better call **SOL**

SHAPE ORIENTED LANGUAGE REFERENCE MANUAL

Aditya Naraynamoorthy
an2753

Erik Dyer
ead2174

Gergana Alteva
gla2112

Kunal Baweja
kb2896

October 15, 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. 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* - The variables for SOL progamming language and words or concept being defined are denoted in italics.

The following are conventions used while drawing and animating objects, used in 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 cannot be a digit. 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	parent
char	func	translate	extends
string	construct	rotate	
	return	render	
		wait	
		drawPoint	
		drawCurve	
		print	
		length	
		consolePrint	

3.4 *Integer Constants*

A sequence of one or more digits representing a number in base-10.
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.
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 series of *characters* within double quotation marks. Its type is an array (defined in Section 6.2) of characters. The compiler places a null byte (`\0`) at the end of a string literal to mark its end.
Eg: `cat`

3.9 *Operators*

SOL has mainly four categories of operators defined below:

3.9.1 *Assignment Operator*

An *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 the expression on the right to the variable on the left.

3.9.2 *Arithmetic Operators*

SOL has the following *binary arithmetic operators*. A *binary arithmetic operator* operates on two *arithmetic expressions* specified before and after the operator respectively. The said expressions must be of type *int* or *float*.

Operator	Definition
+	Addition
-	Subtraction
	Multiplication
/	Division
%	Modulo

3.9.3 *Comparison Operators*

The comparison operators are binary operators for comparing values of operands defined as expressions.

Operator	Definition
==	Equality
!=	Not Equals
¡	Less than
¿	Greater than
¡=	Less than or equals
¿=	Greter than or equals

3.9.4 *Logical Operators*

The logical operators evaluate boolean expressions and return an integer as the result - with 0 as False and 1 as True. The AND (&&) and OR (||) operators are binary, while the NOT (!) operator is unary.

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 *Typecasting*

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

5.2 *Precedence and Associativity*

SOL expressions are evaluated with the following rules:

- Expressions are evaluated from left to right
- Multiplication, division and modulo operations take precedence over addition and subtraction
- Parentheses override all precedence rules
- Logical NOT has precedence over logical AND, which has precedence over logical OR

5.3 *Dot Accessor*

To access members of a declared `shape` (further described in section 7), use the dot accessor '.'.
Eg: `shape_object.point1 /* This accesses the variable point1 within the object shape_object */`

6 Declarations

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 *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). In a function signature, the size of the array should not be specified. Arrays are most commonly used in SOL to specify coordinates with two integers.
Eg: `int[2] coor; /* Array of two integers */`

6.3 *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 returns nothing. Functions are given a name (which is a valid identifier) followed by function 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.

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

6.4 *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. Constructors are given the same name as the class and followed by arguments, within parenthesis as a comma-separated list of variable declarations, similar to function definitions
3. Constructors do not have a return type specified

Eg:
```
construct Triangle(int[] a_init, int[] b_init,
   int[] c_init) {
    a = a_init;
    b = b_init;
    c = c_init;
}
```

6.5 *Definitions*

A definition of an object or type includes a value, assigned by the assignment operator '='.

Eg:
```
int x = a;      /* declaration & definition*/
char y;         /* declaration */
y = 'b';        /* definition */
float z = 3.4;
int[3] w = [5, 2, 0];
string f = "cats";
```

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: `int x = 5; /* assign 5 to variable x */`

7.2 *If Statement*

An *if* statement is a conditional statement. Given a condition for the *if* statement, if it evaluates to non-zero value then it is considered valid and the code block associated with the *if* statement is executed.
Eg:

```
int 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.
Eg:

```
int 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).

Eg:
```
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 call other functions written in the program. The `main` function does not take inputs as SOL programs do not depend on user input. The `main` function does not allow for member variables of `shape` objects to be changed.
Arguments: None

8.2 *length*

Returns the number of elements in an `array`.
Arguments: Array of elements

8.3 `setFramerate`

Defined once at the start of the program, to specify the frames rendered per second. May only be defined once.
Arguments: `rate (float)`

8.4 `consolePrint`

Prints a string to the console. Commonly used to print error messages.
Arguments: `text (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*

Draws a Bézier curve in the specified color defined by three coordinates, which are the three control points of the curve in order.
Arguments: `pt1 (int[2]), pt2 (int[2]), pt3 (int[2]), color (int[3])`

9.3 *print*

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

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]) /* horizontal, vertical */`

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. Every `shape` has a user-defined `draw` function that specifies how shapes are statically rendered, using multiple `drawPoint`, `drawCurve` and `print` commands. The class may contain multiple member variables that could be used to draw the shape. These member variables are defined in a constructor, specified by the keyword `construct`. It is also

possible to declare member functions for a shape. 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.
Once a `shape` object has been instantiated, these member variables cannot be changed, but may still be accessed later, using the dot accessor, '.'.
Eg:

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

    func int[] findCentre(int[] x, int []y) {
        if(length(a) != length(b)) {
            consolePrint("Arrays size mismatch.
                Abort");
            return int[0];
        }

        int i = 0;
        int c[length(a)];
        while(i < length(x)) {
            c[i] = a[i] + b[i] / 2;
            i = i + 1;
        }
        return c;
    }

    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]);
        }
    }
```

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`.

Eg:

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

    construct Line(int[] a_init, int[] b_init)
       {
        a = a_init;
        b = b_init;
    }

    func int[] findCentre(int[] x, int[] y) {
        if(length(a) != length(b)) {
            consolePrint("Arrays size mismatch!
                Abort !");
            return int[0];
        }

        int i = 0;
        int c[length(a)];
        while(i < length(x)) {
            c[i] = a[i] + b[i] / 2;
            i = i + 1;
        }
        return c;
```

```
    }

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

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

11.2.1 *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`.