

DesignScript Language Specification 2.1

1. [Introduction](#)
2. [Lexical elements](#)
 - 2.1 [Comments](#)
 - 2.2 [Semicolons](#)
 - 2.3 [Identifiers](#)
 - 2.4 [Keywords](#)
 - 2.5 [Bool literal](#)
 - 2.6 [Integer literal](#)
 - 2.7 [Floating-point literal](#)
 - 2.8 [String literal](#)
3. [Types](#)
 - 3.1 [Primitive Types](#)
 - 3.2 [User defined types](#)
 - 3.3 [Collections](#)
 - 3.3.1 [List](#)
 - 3.3.1.1 [Rank](#)
 - 3.3.1.2 [Indexing](#)
 - 3.3.1.3 [Dynamic list](#)
 - 3.3.1.4 [List Immutability](#)
 - 3.3.2 [Dictionary](#)
 - 3.4 [Type conversion rules](#)
 - 3.4.1 [Non-array case](#)
 - 3.4.2 [Rank promotion](#)
4. [Variables](#)
 - 4.1 [Dynamic](#)
 - 4.2 [Scope](#)
 - 4.2.1 [Scope resolution](#)
5. [Functions](#)
 - 5.1 [Function declaration](#)
 - 5.2 [Default arguments](#)
 - 5.3 [Function overloads](#)
 - 5.4 [Function resolution](#)
6. [Expressions](#)
 - 6.1 [List creation expression](#)
 - 6.2 [Range expression](#)
 - 6.3 [Inline conditional expression](#)
 - 6.4 [Member access expression](#)
 - 6.5 [List access expression](#)
 - 6.6 [Operators](#)
 - 6.6.1 [Arithmetic operators](#)
 - 6.6.2 [Comparison operators](#)

[6.6.3 Logical operators](#)

[7. Statements](#)

[7.1 Empty statements](#)

[7.2 Import statements](#)

[7.3 Expression statements](#)

[7.4 Assignments](#)

[7.5 Flow statements](#)

[7.5.1 Return statements](#)

[7.5.2 Break statements](#)

[7.5.3 Continue statement](#)

[7.6 If statements](#)

[7.7 Loops](#)

[7.7.1 While statements](#)

[7.7.2 For statements](#)

[8. Language blocks](#)

[8.1 Default associative language block](#)

[8.2 Nested associative language block](#)

[8.3 Imperative language block](#)

[9. Replication and replication guide](#)

[9.1 Replication and replication guide](#)

[9.2 Function dispatch rule for replication and replication guide](#)

[10. List@Level](#)

[11. Built-in functions](#)

[12. Internals](#)

[12.1 Associative update](#)

[12.1.1 Associativity establishment](#)

[12.1.2 Update by re-execution](#)

[12.1.3 Dependents](#)

[12.1.4 Entities that could be depended on](#)

[12.1.4.1 Variables](#)

[12.1.4.2 Function](#)

[12.1.4.3 Properties](#)

[12.1.5 Associativity scope](#)

[12.2 Classes](#)

[12.2.1 Class declaration](#)

[12.2.2 Inheritance](#)

[12.2.3 Constructors](#)

[12.2.4 Properties](#)

[12.2.5 Member functions](#)

[12.2.6 Access modifiers](#)

[12.2.7 Static members](#)

[12.2.8 `_Dispose\(\)` method](#)

1. Introduction

This is the documentation for the DesignScript programming language. DesignScript is dynamic, garbage-collected and associative language, and provides strong support for a flow based 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 (Zero Touch methods). The latter 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)

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

- Titlecase letter (Lt)
- Modifier letter (Lm)
- Other letter (Lo)
- Letter number (NI)

“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 being keywords and cannot be used as user defined variables.

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

2.5. Bool literal

Boolean literal can only be either of the following fixed values:

```
true, false
```

2.6. Integer literal

Integer literal represents an integer constant. It is in decimal base.

```
digit = '0'..'9'
hex_digit = '0'..'9' + 'a'..'f' + 'A'..'F'
decimal_number = digit { digit }
```

Example:

```
123;
```

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;  
0.123;
```

2.8. String literal

String literal represents a string constant. It is obtained by putting sequence of characters in between double quotes (“”).

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 character²:

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

Example:

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

3. Types

3.1. Primitive Types

Primitive types refer to the most basic data types that are available in the language. The type system in DesignScript is dynamic and object-oriented, meaning type checking of a program is performed at runtime based on the data returned. DesignScript supports following primitive types:

Type	Value range*	Default value
string	n.a.	“”

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

*These are implementation details and are not specific to the language spec

integer	$-2^{63} - 2^{63} - 1$	0
floating-point	IEEE 754 double-precision ³	0
bool	true, false	false
null	null	null
var	n.a.	null

If the language implementation supports FFI (foreign functions), the default value of all other imported FFI types is 'null'.

3.2. User defined types

User defined types are supported through [class mechanism](#). These types, also referred to as classes, may contain

- Properties
- Instance methods
- Static members
- Constructors

Only single inheritance is allowed in DesignScript.

3.3. Collections

3.3.1. List

```
List = "[" exprList? "]"
exprList = Expression ("," Expression)*
```

A List is a variable type that holds one or more values of a given type or an assortment of types. It can also contain values that are Lists by themselves or [Dictionaries](#). It can be regarded as a dynamic array whose size is not fixed and is allocated at runtime. Lists are created with the square bracket operator ([]) around a set of values separated by commas, or left empty to create an empty collection:

```
a = []; // empty list
a = [1, 2, 3]; // create a list with explicit values
```

Lists can also be created using [Range Expressions](#).

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

3.3.1.1. Rank

If a type has square brackets, “[]”, suffixed after it, it declares a list. The number of “[]” specifies the rank or dimension of the list. A “[] . []” specifies arbitrary rank, which means that the list can be of any rank including a single value or rank 0. 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
```

A List is a unique type in DesignScript as it is the only type that supports [replication](#). Functions can replicate over list arguments. The rank of an argument type decides how [replication](#) and [replication guides](#) work on the argument during function dispatch.

3.3.1.2. Indexing

It is possible to index into any location of the list using the square bracket ([]) operator following the variable name. Indexing starts at 0, meaning the first element of a list ‘a’ is accessed with: ‘a[0]’. Lists can also be used as the indices to generate new sublists from a list. For instance, a list containing the numbers [1, 3, 5, 7], when used as the index of a list, would extract the 2nd, 4th, 6th, and 8th elements from a list:

```
// Create a new list c, from a collection of indices
a = 1..10;
b = [1, 3, 5, 7];
c = a[b]; // c = [2, 4, 6, 8]
```

3.3.1.3. Dynamic List

A list in DesignScript is dynamic. The individual members of a list can be modified using the same index operator after the list has been created. If setting a value to an index which is beyond the length of the list, it will automatically expand and the indices in-between will be padded with “null” values. This is also called “null-padding”. For example:

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

It is also possible to dynamically change the rank of a list. In the following example, ‘x’ is originally a list of rank 1. When it is set to a value at an index for a dimension that exceeds the bound/rank of the list, the intermediate indices are set to ‘null’ values due to null-padding:

```
x = [1, 2, 3];  
x[1][2] = 99; // x = [1, [2, null, 99], 3]
```

Note: The dynamic nature of a List should not be confused with List immutability covered in the next section.

3.3.1.4. List Immutability

Lists in DesignScript are immutable. When copying a list from one variable to the other variable, it is a deep copy operation, i.e. all elements in the list are copied as well. The same holds true when DesignScript lists interact with C# lists. The original list remains immutable and any operation on it creates a new List. Example:

```
a = 1..5;  
b = a;  
b[7] = 99; // b = [1, 2, 3, 4, 5, null, null, 99], a = [1, 2, 3, 4, 5]
```

3.3.2. Dictionary

```
Dictionary =  
    "{" keyValues? "  
keyValues =  
    keyValue ("," keyValue)*  
keyValue =  
    string ":" Expression
```

A DesignScript Dictionary is a unordered map of key-value pairs, where keys are unique in the Dictionary and can only be of 'string' type. The values in the Dictionary can be of any type including lists and dictionaries by themselves. Dictionaries are initialized using curly braces, '{ }'. Empty curly braces create an empty Dictionary. Dictionaries can be initialized and queried using the following syntax:

```
x = {"foo" : 1, "bar" : 2, "baz" : 3};  
a = x["foo"]; // a = 1  
b = x["baz"]; // b = 3
```

Unlike Lists, Dictionaries cannot be dynamically modified once initialized. They are strictly immutable. Replication is not supported on Dictionaries.

3.4. Type conversion rules

DesignScript has certain type conversion rules built into it. Type conversion (also called type coercion) can be either explicit or implicit. DesignScript does not support explicit type conversion.

Since DesignScript is a dynamically-typed language, values can be converted between different types automatically at runtime, which is referred to as implicit type coercion. Implicit type conversion can happen to values when they interact with each other through operators such as '==' or triggered by the surrounding context, like with 'if (value) {...}', where 'value' is coerced to boolean. Type conversion also happens to function arguments when they are passed into typed parameter positions in the function and are forced to assume the type of the parameter. Examples:

```
// A non-zero double value converts to a true value when coerced to a 'bool' type
a = 3.0 == true; // a = true

// a non-empty string converts to a true value when coerced to a 'bool' type in the
// inline conditional statement
i = "hello" ? "non-empty" : "empty"; // i = "non-empty"
```

The following implicit type conversion rules specify the result of converting an object of one type to that of another with similar content:

3.4.1. Non-list case

“yes” means convertible, “no” means not convertible.

To From	var	int	double	bool	string	user defined
var	yes	Yes, if actual type can be coerced to int	Yes, if actual type can be coerced to double	Yes, if actual type can be coerced to bool	Yes, if actual type can be coerced to string	Yes, if actual type can be coerced to user-defined type
int	yes	yes	yes	x != 0	no	no
double	yes	warning	yes	x != 0 && x != NaN	no	no
bool	yes	no	no	yes	no	no
string	yes	no	no	x != ""	yes	no
user defined	yes	no	no	x != null	no	Covariant

3.4.2. Rank promotion

Single value arguments when passed into function parameter positions with list (array) types are promoted to lists. Similarly lower ranked list arguments are promoted to higher ranked lists when passed to function parameters with higher rank. Array promotion and type coercion rules also apply when typed identifiers (variables) with higher ranked types are assigned lower ranked values. The following table shows all the array promotion and type coercion rules in DesignScript:

```
a : int[] = 123; // a = [123] as the value assigned to 'a' is promoted to a list
```

To From	var[]	int[]	double[]	bool[]	string[]	user defined[]
var	yes	Yes, if actual type can be coerced to int	Yes, if actual type can be coerced to double	Yes,if actual type can be coerced to bool	Yes, if actual type can be coerced to string	Yes, if actual type matches user defined type
int	yes	yes	yes	x != 0	no	no
double	yes	warning	yes	x != 0 && x != NaN	no	no
bool	yes	no	no	yes	no	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. If a variable is not declared with a type, it is free to be assigned any type and the type is totally run-time dependent. Example:

```
a = 123; // a is an 'int'  
a = "hello"; // a is now a 'string'
```

In the following example, since 'a' is declared as 'int' and there is no conversion from 'string' to 'int', 'a' is 'null' in this case:

```
a : int = "string"; // a is 'null'
```

4.2. Scope

DesignScript uses block scope⁴. The scope of a defined variable in DesignScript is limited to a block (such as a language block) or a function where it is defined. A variable can be used before it is defined because of [associativity](#) and the DesignScript virtual machine will ensure to propagate the value update to all its references.

```
x = y; // x = 2, this statement will execute after 'y' due to associativity
y = 2;
z = x + y; // z = 4, as it will execute after 'x'
```

In the case of [imperative scope](#), statements are executed in sequence unlike in associative code. All global variables (defined in the outer scope) that are used in the imperative block are copied locally and any changes to them within the imperative block do not affect their values in the outer block. In other words these global variables are read-only (immutable) within the imperative block:

```
x = 1;
y = 2;
def foo(x) {
    z = 3; // "z" is local variable
    return x + y; // "x" is parameter
                // "y" is in outer scope and not defined inside function
}
[Imperative] {
    x = 3; // A local copy of "x" is made and the global "x" is unchanged
    m = n; // "m", "n" are local variables.
    n = 4; // the order of statements matters in Imperative code
          // since 'm' is assigned before 'n', its value remains 'null'
    if (true) {
        y = x + 100; // OK, "x" is visible here
    }

    return x + y + m; // a local copy of 'y' is made inside the imperative block
}
```

4.2.1. Scope resolution

The search order of an identifier is

- Innermost scope.
- Each progressive outer scope, including class scope.

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

- Classes that the current class extends.
- The global scope.

5. Functions

5.1. Function declaration

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

ParameterList =
    "(" [Parameter {"," Parameter}] ")"

Parameter =
    identifier [":" TypeSpecification] [DefaultArgument]

StatementBlock =
    "{" { Statement } "}"
```

The return type of a function is optional. By default the return type is an arbitrary ranked list, "var[] . . []". If the return type is specified, [type conversion](#) may happen. It is not encouraged to specify return type unless it is necessary.

A Function must always be defined in the global [associative language block](#) or top level block.

For parameters, if their types are not specified, by default they are of type '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 there is no replication on this parameter.

Example:

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

5.2. Default arguments

Function declaration allows default arguments for its parameters, but with one restriction: all default arguments should be applied to the rightmost parameters.

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;
}
```

5.3. Function overloads

DesignScript supports function overload, i.e., functions with the same name but with different types/number of parameters. 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 based on the type of arguments and the type of all parameters of all function candidates.

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);
```

Function overloads with the exception of those with parameters that only differ by rank are illegal. This means that for overloaded functions that have the same number and types of parameters but have one or more parameters that differ only in rank, the overload that is defined first always wins while the rest are discarded by the compiler. For example,

```
def foo(x: number, y: bool)
{
  ...
}

// invalid overload, as only ranks of parameters are different
def foo(x: number[], y: bool[])
{
  ...
}

// valid overload
def foo(x: bool, y: number)
{
  ...
}
```

The higher ranked node can be used in both a replicated as well a non-replicated call and is therefore always preferred to a lower ranked overload. As a rule of thumb therefore, users are always advised to drop lower ranked overloads in favor of higher ranked methods so that the DesignScript compiler always calls the higher ranked method as the first and only one that it finds.

5.4. Function resolution

The order of resolving a function is:

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

As functions can be overloaded, the result of function resolution will return a list of function candidates. The virtual machine will find 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 ascending order which starts at `start_value`, and ends in a value `<= end_value` with an increment of 1.
 - If `start_value > end_value`, it generates a list in descending order which starts at `start_value` and ends in a value `>= end_value` with an increment of -1.
- `start_value..end_value..increment`
 - If `start_value < end_value`, then the increment must be a positive value. It will generate a list which starts at `start_value` and last value `<= end_value`, incrementing by the increment value specified in the range expression.
 - If `start_value > end_value`, then the increment must be a negative value. It will generate a list which starts at `start_value` last value `>= end_value`, decrementing by the increment value specified in the range expression.
- `start_value..#number_of_elements..increment`: generates a list that contains `number_of_elements` elements. Element starts at `start_value` and the increment is `increment`. The value of `number_of_elements` is always rounded off to the nearest integer.
- `start_value..end_value..#number_of_elements`: generates a list that contains `number_of_elements` equally spaced elements between `start_value` and `end_value` including `start_value` and `end_value`. Depending on if `start_value <= end_value`, the generated list will be in ascending or descending order. The value of `number_of_elements` is always rounded off to the nearest integer.

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..10..2; // [1, 3, 5, 7, 9]
0..3..0.8; // [0, 0.8, 1.6, 2.4]
10..1..-2; // [10, 8, 6, 4, 2]
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"]
```

If a generated range expression is required to start and end precisely at the minimum and maximum range expression values, we can approximate an increment, coming as close as possible to the increment while still maintaining an equal distribution of numbers between the range boundaries. This can be done with the tilde sign (~) before the third parameter.

```
0..7..~0.75; // [0, 0.777, 1.555, 2.333, 3.111, 3.888, 4.666, 5.444, 6.222, 7]
```

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.

Example:

```
p = Circle.ByCenterPointRadius(Point.ByCoordinates(5, 0, 0), 10);
x = p.CenterPoint.X; // member access expression, x = 5
```

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 indices, for example the rank of “a” is 2 but the expression is `a[x][y][z]`, there will be an “`IndexOutOfRangeException`” 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 indices, “a” will be extended or its dimension will be 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	

6.6.1. 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"
```

6.6.2. Comparison operators

Statements using these operators result in a “true” or “false” value.

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

6.6.3. Logical operators

Every statement using a logic operator can only return either a “true” or “false” value.

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

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");
```

Note that import statements can only be used in DesignScript files (with a ".DS" extension), which can be imported into Dynamo. Import statements cannot be used in code block nodes in Dynamo.

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](#).

7.5.1. 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.

7.5.2. Break statements

```
BreakStatement = "break" ";"
```

A "break" statement terminates the execution of the innermost ["for"](#) loop or ["while"](#) loop.

Example:

```
i = [Imperative]
{
  a = 0;
  while(true) // this will make the while loop run indefinitely
  {
    a = a+1;
    if(a == 10)
      break; // when a equals 10, this statement breaks out of the loop
  }
  return a; // a = 10
};
```

```
i = [Imperative]
{
  a = 0;
  l = 1..100;
  for(i in l)
  {
    a = i;
    if(a == 15)
      break;
  }
  return a; // a = 15 and not 100
};
```

7.5.3. Continue statement

```
ContinueStatement = "continue" ";"
```

Example:

```
i = [Imperative]
```

```

{
  a = 0;
  l = 1..10;
  for(i in l)
  {
    if(i%2 != 0) // skips all odd numbers
      Continue; // continue next iteration of loop, skip remaining body

    a = a + i;
  }
  return a; // a = 30, sum of all even numbers upto 10
};

```

A “continue” statement begins the next iteration of the innermost “[for](#)” loop or “[while](#)” loop.

7.6. 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 ]

```

Example:

```

x = 5;
if (x > 10) {
  y = 1;
}
elseif (x > 5) {
  y = 2;
}
elseif (x > 0) {
  y = 3;
}
else {
  y = 4;
}

```

7.7. Loops

In Imperative DesignScript code a programmer has the option to write code that manually iterates over the list, extracting individual elements of the collection one at a time. Loops are language constructs to repeat execution over a block of code. The number of times a loop is called can be governed by the length of the list (“for” loop) where the loop is called with each element of the list as input. The number of iterations of a loop can also be determined by a Boolean expression (“while loop”), where the loop is called until the Boolean expression returns false. Loops can be used to generate collections, search for a solution, or repetitively call a certain block of code.

7.7.1. While statements

“while” statements repeatedly execute a block in a loop until the condition becomes false. “while” statements are only valid in [imperative language block](#). A *while* loop can run infinitely if the condition does not evaluate to *false* so it is important to have a terminating condition to break out of the loop unless there is an explicit [break](#) statement inside the loop that forces execution to break out of the *while* loop.

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

Example:

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

7.7.2. 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](#). Just like *while* statements, one can break out of a *for* loop using a [break](#) statement in the body of the *for* statement.

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

Example:

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

8. Language blocks

DesignScript has two language paradigms - Associative and Imperative. These languages are each written in their own respective language blocks.

8.1. Default associative language block

By default, all statements are in a default top level [associative language block](#), so [associative update](#) is enabled by default.

Unlike a nested language block, there is no return statement in the top level associative language block.

8.2. Nested associative language block

It is valid to explicitly define a nested language block, either associative or imperative, in the default associative language block or in a function. It is also valid to define a nested imperative language block within a nested associative language block and vice versa.

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

The statements in the default associative language block are executed in the order that statements depend on each other. This means that all dependencies of a statement are executed before the statement itself irrespective of the order in which the statements occur in the associative block. Statements in a nested associative block execute sequentially until a [return statement](#) returns a value; otherwise the language block returns null.

It is invalid to define a nested language block within a nested block of the same type.

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. An Imperative language block can only be defined in an outer associative language block or the default associative language block.

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

- [Associative update](#) is disabled in imperative language block.
- “if”, “for” and “while” statements are only available in imperative language blocks.

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 a 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. Replication and replication guide

9.1. 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 $x = [x_1, x_2, \dots, x_n]$ and $y = [y_1, y_2, \dots, y_n]$, when calling function $f(x, y)$ with zip replication, it is equivalent to $[f(x_1, y_1), f(x_2, y_2), \dots, f(x_n, y_n)]$. 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 $x = [x_1, x_2, \dots, x_n]$ and $y = [y_1, y_2, \dots, y_n]$, when calling function $f(x, y)$ 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 $[f(x_1, y_1), f(x_1, y_2), \dots, f(x_1, y_n), f(x_2, y_1), f(x_2, y_2), \dots, f(x_2, y_n), \dots, f(x_n, y_1), f(x_n, y_2), \dots, f(x_n, y_n)]$.

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 = add(xs, ys); // r1 = [4, 6]

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

// 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 = add(xs<1L>, zs<1L>); // r3 = [6, 8, 9]

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

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

```

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 parameter. 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.

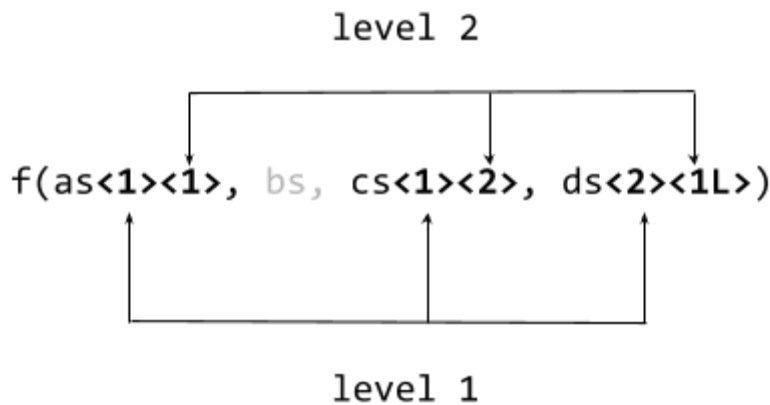
9.2. Function dispatch rule for replication and replication guide

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 heterogeneous 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 ascending 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: $r_1 = \text{rank}(a_1)$, $r_2 = \text{rank}(a_2)$, ..., $r_n = \text{rank}(a_n)$; for each rank, update it to $r = r - \langle \text{number of replication guide on argument} \rangle$. The final list $[r_1, r_2, \dots, r_n]$ 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 lists whose element values are less than or equal to the corresponding reduction value in the base reduction list. For each reduction list $[r_1, r_2, \dots, r_n]$, iteratively do the following computation to generate replications:
 - a. For any $r_i > 0$, $r_i = r_i - 1$. 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.

10. List@Level

List@Level is a special list feature that returns a flattened list at a specified level. If an input is specified as a dominant list, the structure of output will be the same as the structure of dominant list.

For example, for two inputs *xs* (Rank 4 list) and *ys* (Rank 3 list), we want to call function *f()* with the second level of *xs* and *ys*, where *xs* is the dominant list: $+ (xs@@L2, ys@L2)$. The output preserves the list structure of *xs* as it is the dominant list (represented by the double '@' symbol):

```
xs = [[["a","b"], ["c","d"]], [{"e","f"}, [{"g","h"}]], [{"i","j"}]];
ys = [{"1","2"}, [{"3","4"}, [{"5","6"}]], [{"7","8"}, [{"9","10"}, [{"11","12"}]];
x = xs@@L2 + ys@L2;
```

xs

```
[0]
  [0]
    [0]
      [0]a
      [1]b
    [1]
      [0]c
      [1]d
  [1]
    [0]
      [0]e
      [1]f
    [1]
      [0]g
      [1]h
  [1]
    [0]
      [0]
        [0]i
        [1]j
```

ys

```
[0]
  [0]
    [0]1
    [1]2
  [1]
    [0]3
    [1]4
  [2]
    [0]5
    [1]6
  [1]
    [0]
      [0]7
      [1]8
    [1]
      [0]9
      [1]10
    [2]
      [0]11
      [1]12
```

ys@L2 returns:

```
[0]
  [0]
    [0]
      [0]a
      [1]b
    [1]
      [0]c
      [1]d
  [1]
    [0]
      [0]e
      [1]f
    [1]
      [0]g
      [1]h
  [1]
    [0]
      [0]
        [0]i
        [1]j
```

```
[0]
  [0]
    [0]1
    [1]2
  [1]
    [0]3
    [1]4
  [2]
    [0]5
    [1]6
  [1]
    [0]
      [0]7
      [1]8
    [1]
      [0]9
      [1]10
    [2]
      [0]11
      [1]12
```

```
[
  [1, 2]
  [3, 4]
  [5, 6]
  [7, 8]
  [9, 10]
  [11, 12]
]
```

xs@L2 returns:

```
[0]
  [0]
    [0] [0]a
    [1] [1]b
  [1]
    [0] [0]c
    [1] [1]d
  [1]
    [0] [0]e
    [1] [1]f
  [1]
    [0] [0]g
    [1] [1]h
  [1]
    [0] [0]i
    [1] [1]j
```

```
[0]
  [0]
    [0] [0]1
    [1] [1]2
  [1]
    [0] [0]3
    [1] [1]4
  [2]
    [0] [0]5
    [1] [1]6
  [1]
    [0] [0]7
    [1] [1]8
  [1]
    [0] [0]9
    [1] [1]10
  [2]
    [0] [0]11
    [1] [1]12
```

```
[
  [a, b]
  [c, d]
  [e, f]
  [g, h]
  [i, j]
]
```

`x = xs@L2 + ys@L2` returns:

```
[0]
 [0]
  [0]
   [0]a1
   [1]b2
  [1]
   [0]c3
   [1]d4
 [1]
  [0]
   [0]e5
   [1]f6
  [1]
   [0]g7
   [1]h8
 [1]
  [0]
   [0]
    [0]i9
    [1]j10
```

11. Built-in functions

The following functions are built-in global functions that are native to DesignScript and are used internally in the VM. These are not exposed in the Dynamo UI but can still be invoked from code block nodes:

- `Concat:var[]..[](list1: var[]..[], list2: var[]..[])`
Concatenates list1 and list2 and returns a new list.
- `Count:int(list: var[]..[])`
Returns the number of elements in the specified list.
- `Evaluate:var[]..[](fptr: fptr, params: var[]..[], unpack: bool)`
Evaluates a function pointer with specified params.
- `Sleep(x: int)`
Put the virtual machine to sleep for x milliseconds.
- `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.

The following built-in functions are exposed to the end-user as inheriting from the FFI List class:

- `Equals:bool(objectA: var, objectB: var)`
Determines whether two object instances are equal.
- `GroupByFunction(list: var[]..[], func: Function)`
Groups items in a list as determined by a predicate function.

- `MaximumItemByKey(list: var[]..[], keyProjector: Function)`
Determines the maximum item in a list that satisfies a predicate function.
- `MinimumItemByKey(list: var[]..[], keyProjector: Function)`
Determines the minimum item in a list that satisfies a predicate function.
- `Rank(list: var[]..[])`
Counts the maximal rank of the specified list.
- `RemoveIfNot:var[]..[](list: var[]..[], type:string)`
Removes the members of the list which are not members of the specified type.
- `SortByFunction(list: var[]..[], func: Function)`
Sorts items in a list in accordance with a given function.
- `TrueForAll: bool(list: var[]..[], predicate: Function)`
Returns true if the given predicate is satisfied for all items in the list.
- `TrueForAny: bool(list: var[]..[], predicate: Function)`
Returns true if the given predicate is satisfied for any of the items in the list.

12. Internals

12.1. Associative update

Note that Associative update is disabled in code block nodes by disallowing variable redefinition however the language itself supports it. Associative update can be seen in action by writing DesignScript files and executing them.

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

Associative update example:

```
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

12.1.1. 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. 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.

12.1.2. Update by re-execution

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

12.1.3. Dependents

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

```
class Foo
{
    x;
}

f = Foo();
m = 21;

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

m = 42;

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

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

12.1.4. Entities that could be depended on

12.1.4.1. Variables

The change of variable will trigger updates of its dependents. Variable could appear 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 5 now.
5. "x" is increased by 1. The update for "z" will be triggered, "z" is 6 now.

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

12.1.4.2. 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).

12.1.4.3. 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)
    {
        foo = Foo(_x);
    }
}

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

b.foo.x = 1; // update "x"
b.foo = Foo(2); // update "foo"
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”.

12.1.5. 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: }
```

12.2. Classes

Note: Classes can only be declared in DesignScript script files or files that have a .DS extension. Class declarations are not allowed in code block nodes in Dynamo. DS files can be imported into a Dynamo session to be consumed as nodes.

12.2.1. Class Declaration

```
ClassDeclaration =
  "class" identifier ["extends" identifier]
  "{ " [ClassMemberDeclaration] "}"

ClassMemberDeclaration =
  ConstructorDeclaration
  | MemberDeclaration

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

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

AccessModifier = "public" | "protected" | "private"
```

A Class can only be defined in the top-level [associative language block](#).

12.2.2. Inheritance

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

Example:

```
class Point2D
{
  ...
}

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

12.2.3. 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();
```

12.2.4. 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;

```

12.2.5. Member functions

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

12.2.6. 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;

```

12.2.7. 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();
```

12.2.8. `_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 Disposable
{
    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;
```