      (L.const_int i32_t 20) "floatToString" builder in
      let flt_fmt_ptr =
        L.build_in_bounds_gep float_format_str [| const_zero
           ; const_zero |] "tmp" builder in
      ignore(L.build_call sprintf_func (Array.of_list (result
          :: flt_fmt_ptr :: actuals)) "floatToStringResult"
         builder);
      result
  | "charToString" -> let result = L.build_array_alloca i8_t
     (L.const_int i32_t 2) "charToString" builder in
      let char_fmt_ptr =
        L.build_in_bounds_gep char_format_str [| const_zero ;
            const_zero |] "tmp" builder in
      ignore(L.build_call sprintf_func (Array.of_list (result
          :: char_fmt_ptr :: actuals)) "charToStringResult"
         builder);
      result
  | "drawCurve" -> L.build_call drawCurve_func (Array.of_list
     (actuals)) "drawCurve_result" builder
  | "drawPoint" -> L.build_call drawPoint_func (Array.of_list
     (actuals)) "drawPoint_result" builder
```

```
        | "print" -> L.build_call print_func (Array.of_list(actuals
          )) "print_result" builder
        | "setFramerate" -> L.build_call setFramerate_func (Array.
          of_list(actuals)) "" builder
        | "getFramerate" -> L.build_call getFramerate_func (Array.
          of_list(actuals)) "getFramerate_result" builder
        | _ -> let (fdef, fdecl), actuals =
            (try StringMap.find f_name function_decls, actuals
            with Not_found ->
            (* In this case, another member function is being
               called from inside a member function *)
            let (sname, inst_name) = (match List.hd sfdecl.S.
               sformals with
                (A.Shape(s), n) -> (s, n)
              | _ -> (* In this case, the member function is being
                 called from the constructor *)
                  match List.hd sfdecl.S.slocals with
                      (A.Shape(s), n) -> (s, n)
                    | _ -> raise(Failure("SCall Not_found"))
            )
            in
            let new_f_name = sname ^ "__" ^ f_name in
            (StringMap.find new_f_name function_decls, (lookup
               inst_name) :: actuals)) in
                let result = (match fdecl.S.styp with A.Void -> ""
                                                | _ -> f_name ^ "
                                                  _result") in
          L.build_call fdef (Array.of_list actuals) result builder)
      | S.SShape_fn(s, styp, s_f, act), _ -> let obj = lookup s in
          let f_name = (match styp with
              A.Shape(sname) -> sname
            | _ -> raise(Failure("Non-shape type object in member
               function call!"))) ^ "__" ^ s_f.S.sfname in
          let actuals = List.rev (List.map
            (fun (s_e, t) ->
              (* Send a pointer to array types instead of the
                 actual array *)
              match t with
                A.Array(_) -> expr builder false (s_e, t)
              | _ -> expr builder true (s_e, t))
              (List.rev act)) in
          let (fdef, fdecl) = (try StringMap.find f_name
             function_decls with Not_found -> raise(Failure("
             SShape_fn Not_found!"))) in
          let result = (match fdecl.S.styp with A.Void -> ""
                                          | _ -> f_name ^ "
                                            _result") in
```

```
          L.build_call fdef (Array.of_list (obj :: actuals)) result
              builder
    | S.SLval(l), _ -> let lval = lval_expr builder l in
          if loadval then L.build_load lval "tmp" builder
          else lval
    | S.SInst_shape(_, sactuals), A.Shape(sname) ->
          let actuals =
          List.rev (List.map (fun (s_e, t) -> let ll_expr = expr
              builder true (s_e, t) in
             (* Send a pointer to array types instead of the actual
                array *)
             match t with
               A.Array(_) -> let copy = L.build_alloca (ltype_of_typ
                   t) "arr_copy" builder in
                 ignore(L.build_store ll_expr copy builder); copy
             | _ -> ll_expr)
             (List.rev sactuals)) in
          (* Call the constructor *)
          let (constr, _) = (try StringMap.find (sname ^ "
              __construct") function_decls with Not_found -> raise(
              Failure("SInst_shape Not_found!"))) in
          let new_inst = L.build_call constr (Array.of_list actuals
              ) (sname ^ "_inst_ptr") builder in
          L.build_load new_inst (sname ^ "_inst") builder
    | S.SInst_shape(_, _), _ -> raise(Failure("Cannot instantiate
          a shape of non-shape type!"))


  and lval_expr builder = function
      S.SId(s), _ -> lookup s
    | S.SAccess(id, idx), _(* el_typ *) ->
        (* ignore(print_string "access"); *)
        let arr = lookup id in
        let idx' = expr builder true idx in
        (* let arr_len = L.array_length (ltype_of_typ el_typ) in
        if (idx' < const_zero || idx' >= (L.const_int i32_t arr_len
            ))
          then raise(Failure("Attempted access out of array bounds
              "))
          (* TODO: figure out how to check for access out of array
              bounds *)
          else  *)L.build_gep arr [| const_zero ; idx' |] "tmp"
              builder
        (*let id' = lookup id
        and idx' = expr builder idx in
        if idx' < (expr builder (A.Int_literal 0)) || idx' > id
           '.(1) then raise(Failure("Attempted access out of array
```

```
         bounds"))
      else L.const_int i32_t idx'*)
  | S.SShape_var(s, v), s_t ->
      let rec resolve_shape_var obj var obj_type =
        (* Find index of variable in the shape definition *)
        match obj_type with
            A.Shape(sname) -> let sdef = shape_def sname in
            (match var with
                S.SId(v_n), _ -> let index = index_of (fun (_,
                   member_var) -> v_n = member_var) sdef.S.
                   smember_vs 0 in
                  L.build_struct_gep obj index "tmp" builder
              | S.SAccess(v_n, idx), _ -> let index = index_of (
                 fun (_, member_var) -> v_n = member_var) sdef.S.
                 smember_vs 0 in
                  let arr = L.build_struct_gep obj index "tmp"
                     builder in
                  let idx' = expr builder true idx in
                  L.build_gep arr [| const_zero ; idx' |] "tmp"
                     builder
              | S.SShape_var(member_n, member_v), member_t ->
                  let index = index_of (fun (_, member_var) ->
                     member_n = member_var) sdef.S.smember_vs 0
                     in
                  let id = L.build_struct_gep obj index "tmp"
                     builder in
                  resolve_shape_var id member_v member_t
            )
          | _ -> raise(Failure("Cannot access a shape variable of
              a non-shape type object!"))
      in
      resolve_shape_var (lookup s) v s_t


  in

  (* Invoke "f builder" if the current block doesn't already
     have a terminal (e.g., a branch). *)
  let add_terminal builder f =
    match L.block_terminator (L.insertion_block builder) with
            Some _ -> ()
    | None -> (* ignore(print_string "Found no return statement
       !"); *)ignore (f builder) in

  (* Build the code for the given statement; return the builder
     for
     the statement's successor *)
```

```
  let rec stmt builder = function
          S.SBlock sl -> List.fold_left stmt builder sl
    | S.SExpr e -> ignore (expr builder true e); builder
    (* | S.SVDecl ((t, n), e) -> let var = L.build_alloca (
       ltype_of_typ t) n builder in
        let e' = expr builder e in
        ignore(L.build_store e' var builder); builder *)
    | S.SReturn e -> ignore (match sfdecl.S.styp with
            A.Void -> L.build_ret_void builder
          | _ -> L.build_ret (expr builder true e) builder);
            builder
    | S.SIf (predicate, then_stmt) ->
        let pred' = expr builder true predicate in
        let llty_str = L.string_of_lltype (L.type_of pred') in (*
            TODO: Find a less hack-y way to do this! *)
        let bool_val =
          (match llty_str with
              "i32" -> (L.build_icmp L.Icmp.Ne pred' const_zero "
                 tmp" builder)
            | "i1" -> pred'
            | _ -> raise(Failure("Type of predicate is wrong!")))
               in

              let merge_bb = L.append_block context "merge"
                 the_function in

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

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

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

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

```
        let pred_builder = L.builder_at_end context pred_bb in
        let pred' = expr pred_builder true predicate in
        let llty_str = L.string_of_lltype (L.type_of pred') in (*
            TODO: Find a less hack-y way to do this! *)
        let bool_val =
          (match llty_str with
              "i32" -> (L.build_icmp L.Icmp.Ne pred' const_zero "
                 tmp" pred_builder)
            | "i1" -> pred'
            | _ -> raise(Failure("Type of predicate is wrong!")))
                in

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

  in

  (* Build the code for each statement in the function *)
  let builder = stmt builder (S.SBlock sfdecl.S.sbody) in

  (* SPECIAL CASE: For the main(), add in a call to the main
     rendering of the SDL window, return its result *)
  let _ = match sfdecl.S.sfname with
      "main" ->
      (* Find all shape objects within scope *)
      let final_objs = List.rev (List.fold_left (fun lst (t, n)
         -> match t with
          A.Shape(sname) -> let inst = lookup n in
            let rec find_child_objs p_lst p_inst p_sname =
              let v_lst = (p_sname, p_inst) :: p_lst in
              (* Look through the shape's member variables, to
                 see if it has any other shape members *)
              let sdef = shape_def p_sname in
              List.fold_left (fun l (v_t, v_n) -> match v_t with
                  A.Shape(v_sname) -> (* Find reference to
                     variable shape *)
                    let index = index_of (fun (_, member_var) ->
                       v_n = member_var) sdef.S.smember_vs 0 in
                    let v_inst = L.build_struct_gep p_inst index
                       "tmp" builder in
                    (v_sname, v_inst) :: (find_child_objs l
                       v_inst v_sname)
                | _ -> l) v_lst sdef.S.smember_vs
            in find_child_objs lst inst sname
```

```
        | _ -> lst)
      [] (List.rev sfdecl.S.slocals)) in

      (* Create a loop while program is running *)
      let pred_bb = L.append_block context "while" the_function
        in
      ignore (L.build_br pred_bb builder);

      let body_bb = L.append_block context "while_body"
        the_function in
      let builder_body = L.builder_at_end context body_bb in
      (* Build call to onRenderStartSDL() *)
      ignore(L.build_call onRenderStartSDL_func [|  |] ""
        builder_body);

      (* Build a call to each shape's draw function *)
      List.iter (fun (n, o) ->
        let (draw_fn, _) = StringMap.find (n ^ "__draw")
          function_decls in
        ignore(L.build_call draw_fn [| o |] "" builder_body) )
      final_objs;

      (* Build call to onRenderFinishSDL() *)
      ignore(L.build_call onRenderFinishSDL_func [|  |] ""
        builder_body);
      ignore(L.build_br pred_bb builder_body);

      let pred_builder = L.builder_at_end context pred_bb in
      let merge_bb = L.append_block context "merge" the_function
        in
      ignore (L.build_cond_br
        (L.build_load (StringMap.find "_Running" global_vars) "
          _Running_val" pred_builder)
      body_bb merge_bb pred_builder);
      let builder = L.builder_at_end context merge_bb in

      let stopSDL_ret = L.build_alloca i32_t "stopSDL_ret"
        builder in
        ignore(L.build_store (L.build_call stopSDL_func [|  |] "
          stopSDL_ret" builder) stopSDL_ret builder);
        ignore(L.build_ret (L.build_load stopSDL_ret "stopSDL_ret
          " builder) builder)
    | _ -> () in

  (* Add a return if the last block falls off the end *)
  (* add_terminal builder (match sfdecl.S.styp with
      A.Void -> L.build_ret_void
```

```
    | _ -> L.build_ret const_zero(* L.build_ret (L.const_int (
        ltype_of_typ t) 0) *)) *)
  match sfdecl.S.styp with
      A.Void -> add_terminal builder L.build_ret_void
    | _ -> ()
in

let build_object_function_body sfdecl sdecl =
  let sname = sdecl.S.ssname in
  let stype = shape_type sname in
  let (the_function, _) = (try StringMap.find sfdecl.S.sfname
     function_decls with Not_found -> raise(Failure("
     build_object_function_body Not_found!"))) in
  let builder = L.builder_at_end context (L.entry_block
     the_function) in
  let construct_name = sname ^ "__construct" in
  let inst_name = "__" ^ sname ^ "_inst" in

  let shape_inst =
    if sfdecl.S.sfname = construct_name
    (* SPECIAL CASE: For the construct(), add creation of an
       object of the required type *)
    then L.build_alloca stype inst_name builder
      (* In all other cases, return the first argument of the
         function *)
    else let inst = Array.get (L.params the_function) 0 in ignore
       (L.set_value_name inst_name inst); inst
  in

  let sfdecl =
    if sfdecl.S.sfname = construct_name
    then {sfdecl with S.styp = A.Shape(sname); S.slocals = (A.
       Shape(sname), inst_name) :: sfdecl.S.slocals}
    else {sfdecl with S.sformals = (A.Shape(sname), inst_name) ::
        sfdecl.S.sformals}
  in

  (* Create pointers to all member variables *)
  let member_vars = List.fold_left
    (fun m ((_, n), i) -> let member_val = L.build_struct_gep
       shape_inst i n builder in
      StringMap.add n member_val m)
    StringMap.empty (List.mapi (fun i v -> (v, i)) sdecl.S.
       smember_vs) in
  let member_vars = if sfdecl.S.sfname = construct_name
    then StringMap.add inst_name shape_inst member_vars
    else member_vars
```

```
    in

    (* Build rest of the function body *)
    build_function_body sfdecl member_vars;

    (* SPECIAL CASE: For the construct(), return the instantiated
       object *)
    if sfdecl.S.sfname = construct_name
    then let bbs = L.basic_blocks the_function in
      let builder = L.builder_at_end context (Array.get bbs ((Array
         .length bbs) - 1)) in
      (* build_function_body would have inserted a void return
         statement at the end; remove this *)
      match L.block_terminator (L.insertion_block builder) with
        Some ins -> (L.delete_instruction ins);
          ignore(L.build_ret shape_inst builder)
      | None -> ignore(L.build_ret shape_inst builder)
    else ()

  in

  List.iter (fun f -> build_function_body f StringMap.empty)
     functions;
  List.iter (fun s ->
    build_object_function_body s.S.sconstruct s;
    build_object_function_body s.S.sdraw s;
    List.iter (fun f -> build_object_function_body f s) s.S.
       smember_fs;)
  shapes;
  the_module
```

A.8 predefined.h

```
/*
 * @author: Kunal Baweja
 */
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <stdint.h>
#include <math.h>
#include <unistd.h>

#include "SDL2_gfxPrimitives.h"
#include "SDL2_imageFilter.h"
#include "SDL2_framerate.h"
```

```
#include "SDL2_rotozoom.h"

typedef struct {
    int framerate;
    long int frame_interval; // In nicroseconds
    SDL_Window* window;
    SDL_Renderer* renderer;
    SDL_Event Event;
} GAME;

/* Global variables for graphics management */
GAME theGame;
/* Boolean to keep track of whether the program is still running */
extern bool _Running;
FPSmanager fpsmanager;

int startSDL();
bool onInitSDL();
bool LoadContent();
void onEventSDL(SDL_Event* Event);
void clearSDL();
void onRenderStartSDL();
void onRenderFinishSDL();
int stopSDL();

/* Framerate functions */
int setFramerate(int rate);
int getFramerate();

/* Internal Draw functions of SDL */

bool drawPointUtil(const int point[2], const int rgb[3], const int
   opacity);
bool drawPoint(const int point[2], const int rgb[3]);

bool drawCurveUtil(const Sint16 *vx, const Sint16 *vy, const int
   num,
    const int steps, const int rgb[2], const int opacity);

bool drawCurve(const int start[2], const int mid[2], const int end
   [2],
    const int steps, const int rgb[3]);

/* print on SDL window; returns 0 on success, -1 on failure */
int print(const int pt[2], const char *text, const int color[3]);

/* rotate a coordinate */
```

```
void rotateCoordinate(int pt[2], const int axis[2], const double
   degree);

/* rotate a curve */
void rotateCurve(int start[2], int mid[2], int end[2], const int
   axis[2],
    const double degree);

/* translate a point */
void translatePoint(int pt[2], const int displace[2]);

/* translateCurve */
void translateCurve(int start[2], int mid[2], int end[2],
    const int displace[2]);
```

A.9 predefined.c

```
/*
 * @author: Kunal Baweja
 * Pre-defined functions for SOL
 */

#include "predefined.h"

bool onInitSDL() {
    if(SDL_Init(SDL_INIT_EVERYTHING) < 0) {
        return false;
    }

    if((theGame.window = SDL_CreateWindow("Shape Oriented Language"
       ,100,100,640, 480, SDL_WINDOW_SHOWN)) == NULL) {
        return false;
    }
    //SDL Renderer
    theGame.renderer = SDL_CreateRenderer(theGame.window, -1,
       SDL_RENDERER_ACCELERATED | SDL_RENDERER_PRESENTVSYNC);
    if (theGame.renderer == NULL){
        printf("%s \n", SDL_GetError());
        return 1;
    }
    // Set default values for framerate
    theGame.framerate = 30;
    theGame.frame_interval = 33333;
    return true;
}
```

```
void onEventSDL(SDL_Event* Event) {
    if(Event->type == SDL_QUIT) {
        _Running = false;
    }
}

void clearSDL()
{
    /* clear screen before drawing again */
    SDL_SetRenderDrawColor(theGame.renderer, 242, 242, 242, 255);
    SDL_RenderClear(theGame.renderer);
}

void onRenderStartSDL() {
    while(SDL_PollEvent(&theGame.Event)) {
        onEventSDL(&theGame.Event);
    }

    clearSDL();
}

void onRenderFinishSDL()
{
    SDL_RenderPresent(theGame.renderer);
    // Enforce frame rate by sleeping
    usleep(theGame.frame_interval);
}

int stopSDL()
{
    SDL_DestroyRenderer(theGame.renderer);
    SDL_DestroyWindow(theGame.window);
    SDL_Quit();
    return 0;
}

int startSDL() {

    theGame.window = NULL;
    _Running = true;

    if(onInitSDL() == false) {
        return -1;
    }

    /* initialize frame rate manager */
    SDL_initFramerate(&fpsmanager);
```

```
    return 0;
}

/* draw a point in SOL */
bool drawPointUtil(const int point[2], const int rgb[3], const int
   opacity) {
    pixelRGBA(theGame.renderer, (Sint16)point[0], (Sint16)point[1],
        (Uint8)rgb[0], (Uint8)rgb[1], (Uint8)rgb[2], opacity);
    return true;
}

bool drawPoint(const int point[2], const int rgb[3]) {
    return drawPointUtil(point, rgb, 255);
}

/* helper function to draw a bezier curve in SOL */
bool drawCurveUtil(const Sint16 *vx, const Sint16 *vy, const int
   num,
    const int steps, const int rgb[3], const int opacity) {

    // pass arguments to SDL gfx
    bool res = bezierRGBA(theGame.renderer, vx, vy, num, steps, (
       Uint8)rgb[0],
        (Uint8)rgb[1], (Uint8)rgb[2], (Uint8)opacity);

    return res;
}

/* draw a bezier curve with 3 control points */
bool drawCurve(const int start[2], const int mid[2], const int end
   [2],
    int steps, const int rgb[3]) {
    // printf("(%d, %d), (%d, %d), (%d, %d), %d, (%d, %d, %d)\n",
    //     start[0], start[1], mid[0], mid[1], end[0], end[1],
       steps, rgb[0], rgb[1], rgb[2]);

    const int num = 3;

    Sint16 *vx = NULL;
    Sint16 *vy = NULL;

    // accumulate x and y coordinates
    if ((vx = (Sint16*)malloc(num * sizeof(Sint16))) == NULL)
        return false;

    if ((vy = (Sint16*)malloc(num * sizeof(Sint16))) == NULL) {
```

```
        free(vx);
        return false;
    }

    // x coordinates
    vx[0] = start[0];
    vx[1] = mid[0];
    vx[2] = end[0];

    // y coordinates
    vy[0] = start[1];
    vy[1] = mid[1];
    vy[2] = end[1];

    bool res = drawCurveUtil(vx, vy, num, steps, rgb, 255);

    // memory cleanup
    free(vx);
    free(vy);

    return res;
}


/*
 * set frames per second (positive integer)
 * returns 0 for sucess and -1 for error
 */
int setFramerate(int rate) {
    theGame.framerate = rate;
    theGame.frame_interval = 1e6 / rate;
    return SDL_setFramerate(&fpsmanager, (Uint32)rate);
}


/* get current frame ratre per second */
int getFramerate() {
    return SDL_getFramerate(&fpsmanager);
}

/*
 * print on SDL window
 * returns 0 on success, -1 on failure
 */
int print(const int pt[2], const char *text, const int color[3]) {
    return stringRGBA(theGame.renderer, (Sint16)pt[0], (Sint16)pt
       [1], text,
```

```
        (Uint8)color[0], (Uint8)color[1], (Uint8)color[2], 255);
}

/*
 * rotate a coordinate clockwise by degree
 * around the axis point
 */
void rotateCoordinate(int pt[2], const int axis[2], const double
   degree) {
    // account for actual rotation to perform
    int _d = ((int)(degree * 100)) % 36000;
    double _degree = _d / 100.0;
    _degree *= M_PI / 180.0;

    // translate back to origin
    pt[0] -= axis[0];
    pt[1] -= axis[1];

    // rotate and round off to nearest integers
    pt[0] = (int)nearbyint(pt[0] * cos(_degree) - pt[1] * sin(
       _degree));
    pt[1] = (int)nearbyint(pt[0] * sin(_degree) + pt[1] * cos(
       _degree));

    // translate point back
    pt[0] += axis[0];
    pt[1] += axis[1];
}

/* rotate a bezier curve control points */
void rotateCurve(int start[2], int mid[2], int end[2], const int
   axis[2],
    const double degree) {
    rotateCoordinate(start, axis, degree);
    rotateCoordinate(mid, axis, degree);
    rotateCoordinate(end, axis, degree);
}

/* translate a point by given displacement */
void translatePoint(int pt[2], const int displace[2]) {
    pt[0] += displace[0];
    pt[1] += displace[1];
}

/* translate a bezier curve control points */
void translateCurve(int start[2], int mid[2], int end[2],
    const int displace[2]) {
```

```
    translatePoint(start, displace);
    translatePoint(mid, displace);
    translatePoint(end, displace);
}
```

A.10 Makefile

```
# @author: Kunal Baweja

# Make sure ocamlbuild can find opam-managed packages: first run
# eval `opam config env`
# Easiest way to build: using ocamlbuild, which in turn uses
   ocamlfind

CC = gcc
CFLAGS = -std=c99 -O2 -D_REENTRANT -I/usr/include/SDL2 -Wno-
   implicit
LIBS =
LFLAGS = -lSDL2 -lSDL2_gfx -lm

all : sol.native predefined.o

sol.native:
        ocamlbuild -use-ocamlfind -pkgs llvm,llvm.analysis -cflags
           -w,+a-4 \
                sol.native

sol.d.byte:
        ocamlbuild -use-ocamlfind -pkgs llvm,llvm.analysis -cflags
           -w,+a-4 \
                sol.d.byte

# "make clean" removes all generated files

.PHONY : clean

clean:
        ocamlbuild -clean
        rm -rf testall.log *.diff sol scanner.ml parser.ml parser.
           mli
        rm -rf *.cmx *.cmi *.cmo *.cmx *.o *.s *.ll *.out *.exe *.
           err *.diff

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

```
OBJS = ast.cmx codegen.cmx parser.cmx scanner.cmx semant.cmx sol.
   cmx

sol: $(OBJS)
        ocamlfind ocamlopt -linkpkg -package llvm -package llvm.
           analysis $(OBJS) -o sol

scanner.ml : scanner.mll
        ocamllex scanner.mll

parser.ml parser.mli : parser.mly
        ocamlyacc parser.mly

%.cmo : %.ml
        ocamlc -c $<

%.cmi : %.mli
        ocamlc -c $<

%.cmx : %.ml
        ocamlfind ocamlopt -c -package llvm $<

predefined.o: predefined.c
        $(CC) -c $^ $(CFLAGS) $(LIBS) $(LFLAGS)

# Testing the "bindings" example

### Generated by "ocamldep *.ml *.mli" after building scanner.ml
   and parser.ml
ast.cmo :
ast.cmx :
codegen.cmo : ast.cmo
codegen.cmx : ast.cmx
sol.cmo : semant.cmo scanner.cmo parser.cmi codegen.cmo ast.cmo
sol.cmx : semant.cmx scanner.cmx parser.cmx codegen.cmx ast.cmx
parser.cmo : ast.cmo parser.cmi
parser.cmx : ast.cmx parser.cmi
scanner.cmo : parser.cmi
scanner.cmx : parser.cmx
semant.cmo : ast.cmo
semant.cmx : ast.cmx
parser.cmi : ast.cmo
```

Appendix B

Environment Setup

The following scripts can be used for installing dependencies and setting up environment.

B.1 install-llvm.sh

```
#!/bin/bash

#@author: Kunal Baweja

# update repo data
sudo apt update

# install llvm runtime and m4 preprocessor
sudo apt install --yes llvm-3.8-dev \
    llvm-3.8     \
    llvm-runtime\
    m4           \
    llvm

# install ocaml llvm binding
opam install --yes llvm.3.8
```

B.2 install-sdl-gfx.sh

```
#!/bin/bash

#@author: Kunal Baweja

SDL_GFX="SDL2_gfx-1.0.3"
SDL_GFX_TAR=$SDL_GFX".tar.gz"

# install sdl
sudo apt install --yes libegl1-mesa-dev \
    libgles2-mesa-dev\
```

```
    sdl2-2.0          \
    libsdl2-dev       \
    xdotool


# untar the file folder
tar xvzf $SDL_GFX_TAR

# step into directory
cd $SDL_GFX

# generate
./autogen.sh

# configure
./configure --prefix=/usr

# make
make

# install
sudo make install
```

Appendix C

Automated testing

The first two scripts are used for automated testing on Travis CI. For individual test cases, the author names are mentioned as first line of each test case.

C.1 .travis.yml

```yaml
# @author: Kunal Baweja

language: c

sudo: required

os:
  - linux

env:
  - OCAML_VERSION=4.02

before_install:
  - wget https://raw.githubusercontent.com/ocaml/ocaml-ci-scripts/
    master/.travis-ocaml.sh
  - wget http://www.ferzkopp.net/Software/SDL2_gfx/SDL2_gfx-1.0.3.
    tar.gz

install:
  - bash -ex .travis-ocaml.sh
  - bash -ex install-llvm.sh
  - bash -ex install-sdl-gfx.sh

before_script:
  - eval `opam config env`
  - "export DISPLAY=:99.0"
  - "/sbin/start-stop-daemon --start --quiet --pidfile /tmp/
    custom_xvfb_99.pid --make-pidfile --background --exec /usr/bin
    /Xvfb -- :99 -ac -screen 0 1280x1024x24"
```

```
  - sleep 3

script:
  - make clean all
  - ./testall.sh
  - cat testall.log

notifications:
  email: false
```

C.2 testall.sh

```
#!/bin/bash

#@author: Kunal Baweja

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

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

# Path to the LLVM compiler
LLC="llc"

# Path to the C compiler
CC="cc"

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

SOL="./sol.native"

LIB="predefined.o"

SDL_FLAGS="-lSDL2 -lSDL2_gfx -lm"

# Set time limit for all operations
ulimit -t 30

globallog=testall.log
rm -f $globallog
```

```
error=0
globalerror=0

keep=0

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

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

# close sdl window
closeWindow() {
    # sleep 2 && xdotool key --clearmodifiers --delay 100 alt+F4
    xdotool sleep 2 && xdotool windowactivate --sync $(xdotool
       search --name "Shape Oriented Language") key --
       clearmodifiers --delay 100 alt+F4
}

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

# Run <args>
# Report the command, run it, and report any errors
Run() {
    echo $* 1>&2
    if [[ "$1" == *exe && "$1" != *mnl-* ]]; then
        closeWindow &
    fi
```

```
    eval $* || {
            SignalError "$1 failed on $*"
            return 1
    }
}

# RunFail <args>
# Report the command, run it, and expect an error
# Command may fail, we do not enforce by SignalError
# if it does not fail here
RunFail() {
    echo $* 1>&2
    if [[ "$1" == *exe ]]; then
        closeWindow &
    fi
    eval $* && {
        error=1
        return 1
    }
    return 0
}

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

    echo -n "$basename..."

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

    generatedfiles=""

    generatedfiles="$generatedfiles ${basename}.ll ${basename}.s ${
       basename}.exe ${basename}.out" &&
    Run "$SOL" "$1" ">" "${basename}.ll" &&
    Run "$LLC" "${basename}.ll" ">" "${basename}.s" &&
    Run "$CC" "-o" "${basename}.exe" "${basename}.s" "$LIB" "
       $SDL_FLAGS"&&
    Run "./${basename}.exe" ">" "${basename}.out" &&
    Compare ${basename}.out ${reffile}.gold ${basename}.diff

    # Report the status and clean up the generated files
```

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

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

    echo -n "$basename..."

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

    generatedfiles="${basename}.ll ${basename}.s ${basename}.err ${
       basename}.exe"
    RunFail "$SOL" "$1" "1>" "${basename}.ll" "2>" "${basename}.err
       "
    if [ $error -eq 1 ];
    then
        Run "$LLC" "${basename}.ll" "1>" "${basename}.s" &&
        Run "$CC" "-o" "${basename}.exe" "${basename}.s" "$LIB" "
           $SDL_FLAGS" &&
        RunFail "./${basename}.exe" "1>" "${basename}.err" "2>" "${
           basename}.err"
        error=0
    fi
    Compare ${basename}.err ${reffile}.err ${basename}.diff

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

```
        globalerror=$error
    fi
}

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

shift `expr $OPTIND - 1`

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

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

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

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

# ignore unknown files
for file in $files
do
    case $file in
        *test-*.sol|*mnl-*.sol)
            Check $file 2>> $globallog
            ;;
```

```
        *fail-*.sol)
            CheckFail $file 2>> $globallog
            ;;
    esac
done

exit $globalerror
```

C.3 fail-array-assign.sol

```
/*@author: Kunal Baweja*/

func main() {
    int [5]arr;
    int i;
    string s;

    /* array upper bound checking */
    i = 0;
    while(i < 6) {
        arr[i] = i;
        i = i + 1;
    }
}
```

C.4 test-array-of-shape.sol

```
/* @author: Kunal Baweja */

/* Test arrays of shapes */

/* Define a line */
shape Line {
    int [2]start;
    int [2]mid;
    int [2]end;

    construct(int [2]first, int [2]second) {
        start = first;
        end = second;
        /* line mid point */
        mid[0] = (start[0] + end[0]) / 2;
        mid[1] = (start[1] + end[1]) / 2;
    }

```

```
    draw(){}

    func describe() {
        consolePrint(intToString(start[0]));
        consolePrint(intToString(start[1]));
        consolePrint(intToString(mid[0]));
        consolePrint(intToString(mid[1]));
        consolePrint(intToString(end[0]));
        consolePrint(intToString(end[1]));
    }
}

func main() {

    /* four lines to describe a square */
    Line [4]sq;
    int i;

    /* describe four sides */
    sq[0] = shape Line([1,1], [3,1]); /* top */
    sq[1] = shape Line([3,1], [3,3]); /* right */
    sq[2] = shape Line([3,3], [1,3]); /* bottom */
    sq[3] = shape Line([1,3], [1,1]); /* left */

    /* print end and midpoit of each side */
    i = 0;
    while (i < 4) {
        sq[i].describe();
        i = i + 1;
    }
}
```

C.5 test-char-to-string.sol

```
/*@author: Kunal Baweja*/

func main() {
    char c;
    string s;

    c = 'h';
    s = charToString(c);
    consolePrint(s);
}
```

C.6 fail-div-semantic.sol

```
/*@author: Kunal Baweja*/

func main() {
    /* fail: numerator and denominator of different types */
    float x;
    x = 1.0 / 3;
}
```

C.7 test-add.sol

```
/*@author: Erik Dyer & Kunal Baweja*/

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

func float fadd(float x, float y) {
    return x + y;
}

func main() {
    int x;
    float y;

    /* integer addition */
    x = add(40, 2);
    if (x == 42) {
        consolePrint("CORRECT");
    }
    if (x != 42) {
        consolePrint("INCORRECT");
    }


    /* float addition */
    y = fadd(38.0, 4.0);
    if (y == 42.0) {
        consolePrint("CORRECT");
    }
    if (y != 42.0) {
        consolePrint("INCORRECT");
    }
}
```

C.8 test-precedence.sol

```
/*@author: Kunal Baweja*/

func checkEqual(int x, int y) {
    if (x == y) {
        consolePrint("CORRECT");
    }
    if (x != y) {
        consolePrint("INCORRECT");
    }
}

func main() {
    int x;

    x = 1 + 20 * 3;     /* 61 */
    checkEqual(x, 61);

    x = 1 - 20 * 3;      /* -59 */
    checkEqual(x, -59);

    x = 1 + 18 / 3;      /* 7 */
    checkEqual(x, 7);

    x = 1 - 18 / 3;      /* -5 */
    checkEqual(x, -5);

    /* parenthesis override */
    x = (1 + 5) / 3;     /* 2 */
    checkEqual(x, 2);

    x = (1 - 7) / 3;     /* -2 */
    checkEqual(x, -2);

    /* for same precedence left to right associativity */
    x = 1 - 7 + 3;
    checkEqual(x, -3);

    x = 30 / 3 * 2;
    checkEqual(x, 20);

    /* unary negation precedes other arithmetic operators*/
    x = 3 + -2;
    checkEqual(x, 1);

    x = 3 - -2;
```

```
    checkEqual(x, 5);

    x = 3 * -2;
    checkEqual(x, -6);

    x = 3 / -1;
    checkEqual(x, -3);
}
```

C.9 test-if.sol

```
/*@author: Kunal Baweja*/

func main() {
    if (1) {
        consolePrint("INSIDE IF BLOCK");
    }
}
```

C.10 fail-prod-semantic.sol

```
/*@author: Kunal Baweja*/

func main() {
    int x;
    x = 1.0 * 1;    /* can not multiply float to int */
}
```

C.11 test-empty-function.sol

```
/*@author: Kunal Baweja*/

func empty(){}

func main(){
    consolePrint("BEFORE");
    empty();
    consolePrint("AFTER");
}
```

C.12 test-shape-member-shape.sol

```
/* @author: Kunal Baweja */

/* Test member shapes */

/* Define a line */
shape Line {
    int [2]start;
    int [2]end;

    construct(int [2]first, int [2]second) {
        start = first;
        end = second;
    }

    draw(){}

    func describe() {
        consolePrint(intToString(start[0]));
        consolePrint(intToString(start[1]));
        consolePrint(intToString(end[0]));
        consolePrint(intToString(end[1]));
    }

    func move(int [2]d) {
        int i;
        i = 0;
        while (i < 2) {
            start[i] = start[i] + d[i];
            end[i] = end[i] + d[i];
            i = i + 1;
        }
    }
}

/* Define rectange as a collection of lines */
shape Rectangle {
    Line top;
    Line right;
    Line bottom;
    Line left;

    construct(Line t, Line r, Line b, Line l) {
        top = t;
        right = r;
        bottom = b;
        left = l;
```

```
    }

    draw(){}

    func move(int [2]d) {
        top.move(d);
        right.move(d);
        bottom.move(d);
        left.move(d);
    }

    func describe() {
        top.describe();
        right.describe();
        bottom.describe();
        left.describe();
    }
}

func main() {
    Line t;
    Line b;
    Line r;
    Line l;
    Rectangle sq;

    /* define lines */
    t = shape Line([1,1], [3,1]);
    r = shape Line([3,1], [3,3]);
    b = shape Line([3,3], [1,3]);
    l = shape Line([1,3], [1,1]);

    /* initialize square */
    sq = shape Rectangle(t,r,b,l);
    sq.describe();

    /* move square */
    consolePrint("MOVE [2,2]");
    sq.move([2,2]);

    /*confirm all members are called*/
    sq.describe();
}
```

C.13 fail-array-access-pos.sol

```
/*@author: Erik Dyer*/

func main() {
    int i;
    int [5]array;

    /* assign array elements */
    array = [0,1,2,3,4];

    /* print array elements */
    i = 0;
    consolePrint(intToString(array[5]));
}
```

C.14 fail-parameter-floatint.sol

```
/*@author: Kunal Baweja*/

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

func main() {
    int x;
    x = add(40, 2.5); /* Fail: passing a float to a func that
       expects int */
}
```

C.15 fail-recursion.sol

```
/*@author: Kunal Baweja*/

/* find sum of 1 to n, inclusive */
func int series(int n) {
    /* fail no terminating condition */
    return n + series(n-1);
}

func main() {
    /* crash due to stack overflow */
    consolePrint(intToString(series(-1)));  /*-1*/
}
```

C.16 test-while.sol

```
/*@author: Kunal Baweja*/

func main() {
    int x;
    x = 5;
    while (x > 0) {
        consolePrint("INSIDE WHILE");
        x = x - 1;
    }
}
```

C.17 fail-return-void-int.sol

```
/*@author: Erik Dyer*/

func somefun() {
    return 42; /* Fail: return int from void function */
}

func main() {
   somefun();
}
```

C.18 test-function-shape-formal.sol

```
/* @author: Kunal Baweja */

/* Shapes should be passed by reference */

/* Define a line */
shape Line {
    int [2]start;
    int [2]end;

    construct(int [2]first, int [2]second) {
        start = first;
        end = second;
    }

    draw(){}

    func describe() {
        consolePrint(intToString(start[0]));
```

```
            consolePrint(intToString(start[1]));
            consolePrint(intToString(end[0]));
            consolePrint(intToString(end[1]));
        }
}

/* displace line by d */
func moveLine(Line l, int[2] d) {
    l.start[0] = l.start[0] + d[0];
    l.start[1] = l.start[1] + d[1];

    l.end[0] = l.end[0] + d[0];
    l.end[1] = l.end[1] + d[1];
}

func main() {
    Line l;
    l = shape Line([1,1], [5,5]);
    moveLine(l, [2, 2]);

    /* confirm modified values for pass by reference */
    l.describe();
}
```

C.19 test-product.sol

```
/*@author: Kunal Baweja*/

func int mult(int x, int y) {
    return x * y;
}

func float fmult(float x, float y) {
    return x * y;
}

func checkInt(int x, int y) {
    if (x == y) {
        consolePrint("CORRECT");
    }
    if (x != y) {
        consolePrint("INCORRECT");
    }
}

func checkFloat(float x, float y) {
```

```
    if (x == y) {
        consolePrint("CORRECT");
    }
    if (x != y) {
        consolePrint("INCORRECT");
    }
}

func main() {
    int x;
    float y;

    /* integer multiplication */
    x = mult(40, 2);
    checkInt(x, 80);

    x = mult(1, 0);
    checkInt(x, 0);

    /* float multiplication */
    y = fmult(-3.0, 2.0);
    checkFloat(y, -6.0);

    y = fmult(0.0, 1.0);
    checkFloat(y, 0.0);
}
```

C.20 fail-array-access-neg.sol

```
/*@author: Erik Dyer*/

func main() {
    int i;
    int [5]array;

    /* assign array elements */
    array = [0,1,2,3,4];

    /* print array elements */
    consolePrint(intToString(array[-1]));
}
```

C.21 test-logical.sol

```
/*@author: Kunal Baweja*/

func main() {
    if (1 == 1 && 2 == 2) {
        consolePrint("AND");
    }
    if (1 == 1 || 1 == 0) {
        consolePrint("OR");
    }
    if (!(1 == 0)) {
        consolePrint("NOT");
    }
}
```

C.22 test-escape-chars.sol

```
/* @author: Kunal Baweja */

/* test escape sequences */

func main() {
    consolePrint("\\");
    consolePrint("new\nline");
    consolePrint("\'");
    consolePrint("\"");
    consolePrint("\ttableft");
}
```

C.23 fail-return-int-string.sol

```
/*@author: Erik Dyer*/

func int somefun() {
    return "should return int";
}

func main() {
    somefun();
}
```

C.24 test-array-pass-ref.sol

```
/*@author: Kunal Baweja*/
/* test arrays passed by reference */

func assign(int [5]b) {
    int i;
    i = 0;

    while(i < 5) {
        i = b[i] = i + 1;
    }
}

func main() {
    int [5]a;
    int i;

    /* pass for assignment */
    assign(a);

    /* confirm assigned values */
    i = 0;
    while (i < 5) {
        consolePrint(intToString(a[i]));
        i = i + 1;
    }
}
```

C.25 test-int-to-string.sol

```
/*@author: Kunal Baweja*/

func main() {
    string s;

    s = intToString(-2147483648);
    consolePrint(s);

    s = intToString(-2147483648 + 2147483647);
    consolePrint(s);

    s = intToString(0);
    consolePrint(s);

    s = intToString(2147483647);
    consolePrint(s);
```

```
}
```

C.26 test-shape-function.sol

```
/* @author: Kunal Baweja*/
/* confirm return values of member functions */

shape Rectangle {
    int [2]a;
    int [2]b;
    int [2]c;
    int [2]d;

    construct (int [2]w, int[2]x, int[2]y, int[2]z) {
        a = w;
        b = x;
        c = y;
        d = z;
    }

    draw () {}

    /* get area */
    func int area() {
        int h;
        int w;

        h = a[1] - d[1];
        if (h < 0) {
            h = -h;
        }

        w = b[0] - a[0];
        if (w < 0) {
            w = -w;
        }

        return w * h;
    }
}

func main () {
    Rectangle r;
    r = shape Rectangle([1,1],[4,1],[4,4],[1,4]);
    consolePrint(intToString(r.area()));
}
```

C.27 test-float-to-string.sol

```
/*@author: Kunal Baweja*/

func main() {
    float f;
    string s;

    f = -10.0;
    s = floatToString(f);
    consolePrint(s);

    f = 0.0;
    s = floatToString(f);
    consolePrint(s);

    f = 10.0;
    s = floatToString(f);
    consolePrint(s);
}
```

C.28 test-framerate.sol

```
/* @author Kunal Baweja */

func main() {
    /* test set and get framerate functions */
    consolePrint(intToString(getFramerate()));
    setFramerate(24);
    consolePrint(intToString(getFramerate()));
}
```

C.29 fail-if.sol

```
/*@author: Kunal Baweja*/

func main() {
    /* if condition expects integer expression */
    if (1.0) {
        consolePrint("INVALID CONDITION");
    }
}
```

C.30 test-shape-array.sol

```
/* @author: Kunal Baweja */

/* test initializing a shape with an array of points
 * pass an array to the constructor and ensure that
 * the object makes a copy of the array. The contents
 * of array should not change
 */

shape Line {
    int [2]start;
    int [2]mid;
    int [2]end;

    construct(int [2]first, int [2]second) {
        start = first;
        end = second;
        /* line mid point */
        mid[0] = (start[0] + end[0]) / 2;
        mid[1] = (start[1] + end[1]) / 2;
    }

    draw(){}

    func describe() {
        consolePrint(intToString(start[0]));
        consolePrint(intToString(start[1]));
        consolePrint(intToString(mid[0]));
        consolePrint(intToString(mid[1]));
        consolePrint(intToString(end[0]));
        consolePrint(intToString(end[1]));
    }
}

func main() {
    Line l;
    int [2]first;
    int [2]second;

    first = [1, 1];
    second = [9, 9];

    l = shape Line(first, second);

    /* modify source array */
    first[0] = -1;
```

```
    first[1] = -1;
    second[0] = -9;
    second[1] = -9;

    /* verify shape remains unchanged */
    l.describe();
}
```

C.31 fail-add-semantic.sol

```
/*@author: Kunal Baweja*/

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

func main() {
    float x;
    x = add(40, 2.5);
}
```

C.32 test-hello.sol

```
/* @author: Erik Dyer */

func main() {
    consolePrint("Hello World");
}
```

C.33 fail-assign-stringint.sol

```
/*@author: Erik Dyer*/

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

func main() {
    int x;
    string y;
    int z;
    y = "foo";
    x = add(10, 2);
    z = "bar"; /* cant assign string to int*/
```

```
}
```

C.34 test-assign-variable.sol

```
/*@author: Kunal Baweja*/

func main() {
    int x;
    int y;
    float f;
    float g;
    string s;
    string p;
    string q;

    /* integer assignment */
    x = 5;
    y = x;
    s = intToString(y);
    consolePrint(s);

    /* string variable assignment */
    p = "Hello World";
    q = p;
    consolePrint(q);

    f = 4.2;
    g = f;
    consolePrint(floatToString(g));
}
```

C.35 test-set-array.sol

```
/*@author: Erik Dyer*/

func main() {
    int[2] x;
    int[2] y;

    y[0] = 4;
    y[1] = 2;

    x = y;
    consolePrint(intToString(x[0]));
    consolePrint(intToString(x[1]));
```

```
}
```

C.36 test-array-assign.sol

```
/*@author: Kunal Baweja*/

func main() {
    int [5]arr;
    int i;
    string s;

    i = 0;
    while(i < 5) {
        arr[i] = i;
        i = i + 1;
    }

    i = 4;
    while(i >= 0) {
        s = intToString(arr[i]);
        consolePrint(s);
        i = i - 1;
    }
}
```

C.37 test-comparison.sol

```
/*@author: Kunal Baweja*/

func main() {
    /* Integer comparisons */
    if (0 == 0) {
        consolePrint("EQUALITY");
    }
    if (-1 != 0) {
        consolePrint("INEQUALITY");
    }
    if (2 > 1) {
        consolePrint("GREATER THAN");
    }
    if (-2 < -1) {
        consolePrint("LESS THAN");
    }
    if (1 <= 2) {
        consolePrint("LESS THAN OR EQUAL");
```

```
    }
    if (5 >= 3) {
        consolePrint("GREATER THAN OR EQUAL");
    }

    /* float logical comparison */
    if (0.0 == 0.0) {
        consolePrint("FLOAT EQUALITY");
    }
    if (-1.0 != 0.0) {
        consolePrint("FLOAT INEQUALITY");
    }
    if (2.0 > 1.0) {
        consolePrint("FLOAT GREATER THAN");
    }
    if (-1.1 < -1.0) {
        consolePrint("FLOAT LESS THAN");
    }
    if (1.0 <= 2.0) {
        consolePrint("FLOAT LESS THAN OR EQUAL");
    }
    if (5.0 >= 3.0) {
        consolePrint("FLOAT GREATER THAN OR EQUAL");
    }
}
```

C.38 fail-add-intstring.sol

```
/*@author: Erik Dyer*/

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

func main() {
    float x;
    string y;
    y = "foo";
    x = add(40, y); /* cant add string and int */
}
```

C.39 test-recursion.sol

```
/*@author: Kunal Baweja*/
```

```
/* find sum of 1 to n, inclusive */
func int series(int n) {
    if (n < 2)
        return n;
    return n + series(n-1);
}

func main() {
    consolePrint(intToString(series(-1)));  /*-1*/
    consolePrint(intToString(series(0)));   /*0*/
    consolePrint(intToString(series(1)));   /*1*/
    consolePrint(intToString(series(2)));   /*3*/
    consolePrint(intToString(series(10)));  /*55*/
}
```

C.40 test-division.sol

```
/*@author: Kunal Baweja*/

func int div(int x, int y) {
    return x / y;
}

func float fdiv(float x, float y) {
    return x / y;
}

func checkInt(int x, int y) {
    if (x == y) {
        consolePrint("CORRECT");
    }
    if (x != y) {
        consolePrint("INCORRECT");
    }
}

func checkFloat(float x, float y) {
    if (x == y) {
        consolePrint("CORRECT");
    }
    if (x != y) {
        consolePrint("INCORRECT");
    }
}

func main() {
```

```
    int x;
    float y;

    /* integer diviplication */
    x = div(40, 2);
    checkInt(x, 20);

    x = div(2, 5);
    checkInt(x, 0);

    /* float division */
    y = fdiv(-4.0, 2.0);
    checkFloat(y, -2.0);

    y = fdiv(0.0, 1.0);
    checkFloat(y, 0.0);
}
```

C.41 test-associativity.sol

```
/*@author: Kunal Baweja*/

func main() {
    int x;
    x = 1 + 2 - 3;  /* 0 */
    if (x == 0) {
        consolePrint("CORRECT");
    }

    x = 21 * 3 % 80 / 9; /* 7 */
    if (x == 7) {
        consolePrint("CORRECT");
    }
}
```

C.42 test-shape-define.sol

```
/*@author: Kunal Baweja*/

shape Circle{
    int [2]center;
    int radius;
    construct(int [2]c, int r) {
        center[0] = c[0];
        center[1] = c[1];
```

```
        radius = r;
    }

    draw() {}

    func describe() {
        consolePrint("Center X");
        consolePrint(intToString(center[0]));
        consolePrint("Center Y");
        consolePrint(intToString(center[1]));
        consolePrint("Radius");
        consolePrint(intToString(radius));
    }
}
func main() {
    Circle c;
    int a;

    c = shape Circle([3, 5], 5);
    c.describe();

    /* change member variables */
    c.center[0] = -3;
    c.center[1] = -5;
    c.radius = 30;
    c.describe();
}
```

C.43 test-array-access.sol

```
/*@author: Kunal Baweja*/

func main() {
    int i;
    int [5]array;

    /* assign array elements */
    array = [0,1,2,3,4];

    /* print array elements */
    i = 0;
    while(i < 5) {
        consolePrint(intToString(array[i]));
        i = i  + 1;
    }
}
```

Appendix D

Manual Testing

D.1 mnl-composite-square.sol

```
/* @author: Kunal Baweja */

/* Test member shapes */

/* Define a line */
shape Line {
    int [2]start;
    int [2]mid;
    int [2]end;
    int [3]color;

    construct(int [2]first, int [2]second, int [3]clr) {
        start = first;
        end = second;
        color = clr;

        mid[0] = (start[0] + end[0]) / 2;
        mid[1] = (start[1] + end[1]) / 2;
    }

    draw(){
        drawCurve(start, mid, end, 2, color);
    }

}

/* Define rectange as a collection of lines */
shape Rectangle {
    Line top;
    Line right;
    Line bottom;
    Line left;
```

```
    construct(int [2]a, int [2]b, int [2]c, int [2]d) {
        top = shape Line(a, b, [150, 0, 0]);
        right = shape Line(b, c, [0, 150, 0]);
        bottom = shape Line(c, d, [0, 0, 150]);
        left = shape Line(d, a, [150, 150, 150]);
    }

    draw(){}

}

func main() {
    Rectangle sq;

    /* define lines */

    /* initialize square */
    sq = shape Rectangle([10,10], [300,10], [300, 300], [10, 300]);
}
```

D.2 mnl-drawpoint.sol

```
/* @author: Kunal Baweja */

shape Points {
    int [2]a;

    construct (int [2]x) {
        a = x;
    }

    draw() {
        int i;
        i = 0;
        while (i < 50) {
            a[0] = a[0] + i;
            a[1] = a[1] + i;
            drawPoint(a, [0, 0, 0]);
            i = i + 1;
        }
    }
}

func main() {
    Points p;
```

```
    p = shape Points([100, 100]);
    setFramerate(2);
}
```

D.3 test-triangle-translate.sol

```
/*@author: Erik Dyer*/

/* Test Triangle translate */
/* Run this code, */
shape Triangle {
    int[2] a;
    int[2] b;
    int[2] c;
    /* How much to translate triangle */

    construct Triangle(int[2] a_init, int[2] b_init, int[2] c_init)
       {
        a = a_init;
        b = b_init;
        c = c_init;
        disp = disp_init
    }

    func int[] findCentre(int[2] x, int [2]y) {
        int i = 0;
        int [2] center;
        while(i < 2) {
            center[i] = a[i] + b[i] / 2;
            i = i + 1;
        }
        return center;
    }
    func draw() {
        /* Draw lines between the three vertices of the triangle */
         drawcurve(a, findCentre(a, b), b, [255, 0, 0]);
        drawcurve(b, findCentre(b, c), c, [0, 255, 0]);
        drawcurve(c, findCentre(c, a), a, [0, 0, 255]);
   }
}

func main(){
        int[2] disp;
        Triangle t;

        disp = [1,1];
```

```
        Triangle t = Triangle([0, 0], [2, 0], [1, 1]);

        t.render = {
                translate(disp);
        }
}
```

D.4 mnl-hello.sol

```
/* @author: Kunal Baweja */

shape Text {
    int [2]loc;
    int [3]clr;
    string text;

    construct(int [2]l, string t, int [3]c) {
        loc = l;
        clr = c;
        text = t;
    }

    draw(){
        print(loc, text, clr);
    }
}

func main() {
    Text t;
    t = shape Text([280, 240], "Welcome to SOL !", [0, 100, 100]);
}
```

D.5 mnl-line.sol

```
/*@author: Kunal Baweja*/

/* Test drawing a line */

shape Line {
    int [2]start;
    int [2]mid;
    int [2]end;

    construct(int [2]s, int [2]e) {
        start = s;
```

```
        end = e;
        mid[0] = (start[0] + end[0]) / 2;
        mid[1] = (start[1] + end[1]) / 2;
    }

    draw(){
        drawCurve(start, mid, end, 100, [0,150,0]);
    }
}


func main() {
    Line l;
    l = shape Line([2,2], [200,200]);
}
```

D.6 mnl-thick-line.sol

```
/*@author: Kunal Baweja*/

/* Test drawing a line */

shape Line {
    int [2]start;
    int [2]end;

    construct(int [2]s, int [2]e) {
        start = s;
        end = e;
    }

    draw(){
        int i;
        int [2]s;
        int [2]m;
        int [2]e;
        i = 0;

        s = start;
        e = end;
        while (i < 20) {
            s[0] = s[0] - 1;
            s[1] = s[1] + 1;

            e[0] = e[0] - 1;
            e[1] = e[1] + 1;
```

```
            m[0] = (s[0] + e[0]) / 2;
            m[1] = (s[1] + e[1]) / 2;

            drawCurve(s, m, e, 2, [0,150,0]);
            i = i + 1;
        }
    }
}


func main() {
    Line l;
    l = shape Line([100,2], [200,200]);
    setFramerate(10);
}
```

D.7 mnl-triangle.sol

```
/*@author: Erik Dyer & Kunal Baweja */

/* Test Triangle*/
/* Run this code, */


func findCenter(int [2]m, int[2]x, int[2]y) {
    m[0] = (x[0] + y[0]) / 2;
    m[1] = (x[1] + y[1]) / 2;
}

shape Triangle {
    int[2] a;
    int[2] b;
    int[2] c;

    int[2] abm;
    int[2] bcm;
    int[2] acm;

    construct (int[2] a_init, int[2] b_init, int[2] c_init){
        int i;
        a = a_init;
        b = b_init;
        c = c_init;

        i = 0;
```

```
        while (i < 2) {
            abm[i] = (a[i] + b[i]) / 2;
            bcm[i] = (c[i] + b[i]) / 2;
            acm[i] = (a[i] + c[i]) / 2;
            i = i + 1;
        }

        findCenter(abm, a, b);
        findCenter(acm, a, c);
        findCenter(bcm, c, b);
    }

    draw() {
        /* Draw lines between the three vertices of the triangle*/
        drawCurve(a, abm, b, 2, [150, 0, 0]);
        drawCurve(b, bcm, c, 2, [0, 150, 0]);
        drawCurve(c, acm, a, 2, [0, 0, 150]);
    }
}

func main(){
        Triangle t;
        t = shape Triangle([170, 340], [470, 340], [320, 140]);

}
```