DesignScript Documentation (Draft)

1. Introduction

This is the documentation for DesignScript programming language. DesignScript is dynamic, garbage-collected and associative language, and provides strong support for visual programming environment.

The grammar in this specification is in Extended Backus-Naur Form (EBNF) [1].

This document doesn't contain information about APIs and Foreign Function Interface (FFI). The later is implementation dependent.

2. Lexical elements

2.1. Comments

DesignScript supports two kinds of comments.

- Single line comment starts with `//` and stop at the end of the line.
- Block comments starts with `/*` and ends with `*/`.

Example:

```
x = 1; // single line comment

/*
   Block comments
*/
y = 2;
```

2.2. Semicolons

Semicolon ";" is used as a terminator of a statement.

2.3. Identifiers

Identifiers in DesignScript name variables, types, functions and namespaces.

```
Identifier =
    identifier_start_letter { identifier_part_letter }
```

"identifier_start_letter" is the unicode character in the following categories:

- Uppercase letter (Lu)
- Lowercase letter (Ll)
- Titlecase letter (Lt)
- Modifier letter (Lm)
- Other letter (Lo)
- Letter number (Nl)

[1] https://en.wikipedia.org/wiki/Extended_Backus-Naur_Form

“identifier_part_letter” is the unicode character in the categories of “identifier_start_letter” including the following categories:

- Combining letter (Mn, Mc)
- Decimal digital letter (Nd)
- Connecting letter (Nc)
- Zero Width Non-Joiner
- Zero Width Joiner

2.4. Keywords

The following words are reserved as keywords

```
break, class, constructor, continue, def, else, elseif, extends, for, from, if,
import, in, return, static, while,
```

2.5. Bool literal

```
true, false
```

2.6. Integer literal

Integer literal represents an integer constant. It is in decimal base~~, or in hexadecimal base~~.

```
digit = ‘0’..’9’

hex_digit = ‘0’..’9’ + ‘a’..’f’ + ‘A’..’F’

decimal_number = digit { digit }

~~hexadecimal_number = (“0x” | “0X”) hex_digit { hex_digit }~~
```

~~In DesignScript, the range of integer value is -2^{63} – 2^{63} – 1~~.

Example:

```
123;
0xff;  // 255
0XFF;  // 255
```

2.7. Floating-point literal

Floating-point literal represent a floating-point constant in decimal base.

```
floating_point =
    digit { digit } ‘.’ [digit { digit }] [ exponent ]
  | digit { digit } exponent
  | ‘.’ digit { digit } [ exponent ]
```

```
exponent = (‘E’ | ‘e’) [‘+’ | ‘-’] digit { digit }
```

Example:

```
1.2e3;    // 1200.0;
1.234;
.123;     // 0.123
```

2.8. String literal

String literal represents a string constant. It is obtained by putting character sequence between double quote (“).

Any character can be in the sequence except newline and double quote (“). Backslash character in the sequence could be combined with the next character to become an escape character[2]:

- \a
- \b
- \f
- \n
- \t
- \v
- \r

Example:

```
// “Hello        DesignScript
// Language”
“\”Hello\tDesignScript\nLanguage\””;
```

3. Types

3.1. Primitive Types

The type system in DesignScript is dynamic and object-oriented. DesignScript supports following primitive types

Type	Value range	Default value
string	n.a.	“”
integer	-2^{63} - 2^{63} - 1	0

[2] https://en.wikipedia.org/wiki/Escape_character

floating-point	IEEE 754 double-precision[3]	0
bool	true, false	false
null	null	null
var	n.a.	null

The default value of all other types is “null”.

3.2. User defined types

User defined types are supported through class mechanism . Objects, instances of classes, may contain

- Properties
- Instance methods
- Static members
- Constructors

Only single inheritance is allowed in DesignScript.

3.3. List

3.3.1. Rank

If a type has rank suffix, it declares a list. The number of “[]” specifies the number of rank. “[]..[]” specifies arbitrary rank. For example,

```
int[]     // an integer list whose rank is 1
int[][]   // an integer list whose rank is 2
int[]..[] // an integer list with arbitrary rank
```

The rank of type decides how do replication and replication guide work in function dispatch.

3.3.2. Dynamic list

The list in DesignScript is dynamic. It is possible to index into any location of the list. If setting value to an index which is beyond the length of list, list will be automatically expanded. For example,

```
x = {1, 2, 3};
x[5] = 4;      // x = {1, 2, 3, null, null, 4};
```

3.3.3. Use as a dictionary

[3] https://en.wikipedia.org/wiki/Double-precision_floating-point_format

List in DesignScript is just a special case of dictionary whose keys are integers. When indexing a list, the type of key could be any type. For example:

```
x = {1, 2, 3};
x[“foo”] = 4;
x[false] = 5;
```

When a dictionary is used in “in” clause of “for” loop, it returns all values associated with keys.

3.4. Type conversion rules(TBD)

Following implicit type conversion rules specify the result of converting one type to another:

3.4.1. Non-list case

“yes” means convertible, “no” means no convertible.

To From	var	int	double	bool	string	user defined
var	yes	no	no	no	no	no
int	yes	yes	yes	x != 0	no	no
double	yes	warning	yes	x != 0 && x != NaN	no	no
bool	yes	no	no	yes	yes	no
string	yes	no	no	x != “”	yes	no
user defined	yes	no	no	x != null	no	Covariant

3.4.2. Array promotion

To From	var[]	int[]	double[]	bool[]	string[]	user defined[]
var	yes	no	no	no	no	no
int	yes	yes	yes	x != 0	no	no
double	yes	warning	yes	x != 0 && x != NaN	no	no
bool	yes	no	no	yes	yes	no
string	yes	no	no	x != “”	yes	no

user defined	yes	no	no	x != null	no	Covariant

3.4.3. Array demotion

To From	var	int	double	bool	string	user defined
var[]	yes	no	no	no	no	no
int[]	yes	yes	yes	x != 0	no	no
double[]	yes	warning	yes	x != 0 && x != NaN	no	no
bool[]	yes	no	no	yes	yes	no
string[]	yes	no	no	x != “”	yes	no
user defined[]	yes	no	no	x != null	no	Covariant

4. Variables

4.1. Dynamic

Variables in DesignScript are dynamic. It is free to assign any kinds of objects to any variable, and the type of a variable is totally run-time dependent.

4.2. List Immutability

Lists in DesignScript are immutable. That is, when copying a list from one variable to the other variable, it is deep copy operation: all elements in the list are copied as well.

4.3. Scope

DesignScript uses block scope[4], and blocks are either functions or language blocks.Because of associativity, a variable could be used before it is defined, the DesignScript virtual machine will ensure to propagate the value update to all its references.

Example:

```
x = 1;
y = 2;

def foo(x) {
```

[4] https://en.wikipedia.org/wiki/Scope_(computer_science)#Block_scope

```
    z = 3;          // “z” is local variable
    return = x + y; // “x” is parameter
                    // “y” is defined in the global associative language block
}

[Imperative] {
    x = 3;          // update the global “x”
    m = n;          // “m”, “n” are local variables. It is fine to use “n”
    n = 4;          // the VM ensures “m” finally is 4
    return = x + y + m;
}
```

4.4. Scope resolution

The search order of an identifier is

- Innermost scope.
- Each progressive outer scope, including class scope.
- Classes that the current class extends.
- The global scope.

5. Declarations

5.1. Function declaration

```
FunctionDeclaration =
    “def” identifier [“:” TypeSpecification] ParameterList StatementBlock.

ParameterList =
    “(“ [Parameter {“,” Parameter}] “)”

Parameter =
    identifier [“:” TypeSpecification] [DefaultArgument]

StatementBlock =
    “{“ { Statement } “}”
```

The return type of function is optional. By default the return type is `var`. If the return type is specified, type conversion may happen. It is not encouraged to specify return type unless it is necessary.

Function must be defined in the global associative language block.

For parameters, if their types are not specified, by default is `var`. The type of parameters should be carefully specified so that replication and replication guide will work as desired. For example, if a parameter’s type is `var[]..[]` (arbitrary rank), it means no replication on this parameter.

Example:

```
def foo:var(x:int[]..[], y:int = 3)
{
    return = x + y;
}
```

5.1.1. Default parameters

Function declaration allows to have default parameter, but with one restriction: all default parameters should be the rightmost parameters.

For example:

```
// it is valid because “y” and “z” are the rightmost default parameters
def foo(x, y = 1, z = 2)
{
    return = x + y + z;
}

// it is invalid because “x” is not the rightmost default parameter
def bar(x = 1, y, z = 2)
{
    return = x + y + z;
}
```

5.1.2. Function overloads

DesignScript supports function overload, i.e., functions with a same name but with different types/number of parameters, but which function will be called finally is totally run-time dependent,, especially if replication happens. DesignScript virtual machine will try to find out the best match one based on the type of arguments and the type of all parameters of all function candidates.

Following code shows a function overload example:

```
def foo(x: int, y:int)
{
    return = x + y;
}

def foo(x: double, y: double)
{
    return = x * y;
}

// will call foo(x:int, y:int)
r1 = foo(2, 3);

// will call foo(x:double, y:double)
r2 = foo(2.1, 3.2);
```

5.2. Class declaration

```
ClassDeclaration =
    “class” identifer [“extends” identifier]
    “{“ [ClassMemberDeclaration] “}”

ClasMemberDeclaration =
    ConstructorDeclaration
  | MemberDeclaration

ConstructorDeclaration =
    “constructor” identifier ParameterList
    [“:” “base” “(“ ArgumentList “)”]
    StatementBlock

MemberDeclaration =
    [AccessModifier] [static] (FunctionDeclaration | VariableDeclaration “;”)

AccessMmodifier = “public” | “protected” | “private”
```

Class should be defined in the top associative language block.

5.2.1. Inheritance

DesignScript only supports single inheritance. That is, only one base class is allowed.

Example:

```
class Point2D
{
    ...
}

class Point3D: extends Point2D
{
    ...
}
```

5.2.2. Constructors

Although it is not forced, but it is suggested that the name of constructor starts with “`By...`” to indicate how to create the object. For example, the constructor of creating a `Point` may be “`ByCoordinates()`”.

If the class inherits from the other class, it is possible to call the base class's constructor by calling “`base(...)`”.

To create an instance of class, use object creation statement `ClassName.ConstructorName(...)`. If the name of constructor is the same as class name, or a class is without constructor (in this case a

default constructor will be created internally), in both cases object creation statement could be simplified as `ClassName(...)`.

Example:

```
class Point2D
{
    constructor ByCoordinates(x, y)
    {
        ...
    }
}

class Point3D: extends Point2D
{
    constructor ByCoordinates(x, y, z) : base(x, y)
    {
        ...
    }

    constructor Point3D()
    {
        ...
    }
}

// create an instance of Point3D
p1 = Point3D.ByCoordinates(1, 2, 3);

// create the other instance of Point3D
p2 = Point3D();
```

5.2.3. Properties

Variables defined in class are properties. The initialization of properties could either be in constructor or in definition.

To distinguish properties from other kinds of variables, member access expression "`this.property_name`" or "'`base.property_name`" could be used to indicate "`property_name`" is a property of the class or a property of base class.

Extending previous `Point2D` and `Point3D` example:

```
class Point2D
{
    x = 0;   // initialize property "x"
    y = 0;

    constructor ByCoordinates(x, y)
```

```
    {
        this.x = x;
        this.y = y;
    }
}

class Point3D: extends Point2D
{
    z = 0;

    constructor ByCoordinates(x, y, z) : base(x, y)
    {
        this.z = z;
    }

    constructor Point3D()
    {
        x = 1;       // it is property “x” of base class
        base.y = 1;  // it is property “y” of base class
        z = 1;
    }
}

// create the other instance of Point3D
p1 = Point3D();

// x == 1
x = p1.x;

// update property “y”
p1.y = 2;
```

5.2.4. Member functions

The definition of member functions is the same as normal function definition. All properties are accessible in member functions.

5.2.5. Access modifiers

The access restriction to class properties and member functions are specified by labels `public`, `protected` and `private`. If no access specifier is specified for members, by default they are public.

The following rules apply:

- Public members are accessible from anywhere inside and outside the class.
- Protected members are only accessible in the class or in the derived class.
- Private members are only accessible in the class.

Example:

```
class Base
{
    private prop1;
    protected prop2;
    public prop3;

    private def foo1()
    {...}

    protected def foo2()
    {...}

    // by default it is public
    def foo3()
    {
    }
}

class Derived: extends Base
{
    def bar()
    {
        // as "prop1" is private in base, it is not accessible
        x = prop1;

        // "prop2" is accessible in derived class
        y = prop2;

        // "prop3" is accessible in derived class
        z = prop3;
    }
}

b = Base();

// as "prop1" is private, it is not accessible outside the class
// p1 is null
p1 = b.prop1;

// as "prop2" is protected, it is not accessible outside the class
// p2 is null
p2 = b.prop2;

// as "prop3" is public, it is accessible outside the class
p3 = b.prop3;
```

5.2.6. Static members

Static properties and static member functions are defined on class level. That is, the access expression is in the form of "`ClassName.X`" or "`ClassName.Y()`'

There are some rules:

- It is invalid to access static members on instance level.
- It is invalid if a static property has the same name as a non-static property, or a static member function has the same signature as a non-static member function.
- It is not necessary to have "`ClassName`." prefix when calling static members in the class.

The example belows shows these rules:

```
class Foo
{
    x;

    // error: x has been defined as a non-static property
    static x;

    static Y;

    def foo()
    {
    }

    // error: foo() has been defined as a non-static member function
    static def foo()
    {
    }

    static def bar()
    {
        // error: it is invalid to call static member function in non-static
        // member function
        qux();
    }

    def qux()
    {
        // it is fine to call static member function in non-static member
        // function
        bar();
    }
}

f = Foo();

// error: it is invalid to access static members on instance level
r = f.Y;

// get the value of static member "Y"
r = Foo.Y;

// error: it is invalid to access static members on instance level
r = f.bar();
```

```
// call static member function "bar()"
r = Foo.bar();
```

5.2.7. _Dispose() method

If a public `_Dispose()` method is defined in the class, when the instance is garbage collected, this method will be called, so `_Dispose()` could be used to release resources acquired by instance. Typically, this method will be generated automatically for FFI classes.

Example:

```
class Diposable
{
    static Flag = false;

    def _Dispose()
    {
        Flag = true;
    }
}

[Imperative]
{
    d = Disposable();

    // builtin function to force garbage collection.

    __GC();
    // "d" will be garbage collected
}

// Disposable.Flag is true
r = Disposable.Flag;
```

5.3. Function resolution

The order of resolve a function is:

- Innermost scope
- Each progressive outer scope
- Class instance scope
- Class that the class extends
- The global scope

As function could be overloaded, the result of function resolution will return a list of function candidates. The virtual machine will find out the best match depending on the type of arguments, the types of parameters and replication guide.

6. Expressions

6.1. List creation expression

```
ListCreationExpression = “{“ [Expression { “,” Expression } ] “}”
```

List creation expression is to create a list. Example:

```
x = {{1, 2, 3}, null, {true, false}, “DesignScript”};
```

6.2. Range expression

Range expression is a convenient way to generate a list.

```
RangeExpression =
     Expression [“..” [“#”] Expression [“..” [“#”] Expression]
```

There are three forms of range expressions:

- `start_value..end_value`
  - If `start_value` < `end_value`, it generates a list in ascendant order which starts at `start_value`, and the last value < `end_value`, the increment is 1
  - If `start_value` > `end_value`, it generates a list in descendent order which starts at `start_value` and the last value >= `end_value`, the increment is -1.
- `start_value..#number_of_elements..increment:` it generates a list that contains `number_of_elements` elements. Element starts at `start_value` and the increment is increment.
- `start_value..end_value..#number_of_elements`: it generates a list that contains `number_of_elements` elements. Element starts at start_value and ends at end_value. Depending on if `start_value` <= `end_value`, the generated list will be in ascendant order or in descendent order.

Example:

```
1..5;       // {1, 2, 3, 4, 5}

5..1;       // {5, 4, 3, 2, 1}

1.2 .. 5.1; // {1.2, 2.2, 3.2, 4.2}

5.1 .. 1.2; // {5.1, 4.1, 3.1, 2.1}
```

```
1..#5..2;   // {1, 3, 5, 7, 9}

1..5..#3;   // {1, 3, 5}
```

Range expression is handled specially for strings with single character. For example, following range expressions are valid as well:

```
“a”..”e”;     // {“a”, “b”, “c”, “d”, “e”}

“a”..#4..2;   // {“a”, “c”, “e”, “g”}

“a”..”g”..#3; // {“a”, “d”, “g”}
```

6.3. Inline conditional expression

```
InlineConditionalExpression = Expression ? Expression : Expression;
```

The first expression in inline conditional expression is condition expression whose type is bool. If it is true, the value of “?” clause will be returned; otherwise the value of “:” clause will be returned. The types of expressions for true and false conditions are not necessary to be the same. Example:

```
x = 2;
y = (x % 2 == 0) ? “foo” : 21;
```

Inline conditional expressions replicate on all three expressions. Example:

```
x = {true, false, true};
y = {“foo”, “bar”, “qux”};
z = {“ding”, “dang”, “dong”};

r = x ? y : z;  // replicates, r = {“foo”, “dang”, “qux”}
```

6.4. Member access expression

Member access expression is of the form

```
x.y.z
```

“y” and “z” could be properties, or member functions. If they are not accessible, `null` will be returned.

6.5. List access expression

List access expression is of the form

```
a[x]
```

“x” could be integer value or a key of any kind of types (if “a” is a dictionary).

The following rules apply:

- If it is just getting value, if “a” is not a list, or the length of “a” is less than “x”, or the rank of “a” is less than the number of indexer, for example the rank of “a” is 2 but the expression is `a[x][y][z]`, there will be a “IndexOutOfRange” warning and `null` will be returned.
- If it is assigning a value to the list, if “a” is not a list, or the length of “a” is less than “x”, or the rank of “a” is less than the number of indexer, “a” will be extended or its dimension will promoted so that it is able to accommodate the value. For example,

```
a = 1;
a[1] = 2;      // “a” will be promoted, a = {1, 2} now
a[3] = 3;      // “a” will be extended, a = {1, 2, null, 3} now
a[0][1] = 3;   // “a” will be promoted, a = {{1, 3}, 2, null, 3} now
```

6.6. Operators

The following operators are supported in DesignScript:

```
+         Arithmetic addition
-         Arithmetic subtraction
*         Arithmetic multiplication
/         Arithmetic division
%         Arithmetic mod
>         Comparison large
>=        Comparison large than
<         Comparison less
<=        Comparison less than
==        Comparison equal
!=        Comparison not equal
&&        Logical and
||        Logical or
!         Logical not
-         Negate
```

All operators support replication. Except unary operator “!”, all other operators also support replication guide. That is, the operands could be appended replication guides.

```
x = {1, 2, 3};
y = {4, 5, 6};
r1 = x + y;       // replication
                  // r1 = {5, 7, 9}
```

```
r2 = x<1> + y<2>; // replication guide
                  // r2 = {
                  //          {5, 6, 7},
                  //          {6, 7, 8},
                  //          {7, 8, 9}
                  //      }
```

Operator precedence

Precedence	Operator
6	-
5	*, /, %
4	+, -
3	>, >=, <, <=, ==, !=
2	&&
1	||

6.7. Arithmetic operators

```
+, -, *, /, %
```

Normally the operands are either integer value or floating-point value. “+” can be used as string concatenation:

```
s1 = “Design”;
s2 = “Script”;
s = s1 + s2;  // “DesignScript”
```

6.8. Comparison operators

```
>, >=, <, <=, ==, !=
```

6.9. Logical operators

```
&&, ||, !
```

The operand should be bool type; otherwise type conversion will be incurred.

7. Statements

7.1. Empty statements

Empty statement is

```
;
```

7.2. Import statements

Import statements import other DesignScript source file or C# assembly into current namespace.

```
ImportStatement = “import” “(“ (string | (ident from string))“)”
```

If importing a C# assembly, DesignScript virtual machine will generate DesignScript classes for classes defined in the assembly, this is done by FFI.

Import statements could import all the entities found at the location, or for specific named-entity found at the location.

The location may be specified by:

- A relative file path, using local operating system conventions.
- An absolute file path, using local operating system conventions.
- A URI.

Example:

```
import (“/home/dev/libraries/foo.ds”);
import (Point from “Geometry.dll”);
```

7.3. Expression statements

```
ExpressionStatement = Expression “;”
```

Expression statements are expressions without assignment.

7.4. Assignments

```
Assignment = Expression “=” ((Expression “;”) | LanguageBlock)
```

The left hand side of “=” should be assignable. Typically, it is member access expression or array access expression or variable. If the left hand side is a variable which hasn’t been defined before, the assignment statement will define this variable.

7.5. Flow statements

Flow statements change the execution flow of the program. A flow statement is one of the followings:

1. A return statement.
2. A break statement in the block of for or while statement in imperative language block.
3. A continue statement in the block of for or while statement in imperative language block.

7.6. Return statements

```
ReturnStatement = “return” “=” Expression “;”
```

A “return” statement terminates the execution of the innermost function and returns to its caller, or terminates the innermost imperative language block, and returns to the upper-level language block or function.

7.7. Break statements

```
BreakStatement = “break” “;”
```

A “break” statement terminates the execution of the innermost “for” loop or “while” loop.

7.8. Continue statement

```
ContinueStatement = “continue” “;”
```

A “continue” statement begins the next iteration of the innermost “for” loop or “while” loop.

7.9. If statements

“if” statements specify the conditional execution of multiple branches based on the boolean value of each conditional expression. “if” statements are only valid in imperative language block.

```
IfStatement =
    “if” “(” Expression “)” StatementBlock
    { “elseif” “(” Expression “)” StatementBlock }
    [ “else” StatementBlock ]
```

For example:

```
x = 5;
if (x > 10) {
    y = 1;
}
elseif (x > 5) {
```

```
    y = 2;
}
elseif (x > 0) {
    y = 3;
}
else {
    y = 4;
}
```

7.10. While statements

“while” statements repeatedly executes a block until the condition becomes false. “while” statements are only valid in imperative language block.

```
WhileStatement = “while” “(” Expression “)” StatementBlock
```

Example:

```
sum = 0;
x = 0;
while (x < 10)
{
    sum = sum + x;
    x = x + 1;
}
// sum == 55
```

7.11. For statements

“for” iterates all values in “in” clause and assigns the value to the loop variable. The expression in “in” clause should return a list; if it is a singleton, it is a single statement evaluation. “for” statements are only valid in imperative language block.

```
ForStatement = “for” “(” Identifier “in” Expression “)” StatementBlock
```

Example:

```
sum = 0;
for (x in 1..10)
{
    sum = sum + x;
}
// sum == 55
```

8. Language blocks

8.1. Default associative language block

By default, all statements are in a default top associative language block, so associative update is enabled by default.

Not like nested language block, there is no return statement in top language block: all statements will be executed sequentially to the last one.

8.2. Nested associative language block

It is also valid to explicitly define a nested associative language block in the top associative language block, in an imperative language block or in a function.

```
AssociativeLanguageBlock = “[“ “Associative” “]” StatementBlock
```

The statements in associative language block will be executed sequentially until a return statement returns a value; otherwise the language block returns `null`.

Note it is invalid to define an associative language block inside an associative language block, except the later is the default one.

Examples:

```
x = 1;

// z = 3
z = [Associative]
{
    y = 2;
    return = x + y;
}

[Associative]
{
    // invalid associative language block
    [Associative]
    {
    }
}
```

8.3. Imperative language block

Imperative language block provides a convenient way to use imperative semantics. Similar to nested associative language block, imperative language block executes all statements sequentially unless a

statement is a return statement to return a value. Imperative language block can only be defined in the other associative language block, including the top associative language block.

The key differences between associative language block and imperative language block are:

- Associative update is temporarily disabled in imperative language block.
- “if”, “for” and “while” statements are only available in imperative language block.

Example:

```
x = 1;

// define an imperative language block in the top associative language block
y = [Imperative]
{
    if (x > 10) {
        return = 3;
    }
    else if (x > 5) {
        return = 2;
    }
    else {
        return = 1;
    }
}

def sum(x)
{
    // define an imperative language block inside a function, which is in global
    // associative language block
    return = [Imperative]
    {
        s = 0;
        for (i in 1..x)
        {
            s = s + i;
        }
        return = s;
    }
}

[Associative]
{
    // define an imperative language block inside an associative language block
    return = [Imperative]
    {
        return = 42;
    }
}

[Imperative]
```

```
{
    // invalid imperative language block
    [Imperative]
    {
    }
}
```

9. Associative update

9.1. Associativity

Associativity is a feature in DesignScript that allows the propagation of change of a variable to all dependent statements in the program. The propagation of change happens at runtime and is handled by update mechanism in DesignScript virtual machine.

Following code shows a simple associative update case:

```
1: x = 1;
2: y = x;
3: x = 3;
```

If the code is in associative language block, it will be executed as follow:

1. "x" is set to 1.
2. "y" is set to "x" and depends on "x", so "y" is 1.
3. "x" is changed to 3. As "y" depends on "x", "y" will be updated to 3.

At line 2, "y" depends on "x", or "y" is a dependent of "x"; then on line 3, the change of "x" will update "y" through re-executing the corresponding assignment statement that establishes associativity.

On the other hand, if the code is in imperative language block, the change of "x" will not update "y", just like other programming languages.

There are four key elements in associativity definition and update:

1. The establishment of associativity is done by assignment statement.
2. The update is done by re-executing assignment statement.
3. The dependent (on the left hand side of assignment) could be
    - Variables
    - Properties
4. Entities that could be depended on are
    - Variables
    - Properties
    - Functions

9.2. Associativity establishment

Assignment statement in associative language block establishes a relationship between the assigned entities and the entities used in the statement to compute it. Re-redefinition will remove previous associativity (if there is) and redefine associativity.

It is important to note that the assigned entities and the entities that are depended on should be defined in the same scope. See associativity scope.

An example in associative language block

```
1: x = 1;
2: y = 2;
3: z = foo(x, y);
4: x = 3;
5: y = 4;
6: z = 0;
7: x = 5;
```

At line 3, the assignment establishes the associativity between "z" and "foo", "x" and "y", so "z" will depend on "foo", "x" and "y". After updating "x" to 3, line 3 will be re-executed to update "z"; after updating "y" to 4 at line 5, line 3 will be re-executed again.

At line 6, the redefinition of "z" remove previously established associativity. Now "z" no longer depends on other entities, so the following update of "x" at line 7 will not incur re-computation of "z".

Assignment statement in imperative language block changes only variables/properties they are assigned to and has no implicit update.

An example in imperative language block

```
[Imperative]
{
1: x = 1;
2: y = x;
3: x = 3;
}
```

The code is executed line by line, no re-computation will happen in imperative language block. After execution, "x" is 3 and "y" is 1.

9.3. Update by re-execution

The change is propagated to all dependents recursively by re-executing all statements that establish associativity.

9.4. Dependents

A dependent could be a variable or a property. Code below shows the later case.

```
class Foo
{
    x;
}

f = Foo();
m = 21;

// property “x” depends on “m”
f.x = m;

m = 42;

// r is 42
r = f.x;
```

After “m” is updated to 42, assignment statement “`f.x = m`” will be re-executed to update property “x”.

9.5. Entities that could be depended on

9.5.1. Variables

The change of variable will trigger updates of its dependents. Variable could appears in

- Expression
- Parameters: in the form of “`x = f(y, z)`”, where “x” depends on “`f()`” and variables “y” and “z”.
- List indexing: in the form of “`x = y[z]`”, where “x” depends on both variables “y” and array indexer variable “z”.

There is a special case for self-redefinition, i.e., a variable depends on itself. For example, “`a = a + 1`”. This kind of self-redefinition will not remove previously established associativity between “a” and other variables. Besides, the change of “a” will only update its dependents, not “a” itself. Following example illustrates this case:

```
1: x = 1;
2: y = 2;
3: x = x + y;
4: z = x + y;
```

```
5: x = x + 1;
```

The code above will be executed as follow:

1. “x” is set to 1.
2. “y” is set to 2.
3. “x” is changed to “x + y”. So “x” depends on “y”, “x” is 3 now.
4. “z” is set to “x + y”. So “z” depends on “x” and “y”, “z” is 3 now.
5. “x” is increased by 1. The update for “z” will be triggered, “z” is 4 now.

It is also fine that variable appears in function parameter list, just as the example in associativity establishment shows.

9.5.2. Function

In the form of

```
x = f(...);
```

The assignment establishes associativity between “x” and function “f()”. Any update of function body of any overloaded “f()” will cause the re-execution of this statement. Note this feature is only available when live runner is enabled (which is only available in visual programming environment).

9.5.3. Properties

In the form of

```
r = x.y.z;
```

The assignment establishes associativity between “r” and variable “x” and properties “y” and “z”. Any update for “x”, “y” and “z” will update “r”.

Example:

```
class Foo
{
    x;
    constructor Foo(_x)
    {
        x = _x;
    }
}

class Bar
{
    foo;
```

```
    constructor Bar(_x)
    {
        f = Foo(_x);
    }
}

b = Bar(0);
t = b.foo.x;

b.foo.x = 1;   // update “x”
b.f = Foo(2);  // update “f”
b = Bar(3);    // update “b”
```

Each update in the last three statement will re-execute statement “`t = b.foo.x`” to update the value of “`t`”.

9.6. Associativity scope

As the establishment of associativity and update are both at runtime, it is important to note that the associativity establishment and update should only apply to entities in the same scope; otherwise the result is undefined. If there already is an associativity, the associativity may be removed.

There are four undefined cases:

1. The dependent is defined in upper scope and depends on variables defined in the same scope, but associativity is established in function. For example, in below code, “x” and “y” both defined in global associative language block. The execution of “foo()” will execute line 5 to establish associativity between “x” and “y”. The expected behavior of this kind of associativity is undefined, therefore updating “y” to 3 will not necessarily update “x” as the DesignScript virtual machine doesn’t support to just execute line 5 without executing the whole function “`foo()`”.

```
 1: x = 1;
 2: y = 2;
 3: def foo()
 4: {
 5:    x = y;
 6:    return = null;
 7: }
 8:
 9: foo();
10: y = 3;  // the update for “x” is undefined
```

2. The dependent is defined in upper scope and depends on variables defined in the same scope, but associativity is established in nested language block. For example, in below code, “x” and “y” both defined in global associative language block. The execution of nested imperative language will establish associativity between “x” and “y” at line 5, but the expected behavior of this kind of associativity is undefined as well.

```
1: x = 1;
2: y = 2;
3: [Imperative]
4: {
5:    x = y;
6:    return = null;
7: }
8: y = 3;  // the update for “x” is undefined
```

3. The dependent is defined in upper scope but depends on variables defined in function. For example, “x” depends on local variable “y” in “foo()”.

```
1: x = 1;
2: def foo()
3: {
4:    y = 2;
5:    x = y;
6:    return = null;
7: }
8: foo();
9: y = 3;
```

4. The dependent is defined in upper scope but depends on variables defined in nested language block. For example, “x” depends on local variable “y” in nested imperative language block.

```
1: x = 1;
2: [Imperative]
3: {
4:    y = 2;
5:    x = y;
6:    return = null;
7: }
```

10. Replication and replication guide

10.1. Replication and replication guide

Replication is a way to express iteration in associative language block. It applies to a function call when the rank of input arguments exceeds the rank of parameters. In other words, a function may be called multiple times in replication, and the return value from each function call will be aggregated and returned as a list.

There are two kinds of replication:

- Zip replication: for multiple input lists, zip replication is about taking every element from each list at the same position and calling the function; the return value from each function call is

aggregated and returned as a list. For example, for input arguments `{x1, x2, ..., xn}` and `{y1, y2, ..., yn}`, when calling function `f()` with zip replication, it is equivalent to `{f(x1, y1}, f(x2, y2), ..., f(xn, yn)}`. As the lengths of input arguments could be different, zip replication could be

- Shortest zip replication: use the shorter length.
- Longest zip replication: use the longest length, the last element in the short input list will be repeated.

The default zip replication is the shortest zip replication; otherwise need to use replication guide to specify the longest approach.

- Cartesian replication: it is equivalent to nested loop in imperative language. For example, for input arguments `{x1, x2, ..., xn}` and `{y1, y2, ..., yn}`, when calling function `f()` with cartesian replication and the cartesian indices are `{0, 1}`, which means the iteration over the first argument is the first loop, and the iteration over the second argument is the nested loop; it is equivalent to `{f(x1, y1}, f(x1, y2), ..., f(x1, yn}, f(x2, y1), f(x2, y2), ..., f(x2, yn), ..., f(xn, y1), f(xn, y2), ..., f(xn, yn)}`.

Replication guide is used to specify the order of cartesian replication indices; the lower replication guide, the outer loop. If two replication guides are the same value, zip replication will be applied.

```
ReplicationGuide = “<” number [“L”] “>” {“<” number [“L”] “>”}
```

Only integer value is allowed in replication guide. Postfix “`L`” denotes if the replication is zip replication, then use the longest zip replication strategy. The number of replication guide specifies the nested level. For example, replication guide `xs<1><2>` indicates the argument should be at least of 2 dimensional and its nested level is 2; it could also be expressed by the following pseudo code:

```
// xs<1><2>
for (ys in xs)
{
    for (y in ys)
    {
        ...
    }
}
```

Example:

```
def add(x: var, y: var)
{
    return = x + y;
}
```

```
xs = {1, 2};
ys = {3, 4};
zs = {5, 6, 7};

// use zip replication
// r1 = {4, 6}
r1 = add(xs, ys);

// use the shortest zip replication
// r2 = {6, 8};
r2 = add(xs, zs);

// the longest zip replication should be specified through replication guide.
// the application guides should be the same value; otherwise cartesian
// replication will be applied
// r3 = {6, 8, 9}
r3 = add(xs<1L>, zs<1L>);

// use cartesian replication
// r4 = {{4, 5}, {5, 6}};
r4 = add(xs<1>, ys<2>);

// use cartesian replication
// r5 = {{4, 5}, {5, 6}};
r5 = add(xs<2>, ys<1>);
```

Besides replication for explicit function call, replication and replication guide could also be applied to the following expressions:

1. Binary operators like +, -, *, / and so on. All binary operators in DesignScript can be viewed as a function which accepts two parameters, and unary operator can be viewed as a function which accepts one parameters. Therefore, replication will apply to expression "`xs + ys`" if "`xs`" and "`ys`" are lists.
2. Range expression.
3. Inline conditional expression in the form of "`xs ? ys : zs`" where "`xs`", "`ys`" and "`zs`" are lists.
4. Array indexing. For example, `xs[ys]` where `ys` is a list. Replication could apply to array indexing on the both sides of assignment expression. Note replication does not apply to multiple indices.
5. Member access expression. For example, `xs.foo(ys)` where `xs` and `ys` are lists. Replication guide could be applied to objects and arguments. If `xs` is a list, `xs` should be a homogeneous list, i.e., all elements in `xs` are of the same type.

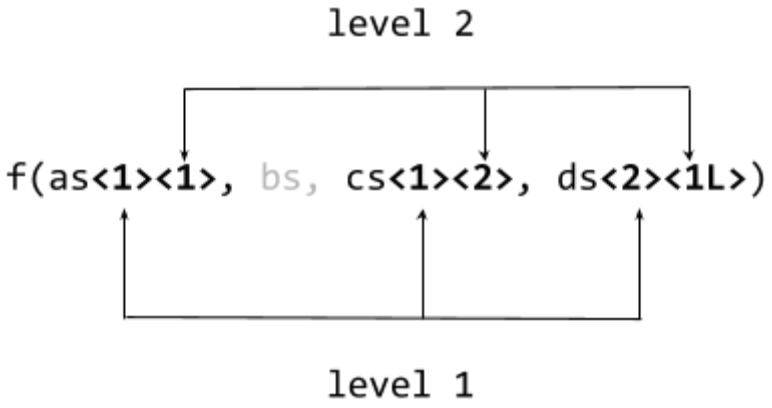
10.2. Function dispatch rule for replication and replication guide

Using zip replication or cartesian replication totally depends on the specified replication guide, the types of input arguments and the types of parameters. Because the input argument could be a heterogenous list, the implementation will compute which replication combination will generate the shortest type conversion distance.

Note if argument is jagged list, the replication result is undefined.

Formally, for a function "`f(x1: t1, x2: t2, ..., xn: tn)`" and input arguments "`a1, a2, ..., an`", function dispatch rule is:

1. Get a list of overloaded function `f()` with same number of parameters. These functions are candidates.
2. If there are replication guides, they will be processed level by level. For example, for function call `f(as<1><1>, bs, cs<1><2>, ds<2><1L>)`, there are two levels of replication guides.



   a. For each level, sort replication guides in ascendant order. If replication guide is less than or equal to 0, this replication guide will be skipped (it is a stub replication guide).
      i. For each replication guide, if it appears in multiple arguments, zip replication applies. By default using shortest lacing. If any replication guide number has suffix "L", longest lacing applies.
      ii. Otherwise cartesian replication applies.
      iii. Repeat step b until all replication guides have been processed.
   b. Repeat step a until all replication levels have been processed.
   c. For this example, following replications will be generated:
      i. Zip replication on `as, cs`
      ii. Cartesian replication on `ds`
      iii. Zip replication on `as, ds`
      iv. Cartesian replication on `ds`
3. After the processing of replication guide, the rank of each input argument is computed: `r1 = rank(a1), r2 = rank(a2), ..., rn = rank(an)`; for each rank, update it to `r = r - <number of replication guide on argument>`. The final list `{r1, r2, ..., rn}` is called a reduction list, each reduction value represents a possible maximum nested loop on the corresponding argument.
4. Based on this reduction list, compute a combination of reduction list whose element value is less than or equal to the corresponding reduction value in base reduction list. For each

reduction list {r1, r2, ..., rn}, iteratively do the following computation to generate replications:

   a. For any ri > 0, ri = ri - 1. If there are multiple reductions whose values are larger than or equal to 1, zip replication applies; otherwise cartesian replication applies.

5. Combine the replications generated on step 3 and step 4, based on the input arguments and the signature of candidate functions, choose the best matched function and best replication strategy. During the process, if the type of parameter and the type of argument are different, the type distance score will be calculated.

11. Built-in functions

- `AllFalse:bool(list: var[]..[])`
  Checks if all elements in the specified list are false.
- `AllTrue:bool(list: var[]..[])`
  Checks if all elements in the specified list are true.
- `Average:double(list: int[]..[])`
  Returns average value of all elements in the specified list.
- `Break()`
  Notifies debugger to break at the point.
- `Concat:var[]..[](list1: var[]..[], list2: var[]..[])`
  Concats list1 and list2 and returns a new list.
- `Contains:bool(list: var[]..[], element: var)`
  Checks if the specified element is in the specified list.
- `Contains:bool(list: var[]..[], element: var[]..[])`
  Checks if the specified element is in the specified list.
- `ContainsKey:bool(list: var[]..[], key: var)`
  Checks if the specified key is present in the specified dictionary.
- `Count:int(list: var[]..[])`
  Returns the number of elements in the specified list.
- `CountTrue:int(list: var[]..[])`
  Returns the number of true values in the specified list.
- `CountFalse:int(list: var[]..[])`
  Returns the number of falsevalues in the specified list.
- `Equals:bool(objectA: var, objectB: var)`
  Determines whether two object instances are equal.
- `Evaluate:var[]..[](fptr: fptr, params: var[]..[], unpack: bool)`
  For internal use. Evaluates a function pointer with specified params.
- `Flatten:var[](list:var[]..[])`
  Returns the flattened 1D list of the multi-dimensional input list.
- `GetElapsedTime:int()`
  Returns elapsed milliseconds in the virtual machine
- `GetKeys:var[]..[](list: var[]..[])`
  Gets all keys from the specified dictionary.
- `GetValues:var[]..[](list: var[]..[])`

Gets all values stored in the specified dictionary and for a simple list it returns all elements.

- `IndexOf:int(list: var[]..[], element: var[]..[])`
  Returns the index of the member in the list.
- `Insert:var[]..[](list: var[]..[], element: var, index: int)`
  Inserts an element into a list at specified index.
- `Insert:var[]..[](list: var[]..[], element: var[]..[], index: int)k`
  Inserts an element into a list at specified index.
- `IsRectangular: bool(list: var[]..[])`
  Checks if each of rows in multidimensional list has the same number of elements.
- `ImportFromCSV:double[][](filePath: string )`
  Imports data from a text file containing comma separated values into two-dimensional list.
- `ImportFromCSV:double[][](filePath: string, transpose:bool)`
  Imports data from a text file containing comma separated values into two-dimensional list and also transpose the output list if specified.
- `IsHomogeneous: bool(list: var[]..[])`
  Checks if all the elements in the specified list are of the same type.
- `IsUniformDepth:bool(list: var[]..[])`
  Checks if the list has a uniform depth.
- `Map:double(rangeMin: double, rangeMax: double, inputValue: double)`
  Maps a value into an input range.
- `MapTo:double(rangeMin: double, rangeMax: double, inputValue: double, targetRangeMin: double, targetRangeMax:double)`
  Maps a value from one range to another range.
- `NormalizeDepth:var[]..[](list: var[]..[])`
  Returns a list with uniform depth as specified by the input depth.
- `NormalizeDepth:var[]..[](list: var[]..[], rank: var)`
  Return multidimensional list according to the rank given.
- `Print(msg: var)`
  Print msg to the console.
- `Rank(list: var[]..[])`
  Counts the maximal rank of the specified list.
- `Remove:var(list: var[]..[], index: int)`
  Removes element at the specified index of the list.
- `RemoveDuplicates:var[]..[](list: var[]..[])`
  Removes duplicate elements in the specified list.
- `RemoveNulls:var[]..[](list: var[]..[])`
  Removes null elements from the specified list.
- `RemoveIfNot:var[]..[](list: var[]..[], type:string)`
  Removes the members of the list which are not members of the specified type.
- `RemoveKey:bool(list:var[]..[], key: var)`
  Returns true if the specified key is removed from the specified list; otherwise returns false.
- `Reorder:var[](list: var[], indice:var[])`
  Reorders the list using the specified indices.

- `Reverse:var[]..[](list: var[]..[])`
  Reverses the specified list.
- `SetDifference:var[](list1: var[], list2: var[])`
  Returns objects that are included in list1 but not excluded in list2
- `SetIntersection:var[](list1: var[], list2: var[])`
  Produces the set intersection of two lists.
- SetUnion:var[](list1: var[], list2: var[])
  Produces the set union of two sequences by using the default equality comparer.
- `Sleep(x: int)`
  Put the virtual machine to sleep for x milliseconds.
- `SomeFalse:bool(list: var[]..[])`
  Returns true if any element in the list is false
- `SomeNulls:bool(list: var[]..[])`
  Returns true if any element in the list is null.
- `SomeTrue:bool(list: var[]..[])`
  Returns true if any element in the list is true.
- `Sort:int[](list: int[])`
  Obsolete.
- `SortIndexByValue:int[](list: double[])`
  Sorts a specified list by values of its members in ascending order.
- `SortIndexByValue:int[](list: double[], ascending: bool)`
  Sorts a specified list by values of its members in either descending or ascending order.
  Sum:int(list: int[]..[])
  Returns the sum of all elements in the specified list.
- `ToString:string(object: var[]..[])`
  Obsolete. Use `__ToStringFromObject()` or `__ToStringFromArray()` instead. Returns object in string representation.
- `Transpose:var[]..[](list: var[]..[])`
  Swaps rows and columns in a list of lists. If there are some rows that are shorter than others, null values are inserted as placeholders in the result list such that it is always rectangular.
- `__GC()`
  Force garbage collection.
- `__ToStringFromObject:string(object: var)`
  Returns object in string representation.
- `__ToStringFromArray:string(list: var[])`
  Returns list in string representation.
- `__TryGetValueFromNestedDictionaries:var[]..[](list: var[]..[], key: var[]..[])`
  Recursively iterate all dictionary elements in the specified list and returns values associated with the specified key.