Version registration

releases	times	note
1.1	20240723	First edition, containing the sections listed and described in the original user's manual

Script Additional Instructions - User's Manual

Clause-by-clause description of some items:

template script

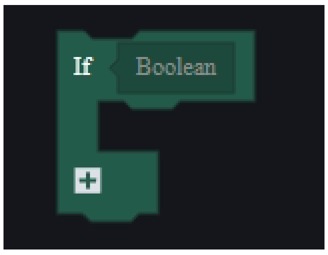
Process

Control

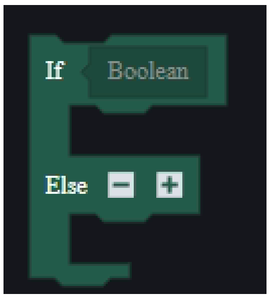
Conditional

Statements:

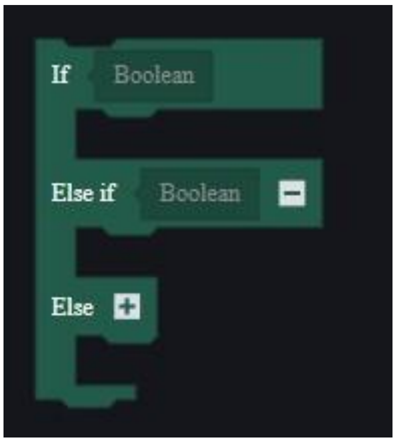
if:



Run the node under IF if the condition is met. Click on the + sign to add ELSE: if the condition is not met, run the node under ELSE:



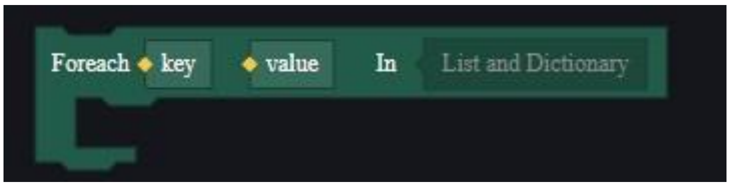
Click + again to add ELSEIF and continue to judge the next condition if it is not met.



Loop Statements:

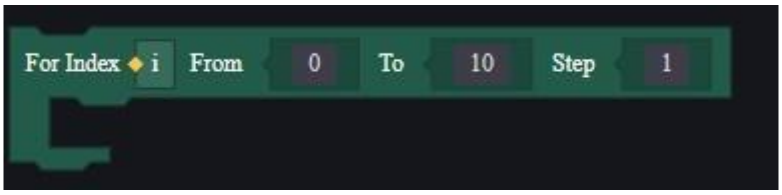
Foreach:

For each KEY in the argument list/dictionary, execute the node below once.



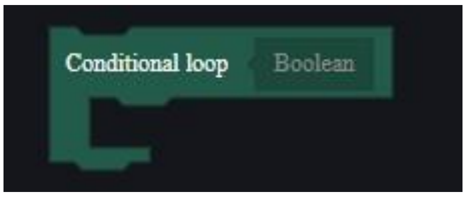
for index:

The parameter i starts from the minimum value, and every time the node below is run then i increases the step value until i is greater than or equal to the maximum value.



> The [min,max) interval of this node is left-closed and right-open, i.e., i is equal to the maximum value that time will not run below the node. As an example, this loop will run ten times, with values of i from 0 to 9.

while loop:



As long as the conditions are met, the nodes below will be run in a loop.

Break:

Interrupts the loop and continues to execute the node after the loop body.

Continue:

Skip this loop and go to the next loop in the loop body.

code script

basic grammar

exegesis

single-line comment

```
// Single-line comments
```

multiline comment

```
/*

Multi-line
comment
s Multi-
line
comment
s
```

identifiers

A token is used to define a variable, function, type name, or other user-defined item. A token consists of a letter, number, or underscore, and cannot begin with a number.

keywords

Reserved keywords cannot be used as user-defined identifiers:

keywords	**instructions**	**typical example**
define	Defining Basic Types	define AssetID : string
alias	Define type aliases that support or and expressions.	alias Number = int | int64 | float
enum	Defining Enumerations	enum SortType { AscIgnoreZero = 0 Desc = 1 Asc = 2 }
component	Defining Components	component XXX{ ... }
abstract	Modifying component definitions, abstracting components	abstract component XXX { }
partial	Modifies the component definition, extends the component, and adds new properties to the original definition of the component	partial component XXX { }
accept	Modifier type definition, other types acceptable to the modified type at the time of the assignment operation	[accept Vector3]
combine	Modifying component definitions, binding components	[combine Visibility]
graph	Defining Scripts	graph XXX{ ... }
import	Importing Scripts	import "StdLibrary.fcc" as stdlibrary
from	Partial import script	import XXX from "EditorGenLib.fcc"
static	Modifying script definitions, static scripts	static graph XXX
readonly	Modifies a component property or variable by declaring it read-only	[readonly]
event	Define or listen to events	event OnAwake(){ ... }
func	Defining functions	func Jump(){ ... }
start	asynchronous function calling	start Jump()
wait	Synchronous function call	wait Jump()
keywords	**instructions**	**typical example**

async	Asynchronous function declarations	async func Jump(){ ... }
var	Defining Local Variables	var int num = 0
out	Declare whether the formal parameter of a function is an output parameter	func Jump(var out speed){ ... }
as	type conversion	var hpFloat = GetHP() as float
thisEntity	Referring to the current entity	var hp = thisEntity.HP
globalEntity	Refers to the global game entity	var tickCount = globalEntity.TickCount
typeof	Getting the value of a type	TypeIs(100, typeof(int))

Other keywords:

if, else, for, in, while, break, continue, return,

object, bool, int, float, string, Vector2, Vector3, Quaternion, List, entity, true, false, nil

member accessor

The member accessor is `.` , and the basic syntax is as follows to point access to object members:

```
Object. Membership
```

For example, imported script libraries, enumeration types, component types, etc., but also a specific instance of a component member or script member, etc.

Access to imported content

Imported content is member accessed in scripts via aliases, which are independent within each script. Take the example of importing a standard library:

```
import "StdLibrary.fcc" as std

graph HelloWorldGraph {
    func SayHello(name string) {
        //std is the alias after the standard
        library import std.LogInfo("Hi,
        " + name)
    }
}
```

Access to component data

Components are data that hangs on an entity, and the basic syntax when accessing data from a component on an entity object is as follows:

```
Entity object <component qualified>. Member
```

Take the example of accessing component data for the current entity:

```
func EntityPropModify(input bool) bool
    { thisEntity<Entity>.EnableLogic = input
    return thisEntity<Entity>.EnableLogic
}
```

Accessing Script Members

Scripts are behaviours that hang on entities, and the basic syntax when accessing members of a script on an entity object is as follows:

```
Entity object <script-qualified>. Member
```

Take the example of a script that accesses the current entity:

```
func MyFunc(){
    //thisEntity is a reserved word,  it refers to the current entity .
    if HasScript(thisEntity, Helper)
        { thisEntity<Helper>.MyHelpFunc()
    }else {
        //Dynamically hook scripts for entities
        AddScript(thisEntity, Helper)
        thisEntity<Helper>.MyHelpFunc()
    }
}
```

data type

basic type

object

object is the base type for all types

bool

Boolean types have only two optional values: true a n d   false.

```
func Demo() {
    var isOK1 = false
    var isOK2 bool = true
    if isOK1 || isOK2 {
        //...
    }
}
```

int

Integer types range from "-2147483648" to "2147483647".

```
func Demo() {
    var num1 = 0
    var num2 int = 5
    if num1 > num2 {
        //...
    }
}
```

float

Decimal types range from approximately "-3.4e38" to "3.4e38".

```
func Demo() {
    var num1 = 0.5
    var num2 float = 5.6
    if num1 > num2 {
        //...
    }
}
```

string

Strings are represented by a pair of double quotes, and when the table double quotes themselves are needed, the escape character "

```
func Demo() {
    var str1 = "Hi, "
    var str2 string = "Tim"
    LogInfo(str1 + str2)
}
```

Vector2

A two-dimensional vector consists of two floats denoted X and Y respectively

```
func Demo() {
    var vec1 = Vector2{0, 0}
    var vec2 Vector2 = Vector2{1, 1}
    var vec3 = vec1 + vec2
    LogInfo(vec3.Y)
}
```

Vector3

The 3D vector consists of three floats denoting X, Y and Z respectively

```
func Demo() {
    var vec1 = Vector3{0, 0, 0}
    var vec2 Vector3 = Vector3{1, 1, 2}
    var vec3 = vec1 + vec2
    LogInfo(vec3.Z)
}
```

Quaternion

A quaternion consists of four floats denoted X, Y, Z, and W respectively

```
func Demo() {
    var qua1 = Quaternion{0, 0, 0, 1}
    var qua2 Quaternion = Quaternion{0, 0, 0, 1}
    LogInfo(qua1.W)
    LogInfo(qua2)
}
```

Enum

An enumeration is the concept of a collection of constant data, such as an attack style enumeration, which can contain options such as melee, shoot, explode, etc. within it. Different enumeration types cannot be assigned to each other, even though they may have the same value, due to different semantics.

```
func Demo() {
    var element int = std.ItemGoodsIDType.AC80
    if element == std.ItemGoodsIDType.AC80 {
        LogInfo("suc")
    }
}
```

dynamic type

List

A list, which represents a set of data, where T is the type of the elements in the list e.g. List, which means it is a list of integers.

```
func Demo() {
    var allItems = List<object>{"a",
    "b"} allItems[0] = "A"
    LogInfo(allItems[0])
    for index, element in
        allItems{ LogInfo(element)
    }
}
```

Map<T1,T2>

Dictionary, represents a set of key-value mapped data, where T1 is the type of the key and T2 is the type of the value e.g. Map<string, int>, indicates that this is a dictionary that looks up the value of int through the string Key

```
func Demo() {
    var allItems Map<string, int> = Map<string, int>{"a":1, "b":2}
    allItems["a"] = 2
    LogInfo(allItems["a"])
    LogInfo(allItems)
}
```

entity<T>

Entity, representing an object where T is the type of component attached to the entity
For example, entity<Player>, indicating that this is an entity with a Player component

```
func Demo() {
    thisEntity<Global>.EcoRoundMoney = 100
    LogInfo(thisEntity<Global>.EcoRoundMoney)

    RevivePlayer(thisEntity<Player>)
}
```

operator (computing)

arithmetic operator

Assume that the value of A is 10 and the value of B is 20.

operator (computing)	descriptions	typical example
+	add up	A + B Output 30
-	subtract from one another	A - B Output result -10
*	multiply (math.)	A * B Output result 200
/	divide (math.)	B / A Output Result 2
%	have to find the remainder (math.)	B % A Output result 0
++	self-addressing	A++ Output 11
–	self-reducing	A- Output results 9

relational operator (computing)

Assume that the value of A is 10 and the value of B is 20.

operator (computing)	descriptions	typical example
==	Checks if two values are equal, if they are equal returns True otherwise returns False.	(A == B) is False
! =	Checks if two values are not equal, if they are not equal returns True otherwise returns False.	(A ! = B) is True
>	Checks if the value on the left is greater than the value on the right, if so returns True otherwise returns False.	(A > B) is False
<	Checks if the left value is less than the right value, if so returns True otherwise returns False.	(A < B) is True
>=	Checks if the left value is greater than or equal to the right value, if so returns True otherwise returns False	(A >= B) is False
<=	Checks if the left value is less than or equal to the right value, if so returns True otherwise returns False	(A <= B) is True

logical operator

Assume that the value of A is True and the value of B is False.

operator (computing)	descriptions	typical example
&&	Logical AND operator. Conditional True if both operands are True, otherwise False	(A && B) for False
||	Logical OR operator. Conditional True if one of the operands on either side is True, otherwise False	(A || B) is True
!	Logical NOT operator. The logical NOT condition is False if the condition is True, otherwise it is True.	! (A && B) for True

assignment operator

operator (computing)	descriptions	typical example
=	Simple assignment operator that assigns the value of an expression to a left value	C = A + B Assigns the result of an A + B expression to C.
+=	Sum and assign	C += A equals C = C + A
-=	Subtract and assign	C -= A equals C = C - A
*=	Multiply and then assign	C *= A equals C = C * A
/=	Dividing and then assigning	C /= A equals C = C / A
%=	Derivation and then assignment	C %= A equals C = C % A

operator priority

Some operators have higher priority, and binary operators all operate from left to right. The following table lists all the operators and their priorities, from top to bottom representing the highest to lowest priority:

priority	operator (computing)
5	*, /, %
4	+, -
3	==, ! =, <, <=, >, >=
2	&&
1	||

process control

conditional statement

Conditional statements require the developer to decide whether or not to execute the specified statement by specifying one or more conditions and testing whether the condition is true, and to execute another statement if the condition is false.

```
func Max(left Number, right Number) Number
    { if left > right {
        return left
    }else if left < right
        { return right
    } else{
        return left
    }
}
```

cyclic statement

In many real-world problems there are many repeated operations with regularity, so it is necessary to repeat certain statements in the programme.

for index

```
func OddNumSumV1(max int) int
    { var sum = 0
    for i = 1, max, 2
        { sum += i
    }
    return sum
}
```

for range

```
func SumOfPrimesUpTo10V3() int
    { var sum = 0
    var nums = List<int>{2, 3, 5, 7}
    for i, num in nums {
        sum += num
    }
    return sum
}
```

while

```
func OddNumSumV2(max int) int
    { var sum = 0
    var loopIndex = 0
    while loopIndex < max
        { loopIndex = loopIndex +
        1 sum += loopIndex
    }
    return sum
}
```