better call **SOL**

SHAPE ORIENTED LANGUAGE FINAL REPORT

Aditya Narayanamoorthy	`an2753`
Erik Dyer	`ead2174`
Gergana Alteva	`gla2112`
Kunal Baweja	`kb2896`

December 20, 2017

Contents

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. SOL uses *point* (visual dot) and Bézier Curves with three control points as the basic drawing controls provided to the programmer for defining shapes. 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.

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* - Italics for variables, words, and concepts being defined.

The following conventions are applied while drawing and animating objects, using internal functions (see Section 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 Lexical Conventions

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

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.

3.2 *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 *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	
char	func	getFramerate	
string	construct	print	
	return	consolePrint	
		intToString	
		floatToString	
		charToString	
		render	
		wait	
		drawPoint	
		drawCurve	
		translate	
		rotate	

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`

3.5 *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`

3.6 *Character Constants*

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

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

3.8 *String constants*

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

3.9 *Operators*

SOL has mainly four categories of operators defined below:

3.9.1 *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 the left.

3.9.2 *Unary Negation Operator*

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

3.9.3 *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 5.1 for precedence and associativity rules.

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

3.9.4 *Comparison Operators*

The comparison operators are left associative binary operators for comparing values of operands defined as expressions. Please refer to section 5.1 for precedence and associativity rules.

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

3.9.5 *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 5.1 for precedence and associativity rules.

Operator	Definition
&&	AND
||	OR
!	NOT

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.

4 Identifier Scope

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.

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.

5 Expressions and Operators

5.1 *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 3.9.2.
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.

5.2 *Dot Accessor*

To access members of a declared `shape` (further described in section 11), use the dot accessor '.'.
Eg: `shape_object.point1 /* This accesses the variable point1 within the object shape_object */`

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.

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)

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 3.2. 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.

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 */`

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.
Functions can also be defined within *shape* definitions in which case they are referred as *member functions* of a class. (see section 11)

Example:

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

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 11 for defining shapes in SOL and creating shape instances.

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 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 */
```

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:

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 */`

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 */
}
```

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;
}
```

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;
}
```

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.

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

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`

8.3 *getFramerate*

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

8.4 *consolePrint*

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

8.5 Type Conversion Functions

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

8.5.1 *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`

8.5.2 *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`

8.5.3 *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`

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.

9.1 *drawPoint*

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

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])`

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. The `drawCurve`, `drawPoint` and `print` functions calls can be used within `draw` definition to describe the actual drawing of an object.
At runtime `draw` functions of all objects instantiated at runtime are called, to create the final scene rendering on screen.

10 Animation Functions

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

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)`

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)`

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

10.4 *wait*

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

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 10.

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.

11.1.1 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 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;
    }
}
```

11.1.2 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 it 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]);
}
```