// test if the two numbers are equal

s = Print(a == b ? "Are equal" : "Are not equal");

The connection between functions and variables is significantly stronger in

DesignScript than other programming languages: once a variable is used in

a function, any changes to that variable later in the program will cause the

original function to update. This is called associativity.

For example, if we create a variable x, and use that variable to create a

Point p, but then later change x, p will change too.

Associativity can accommodate a variety of changes to a variable. For

instance, a variable go from being a single value to a collection, or vice

versa.

By default, DesignScript is an associative language (see: *Associativity*),

which is suited for design exploration and iterative sketching. However,

there are some instances when associative programming leads to complex

algorithms and difficult to read code. In some of these instances, the

complexity can be simplified by a more traditional and straight-forward

programming style called Imperative programming invoked by declaring an

Imperative key word or ‘directive.’ These blocks allow code to execute in a

purely sequential manner: variables modified later in a program have no

effect on the code which preceded it.

Imperative blocks are declared with the [Imperative] command,

wrapping code in curly braces ({}). Like functions, Imperative blocks are

required to return a value with the return = command.

The following example shows how to use an Imperative block to create a

collection of Cuboids by reusing a CoordinateSystem from a previous

operation:

import("ProtoGeometry.dll");

cuboids = [Imperative]

{

c = {};

cs = CoordinateSystem.Identity();

cs = cs.Translate(20, 0, 0);

c[0] = Cuboid.ByLengths(cs, 10, 10, 10);

cs = cs.Rotate(25, Vector.ByCoordinates(1, 1, 1));

cs = cs.Translate(0, 20, 20);

c[1] = Cuboid.ByLengths(cs, 10, 10, 10);

cs = cs.Rotate(35, Vector.ByCoordinates(1, 1, 1));

cs = cs.Translate(0, 20, 20);

c[2] = Cuboid.ByLengths(cs, 10, 10, 10);

cs = cs.Rotate(15, Vector.ByCoordinates(1, 1, 1));

cs = cs.Translate(0, 20, 20);

c[3] = Cuboid.ByLengths(cs, 10, 10, 10);

return = c;

}

DesignScript also allows a programmer to switch from an Imperative block

back to an Associative block, if this leads to more legible code and cleaner

algorithms. Similar to Imperative blocks, Associative blocks are designated

with the [Associative] key word, also wrapping code in curly braces

({}), and also requiring the programmer to return a value with the return

= command.

In associative DesignScript code, if a collection of elements is used as the

input to a function which normally takes a single value, the function is

called individually for each member of a collection. In imperative

DesignScript code a programmer has the option to write code that

manually iterates over the collection, extracting individual collection

members one at a time.

The for statement extracts elements from a collection into a named

variable, once for each member of a collection. The syntax for for is:

for(“extracted variable” in “input collection”)

Replication Guides

The DesignScript language was created as a domain-specific tool for

architects, designers and engineers, and as such has several language

features specifically tailored for these disciplines. A common element in

these disciplines is the prevalence of objects arrayed repetitive grids, from

brick walls and tile floors to façade paneling and column grids. While range

expressions offer a convenient means of generating one dimensional

collections of elements, replication guides offer a convenient means of

generating two and three dimensional collections.

Replication guides take two or three one-dimensional collections, and pair

the elements together to generate one, two- or three-dimensional

collection. Replication guides are indicated by placing the symbols <1>,

<2>, or <3> after a two or three collections on a single line of code. For

example, we can use range expressions to generate two one-dimensional

collections, and use these collections to generate a collection of points:

The order of the replication guide numbers (<1>, <2>, and/or <3>)

determines the order of the underlying collection. In the following example,

the same two one-dimensional collections are used to form two twodimensional

collections, though with the order of <1> and <2> swapped.

Imperative programming is characterized by explicit 'flow control' using **for** loops (for iteration) and **if** statements (for conditionals) as found in familiar scripting and programming languages such as Python. Imperative programming is useful to perform iteration, either stepping through a collection or to perform some iterative feedback or optimisation loop.

Associative programming uses the concept of graph dependencies to establish 'flow control’ and is useful for modeling complex operations (such as geometric processes) applied to collection of objects, in a very succinct programming style with automatic change propagation.

The two styles of programming address different computational tasks and essentially complement each other. The different styles of programming share a common notation which means that in some case the same code can be executed either associatively or imperatively. In addition there are certain computational tasks that benefit from a combination of programming styles within the same program or indeed within the same function or method. DesignScript supports this flexibility by allowing Imperative code to be nested within Associative code and vice versa.

o **DesignScript is a block structured language:** Blocks are defined with {} and may have annotations in []

The outer block in a DesignScript program is interpreted as being Associative.

o **Scope of variables**: The ‘scope’ of a variable is controlled by the block in which it is defined. Variables defined within one block can be used within inner blocks, but not within outer blocks.

Only geometric variables defined within the outermost block (‘global scope’) will be generated and displayed in the DesignScript host application (currently AutoCAD). It is therefore possible to construct, use and dispose of geometric variables without the ‘cost’ of displaying these by not defining these in the outermost block. [Note: the outermost block is by default Associative]

**User defined types**: these are types defined as classes within DesignScript or via imported DLL’s . DesignScript is an object-oriented language and uses the standard terminology of class, subclass, superclass, constructor, method, arguments and instance, as in:

instance = Class.Constructor(arg1, arg2,… argN);

A variable is an instance of a class using a particular named constructor or method with various input arguments.

DesignScript intentionally restricts the change of a single property [such as the X coordinate of a point] in isolation, outside the known context provided by such method.

a.X = 10; // not allowed, instead a method should be used to define a.X in the context of other properties

There are some properties that are ‘writeable’ and can be assigned to, specifically in the DesignScript ProtoGeometry library:

import("ProtoGeometry.dll"); a = Line.ByStartPointEndPoint (Point.ByCartesianCoordinates(WCS, 5, 5, 0), Point.ByCartesianCoordinates(WCS, 10, 5,0));

a.Color = Color.Red; // assign a color to a geometric variable a.Visible = true; // control a variable’s visibility

The users should check the properties of the class he is using to determine which properties are read only or writable.

In summary:

 A variable can be untyped and therefore can be a single value of any type or a collection of any dimension of any type.

 Or a variable can be declared as a single value or as a collection of a specified dimension, or as a collection of any dimension; and the variable can be typed or untyped.

**Declaring variables:**

If it is important to declare the type of the variable and it is anticipated that a variable may be a single value or a collection of values, then the declaration variable : type[]..[]; should be used.

If it is important to declare the type of the variable and it is anticipated that a variable could be of more than one type, then the type declared should be the most specialised common super type of the anticipated values for this variable. For example, if a variable could be a Line or an Arc, then it should be declared as variable : Curve; [as the common super type]

Similarly, if it is important to declare the type of a collection variable and it is anticipated that the variable will be a heterogeneous collection, then the type of the collection should the most specialised common super type of the anticipated members. For example, if a variable collection could contain members that are Line or Arc, then the collection should be declared as variable : Curve[]; [as the common super type] or variable : Curve[]..[]; if it is anticipated that the dimension of the collection may change.

Therefore (more generally) the users should check the class hierarchy of the libraries he is using to ensure that he has selected the appropriate common super type.

Imperative Programming: accessed via the [Imperative]directive applied to a block:

Imperative programming is useful to perform iteration, either stepping through a collection or to perform some iterative feedback or optimisation loop.

Imperative programming uses conventional ***for*** and *while* loops and ***if*** .. *else* statements to explicitly define ‘flow control’.

o **Program execution**: In the absence of such flow control the next statement in the program is executed, for example:

**1** a; b; // define the variables to be output at the top or outer scope

**2** [Imperative] **3** { **4** a = 10; // original value of 'a' **5** b = a \* 2; // calculating ‘b’ based on the current value of ‘a’.. ‘b’ = 20 **6** a = 15; // changing the value of 'a' will NOT cause the value of 'b' to change **7** }

If this code fragment is executed in single step debug, the following sequence of statements will be executed: 4, 5, 6.

o **Iteration**: defined by a ***for*** loop, as: ***for*** *(variable in collection) {…}*

defined by a ***while*** loop, as: ***while****(condition\_is\_true) {..}*

**Associative Programming**: accessed via the [Associative]directive:

Associative programming uses the concept of graph dependencies to establish 'flow control.' Changes to 'upstream' variables are automatically propagated to ‘downstream’ variables. Associative programming in DesignScript also implements two additional concepts: ‘replication’ and ‘modifiers’.

With replication, anywhere a single value is expected a collection may be used instead and the execution is automatically *replicated* over each element. The combined result of dependencies and replication is that is easy to program complex data flows (including geometric operations) involving collections. An upstream variable may change from being a single value to a collection or from a collection to another collection of different dimensions or size, so the downstream dependent variables will automatically follow suit and also become collections of the appropriate dimension and size. This makes Associative programming incredibly powerful, particularly in the context of generating and controlling design geometry.

With modifiers, each variable can have multiple states, which might reflect the geometric modeling sequence. For example a geometric variable might be created (say as a curve) and then it can be ‘modified’ by being trimmed, projected, extended, transformed or translated. Without the concept of modifiers each state or modeling operation would require to be a separate variable and this would force the user to have to make up the names of all these intermediate variables. Modifiers avoid imposing this naming process on the user.

Dependencies, replication and modifiers can all be combined to represent the typical modeling operations found in architecture and constructions. Buildings are composed of collections of components. Typically these collections are often the product of a series of standard operations across all members. On the other hand, within such collections there may be special conditions where different or additional modeling operations are required to be applied to a subcollection of members. Modifiers enable these special conditions to be identified and for additional modeling operation applied.

**1**

[Associative] **2** { **3** a = 10; // original value of 'a' **4** b = a \* 2; // define ‘b’ as dependent on ‘a’.. ‘b’ initially = 10, then = 30 **5** a = 15; // changing the value of 'a' will change the value of 'b' (now = 30) **6** } **7**

If this code fragment is executed in single step debug, the following sequence of statements will be executed: 4, 5, 6, 5.

In associative programming, statement 5 is not just executed once (in sequence, after statement 3).

In associative programming, statement 5 establishes a persistent relationship between the variable b and variable a.

When the value of variable a is re-defined in statement 6, the Associative update mechanism in DesignScript will execute all statements that depend on variable a, which (in this example) includes statement 5. Hence the execution sequence: 4, 5, 6, 5.

**Collections and Replication**: a collection can be used where a single value is expected

a; b; // define the variables to be output at the top or outer scope **1**

[Associative] **2** { **3** a = 10; // original value of ‘a’ **4** b = a \* 2; // define ‘b’ as dependent on ‘a’ **5** a = { 5, 10, 15 }; // redefine a as a collection.. the value of b is now = {10, 20, 30} **6** } **7**

In this example, when a becomes a collection of values, the expression a \* 2 is executed for every member of a and the resulting collection of values are assigned to b. In associative programming, the existence of variable as a collection is propagated to all dependent variables, in this example ‘b’. This propagation of collections is called ‘replication’.

If this code fragment is executed in single step debug, the following sequence of statements will be executed: 4, 5, 6, 5.

Note that variables can be untyped, but if they are declared as typed then DesignScript assumes that the user wants to restrict the values that can be assigned to that variable and will report errors if an erroneous assignment is attempted. For example, if a and b are defined as single int’s, below:

a : int; // explicitly defined as a single int b : int; // explicitly defined as a single int [Associative] { a = 10; b = a \* 2; a = { 5, 10, 15 }; // changed into an array of int's … this will fail }

To overcome this, there are two strategies:

a : int[]..[]; // explicitly defined a variable as a type which could be a single value or a collection b; // declare the variable as untyped

**Zipped replication**:

When there are multiple collections within the same expression, we need to control how these are combined. With ‘zipped’ replication, when there are multiple collections, the corresponding member of each collection is used for each evaluation of the expression. This works well when all collections are the same dimension and length. If collections are of different lengths, then the shortest collections determines the number of times the expression is evaluated, and hence the size of the resulting collection.

a; b; c; // define the variables to be output at the top or outer scope

[Associative] { a = {1, 5 ,9}; b = {2, 4 ,6}; c = a + b; // zipped replication operation .. c = {3, 9, 15} }

Note: replication works well with rectangular collection, but the results may be undefined when used with ragged collections.

**Combining zipped and cartesian replication**

In the following example, we are taking the cartesian product of one 1D array and another 1D array [to create an intermediate 2D array) and then 'zipping' this intermediate 2D array with another 2D array

import("ProtoGeometry.dll"); // use the standard DesignScript geometry library import("Math.dll"); // use the standard DesignScript math library

surfacePoints\_2D\_array : Point[][]; // define a 2D array of Points surface : BSplineSurface; // define surface

[Associative] { xSize = 10; ySize = 15; xHeight = 2; yHeight = 4; numColsX = 8; numColsY = 6; xCoords\_1D\_array = 0..xSize..#numColsX; // 1D array yCoords\_1D\_array = 0..ySize..#numColsY; // 1D array xSineWave\_1D\_array = (Math.Sin(0..180..#numColsX) \* xHeight); // 1D array ySineWave\_1D\_array = (Math.Sin(0..180..#numColsY) \* yHeight); // 1D array zHeight\_2D\_array = xSineWave\_1D\_array<1> + ySineWave\_1D\_array<2>; // using cartesian replication // adding a 1D array to another 1D array creates a 2D array surfacePoints\_2D\_array = Point.ByCoordinates(xCoords\_1D\_array<1>,

yCoords\_1D\_array<2>,

zHeight\_2D\_array<1><2>);

// this operation is taking the cartesian product of xCoords\_1D\_array and yCoords\_1D\_array // [to create a 2D array) and then 'zipping' this 2D array with zHeight\_2D\_array surface = BSplineSurface.ByPoints(surfacePoints\_2D\_array).SetColor(Color.Cyan); // create a surface }

**Modifier blocks**:

The statements that define and modify a variable do not have to be contiguous statements in a program. But this flexibility of separating the statements relating to same variable may eventually lead to a considerable loss of clarity. It may be advantageous to group all the statements which define and modify a specific variable into a single program block, as follows:

import("ProtoGeometry.dll"); // use the standard DesignScript geometry library

a; // define the variable to be output at the top or outer scope

[Associative] { a = { // create the line Line.ByStartPointEndPoint(Point.ByCoordinates(10, 0, 0), Point.ByCoordinates(10, 5, 0)); Trim(0.2, 0.8, false); // trim the line Translate(1, 1, 0); // move the trimmed line } }

Another advantage of the Modifier block is that the name of the variable does not have to be repeated on the left and right hand side of each modifier statement. If the variable name is long, this removes a source of typing errors and improves readability.

o **Right Assign**: allows the labeling and referencing of intermediate states within a modifier block using the **=>** notation.

import("ProtoGeometry.dll"); // use the standard DesignScript geometry library

a; b; c; d; // define the variables to be output at the top or outer scope

[Associative] { a = { Line.ByStartPointEndPoint(Point.ByCoordinates(10, 0, 0),

Point.ByCoordinates(10, 5, 0)) => a@initial;

//'right assign' the initial state of ‘a’ to a new variable

Trim(0.2, 0.8, false) => a@trim; // trim the line and 'right assign' the intermediate state

// of ‘a’ to a new variable

Translate(1, 1, 0); // move the trimmed line } b = a; // 'b' will be the final state of 'a' c = a@initial.Translate(-1, 1, 0); // 'c' is a modification of an intermediate state of 'a'; d = a@trim.Translate(-1, -1, 0); // similarly for 'd' }

o **In-line conditional**: an in-line conditional is defined as:

*variable = boolean\_expression* ***?*** *expression\_if\_true : expression\_if\_false;*

**In-line conditional where the value of one property is used to set the value of another (writable) property**:

In the following example the boolean expression within the in-line conditional is evaluated for each member of the collection and the value of the ‘*expression\_if\_true*’ or value of the ‘*expression\_if\_false*’ is assigned to the writable property of that member of the collection.

import("ProtoGeometry.dll"); curve : Curve[]..[]; // define the output variable as a collection of Curves

[Associative] { start = Point.ByCoordinates(0..10, 0, 0); // a 1D array of point end = Point.ByCoordinates(5, 5, 0); // a single point curve = Line.ByStartPointEndPoint(start, end); // a 1D array of lines curve.Color = curve.Length > 6 ? Color.Red : Color.Blue; // use length to determine color (replicated) }

The length of each member of the array of curves will be individual evaluated to determine its color. This demonstrates a replicated in-line conditional assigning the value of a writable property.

**Understanding the differences between Associative and Imperative programming:**

o Associative programming supports graph based dependencies and uses:

replication and replication guides,

modifiers and modifier blocks

in associative programming a program statement not only defines that the value of a variable will be calculated based on references to other variables, but also defines a persistent dependency relationship between the variable whose value is being computed and the references to the other variables.

once a dependencies has been established, a subsequent change to these other variables in successive statements will cause the variable to be recomputed. In single-step debug mode the execution cursor may apparently move backwards through the source code as statements are executed and the value of variables are recomputed based on these dependencies.

in the absence of graph based dependencies, statements are executed in lexical order

o Imperative programming supports explicit ‘flow control’:

iteration with ***for*** and ***while*** loops

conditionals with ***if..else*** statements

in the absence of such flow explicit control statements are executed in lexical order

in imperative programming a program statement defines the value of a variable to be calculated based on references to other variables, but this is a ’one-time’ operation. A subsequent change to these other variables in successive statements does not cause the variable to be recomputed.

forward references are not allowed: a variable cannot be computed from variables which have yet to been defined

**Function Overloading** it is possible to have multiple definitions of the same function with different types of arguments,

for example:

def foo(x : int) = x \* 5; // x as an int def foo(x : double) = x \* 4.0; // x as an double def foo(x : Point) = x.Translate( 6.0, 0, 0); // x as a Point

In addition the type of the return argument can be explicitly defined, for example:

def foo : int (x : int) = x \* 5; // x as an int def foo : double (x : double) = x \* 4.0; // x as an double def foo : Point (x : Point) = x.Translate( 6.0, 0, 0); // x as a Point

The function can be called with different arguments and the appropriate version of the function will automatically be called for arguments of the specific type, but if an argument is provided for which there is no overloaded method, then this will fail. a = foo(10); // a = 50.. with an int: DesignScript will call: def foo:int (x : int) b = foo(10.1); // b = 50.5.. with a double: DesignScript will call: def foo:double(x : double) c = foo(myPoint); // c = myPoint translated : DesignScript will call: def foo:Point(x : Point) d = foo(yourLine); // d = null.. DesignScript will call: def foo(x) which will fail

**Scoping issues with functions**. If we consider the function definition foo, above and within the same script have the statements

y = 1; z = foo(y+10);

then because the variable y is defined both in the outer scope of the function and within the function, the execution of foo will change the value of y in the outer scope. This kind of side effect is potentially a source of errors.

In the case above, a variable was being assigned to within a function and had the same name as a variable in the main script, but whether or not the variable was present in the main script the function would still execute.

**Inheritance**: classes can be defined: by inheritance, by using the extend key word and specifying a ***base\_class\_name***

Classes need not have a specified base class (the class will assume to be derived from the universal base class, var).

In this case a class can be defined by composition, by including existing classes as properties

In this environment, users are permitted to inter-changeably pass both lists and singletons into certain methods for the crea-tion of geometry. This feature, while not supported by most general-purpose programming languages and considered questionable practice by expert pro-grammers, is championed by the authors of DesignScript (Aish, 2012) who argue that such features would be considered desirable by the target user base accustomed to an associative CAD-like approach to collections of ob-jects. In what follows, we describe Decodes, a computational geometry envi-ronment in development that is built upon three principles that the authors consider critical for the specific needs of the target user base of the architec-tural design community.

• Building up from Scratch: a tabula rasa approach that involves developing a set of primitive elements independent of any CAD environment, sidesteps  
DECODES 5  
the associated unpleasantries of working with the complexities of a full-featured CAD environment. The result is a stand-alone environment that is lightweight and focused on foundational concepts to computational design.  
• Broadening the User Base: this environment is written in a non-compiled scripting language able to be interpreted on the most widely used operating systems (windows, OSX, Unix) in architectural practice, thus encouraging an active use culture that transcends the boundaries between software platforms and operating systems.  
• Interoperating: a set of routines for translating geometry in and out of a se-lected set of CAD software facilitates integration into the native scripting en-vironments in such a way that it effectively replaces host-native scripting.

The Decodes kernel accommodates the needs of the computational design community through two mechanisms. The primary mechanism is the natural extensibility of Decodes and the anticipated user community actively con-tributing extensions. As highly valuable extensions are developed and re-fined, they may be integrated into the core library. The secondary mecha-nism is the framework of the core library itself, which was developed through an examination of use-cases specific to architectural design. A con-current thread of research examines situations commonly encountered in ar-chitectural design, and has identified seven broad categories of computation-al strategies that encapsulate tactics that are commonly employed in practice: