SalaScript Manual

Alasdair Turner

September 2007

Abstract

SalaScript is the scripting language for UCL Depthmap. It allows users to write their own formulae to replace attributes calculated by UCL Depthmap, and to select graph objects according to conditions. However, it is also a fully fledged scripting language, which means that formulae can run into several lines of instructions. This enables users to write algorithms to replace attribute values with graph measures, such as the total depth or connectivity value for each node. This report describes how to use SalaScript, from basic single line formulae to multiple line scripts, and, for more advanced users, the technicalities of the language, as well as providing a reference for the functions and commands available in SalaScript. It assumes some knowledge of UCL Depthmap and the principles of graph-based analysis, but no knowledge of scripting.

Manual revision 1.0, 1-Sep-2007
SalaScript version 1.0
Depthmap version 7.09.00r

Contents

Chapter 1

Introduction

SalaScript is part of the UCL Depthmap software package to perform spatial network analysis. UCL Depthmap primarily creates and analyses networks according to space syntax techniques, but it is able to analyse any network which has spatial representation of nodes. SalaScript further allows the user to modify measures calculated by UCL Depthmap, and to perform their own analysis of graphs within UCL Depthmap.

At its most basic, SalaScript allows manipulation of measures calculated by UCL Depthmap, or the selection of nodes according to values assigned by UCL Depthmap. For example, rather than *connectivity*[1], you may want to look at *connectivity* squared, or you may want to select all the nodes where the *connectivity* is greater than 3. For column replacement, SalaScript is invoked when you edit an attribute in UCL Depthmap, for example, right-clicking on the attribute in the sidebar and choosing 'Edit...' from the popup menu, or choosing 'Edit...' from the 'Attributes' menu. For querying, SalaScript is invoked from the 'Edit' menu, and choosing 'Select by Query'. You are presented with a box to enter your formula and a list of existing attribute columns for the map.

If replacement of attribute values or selection by query is the sort of usage you have in mind, then you can dive straight into Chapter 2 now. For the vast majority of users this will be their only mode of encounter with SalaScript. However, more advanced users may wish to calculate their own measures of the graph. In SalaScript version 1.0, those measures are restricted to 'per node' calculations. So you are given a clean sheet for each node, and your script is run for it. If this is your sort of usage, then you will probably be able to guess

[1] In space syntax terms, the *connectivity* of a node is the number of adjacent nodes.

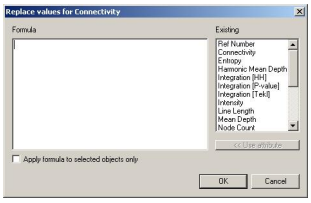

Figure 1.1: The attribute editor dialog box. The free text 'Formula' box is used to enter SalaScript formulae.

the information in Chapter 2, and will be able to go straight to Chapter 3, with judicious reading of the Appendices, which give more technical information about the SalaScript language.

SalaScript is intended to transfer some of the power of the programmer to the user of UCL Depthmap. Until now, only if one knew how to program in some detail could one create a new spatial measure for testing, be it via a DLL in Depthmap, or writing a program from scratch. Most original measures were simply out of reach of the amateur (in perhaps the original sense of the word, as an enthusiast). It is hoped that SalaScript allows the user of UCL Depthmap to be able to test new ideas in a way not previously possible.

The look and feel of SalaScript is 'Pythonesque'. If you have written any Python scripts, then you will recognise the syntax immediately. Do remember, though, that SalaScript *is not* Python. The rule is that if it works in SalaScript then it will work in Python, but not necessarily vice versa. For example, in SalaScript you can access the first element of a list called `fred` with the syntax `fred[0]`. This is also true in Python. However, in Python, you can access the range of elements from the second to the fourth with `fred[1:3]`. If you try this in SalaScript, it will object (there is 'an unexpected colon' in your expression). That said, SalaScript is somewhat more mathematical than Python, so it sometimes uses mathematical terminology for function names where Python might use a more traditional computing name, so always check the function in Appendix A before using it.

When errors do occur in your code, SalaScript tries to be helpful with its error messages. Many languages seem to adopt a very technical attitude to errors, but SalaScript will always try to identify the line where the error occurred, and give you as much information as it has available at the time (e.g., did the error occur within a call to a function, etc.). In making the language helpful and permissive, some optimisation has to be lost. If you really want to make a serious and fast graph calculation to run in UCL Depthmap, then it is best to write a DLL plug-in for UCL Depthmap, using the UCL Depthmap Software Developers Kit, available from `http://www.vr.ucl.ac.uk/depthmap/sdk.html`. When thinking about optimising your script performance, it is better not to try to second guess how SalaScript works internally, as this may change in future implementations. Never the less, any script functionality described herein will continue to be supported in any future version of SalaScript, so you can write safe in the knowledge that your script will always work despite updates to the language.

Note that graphs in UCL Depthmap can be considered either as maps of geometrical objects with attributes ('graph objects' in SalaScript) or as tables of rows (one for each graph object) and columns (one for each attribute). Which is more appropriate to use at any one can vary: UCL Depthmap can display graphs as both maps of graph objects and as tables of rows and columns. In this document, the table metaphor is used when talking about attributes, and the 'map' when discussing graph properties, such as the connections[2] for each node. Thus, basic usage tends to be discussed in terms of tables — you are using SalaScript to go through each row in the table to change the values of the attributes in a particular column — whereas advanced usage tends to be in terms of maps with connections.

[2]More commonly known as 'neighbours' in graph theory

Chapter 2

Basic Usage

Basic usage of SalaScript is either to update measures on a row by row (that is, object by object, e.g., line by line or point by point) basis, or to select objects on a row by row basis. The following two sections in this chapter describe each of these in turn. After these sections, the third section gives some caveats about when SalaScript may not be as intuitive as expected.

2.1 Replacing Attribute Values

Replacing a column value through SalaScript is done by writing a 'script' in the formula box of the Edit Attribute dialog box (see figure 2.1). However, at first you will probably not realise that what you are about to write is in fact a script: what is described here is simply a formula. Say for example, you think that pedestrian movement might be related to connectivity squared, you might want to replace the connectivity column with connectivity squared. It is probably better to create a new column and keep connectivity intact, as Depthmap needs this information! Once you have a new column[1] you can choose to edit it.

2.1.1 Simple Formulae Using Mathematical Operators

To set each row of the new column to connectivity squared, you might type:

```
value("Connectivity") * value("Connectivity")
```

[1] If you do not know how to create new columns and edit them, online UCL Depthmap tutorials are available at `http://www.vr.ucl.ac.uk/depthmap/tutorials/`.

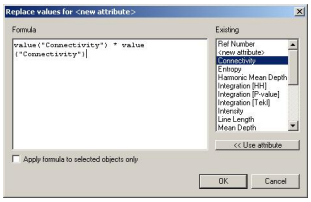

Figure 2.1: A simple formula displayed in the Edit Attribute dialog box

into the formula box (see figure 2.1).

The `*` sign means 'multiply', and is common in computing because there is no dedicated × sign on the keyboard. `value(`x`)` is a function, it retrieves the value of connectivity for this object (whether the object is a point, line or shape does not matter). The x is a column name. In our case we chose 'Connectivity', but it could be the name of any existing column, even the column you are updating. Note that the column name must be between inverted commas: you can write either `value("Connectivity")` or `value('Connectivity')` but not `value(Connectivity)`.

In the Edit Attribute dialog box, the list to the right of the formula box gives the names of existing columns you may want to use (see figure 1.1). Rather than type out `value("Connectivity")` yourself, you can select the attribute you want and then click the 'Use attribute' button to insert this text for you.

As with many scripting languages, there is usually more than one way to do things. Rather than multiplying the value of connectivity with itself, you might like to write:

```
value("Connectivity")^2
```

This means raise the value of 'Connectivity' to the power 2, or more simply simply, square it. The 'raise to the power' notation can be used for other powers, though: to cube, write `value("Connectivity")^3`, or any power you wish. More technically, the `*` symbol and the `^` symbol are called *operators*. You can find a list of all the mathematical operators in Appendix A.1.

2.1.2 Spaces are Ignored

In the previous two examples, each script used a different convention for spacing. The `*` operator was shown with a space either side (e.g., `3 * 3`), but the `^` without the spaces (e.g., `3^3`). However, this is due to the personal taste of the author, rather than any rule in SalaScript. `3*3` is just as valid, as is `3 ^ 3`. You can even write:

```
value  (  "Connectivity"  ) ^  2
```

if you wish. Note, however, that column names must not have extra spaces around them. That is, the word 'Connectivity' must be tightly bound each side by inverted commas as shown in this example. It must also be spelt with an uppercase 'C' if the UCL Depthmap column name has an uppercase 'C'.

2.1.3 Applying Functions to Values

As well as using mathematical operators to make new values, you can use mathematical functions. For example, if you would like the new attribute to be the natural logarithm of the connectivity, you can write:

```
ln(value("Connectivity"))
```

Just like `value(`x`)`, `ln(`x`)` is a function. `ln(`x`)` takes the natural logarithm of whatever x is. In this case, x is the value of the connectivity. You can find a list of all the mathematical functions you can apply in Appendix A.2. Note, however that you will not find the `value` function itself in this list. This is

because the `value` function is actually applied to the graph object, the line, point, or whatever it happens to be, in order to look up the attribute value. You can find it in Appendix B.6 under 'Graph Object Member Functions'. At this stage, it is not necessary to understand why this classification is used, nor the details of the description given, which is intended for advanced users.

In the list of functions it is possible to find `sin`, `cos`, `tan`, and their inverses `asin`, `acos` and `atan`. However, there is no inverse for the natural logarithm function. This is because the inverse is achieved using the 'raise to the power of' operator. You use the mathematical constant `e`, and raise it to the power of a number. So to get the exponential of connectivity, you would write:

```
e^(value("Connectivity"))
```

2.1.4 Floating Point Numbers and Mantissa Exponent Notation

It has remained unsaid in the preceding description that the formula you write must result in a number. It may seem obvious, but, tempting as it might be to try a string (e.g., simply typing `"My value"` into the formula box), UCL Depthmap cannot handle a string attribute, and it will complain. More precisely, SalaScript is expecting a *floating point number*, or 'float' for short. A floating point number is of the form 'mantissa, exponent'. This may sound unfamiliar, but you are probably used to it from calculators: the number `2.0e03` is in mantissa, exponent form. The mantissa is 2.0, and the exponent is 3, and it is short for 2.0×10^3, or 2000. Notice that when you change the exponent, the decimal point 'floats' from right to left, or vice versa: `2.0e03` is the same as 2000.0, `2.0e02` is 200.00, `2.0e01` is 20.000, and so on. All these forms are legal in SalaScript; a perfectly acceptable formula is:

```
value("Connectivity") * 1.25e03 + 1e04
```

Negative exponents are also allowed, for example `1.25e-02` (0.0125). To ensure that SalaScript reads a whole number as a float, add a `.0`, so `2.0` is a float, whereas `2` is simply an integer. In the main, this does not matter too much, as SalaScript can translate an integer to a float, but you should be wary when performing division — see section 2.3.1 for an explanation.

2.2 Selecting by Query

Selecting rows of graph objects through SalaScript is done by writing a 'script' in the query box of the Select by Query dialog box (see figure 2.2). As with the replacement of attribute values, you will probably not notice that your query is in fact a script at first. To get started, choose 'Select by Query' from the main 'Edit' menu in UCL Depthmap.

2.2.1 Simple Queries

You may want to select all the rows with connectivity higher than two, in order to then perform a step depth calculation from them, or to label them with a value for future reference. To do so, type:

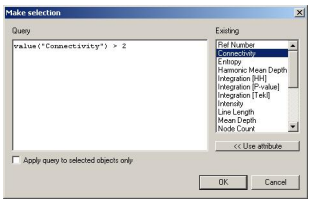

Figure 2.2: A simple query displayed in the Select by Query dialog box

```
value("Connectivity") > 2
```

into the query box, and click the 'OK' button. This query selects all the rows with a connectivity value greater than two.

As with replacing values, there is usually more than one way to perform a query. For example, you might type:

```
value("Connectivity") >= 3
```

This query selects all the rows with a connectivity greater than or equal to three. The `>` and `>=` are, like `+` and `/`, more formally called *operators*. This particular brand of operators are usually called 'comparison operators' because they compare the values of two items. A list of all the comparison operators available in SalaScript is given in Appendix A.4. Note that the comparison operator for equals has two equals signs: `==`, so you would write `value("Connectivity") == 3` to select all graph objects with a connectivity of exactly 3. The reason for the double equals is so that equivalence is not confused with another operation in SalaScript, assignment. You do not need to know the details of assignment, but if you wish to read further it is covered in section 3.2.

2.2.2 Boolean Values and Numbers

Any query in SalaScript is expected to evaluate to either true or false. Connectivity is either greater than 2 (true), or it is not (false). The values 'true' and 'false' are Boolean constants. You can type them into SalaScript if you wish. The query:

```
true
```

will select everything. The query:

```
false
```

will select nothing. Just as comparison operators evaluate to true or false (more technically, they *return* true or false as a result of the comparison), numbers can also evaluate to true or false. In SalaScript the integer 0 and the floating point number 0.0 are considered to be 'false', and any other number is considered to be 'true'. The 'query':

```
value("Connectivity") - 1   # Poor script: do not copy
```

will select every row which does not have a connectivity of exactly 1. Whilst this is strictly possible it is not *advisable*, as it is easy to confuse what this

query is supposed to do. The hash symbol (‘`#`’) in this script is used to indicate a comment in SalaScript: some plain text to make your script more readily understandable to someone else. Here the plain text is used to advise you not to copy a poor example of scripting. A much better way to write the query is with the ‘not equal’ comparison operator, `!=`:

```
value("Connectivity") != 1
```

Note that although numbers can be converted to the Boolean values true or false, strings cannot, so if you try writing `"Connectivity"` as a query, SalaScript will complain.

2.2.3 Chaining Queries Together: Logical Operators

Rather than selecting the graph objects with connectivity above 2, you may want do something more complicated: select the graph objects with connectivity above 3 and below 5. This can be achieved with the script:

```
value("Connectivity") > 2 and value("Connectivity") < 5
```

x `and` y is a *logical operator*. There are three logical operators listed in Appendix A.5. x `and` y returns true if both x and y are true. In our case, if both the comparisons `value("Connectivity") > 2` *and* `value("Connectivity") < 5` are true. So only if connectivity is greater than two and less than five will a row be selected.

The operator x `or` y returns true if either x or y or both x and y are true. For example, you might want to select rows if either the connectivity is higher than 2 or the reference number is greater than 100. In this case you would write:

```
value("Connectivity") > 2 or value("Ref Number") > 100
```

The final logical operator, `not` x can be a little confusing to use at first. It is usually placed before the condition, for example, `not value("Connectivity") > 2` rather than `value("Connectivity") not > 2`, although both are acceptable, and mean the same thing. However, note that it can only be used to negate the whole condition if you put brackets around it. As an example, the following script selects the inverse of the last script:

```
not (value("Connectivity") > 2 or value("Ref Number") > 100)
```

A thorough explanation of why the brackets are required is given in section 2.3.2.

2.3 Caveats for the Unwary

Scripting in the main should be fairly straight-forward, and most scripts you will write will probably do what you intended. However, there are some features of SalaScript of which you need to be aware. These are not specific to SalaScript: almost all programming languages will act in the same way; it is simply a case that by ensuring certain scripts work in one manner, others will suffer as a consequence.

2.3.1 The Difference Between Integer and Floating Point Division

Integers and floating point numbers (floats) were introduced in section 2.1.4. In most cases integers and floats are interchangeable. Even within this document there have been examples of 'loose' scripting style: connectivity was cubed by raising it to the power 3, an integer, rather than the floating point number 3.0. Most of the time SalaScript will work out what you mean, but it is worth ensuring you use floats when you want a floating point answer. For example, you might expect the following script to produce 'one third of connectivity' if you run it to replace values:

```
1/3 * value("Connectivity")   # Poor script: do not copy
```

However, if you do try it, you will find that it instead it sets everything to zero! This is because SalaScript uses *integer division* for 1 divided by 3, and then multiplies it by the connectivity. 1 divided by 3 in integer division is 0 remainder 1. SalaScript simply gives you the 0 part, and discards the remainder. Only after it has performed `1/3`, and evaluated it to 0, does it go on to multiply the result by connectivity. The end result: 0. You can get round this problem by reversing the order of the multiplication and division, but it is not advisable:

```
value("Connectivity") * 1/3   # Poor script: do not copy
```

The answer is 'correct' (as intended): SalaScript first multiplies the float value of connectivity by 1, giving a float; it then divides by 3, but converts everything into floating point notation, because the numerator is a float (it will also convert an integer numerator to a float if the denominator is a float). However, it is not a good idea to rely on this feature, as it is difficult to keep track of when and where integers are integers rather than floats in a long statement. For this reason, it is always better to avoid doubt and write:

```
value("Connectivity") * 1.0/3.0
```

`1.0/3.0` ensures that SalaScript uses floating point division, and always evaluates to a third.

2.3.2 Brackets in Formulae and an Introduction to Operator Precedence

Another possible need is to replace values with the cube root of an attribute. Although there is a function `sqrt(`x`)` to take the square root of a value, there is not cube root function. To obtain the cube root, you need to raise to the power of one third. Bearing in mind the warning in section 2.3.1, you might be tempted to write:

```
value("Connectivity")^1.0/3.0    # Poor script: do not copy
```

However, if you do, you will find that the result is again unintended: this script returns a third of the value of connectivity. This is because of a feature of programming languages called *operator precedence*. The `^` operator has a higher precedence than the `/` operator, meaning that it will be performed first. Thus, the value of connectivity is first raised to the power one, and only then

is it divided by three. Another script, differently spaced, makes the reason apparent:

```
value("Connectivity")^2.0 / 3.0
```

It is clear from the spacing that we want to first square the value of the connectivity, and only then divide by three. However, SalaScript does not understand spacing, and even if it did, the results can still be ambiguous, so it is best to follow precedence rules. The precedence of all the operators in SalaScript is given in Appendix C. These rules also ensure that multiplication is performed before addition, and subtraction before addition. So:

```
value("Connectivity") + 2.0 * 3.0   # Poor script: do not copy
```

first multiplies 2.0 by 3.0 (to give 6.0), and only then adds it to the value of connectivity, even though the `+` operator appears before the `*` operator. Where operators have the same precedence, the order of application is as might be expected, from left to right. So `4 - 3 + 5` is six, not minus four.

In order to override the precedence of operators, and get the cube root of connectivity, or connectivity plus two, multiplied by three, you need to use brackets. To ensure that `1.0/3.0` is done before raising to the power, enclose it in brackets, like so:

```
value("Connectivity") ^ (1.0/3.0)    # Cube root of connectivity
```

Similarly, the addition followed by multiplication can also be solved with brackets:

```
(value("Connectivity") + 2.0) * 3.0
```

This returns us to the issue of the `not` operator in section 2.2.3. As `not` has a higher precedence than `or`, it is performed before it. So:

```
not value("Connectivity") > 2 or value("Ref Number") > 100)
```

is the same as:

```
(not (value("Connectivity") > 2)) or value("Ref Number") > 100)
```

Also note that because `not` has a lower precedence than the `>` operator, it is always performed after it, wherever it appears. So the following four statements are the equivalent, even the rather teenage last one:

```
not value("Connectivity") > 2
value("Connectivity") not > 2
value("Connectivity") > not 2
value("Connectivity") > 2 not
```

In addition, notice that `or` has a higher precedence than `and`, which can lead to confusion. For clarity in logical statements, it is usually best to use brackets, even if that is how SalaScript would have done it anyway:

```
(value("Connectivity") > 2 or value("Ref Number") > 100) and \
(not value("Line Length") < 40.0)
```

The backslash (\) in this example script tells SalaScript that the condition continues on the next line, and has no special meaning for how the script is interpreted.

2.3.3 Ensure the Function Does What You Think it Does

As a final caveat, it is worth checking any function you apply to a value actually does what you expect it does. In particular, `ln(`x`)` returns the natural logarithm, or log base e of x, and `log(`x`)` returns the log base 10 of x. In addition, the trigonometric functions, such as `cos(`x`)`, require that x is in radians, not degrees as might be expected. Radians run from 0 to 2π rather than 0 to 360°.

Chapter 3

Advanced Usage

This chapter contains some quite detailed descriptions of how features of SalaScript work. Most will probably be in *too much* detail, and you may find it easier to pick and choose which sections you read. In particular, you may well want to start with the two scripting examples in sections 3.5.1 and 3.5.2, and work backwards to any details you do not understand.

3.1 Before We Start

3.1.1 SalaScript is Case Insensitive; UCL Depthmap is Not

There is not much more to say: `Value("Connectivity")` is the same as `VaLUE ("Connectivity")` is the same as `value("Connectivity")`. Similarly, `E^3.5` is the same as `e^3.5`. However, just because SalaScript is case insensitive does not mean that UCL Depthmap is case insensitive. In particular, column names *are* case sensitive. If you write `value("connectivity")`, SalaScript will return an 'unknown column' error.

3.1.2 Comments and Line Continuation

Comments in SalaScript begin with the hash symbol (`#`). Everything after the hash symbol is treated as a comment, and not included in the code. This includes the line continuation (backslash) character: if you want to continue a line, you must use `\` before the hash. For example:

```
value("Connectivity") > 2 and \ # Line continued
value("Connectivity") < 5
```

Note that unlike Python, SalaScript does not allow you to break a line unless you use a line continuation character. If you write your line of code over two lines, SalaScript will treat each as a separate (and probably complain) unless you add the backslash.

3.2 Constants and Variables

3.2.1 Variable Assignment and Use

Variables can be assigned in SalaScript using the assignment operator `=`. So, to assign the value '3' to a variable called `x`, you would write:

```
x = 3
```

Once '`x`' has been set to three in this manner, you can use 'x' as a shorthand for three later on. Obviously, you are more likely to use 'x' as shorthand for something longer, such as `value("Connectivity") * 3`. For example:

```
x = value("Connectivity") * 3
x^2
```

This script assigns the value of connectivity multiplied by three to the variable `x`. In the next line, it calculates `x` squared. This second line is the value SalaScript *returns* to UCL Depthmap (see section **??** for more details about return values). So the end product of running this script would be to set each row to the value of connectivity multiplied by three, all squared.

3.2.2 Constants

There are two mathematical constant values in SalaScript version 1.0, `e` and `pi`, ($e = 2.718\ldots$, $\pi = 3.141\ldots$), as well as two truth values `true` and `false`. These cannot be reassigned, so if you write `pi = 2`, SalaScript will complain that you can assign a value to a constant.

3.2.3 this, none and objects

SalaScript has two special variables, `this` and `none`. `this` refers to the current object in SalaScript. The current object is usually the current node in the graph being evaluated or queried. Thus far, the fact that the current node is an 'object' has been implicit. In programming, the object analogy is to an actual object, for example, an orange: an orange has various attributes, it is roughly spherical, it has a colour (orange). Certain 'functions' can been performed on orange, it can be thrown, peeled, eaten and so on. Similarly each node in a graph has properties: each of its attributes, such as connectivity, as well as its list of connections. These can be accessed using functions, e.g., `value(`x`)`. Usually the fact that we are applying the function to *this* node, the current one, is implicit, but you can write it out explicitly: `this.value("Connectivity")` retrieves the value of connectivity for this node (as opposed to any other node in the graph). You can also assign variables to `this`, for example `x = this`. Then write statements such as `x.value("Connectivity")`. Why it might be useful to do this is made apparent in the example of total depth calculation given in section 3.5.1.

`none` is very different to `this`. `none` simply represents nothing value. For example `x = none` sets `x` to nothing. Note that not only does `none` have no value, it has no type either. This makes it impossible to compare `none` with, for example a floating point number. SalaScript complains if you set `x` to `none` and then try to compare it with a value, e.g., `x == 3`. Equally, if you set `x` to 3 and

then compare it `none`, e.g., `x == none`, SalaScript will complain. This can be troublesome when you have a variable that is sometimes `none` and sometimes, for example, numeric. In order to test it, you need to use the `is` comparison operator. `x is none` is always a valid test, regardless of whether or not `x` is assigned to `none` or not. Note that strictly speaking there is only one object `none`: that is, there is only one 'nothingness', thus one variable that is set to none is identical to any other variable set to none. As with `this`, why none is useful is becomes clear in the total depth calculation in section 3.5.1.

3.3 Lists

Lists in SalaScript are lists of various values. For example, `["dog", "cat", 3, 4]` is a list. Note how it is written with square brackets around it. You can assign a variable to a list, e.g.:

```
lst = ["dog", "cat", 3, 4]
```

Having defined a list, you can access a member of the list using the *access* operator, which also uses square brackets. `lst[2]` returns element number 2 from the list. Note, however, that SalaScript, like many other languages, starts counting at element 0. So element 0 in the list is `"dog"`, element 1 is `"cat"`, and element 2 is `3`. Thus, `lst[2]` evaluates to `3`. You could even use this feature to set an entire column to the value `3`, although it is a round about way to do it:

```
lst = ["dog", "cat", 3, 4]
lst[2]
```

Lists can contain functions and can be nested; for example, a valid list is:

```
lst = ["dog", value("Connectivity"), value("Ref Number"), [5, 6]]
```

The element number 3 of this list is `[5,6]`, so `lst[3]` evaluates to `[5,6]`. We can access the '6' by adding another access operator: `lst[3][1]` is `6`.

The length of a list can be found using the `len(`x`)` function. So, in our example `len(lst)` is 4. We might get the last element of the list using the code:

```
lst[len(lst)-1]
```

If the length of `lst` is four, this gets element number 3, or, counting from 0, the fourth element in the list.

Like graph nodes, lists are objects, and we can apply functions to them. The `append(`x`)` function is a *member* of the list type (or class). It can be applied using the same syntax we used with `this`, using a dot:

```
lst = ["dog", "cat", 3, 4]
lst.append(10)
```

In this script, the function `append(`x`)` appends the value `10` to the end of the list, so the resulting list is `["dog", "cat", 3, 4, 10]`.

The function `pop(`x`)` 'pops' the last member of the list — that is, it takes the last member off the end of the list and returns it to us. As a demonstration, we could write:

```
lst = ["dog", "cat", 3, 4]
y = lst.pop()
```

After this example, `lst` is `["dog", "cat", 3]` and `y` is `4`.

Both `append(`x`)` and `pop(`x`)` may not look that useful yet, but when we look at the total depth example in section 3.5.1, their purpose will become apparent.

Other functions belonging to the list type can be found in appendix B.5.

One thing to be slightly wary of with lists is how SalaScript maintains *copies* of lists. This functionality is identical to Python, but it can be confusing at first. When you assign a variable to a list, it does not copy the list, it creates a *reference* to it. To understand this feature, it is probably best to see it in action.

Firstly, let us look at a standard script. In this example, the end result, the `x` at the end, is `3`:

```
x = 3
y = x
y = y + 1
x
```

The way the script works is straight forward: `x` is set to `3`, then `y` is set to `x`, so it is also three. We then add one to the value of `y`, making `y 4`. Finally, the script evaluates `x` to return to UCL Depthmap. As expected, `x` is `3`. If this script were run, every row in the table would be set to 3.

Now let us look at a similar example using lists instead:

```
x = [3,10,8]
y = x
y[0] = y[0] + 1
x[0]
```

In this example, `x` is set to the list `[3,10,8]`. Now, when `y` is set to `x`, rather that copy the list out, SalaScript simply tags `y` as being the same as `x`. Hence, when we change element 0 of `y` on the next line, *we also change* element 0 of `x`. Thus, when the final line, `x[0]`, is evaluated, element 0 of `x`, it is also `4`. If this script were run, every row in the table would be set to 4.

3.4 Program Control

Program control is where the power of SalaScript really takes off. So far, every script has been one line (or possibly a few lines): simply a formula or a query; extra lines have not really done very much. However, when program control is included, SalaScript can respond in a different way according to different circumstances, and can be used to calculate complex graph properties, which comprise many, many lines, each issuing a different command.

3.4.1 if Statements to Replace Values

As a simple start to program control, we can look at the `if` statement used to replace column values. `if` tests a condition, and if it is true, performs the code below it. For example:

```
x = value("Connectivity")
if x > 3:
    x = 0
x
```

In this script, `x` is first assigned the value of connectivity. In the next line, the script tests if the value of connectivity is greater than three. If it is, then `x` is assigned the value 0. Note that this line, the assignment, is indented. The indented line is only performed if the query for the `if` statement is true. You can indent more than one line if you wish; as long as lines are indented, they will be performed only if the condition in the `if` statement is true. However, the final line of the script, `x`, is not indented, so this will be performed whether or not the `if` condition was true. The line is used to tell UCL Depthmap what value should be given to the column: simply set it to whatever `x` is. Thus, this script sets the column to 0 if the connectivity is greater than three, otherwise it sets it to the connectivity.

There is a semantically clearer way to write the same script, according to the logic we have just stated:

```
x = value("Connectivity")
if x > 3:
    0
else:
    x
```

This script gives an identical output to the last script, it just uses a slightly different syntax. The line that reads `0` *returns* 0 to UCL Depthmap if `x` is greater than 3. The `else` statement specifies what to do otherwise: simply return the value of `x`. This follows the logic we wrote originally: if something, otherwise something else.

We have seen `if` and `else`, there is also a statment `elif`, which is used for 'else if', for example, a script might read:

```
if value("Ref Number") < 10:
    100
elif value("Ref Number") < 20:
    200
else:
    value("Connectivity")
```

In this script, if the value of 'Ref Number' is less than 10, SalaScript returns 100 to UCL Depthmap. Otherwise, if 'Ref Number' is less than 20, it returns 200. Finally, if neither of the preceding two conditions are true, it simply returns the value of connectivity.

When beginning to use the `if` statement, the way UCL Depthmap receives the value can be a little confusing at first. Note that the following code does not work as might be expected:

```
x = value("Connectivity")  # Poor script: do not copy
if x > 3:
    0
```

```
x        # this line is always called
```

In this script, if `x > 3` then '0' is evaluated ('0' is, simply 0). However, the next line, because it is not indented, always happens immediately afterwards, regardless of the '0' in the preceding line. Thus, the value of `x` (the connectivity) is returned to UCL Depthmap, and the value assigned to the row. Curiously, the following script does work:

```
x = value("Connectivity")
x
if x > 3:
    0
```

SalaScript always returns the last value it evaluated to UCL Depthmap, so, in this example, it evaluates `x`. Then, only if `x > 3`, it evaluates the line reading '0'. More details about how values are 'returned' to UCL Depthmap can be found in section 3.4.3

3.4.2 for and while Statements

The `if` statement allowed us to change the flow of the program depending on whether or not a condition was true. `for` and `while` allow us to carry out a number of statements in a row depending on whether or not a condition is true.

The `while` statement repeats the indent statements while a condition is true. For example:

```
x = 1
while x < 10:
    x = x * 2
x
```

In this example, the line `x = x * 2` is repeated while `x` is less than 10. So `x` starts as 1, is multiplied by 2, making it 2. Since this is less than 10, the code is repeated, so it becomes 4 (still less than 10), 8 (still less than 10) and finally, 16. Since 16 is greater than 10, the code is not repeated any longer, and the next line is executed: a simple evaluation of `x`, which is 16. If this script is run in UCL Depthmap, every row is set to 16.

The `for` statement is slightly more complicated. It repeats a statement for every element in a list. As an example:

```
x = 0
for y in [1, 2, 3, 4]:
    x = x + y
x
```

In this example, the code loops around the line `x = x + y`, using each value of the list in turn. The first time it sets `y` to `1`, so `x = x + 1`. The value of `x` is thus `1`. The next time through the loop `y` is set to the second element in the list, `2`. `x = x + y` is evaluated with `y` set to `2`, so `x = x + 2`, so `x` now totals `3`. The next time round `y` is `3`, and `x` is set to `6`. The next element `4` is the last in the list, so `x` is finally set to `10`. The final line of the script, the '`x`' on

its own, simply tells SalaScript to return the value of `x` to UCL Depthmap. So this script would set every row in the table to 10.

Another way to produce the same result is to use the `range(`*x,y*`)` function, which creates a list of a series of integers from x up to but not including y. For example, `range(1,5)` creates the list we have just used: `[1,2,3,4]`. So the script could be rewritten:

```
x = 0
for y in range(1,5):
    x = x + y
x
```

Range is useful when you want to create a series, and saves typing, e.g., doing something ten times with the call `range(0,10)`. More details about range can be found in appendix A.8.

So far, the scripts we have seen have not been that useful, so let us turn to a simple example that we might genuinely want to calculate: the sum of the connectivities of a node's neighbours. In order to do this we need to *iterate* through each of the node's neighbours, and sum its connectivity. We can use very similar loops to those above, but rather than using the `range(`*x,y*`)` function, we can use the `connections()` function. The `connections()` function returns the list of connections for the current node. We can go through each connection using a for loop:

```
# Sum of connectivity of neighbours
sum = 0
for neighbour in connections():
    sum = sum + neighbour.value("Connectivity")
sum
```

The only difficult part of this script is the use of `value(`*x*`)` as a *member function*. Just writing `value("Connectivity")` returns the connectivity of the current node. By writing `neighbour.value("Connectivity")` we get the connectivity of the node '`neighbour`'. The for loop ensures that the variable '`neighbour`' is set to each of the current node's neighbours in turn. The final line, which reads simply `sum` on its own returns the value to UCL Depthmap. It is probably worth trying this script on a simple axial or point map to check you understand how it works.

3.4.3 Return Values

You are probably used to the final line of any script being the value we want UCL Depthmap to use by now. For example, this script which calculates the connectivity squared:

```
x = value("Connectivity")
x = x^2
x
```

This convention is actually an implicit form of a standard programming concept, the *return value*. What each script does is calculates a value for the current node or row, and then *returns* it to UCL Depthmap. We can actually write out

the 'return' explicitly, using the keyword `return`, and this is encouraged in complex scripts so it is obvious what is intended. Here is the last example with an explicit return:

```
x = value("Connectivity")
x = x^2
return x
```

Typically clarity can be increased when writing complex `if` statements. For example, recall this example:

```
x = value("Connectivity")  # Poor script: do not copy
if x > 3:
    0
x        # this line is always called
```

This script does not work because the final line is always reached, and always implicitly returns the value `x` to UCL Depthmap. We can convert this so it works using the return keyword:

```
x = value("Connectivity")
if x > 3:
    return 0
x
```

However, it is still not that clear. Far clearer is to spell out every return, and exactly what the script is trying to do, like so:

```
x = value("Connectivity")
if x > 3:
    return 0
else:
    return x
```

3.5 Graph Measure Examples

The preceding sections have been rather theoretical. It is now time to put the theory to use for real, and see each of the program control statements in action, along with lists and variables.

3.5.1 Total Depth Calculation

The total depth is one of the standard measures in space syntax. It is the sum of the lengths shortest paths to all other nodes in the graph.

Here is the script in full:

```
1 # Total Depth Calculation
2 total_depth = 0
3 depth = 0
4 pop_list = [this]
5 push_list = []
```

```
6  setmark(true)
7  while len(pop_list):
8      curs = pop_list.pop()
9      total_depth = total_depth + depth
10     for i in curs.connections():
11         if i.mark() is none:
12             i.setmark(true)
13             push_list.append(i)
14     if len(pop_list) == 0:
15         depth = depth + 1
16         pop_list = push_list
17         push_list = []
18 return total_depth
```

The script works as follows: it assigns two lists a 'pop list' (from which discovered nodes will be 'popped') and a 'push list' (to which newly discovered nodes will be 'pushed'). It starts by putting 'this', the current node, into the pop list. The set mark function on line 6 tells SalaScript to remember that the current node has been seen (we simply set it to the Boolean constant `true` to indicate 'seen'. This sets everything up for a loop, the while loop on line 7, which will keep track of a set of a frontier of nodes fanning out from the original node. This frontier is what is in the 'pop list' (currently just the starting node). When the frontier dies away at the edge of the graph, it will be empty, hence the loop is told only to continue while there are elements in the pop list, that is `while len(pop_list) != 0`. The '`!= 0`' is implicit though, as `0` evaluates to the Boolean constant `false`. We enter the loop, and take one node from the frontier. In the initial instance, just the starting node, `this`, is the frontier. We then add to the total depth the current depth – we are in effect saying this node has been discovered at depth `depth`. For the initial node, this is facile: it is at depth 0, and so 0 is added to the total depth. Then we go through the list of connections of the current node. Each of these is added to the push list if and only if it has not been seen before (the check for this is on line 11). As soon as it is discovered in this way, it is marked as seen on line 12, The node is appended to the push list on line 13. There only remains one trick in the script to make it all work: the push list is the new frontier. Once all the nodes in the current frontier are used, the depth is incremented and the lists are swapped over: the push list becomes the frontier, the pop list and then it is cleared for the process to start over. In this way, on the first iteration of the while loop, all the nodes directly connected to the starting node are added to the new frontier. Immediately the starting node has been evaluated, the frontier is switched to the list of directly connected nodes, and the depth incremented to 1. The loop goes through each of these nodes, adding the depth (1) to the total depth for each of them, and adding any nodes they are connected to that have not been seen yet to another new frontier. When the current frontier is exhausted, the new frontier becomes the current frontier, and so on, until all the nodes in the graph have been seen. Finally, at the end, all that remains is to return the calculated total depth to UCL Depthmap.

3.5.2 Shortest Cycle Calculation

This shortest cycle calculation is a demonstration of about the limit of complexity a SalaScript version 1.0 script can get to. Since functions cannot be defined and there are no debugging tools, a long script is tricky to test. The script that follows took several trials to get right, and required some ingenuity to debug by taking advantage of the `setvalue` function to override the results of earlier calculations.

Note the script is designed for axial and convex maps rather than VGA or segment maps. It can easily be adapted to segment maps, but requires that `connections()` is replaced by `connections("all")`. VGA maps almost always have short cycle lengths, as they are highly interconnected, and so running the script will generally result in setting the value to 3 (the shortest cycle returning via a different node).

Rather than explain exactly how it works, I will sketch the principle. All the connections to the originating axial line are tagged as different paths leading away from it. These paths are then followed, and should two different paths come into contact, then a cycle has been formed. When a cycle is discovered, the length of the two paths that have met is added together, along with adding one for the original node in the cycle, and returned to UCL Depthmap. Should all the paths but one die off, then there is no possibility of forming a cycle, and the main while loop quits. Otherwise the discovery of nodes is very similar to the total depth calculation in section 3.5.1.

The script is as follows:

```
# Shortest Cycle
push_list = []
pop_list = []
live_paths = []
setmark([-1,0])
depth = 1
path_index = 0
for i in connections():
    pop_list.append([path_index,i])
    live_paths.append(1)
    i.setmark([path_index,depth])
    path_index = path_index + 1
if path_index < 2:
    return -1 # no cycle possible
live_path_count = path_index
while live_path_count > 1:
    curs = pop_list.pop()
    path_index = curs[0]
    this_node = curs[1]
    live_paths[path_index] = live_paths[path_index] - 1
    for i in this_node.connections():
        if i.mark() is none:
            i.setmark([path_index,depth+1])
            push_list.append([path_index,i])
            live_paths[path_index] = live_paths[path_index] + 1
        elif i.mark()[0] != path_index and i.mark()[0] != -1:
```

```
27              # found a cycle!
28              return i.mark()[1] + this_node.mark()[1] + 1
29      if live_paths[path_index] == 0:
30          live_path_count = live_path_count - 1
31      if len(pop_list) == 0:
32          depth = depth + 1
33          pop_list = push_list
34          push_list = []
35  return -1 # no cycle found
```

Appendix A

Language Reference

A.1 Mathematical Operators

x* + *y Returns x added to y, so `2 + 3` is 5.

x* - *y Returns y subtracted from x, so `2 - 3` is -1.

x* * *y Returns x multiplied by y, so `2 * 3` is 6.

x* / *y Returns x divided by y. Note that there are two sorts of division, integer division and floating point division. In integer division, 5/2 is 2 remainder 1. So, in SalaScript, `5 / 2` is the result of the division part, 2 (for the remainder part, see the `%` operator). In floating point division, 5/2 is 2.5. So, in SalaScript, `5.0 / 2.0` returns 2.5.

x* % *y Returns x modulo y, that is, the remainder after division by y. So `8 % 6` is 2. Note that SalaScript can also apply this operator to floating point numbers, so `8.4 % 4.1` is 0.2.

x* ^ *y Returns x raised to the power of y, so `2^3` is two cubed, or 8, and `27^(1/3)` is the cube root of 27, or 3.

A.2 Mathematical Functions

Mathematical function names are *reserved words* which means you cannot use them as variable names.

`ln(`***x***`)` Returns the natural logarithm (base e) of x

`log(`***x***`)` Returns the base 10 logarithm of x

`cos(`***x***`)` Returns the cosine of x, where x is in radians

`sin(`***x***`)` Returns the sine of x, where x is in radians

`tan(`***x***`)` Returns the tangent of x, where x in radians

`acos(`***x***`)` Returns the inverse cosine (arccosine) of x in radians

`asin(x)` Returns the inverse sine (arcsine) of x in radians

`atan(x)` Returns the inverse tangent (arctangent) of x in radians

`random()` Returns a random number in the range 0.0 to 1.0

`sqrt(x)` Returns the square root of x

A.3 Mathematical Constants

Mathematical constant names are *reserved words* which means you cannot use them as variable names.

`e` 2.71828182845904523536028747135 27

`pi` 3.14159265358979323846264338327 95

A.4 Comparison Operators

Comparison operators are *reserved words* which means you cannot use them as variable names.

x `==` ***y*** Returns `true` if x is equal to y, so `2 + 3 == 5` is true. Note the double equals sign is used for comparison, whereas the single equals sign is used for assignment of values to variables.

x `!=` ***y*** Returns `true` if x is not equal to y, so `2 + 3 != 5` is false.

x `>` ***y*** Returns `true` if x is greater than y, so `3.0 > 2.0` is true.

x `<` ***y*** Returns `true` if x is less than y, so `3.0 < 2.0` is false.

x `>=` ***y*** Returns `true` if x is greater than or equal to y, so `3.0 >= 2.0` is true.

x `<` ***y*** Returns `true` if x is less than or equal to y, so `3.0 =< 2.0` is false.

x `is` ***y*** Returns `true` if x is identical to y, otherwise `false`. The `is` operator mirrors Python's `is` operator. For simple types, `is` returns `true` if both the type and the value are the same, e.g., `3 is 3` is true, but `3 is 3.0` is false, even though `3 == 3.0` is true. For lists and strings, SalaScript checks whether or not there is an identical reference. Thus `[1,2] is [1,2]` returns `false`, as these are two different instances of the list, whereas if x is set to `[1,2]` and then y set to x, then `x is y` is `true`. `is` is most commonly used where `==` would throw an exception, for example, to check if a variable which may or may not be assigned is `none`.

A.5 Logical Operators

Logical operators are *reserved words* which means you cannot use them as variable names.

x* `and` *y Returns `true` if both x and y are true, so `2+3==5 and 5+6==10` is false, because only part of the condition, `2+3==5`, is true. Conversely, `2+3==5 and 5+6==11` is true, because both the first part and the second part are true.

x* `&&` *y Synonym for the `and` operator.

x* `or` *y Returns `true` if either x or y or both x and y are true, so `2+3==5 or 5+6==10` is true, because the first part of the condition is true. Similarly `2+3==5 or 5+6==11` is true as both parts of the condition are true.

x* `||` *y Synonym for the `or` operator.

`not` ***x*** Returns `true` if x is not true and `false` if x is true. So `not 2+3==5` is false.

`!` ***x*** Synonym for the `not` operator.

A.6 Boolean Constants

Boolean constant names are *reserved words* which means you cannot use them as variable names.

`true` Boolean value 'true'

`false` Boolean value 'false'

A.7 Special Variables

Special variable names are *reserved words* which means you cannot use them for user defined variable names.

`none` Represents a variable with no value. There is just one variable called `none`. So if you set `x = none`, then the test `x is none` is always true; that is, `x` is exactly the same as `none`.

`this` Represents the current object. When applying SalaScript, `this` is usually a graph object or node, such as a line, or a point, which has a set of connections associated with it (see Graph Objects on page 28). `this.connections()` is the same as calling `connections()`.

A.8 Miscellaneous Functions

`range(`***x, y [,z]***`)` Returns a list of integers starting from x and less than y. The optional parameter z sets the step size for the series, which defaults to 1. Note that the square brackets around z indicate that the parameter is optional, not that you should type the square brackets. Examples: `range(1,5)` produces the list `[1, 2, 3, 4]`; `range(2,10,3)` produces the list `[2,5,8]`.

Appendix B

Inbuilt Types

B.1 Booleans

Booleans have just two values: `true` or `false`.

Assignment `x = true` or `y = false`.

B.2 Integers

Integer values are positive and negative whole numbers: $0, 1, 2, 3 \ldots$ and $-1, -2, -3 \ldots$.

Assignment `x = 1` or `y = -2` or `z = +2`

B.3 Floats

'Float' is short for 'floating point number'. That is, a number with a mantissa and an exponent. You are probably familiar with the notation `2.0e05` meaning 2.0×10^5, where 2.0 is the mantissa and 5 is the exponent. In this notation, the decimal point 'floats' as the exponent is changed, so `1.5e-1` is 0.15 but `1.5e-2` is 0.015. UCL Depthmap attribute values are stored as floats.

Assignment When the exponent is omitted, a decimal point is used to distinguish floats from integers. For example, `x = 1.0`. With the exponent, the decimal place is optional, for example, `y = -2.0e05` or `z = +3e-2`

B.4 Strings

Strings are 'strings of characters' joined into a thread. So 's' and 'o' joined into a string is 'so'. In SalaScript, all characters are strings, even single ones.

Assignment `x = "hello"` or `y = 'world'`

Access Operator `[`x`]` The access operator can be used to access the character at position x in the string, starting at index 0. For example, if `x` is set to `"hello¨` then `x[0]` is `"h"` and `x[4]` is `"o"`.

`len(x)` Returns the length of (number of characters in) the string. So `len('world')` is 5.

B.5 Lists

Lists in SalaScript are a set of elements strung together. A list might look like `[1, false, "hello"]`, that is, a list of the integer `1`, the Boolean constant `false` and the string `"hello"`. Note that lists may be nested, so `[1,2,[4,5]]` is a valid list.

Assignment `x = [1, 2, 3]`. That is, square brackets containing a list of values separated by commas. `y = [3]` (single element list) and `z = []` (empty list) are also valid assignments.

Access Operator `[`*x*`]` The access operator can be used to access the list element at position *x* in the list, starting at index 0. For example, if `x` is set to `[1, false, "hello"]` then `x[0]` is `1` and `x[2]` is `"hello"`. Nested lists can be accessed by chaining together access operators, so if *x* is set to `[1,2,[4,5]]`, `x[2][0]` is 4.

`len(x)` Returns the number of elements in a list. So `len([1,3, [4,5]])` is 3.

`range(x, y [,z])` See appendix A.8

B.5.1 List Member Functions

`append(x)` Appends the item *x* to the end of the list, and returns `none`. So if you assign the list `lst = [1,2]` then `lst.append('hello')` appends `'hello'` to the list `lst` so its value would now be `[1,2,'hello']`. Then calling `lst.append([3,4])` would modify `lst` so its value would now be `[1,2,'hello',[3,4]]`.

`extend(x)` Extends the list by adding all the items in list *x* to the end of the list, and returns `none`. So if you assign the list `lst = [1,2]` then `lst.extend([3,4])` would amend `lst` so its value is `[1,2,3,4]`. Note the difference to `append` is that `extend` adds each item in the list, rather than the list itself to the end of the list.

`pop(`*[x]*`)` Pops the last element (or element *x*) from the list and returns it. Note that the square brackets mean that the parameter *x* is optional, not to write the square brackets when using the function. So if `lst` is set to `[1,2,4,5]` then `lst.pop()` returns the last element of the list, 5, and cuts the last element from `lst`, so `lst` is now `[1,2,4]`. Then calling `lst.pop(1)` would return element 1 from the list, 2, and cuts that element from the list, so the value of `lst` would now be `[1,4]`.

B.6 Graph Objects

Graph objects are specific to UCL Depthmap. They have geometry, attributes, and connections to other graph objects. In most contexts in which SalaScript runs, the variable `this` is a graph object.

In SalaScript version 1.0, the attributes and connections of a graph object can be accessed, but not the geometry.

Assignment Graph objects cannot be created in SalaScript version 1.0, so variables can only be assigned to existing graph objects. For examples `x = this` (assuming `this` is a graph object) or `y = connections()[3]` (assuming there are at least 4 items in the list of connections).

B.6.1 Graph Object Member Functions

`value(`***x***`)` Gets the value of the attribute in column x, where x is a string. If `this` is a graph object, then examples might be `this.value("Ref Number")` or, because `this.` is unnecessary, simply `value("Ref Number")` directly.

`setvalue(`***x,y***`)` Set the value of the attribute in column x to the value y, where x is a string and y is a floating point number. For example, if `node` is a graph object, and 'My Attribute' is a column name, `node.setvalue("My Attribute",3.5)` set the value of 'My Attribute' to 3.5 for `node`'s row of the attributes table.

`connections(`*[x]*`)` Returns the list of connections for the graph object. Each connection is also a graph object. For example, if `node` is a graph object in a point map or axial map, `node.connections()` gets the list of all connections. The optional parameter x is dependent on the type of graph. For point maps, it is used to specify the bin, so `node.connections(0)` gets the connections from bin 0. Bins are labelled from 0 to 31. For axial maps, convex maps and shape maps in general, the parameter is unused. For segment maps, one of three options **must** be specified: `'all'`, `'forward'` or `'back'`. `'all'` gets both forward and backward connections from the segement. `'forward'` gets only forward connections, and `'back'` only backward connections. The segment 'direction' is arbitrary, and set by UCL Depthmap: you should not rely on directions being geometrically consistent, however, if a connection is labelled forward from segment x to y, then it is labelled back from segment y to x.

`mark()` Returns a user-defined mark from the graph object. If `this` is a graph object, then `mark()` retrieves the user-defined mark. All user-defined marks are cleared between iterations of SalaScript scripts, so when replacing column contents, the marks are reset as each row of the table is processed. If no mark has been set, `mark()` returns `none`.

`setmark(`***x***`)` Sets a user-defined mark for the graph object, for example, to indicate when a node has been visited in a depth analysis. If `this` is a graph object, then `setmark(true)` sets the user-defined mark to the Boolean value `true`. All user-defined marks are cleared between iterations of SalaScript scripts, so when replacing column contents, the marks are reset as each row of the table is processed.

Appendix C

Operator Precedence

Operator precedence is the order of precedence in which operators are applied. So because the `^` operator has a higher precedence than the `/` operator, it is performed before it. Thus, if you write `27.0^1.0/3.0` you get 9: 27 to the power 1, divided by 3. In order to get what you might have expected, 27 to the power 1/3, that is, the cube root of 27, write `27.0^(1.0/3.0)`, because brackets have a higher precedence than the `^` operator, and are therefore applied first.

The operator precedence in SalaScript is:

1	`=`	Assignment
2	`and`	Logical and
3	`or`	Logical or
4	`not`	Logical not
5	`==`, `is`, `<`, `>`, `<=`, `>=`, `!=`	Comparison operators
6	`+`, `-`	Add and subtract
7	`*`, `/`, `%`	Multiply, divide and modulo
8	`^`	Raise to the power of
9	`()`, `[]`	Bracketed values