SOIL
Simple Object Interaction Language

Language Reference Manual – March 10th, 2009

COMS W4115: Programming Languages and Translators
Professor Stephen A. Edwards

Richard Zieminski
rez2107@columbia.edu

Contents:

1 Introduction

SOIL is a computer language which can be used to teach the concepts of basic object interactions. Using a minimum of operations, a user can create simple objects which are composed of basic shapes, and then assign basic properties which characterize them. Simulations can then be run to see the outcome of the interactions between objects.

2 Conventions In The Document

Text in standard type indicates a keyword or literal. Text in italics indicates a placeholder for some other piece of code.

3 Lexical Conventions

3a Comments

Comments begin with the // character sequence and end with a line feed. Comments may be placed on the same line as source code. Multi-line comments will always begin with the // character sequence.

3b Whitespace

Whitespace characters which include spaces, tabs, and line feed characters may used to separate keywords, operators, and code tokens in the input but are discarded during parsing.

3c Line Breaks and Semicolons

Semicolons serve as a statement separator, and line breaks serve as a terminator. Multiple statements may be put on a single line of source code using semicolons in between each statement.

3d Identifiers

Identifiers represent the names of user defined variables and functions. All identifiers begin with a letter or underscore, followed by zero or more letters, digits, and underscores. Identifiers are case-sensitive. Identifiers can be up to 32 characters in length.

3e Keywords and Reserved Words

The following words are reserved as keywords and may not be used as identifiers. They are case sensitive. Valid keywords are:

if	then	else	
create	destroy		
or	and		
true	false		
run			
world			
function			
for			
shape	extends		
text	number	boolean	shape
elastic			

3f. Built in Functions

SOIL also contains several built in functions which may not be redefined. Valid function names are:

a. toScreen *text, number, etc.*

This function will handle combinations of text and numbers for output to the screen. It will parse the provided text for verbatim output (anything within "") and variables to be output as set.

Example:

```
x = 5;
toScreen "This is a simulation that will run " x " times";
```

Output:
This is a simulation that will run 5 times

b. go(number *seconds*)

This is used to run a simulation. The number of seconds to run is the only parameter.

3g Built in Objects

SOIL also contains built in objects which may not be redefined. Valid function names are:

a. world

*world the extents of the interactive environment in 2 dimensional coordinates (x, y) and needs to be set before running any simulation. It is defined as:

```
world( number value, number value, number value, boolean  value){
        width = $1;
        height = $2;
        strength = $3;
        elastic=$4;
}
```

Once defined, objects can be passed as parameters to other functions and their elements referenced using the '.' operator.

3h Operators

+	-	
/	*	
>	<	
>=	<=	
+=	-=	
=	!=	*shape, boolean or number comparison
.		* dereference fields of a shape
( )		
{ }		

3i Scope

There are two types of scope, local and global. Identifiers declared within a function are local only to that function and may not be used otherwise. Global identifiers which are declared outside any functions may be used anywhere in the program.

3j Constants

Constants are a sequence of digits representing a number or string of characters. Constants can be defined anywhere within the program using the following notation:

#identifier = value
#identifier = "Hello"

Constants are valid from the point of declaration forward. Constants cannot be changed once defined by a following declaration.

4 Primitive Data Types

Supported types will be text, number, boolean, and shape.

number is a 32 bit whole number (+/-). Only whole numbers are supported.
boolean (true/false)

4a Text

Text is a sequence of characters surrounded by double quotes. Text literals may not contain double quotes or span multiple lines.

4b Shape

Shape is a type that may contain any number of user defined fields of possible data types, along with several pre-defined fields and functions. These are:

width, height, speed, direction, symbol, position

These are predefined to give a shape its basic characteristics and therefore, allow it to be compatible for interactions with other shape objects.

Shapes can be defined from base shapes by using the 'extends' keyword.

shape identifier extends *shape identifier*

Functions defined in base shape objects cannot be redefined in extended shape objects. Variables defined in base shape objects can be redefined (overridden) in extended shape objects.

5 Expressions

Expression can be a combination of operators, identifiers and literals. Upon evaluation, an expression will return a value. The value type is dependant on the expressions being combined. Precedence of expressions is as listed in the operators section of this document.

5a Additive Expressions

text + text = text
text + number = text
number + number = number

Only text and numbers can be combined.

6 Declarations

Declarations are used to assign a value (text, number) to an identifier. They have the form:

identifier = value

A text declaration is defined by enclosing the value in quotes “”. There is no need to specify a type.

7 Functions

Functions will be defined through the use of the ‘function’ keyword and only can be added to shape types. This convention was chosen to allow shapes to interact based on their defined functionality as stand alone entities. Function parameters will be accessible via. the ‘.’ dot operator as with many other common languages.

Functions have the form:
function *identifier* (*parameter-list*)
{ *body* }

or

function *identifier* ()
{ *body* }

A function does not return a value. The parameter-list will be of the form (*identifier, identifier, ..*). Parameters are passed by reference and can be modified within the calling function.

To allow for interaction between objects, any function with a ‘shape’ parameter defined in the formal parameter list will be processed on each iteration during a go operation.

Function nesting is supported, but recursive operations are not.

Functions can access only variables that are passed in as arguments as well as locally declared variables.

8 Conditional Statement

There are two forms of the conditional statements:

a.
```
if ( expression ) then
{
        statement1;
        statement2;
        …
}
else
{
        statement1;
        statement2;
        …
};
```

b.
```
if ( expression ) then
{
        statement1;
        statement2;
        …
};
```

*Brackets are always used to enclose conditional statements.

9 Shape Definition, Creation, Destruction, and Operations

Shapes may be created anywhere, even within a shape function, although they will be automatically destroyed upon leaving the function.

A shape is defined and created with the keyword:

```
shape identifier{
}
```

Shape contains no formal parameters as it is meant to act as an autonomous object.

Shapes creation/destruction is done as follows:

Assignment/creation of a shape object:
identifier = create *shape identifier*

Destruction of an object:
destroy *shape identifier*

Shapes can extend their properties from other shape objects. This is done using the 'extends' keyword upon defining a shape object:

shape *shape identifier* extends *shape identifier{*
}

Upon compilation, the extended shape identifier code will be available in the new shape declaration.

10 Sample Program

This program creates two objects, one static, one mobile. Upon issuance of the 'go' command, shapes will begin in motion (if defined) and any function with a 'shape' type defined in it's function(s) parameter list will be called an evaluation on every iteration. This allows shapes to be autonomous while allowing for interactions with other shapes and the environment.

```
shape generic{
        strength = 0;  // Placeholder – will be defined in extended shapes
        waitCount = 0;
        hardness = 0;    // Placeholder – will be defined in extended shapes
        originalSymbol = symbol;

        // Wait to react until shape has had time to process the last change for x iterations
        if ( waitCount == 0 ) then
        {
                function close ( shape ){
                        // If objects are close (1/10 overall world), switch our direction 90
                        // degrees
                        if ( position – shape.position > ( world.width / 10 ) or
                                position – shape.position > ( world.height / 10 ) )
                        // Try to get out of the way
                        then
                        {
                                direction = shape.direction + 90;
                                // Go faster in another direction
                                speed = shape.speed + 1;
                                waitCount = 2;       // delay another reaction for x iterations
                        }
                }

                function far ( shape ){
                        // If objects are far (1/4 overall world), switch our direction 180
                        // degrees and try to return
                        if ( position – shape.position > ( world.width / 4 ) or
                                position – shape.position > ( world.height / 4 ))
                        // Try to get closer
                        then
```

```
                {
                        direction = shape.direction;
                        // Go faster
                        speed = shape.speed + 1;
                        waitCount = 2;      // delay another reaction for x iterations
                }
        }

        function touches ( shape ){
                // If objects touch, reduce health based on evaluation of several
                // variables
                if ( position = shape.position ) then
                {
                        // If we are elastic, then accept minimal damage
                        if ( !elastic and !shape.elastic ) then
                        {
                                // Take into account two factors for strength reductions
                                strength-= shape.strength * hardness;
                        }
                        else
                        {
                                strength-= 1;
                        }
                        waitCount = 2;      // delay another reaction for x iterations
                }
        }
    }else
    {
        waitCount-=1;
    }
}

shape shape1 extends generic{
    strength = 9;
    hardness = 3;
    elastic = true;

    // Default operations
    function default ( shape ){
        // Change symbol to originalSymbol to indicate no change has occurred
```

```
                symbol = originalSymbol;
                // Am I damaged beyond repair?
                if ( strength <= 0 ) then
                {
                        destroy shape1;
                }
        }
}

shape shape2 extends generic{
         strength = 5;
         hardness = 2;
         elastic = false;

        // Default operations
        function default ( shape ){
                // Change symbol to originalSymbol to indicate no change has occurred
                symbol = originalSymbol;
                // Am I damaged beyond repair?
                if ( strength <= 0 ) then
                {
                        destroy shape1;
                }
        }
}

go ( 5 );       // Run a simulation for x seconds
```