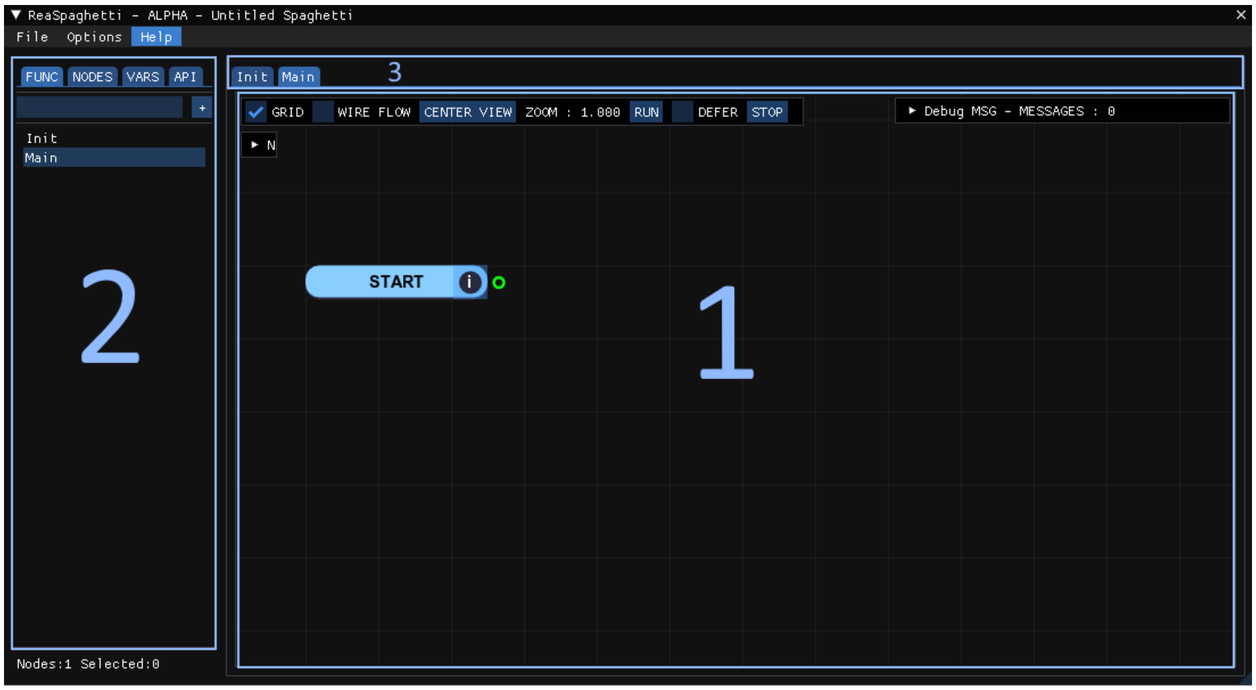
ReaSpaghetti

OVERVIEW:

1. **CANVAS**
2. **SIDEBAR**
3. **FUNCTIONS TAB**



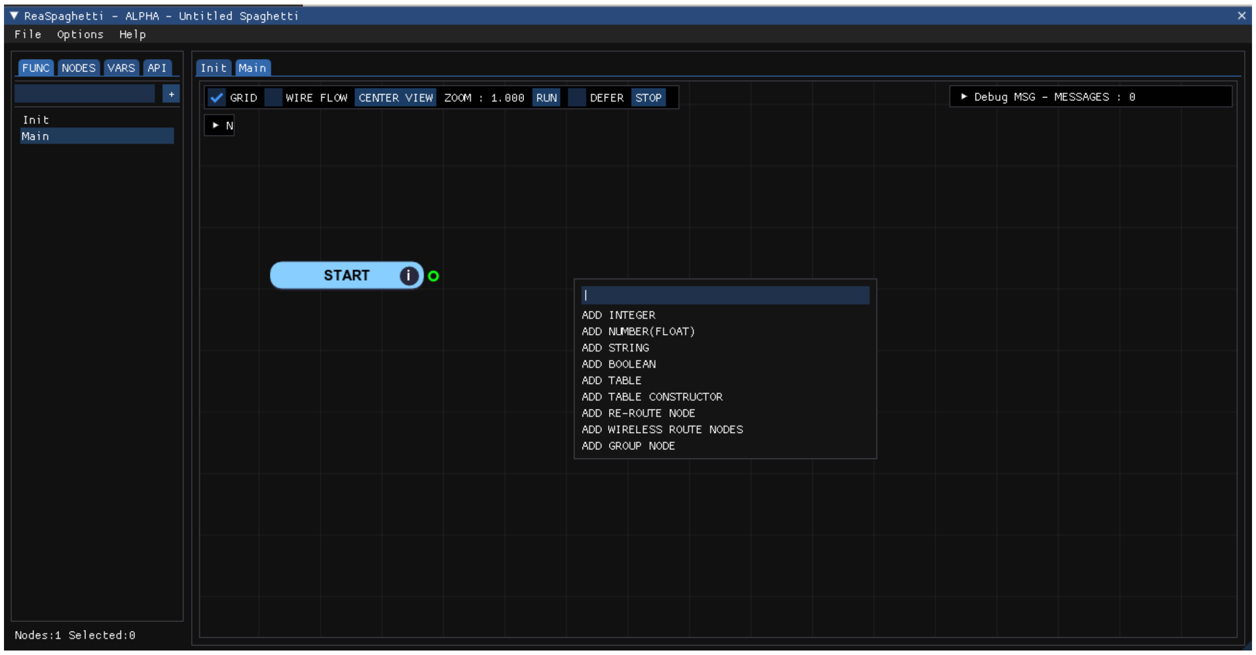
CANVAS

START Node is self explanatory it is the start of the node flow and needs to be connected in order for function to execute. This node cannot be **COPIED OR DELETED** and it is present in every function.

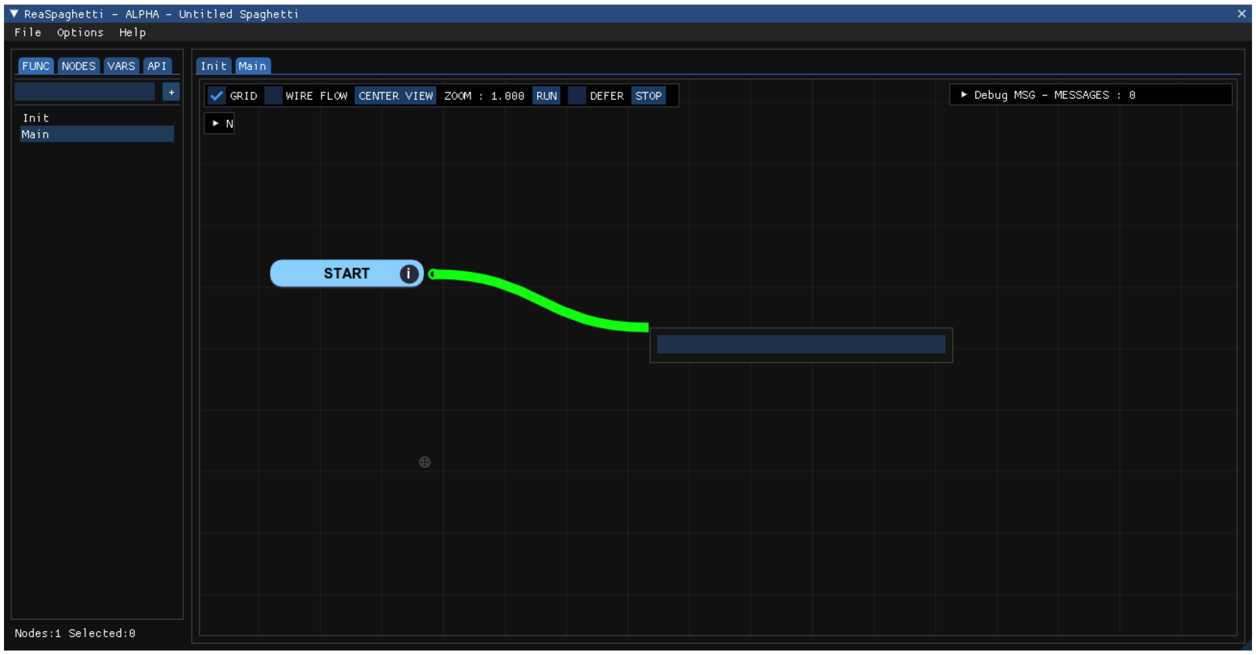
It also acts as **FUNCTION ARGUMENTS INPUT** but that will be discussed later in **FUNCTIONS** section.

Adding new nodes is done in two ways in **CANVAS**:

1. **Right click**



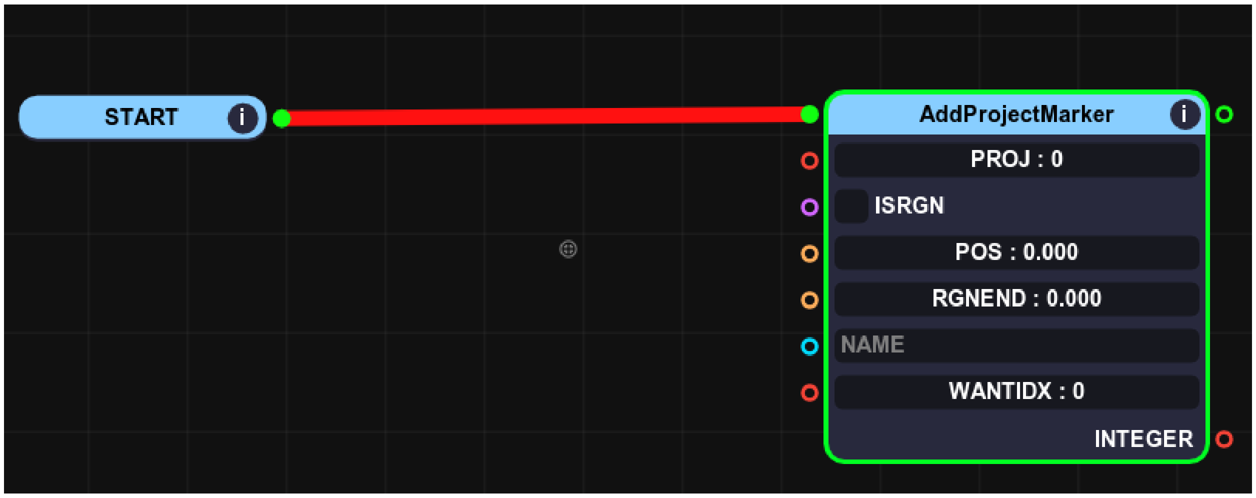
2. **RUN PIN(GREEN)** drag:



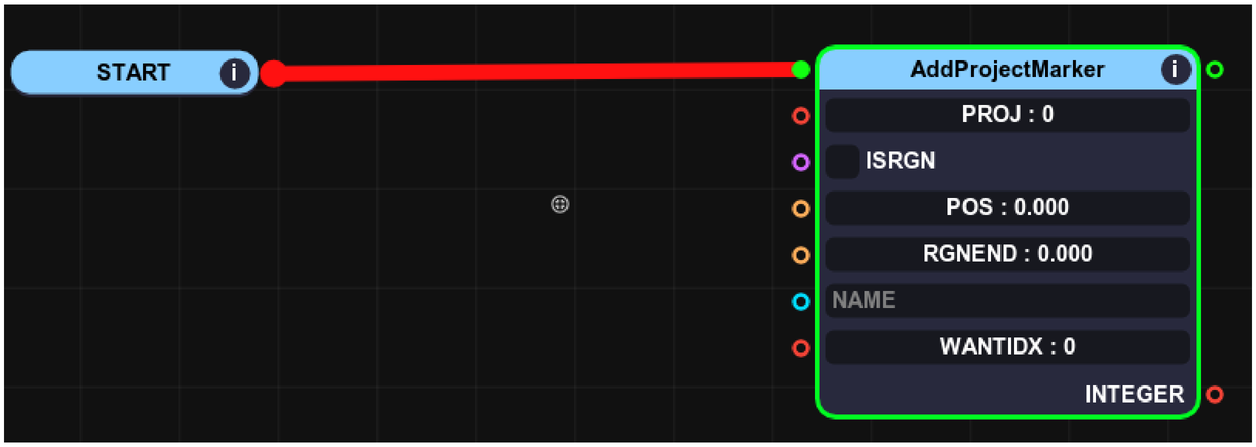
The difference between first and second is that second will auto connect to new node and that node can only be API.

Right click context will show **VARIABLE** and **UTILITY** nodes .

Node connection is deleted by hovering over the wire and **ALT CLICK (RED COLOR INDICATOR WILL APPEAR)**



Or Hovering over pin and **ALT CLICK**



General shortcuts can be found in **Help/Keys and Shortcuts** section but they are universal:

DELETE KEY – Deletes selected node

F2 – Renames Selected Node

CTRL + C/V – Copy/Paste

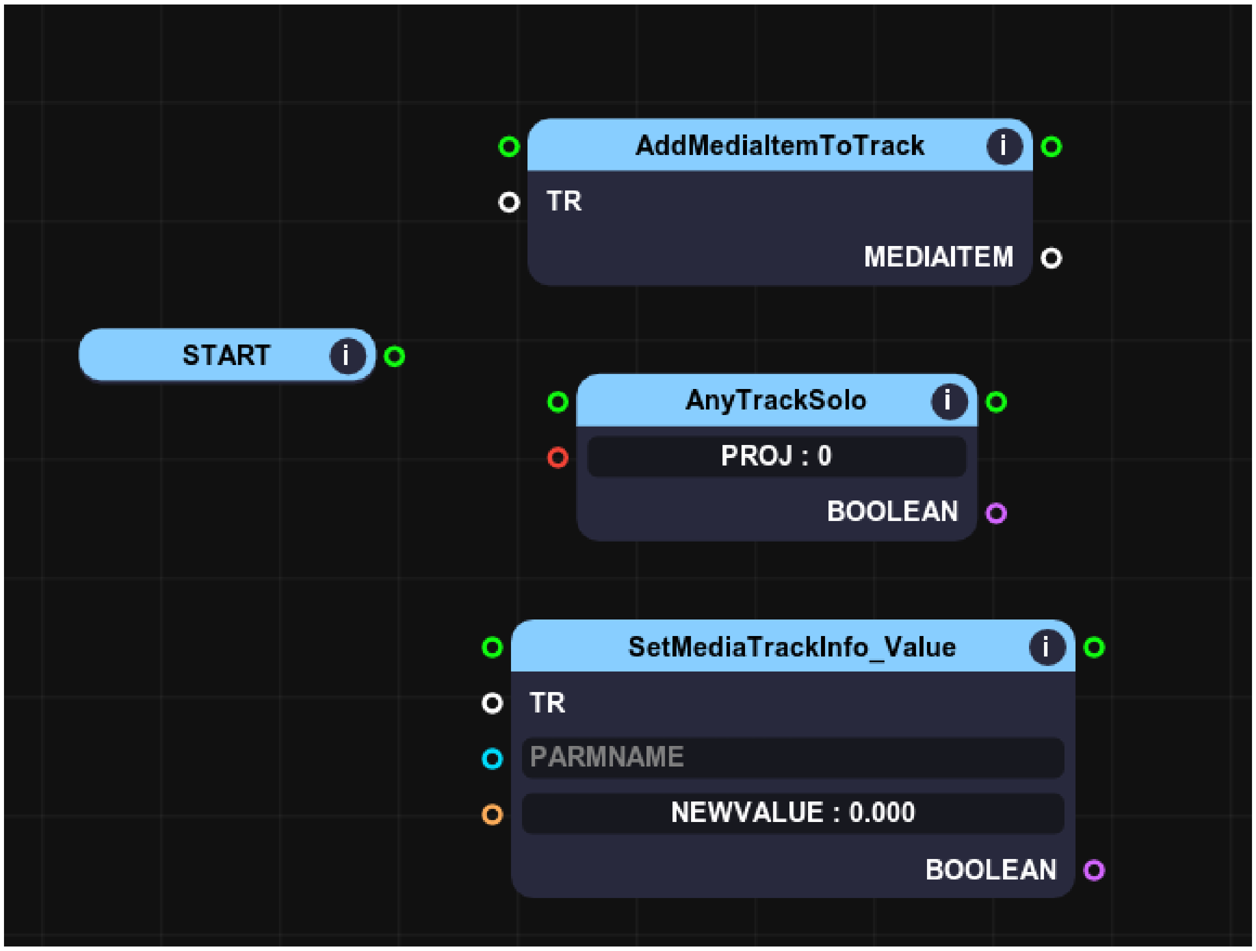
CTRL + A – Selects all nodes

Marquee select – Left DRAG

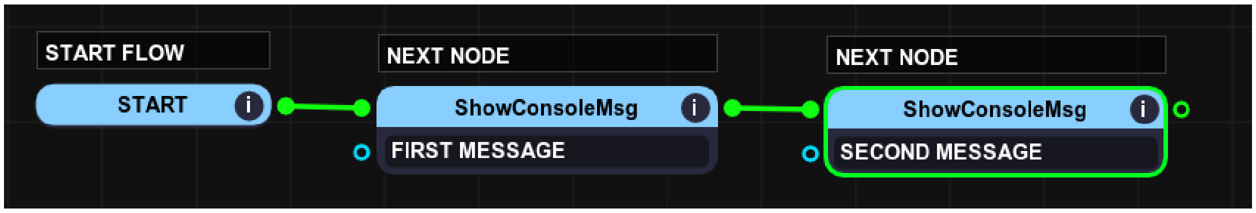
Etc.

NODES

Most nodes have main **RUN(GREEN)** pins:



That pins need to be connected between each Node to create flow.

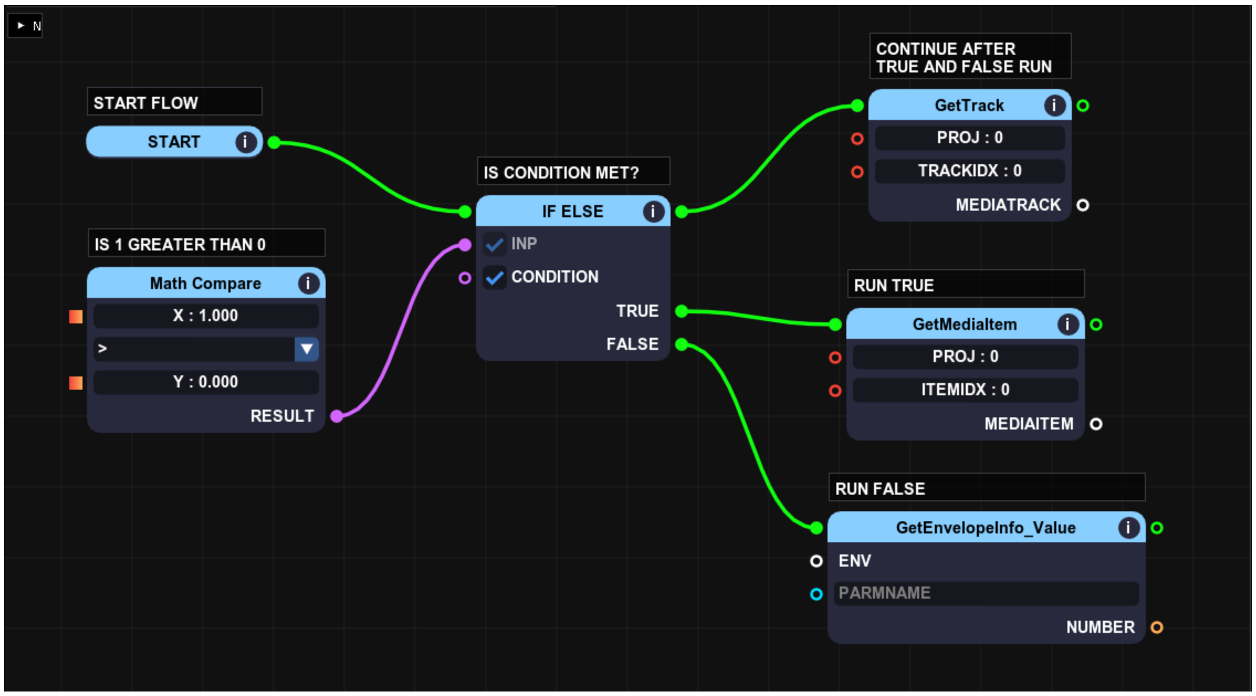


NOTICE : RUN PIN CANNOT BE BRANCHED (CONNECT TO MULTIPLE NODES)!

Some nodes have multiple **RUN PINS** like **CONTROL FLOW NODES:**

1. IF – ELSE
2. NUMERIC LOOP
3. IPAIRS LOOP
4. PAIRS LOOP
5. SWITCH

IF-ELSE



Node has 2 inputs :

1. INP (YOUR CONNECTION)
2. CONDITION – YOU SET THE TRUE OR FALSE (CHECKED TRUE, UNCHECKED FALSE)

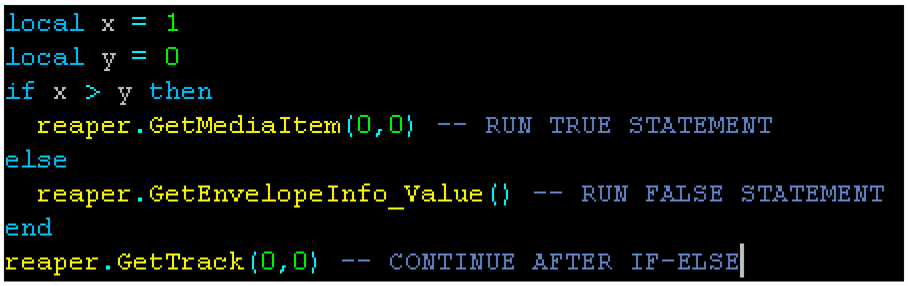
If **INP** and **CODITION** match then appropriate RUN will happen:

If condition is **TRUE** then **TRUE RUN PIN** will be executed along with all flow connected to it.

If condition is **FALSE** then **FALSE RUN PIN** will be executed along with all flow connected to it.

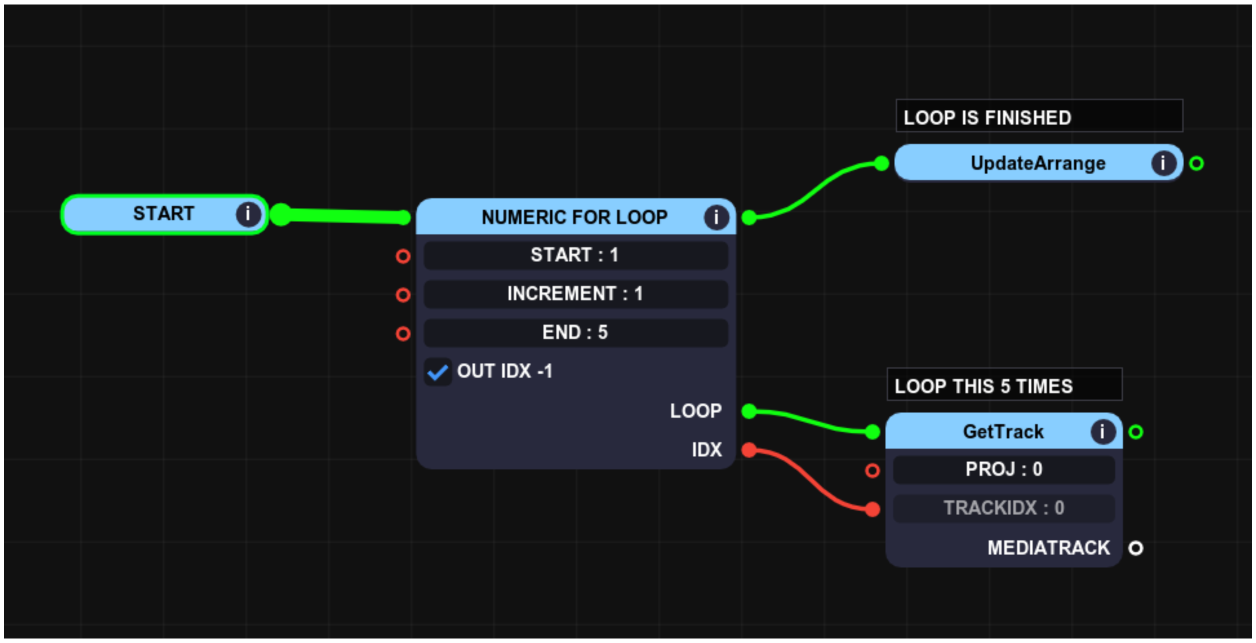
After **TRUE** or **FALSE RUNS** have passed node will continue to execute next node thru **OUT RUN.**

In Lua code it looks like this:



```
local x = 1
local y = 0
if x > y then
  reaper.GetMediaItem(0,0) -- RUN TRUE STATEMENT
else
  reaper.GetEnvelopeInfo_Value() -- RUN FALSE STATEMENT
end
reaper.GetTrack(0,0) -- CONTINUE AFTER IF-ELSE
```

NUMERIC FOR LOOP



Numeric For loop has 3 inputs, 3 outputs and a Variable:

Inputs:

1. START OF THE COUNT
2. COUNT INCREMENTE
3. END OF COUNT (COUNT RANGE)
4. OUT IDX-1 (this is used since all reaper API is 0 based)

NOTICE: OUT IDX-1 SHOULD BE DISABLED IF ITERATING TABLE NODES! BY DEFAULT ITS ON SINCE USERS WILL MOSTLY ITERATE TRACKS,ITEMS,ENVELOPES ETC WHICH IS 0 BASED.

Outputs: 1. LOOP RUN , 2. IDX OUT

This loop above will run **GetTrack** Node 5 times and when its finished it will run **UpdateArrange** node.In lua code it looks like this

```lua
for i = 1, 5 do -- START 1, END 5 (INCREMENT IS 1 BY DEFAULT)
  idx = i-1  -- OUT IDX-1
  reaper.GetTrack(0,idx) -- DO THIS 5 TIMES
end
reaper.UpdateArrange() -- CONTINUE AFTER LOOP IS FINISHED
```

Every other node that has multiple **RUN(GREEN) PINS** follow the same logic:
1.Run all **NON MAIN OUTPUT RUN PINS FIRST**

2.When finished **RUN MAIN OUTPUT RUN PIN LAST (TOP RIGHT PIN)**

SIDEBAR

Sidebar has 4 tabs:

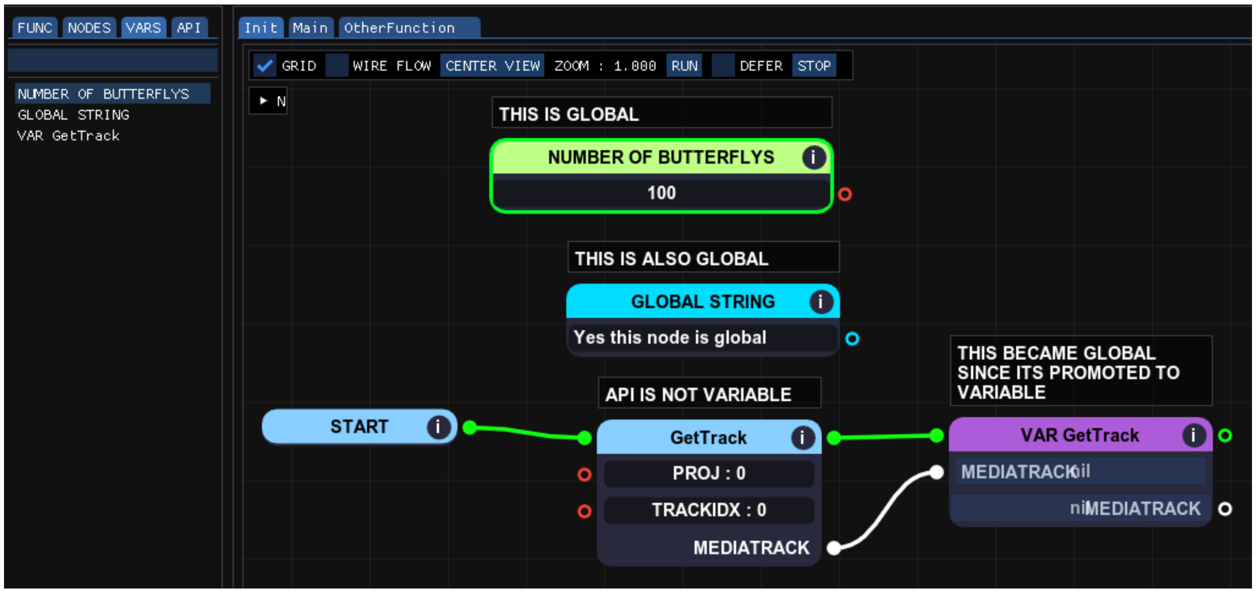
1. **FUNC** – Shows all **FUNCTIONS** in current project.
   a. Nodes can be dragged into canvas
   b. Right Click context **DELETE/RENAME**
2. **NODES** – Overview of all nodes in **CURRENTLY OPENED FUNCTION**
   a. Nodes cannot be dragged into canvas
   b. Double clicking node will **CENTER VIEW THAT NODE**
3. **VARS**
   a. Shows **LOCAL** variables of **CURRENT OPENED FUNCTION**
   b. Shows **GLOBAL** variables of **INIT FUNCTION**
4. **API** – Shows list of available API
   a. Nodes can be dragged into canvas
   b. Double click node will open its **URL** in your **BROWSER**

FUNCTIONS

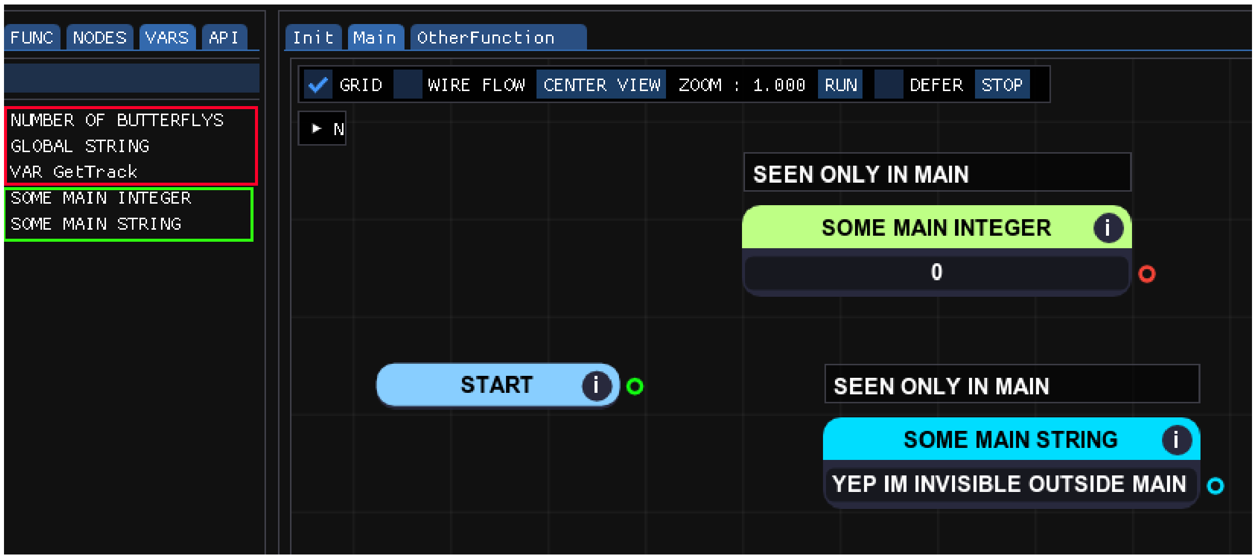
Every project starts with 2 functions **Init** and **Main.**

Init function is global space and it's the first executed function when script is run. Variables inside of it will be **GLOBAL** and visible to all other functions. There is no other way to make **GLOBAL** variables.

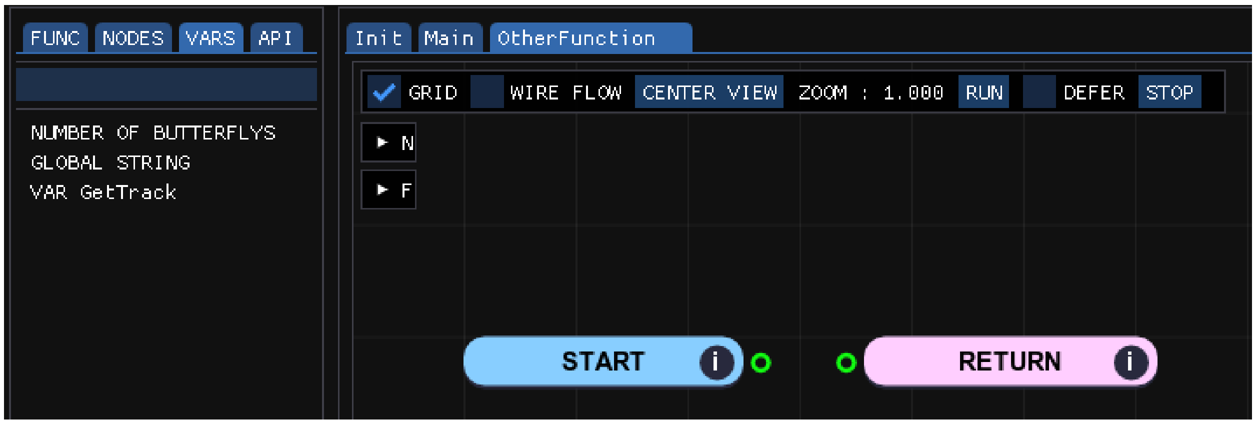
INIT function variables:



View of variables when in **MAIN** function:



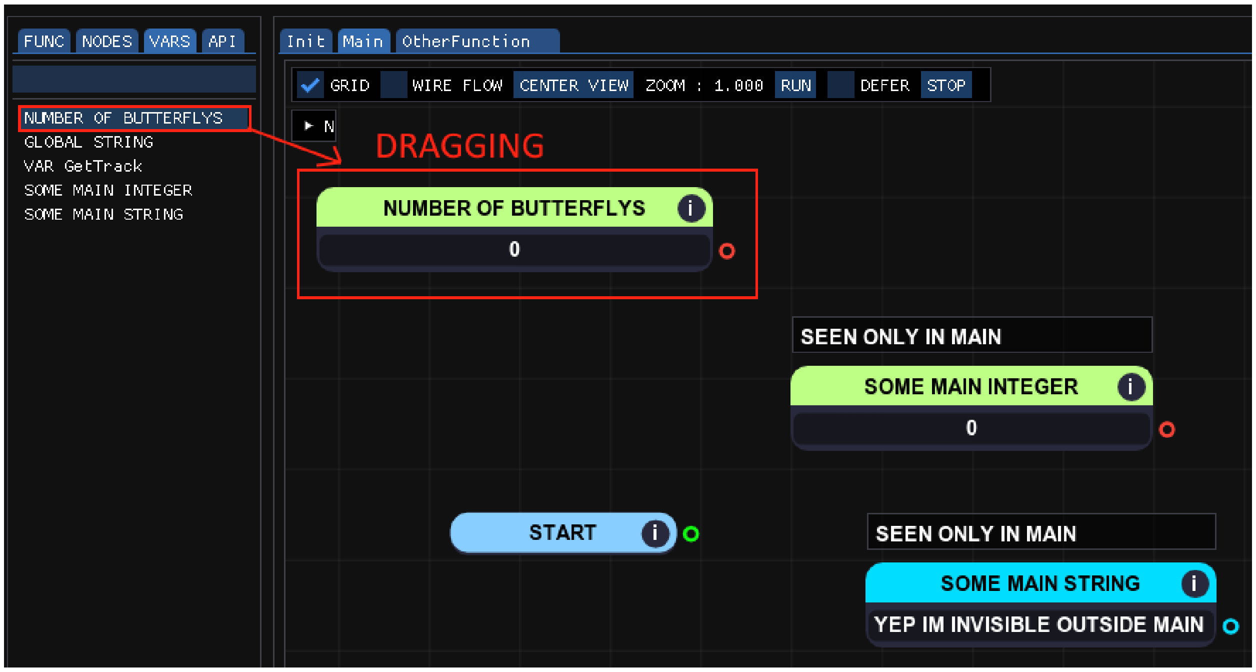
View of nodes in **OtherFunction:**



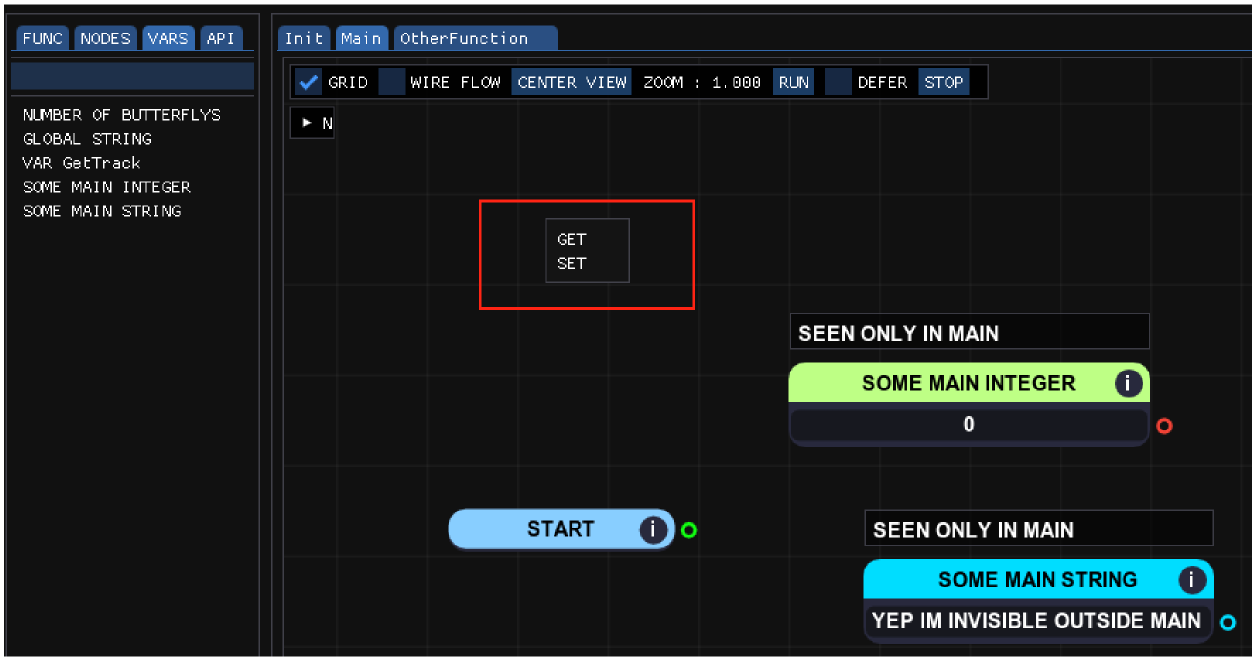
Newly created function have **RETURN** node in them which we will talk soon.

When you want to use global or any variable in your function you **DRAG AND DROP** it and it will give you two options : **GET / SET**

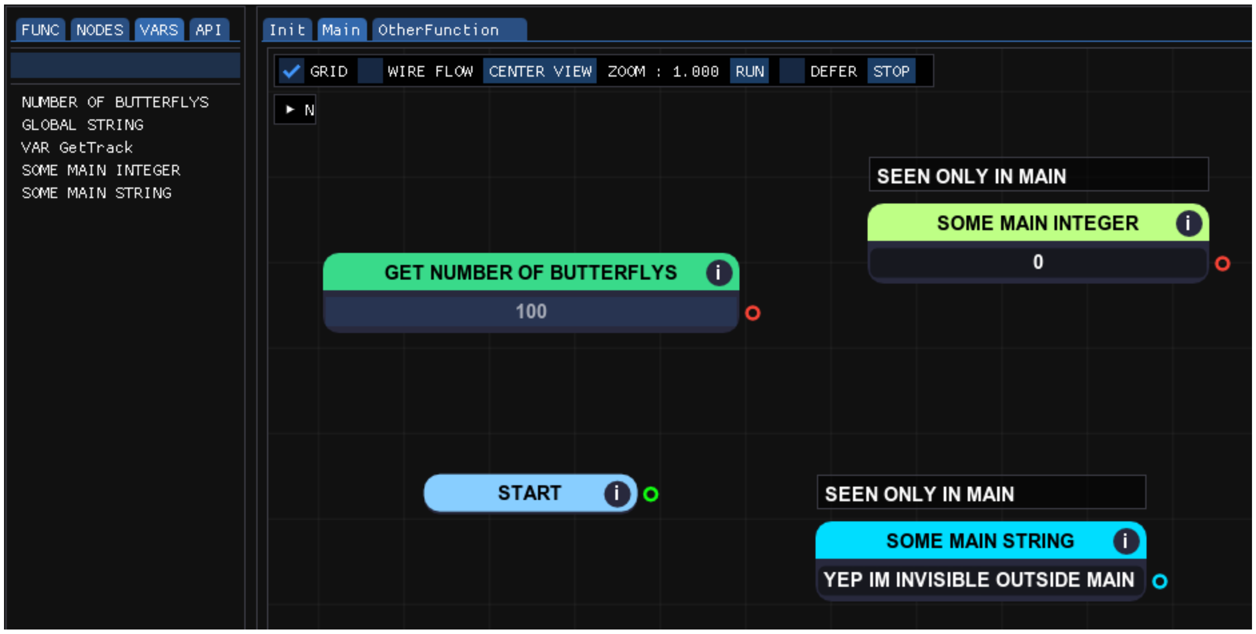
Adding Global variable to **Main**



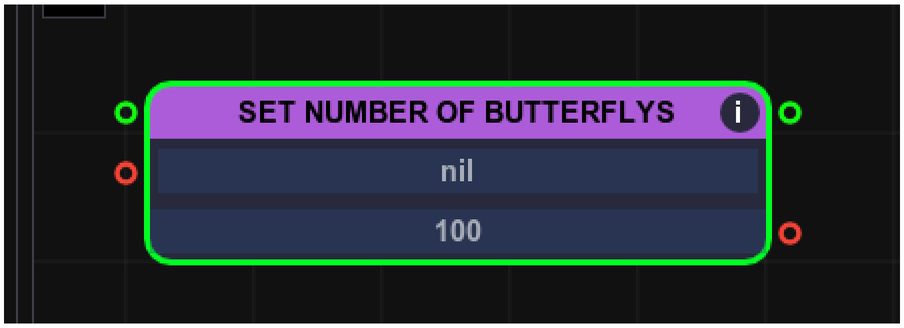
Releasing



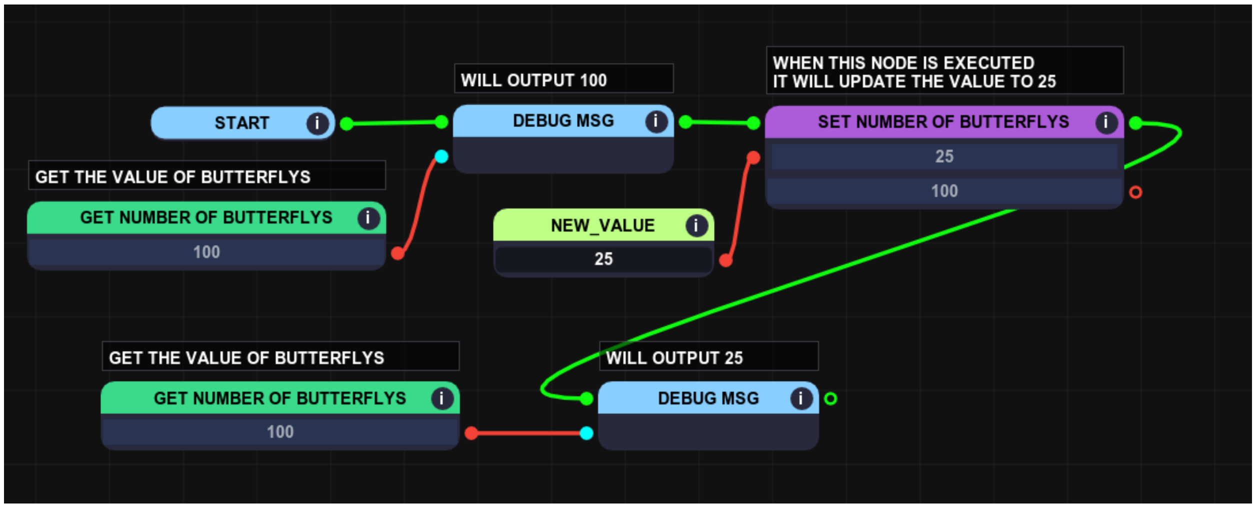
Get is self explanatory it will “**GET**” or “**READ**” its value



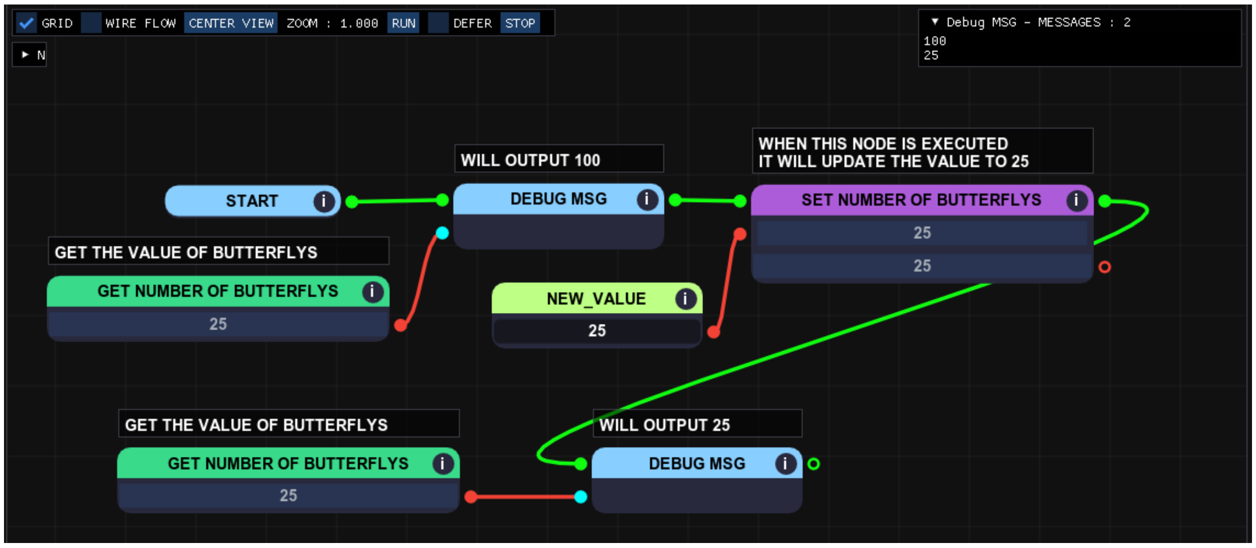
Set on other hand Modifies the value but needs to be executed somewhere in the flow:



Setter has 1 input for setting value and 1 output passthrough (for connecting to next node)



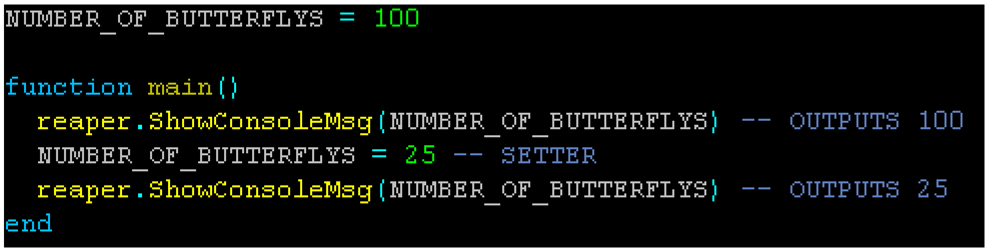
To confim this lets run the script and see the DEBUG MSG view:



```
▼ Debug MSG - MESSAGES : 2
100
25
```

It outputed first original value which was 100 and then updated value wich was 25 (don't mind both **GET** nodes showing 25 its just lua/metatable/implementation reading values…. code wizardry!)

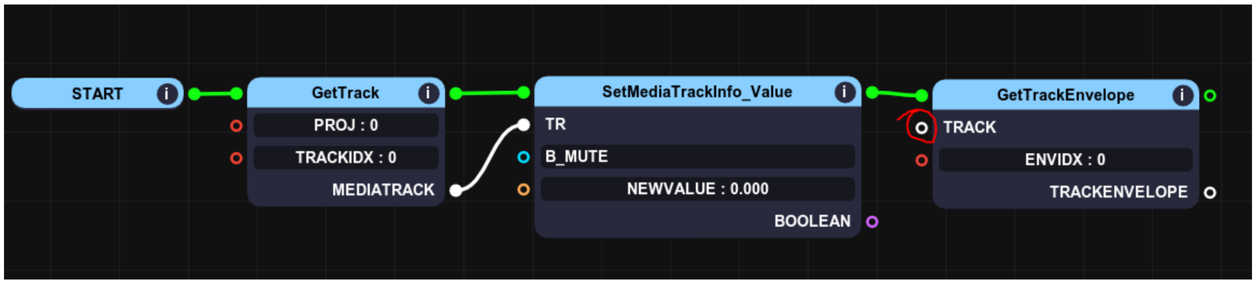
This is equivalent to lua code:



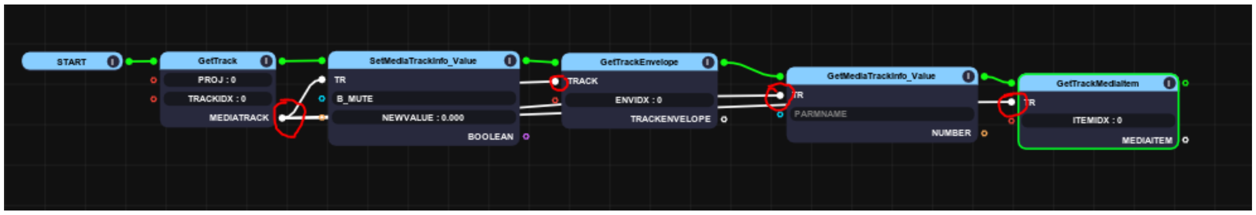
```
NUMBER_OF_BUTTERFLYS = 100

function main()
  reaper.ShowConsoleMsg(NUMBER_OF_BUTTERFLYS) -- OUTPUTS 100
  NUMBER_OF_BUTTERFLYS = 25 -- SETTER
  reaper.ShowConsoleMsg(NUMBER_OF_BUTTERFLYS) -- OUTPUTS 25
end
```

PROMOTING API TO VARIABLE

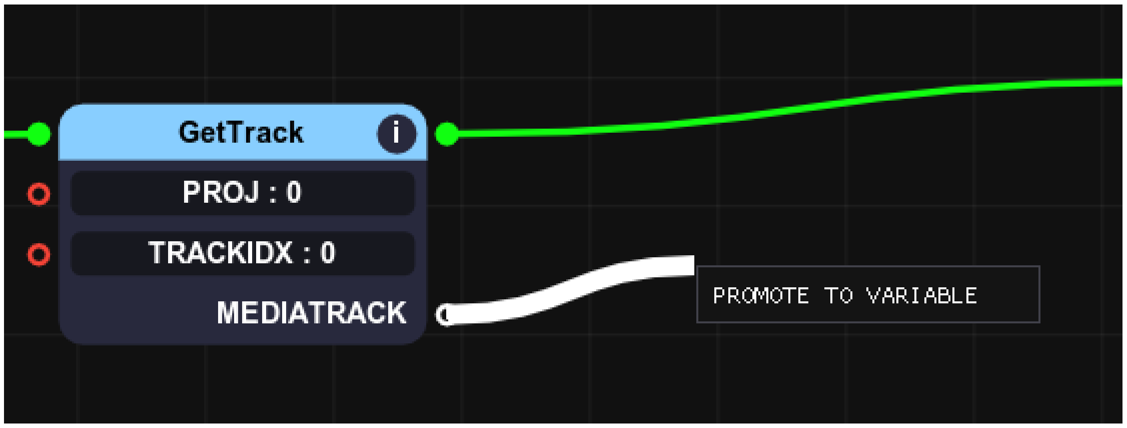
In order to connect your API node on other side of the flow for example lets say you have this flow:



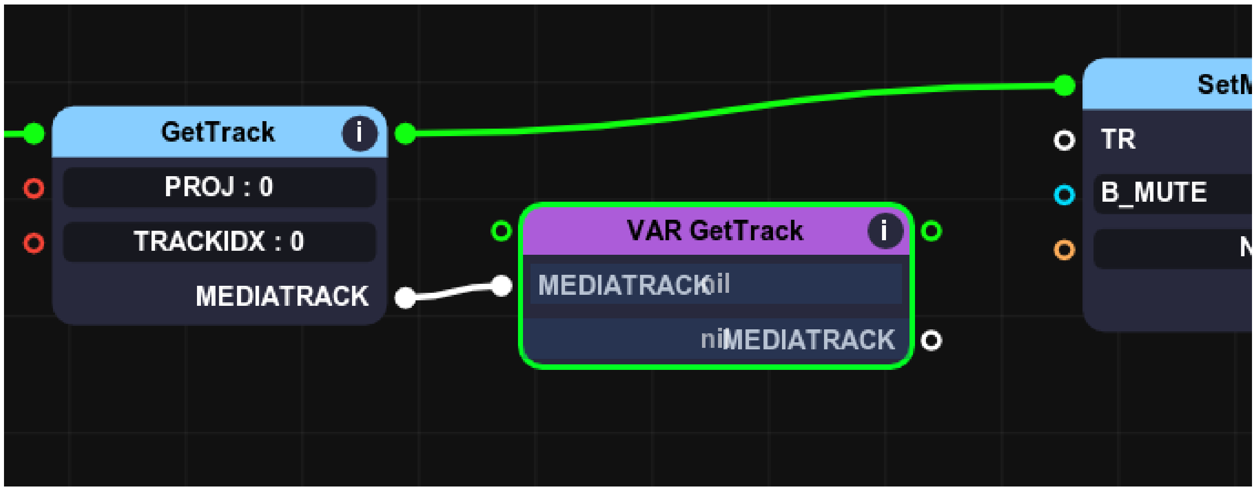
But the same **MEDIATRACK** is needed on **GetTrackEnvelope** node you can connect it directly from **GetTrack** but that would be ugly and you cannot reuse it again unless connecting it again from same output like this:



Solution is to **PROMOTE TO VARIABLE** which is done by dragging the pin to empty space:

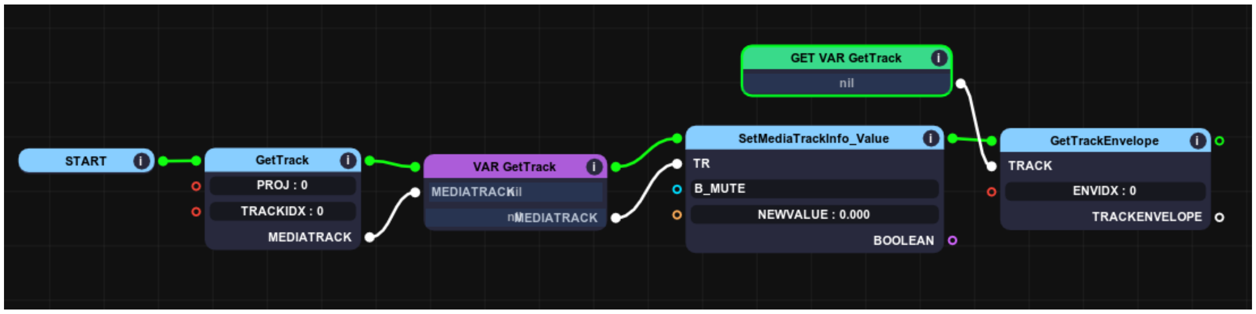


You end up with new node which will be available in **VARS** tab.



API Variables can only be **GET/READ**, you cannot **SET** the API to another value (that is also not possible to do with coding).

After properly connecting it and Dragging the **GET** from VARS tab you get:



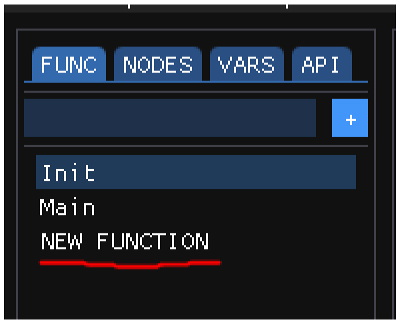
Now you can reuse the **GET** how many times you want and it will look nice and access it anywhere in your flow again.

This concept of **GET/SET** is the core how you manipulate your variables in the script (Other programs like UnrealEngine, Unity etc use same concept)

NEW FUNCTIONS

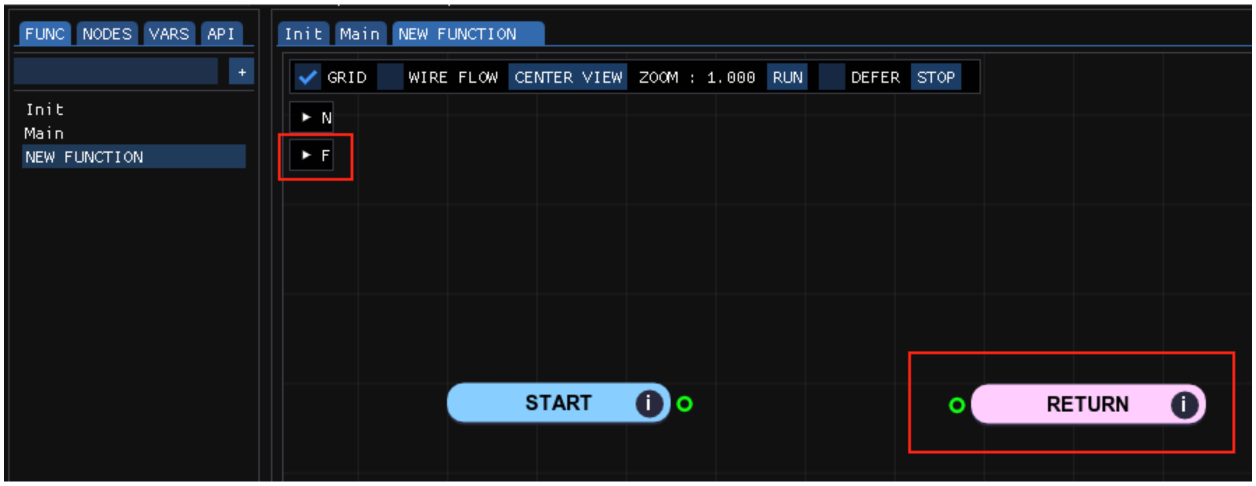
In order to keep your projects clean or you need same functionality of one of your flows but modified for something else so you need to copy paste it etc that's where the **FUNCTIONS** come.

To create new Function you go to FUNC tab and click **+** button.
That will create new function which you can add anywhere and any number of times.

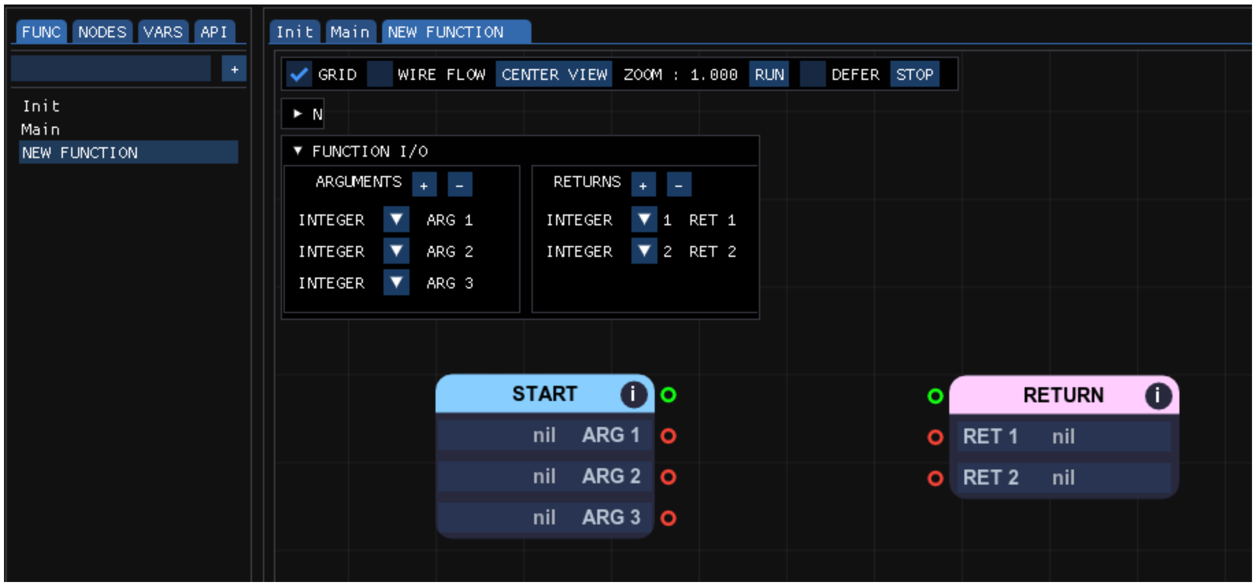


You can rename it or Delete it with Right click context and it will Rename/delete all child functions you dragged into other functions (**Main**,**Init**,other ones etc)

To open the function you Double click on it and you will see one new node there **RETURN** and one more Dynamic button .



Function I/O menu is only available to functions that are not **INIT** and **MAIN** and when you open it you can create function **ARGUMENTS** and **RETURNS:**

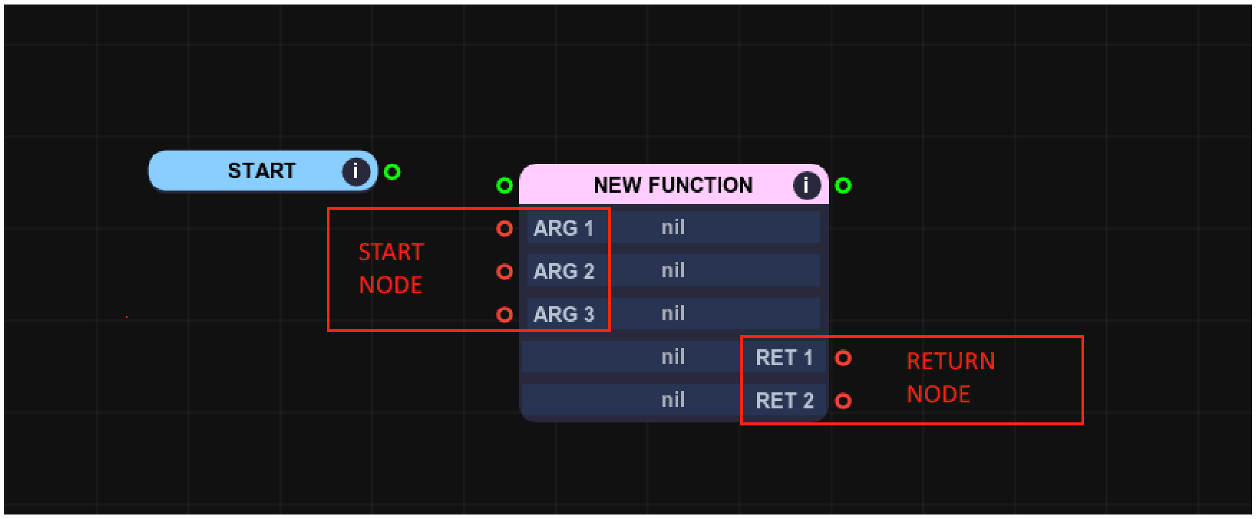


ARGUMENTS	
INTEGER	ARG 1
INTEGER	ARG 2
INTEGER	ARG 3

RETURNS		
INTEGER	1	RET 1
INTEGER	2	RET 2

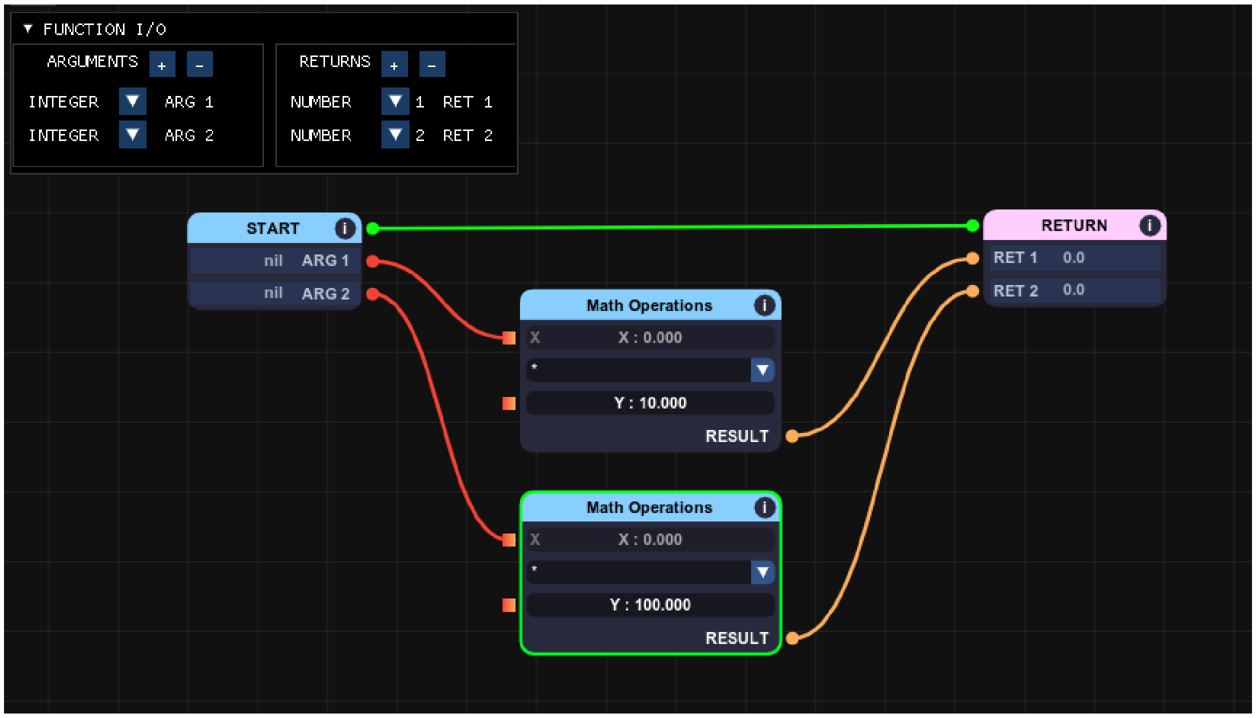
START node will get new inputs and **RETURN** also.

What that new inputs do is pass data connected **IN** the function and **OUT** of it which you can see when you drag your new function somewhere. Lets do that in **MAIN:**



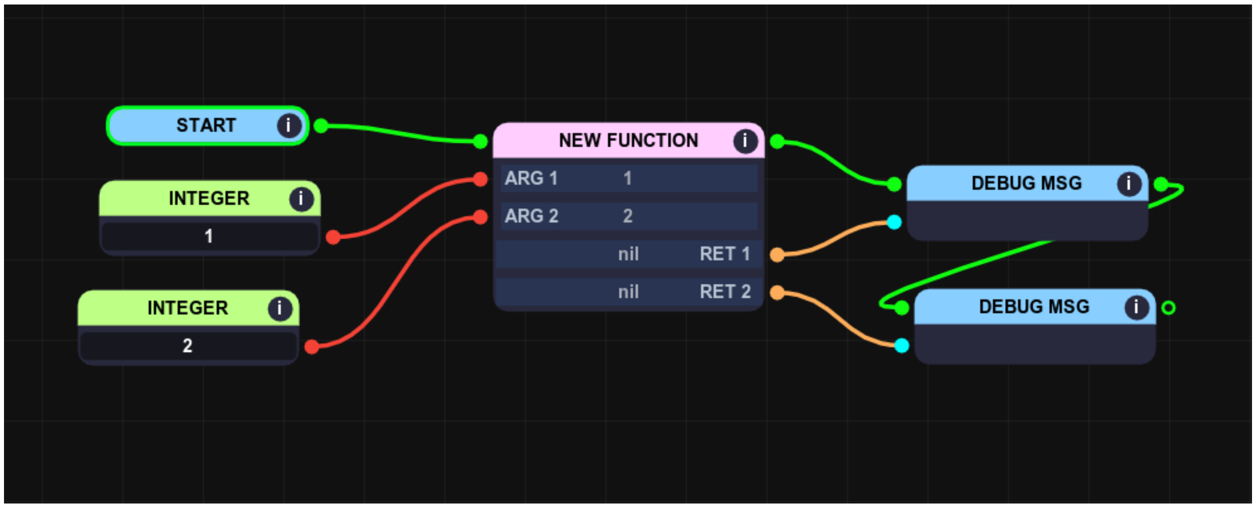
Anything connected to function INPUTS will be passed to START NODE and everything that is connected to function OUTPUTS will receive data from RETURN NODE.

Lets do some very basic math with it:

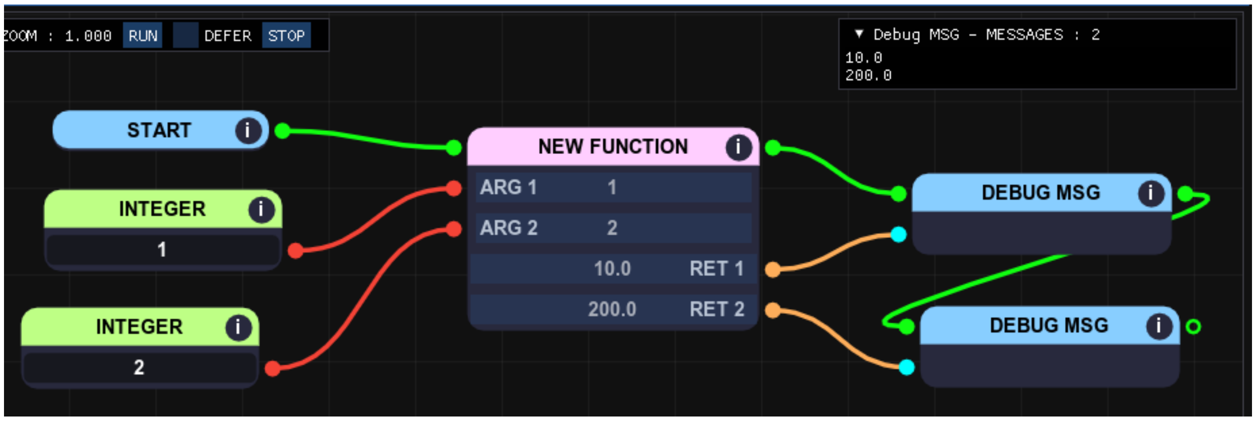


Change number if **ARGUMENTS** and **RETURNS** and set **RETURNS** to be **NUMBER/FLOAT.**

We will multiply each input with 10 and 100 and pass it to return and our function in **Main** connect it this way:

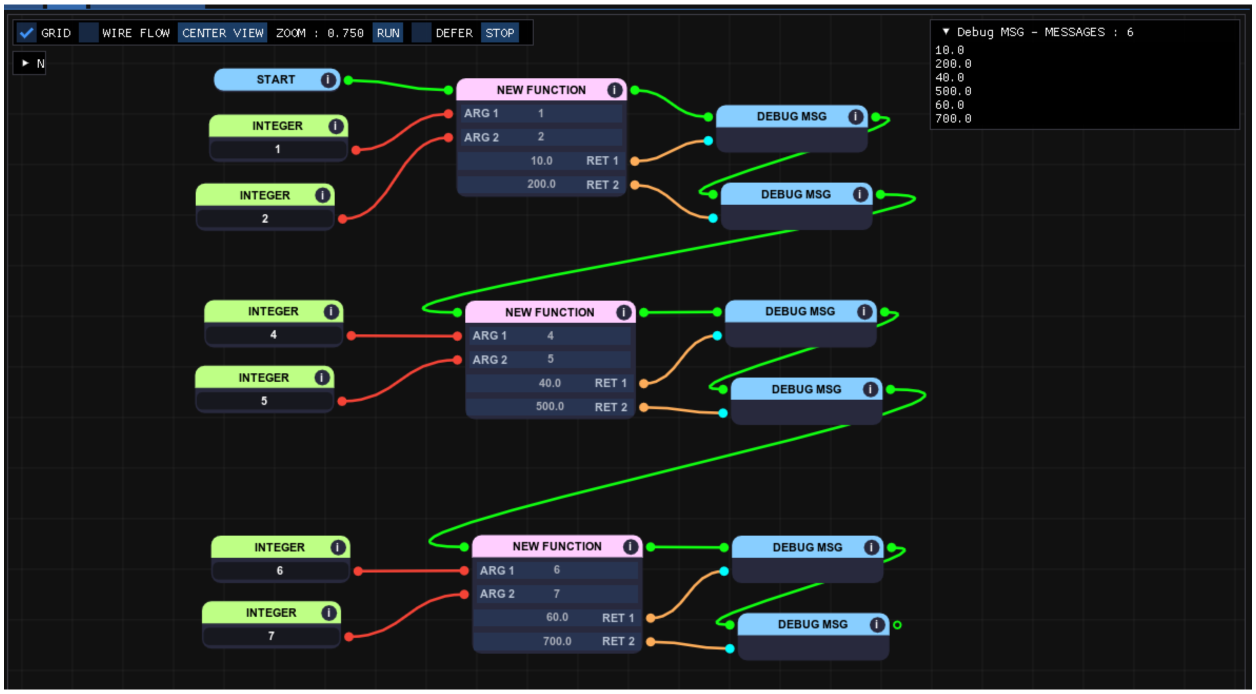


When you run the script the output will be like this:



```
▼ Debug MSG - MESSAGES : 2
10.0
200.0
```

But the fun part now is you can reuse the same function with other values:

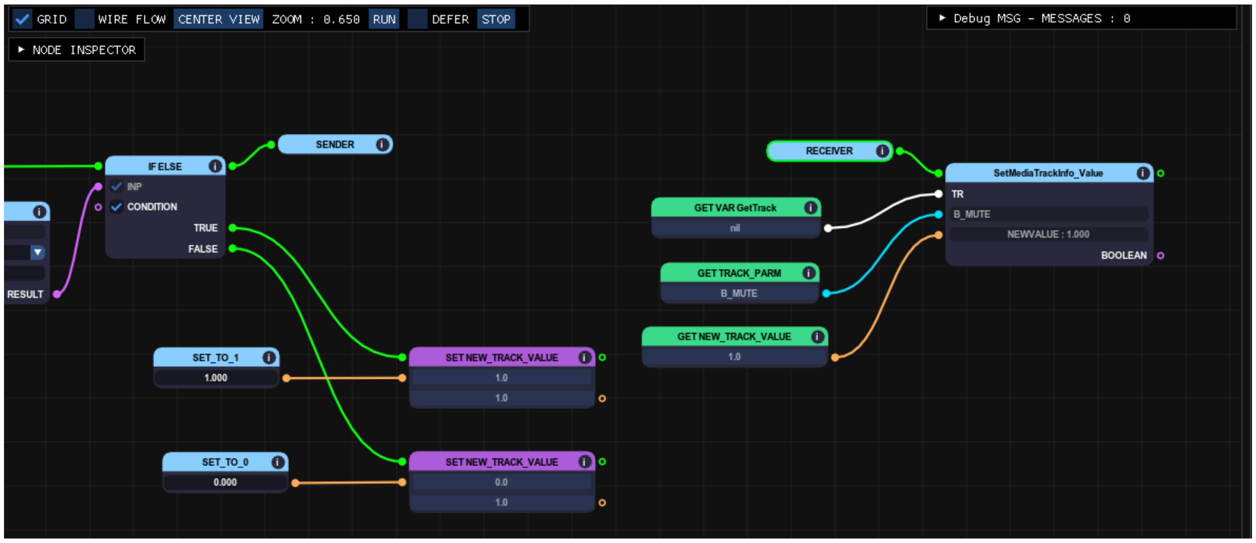


```
▼ Debug MSG - MESSAGES : 6
10.0
200.0
40.0
500.0
60.0
700.0
```

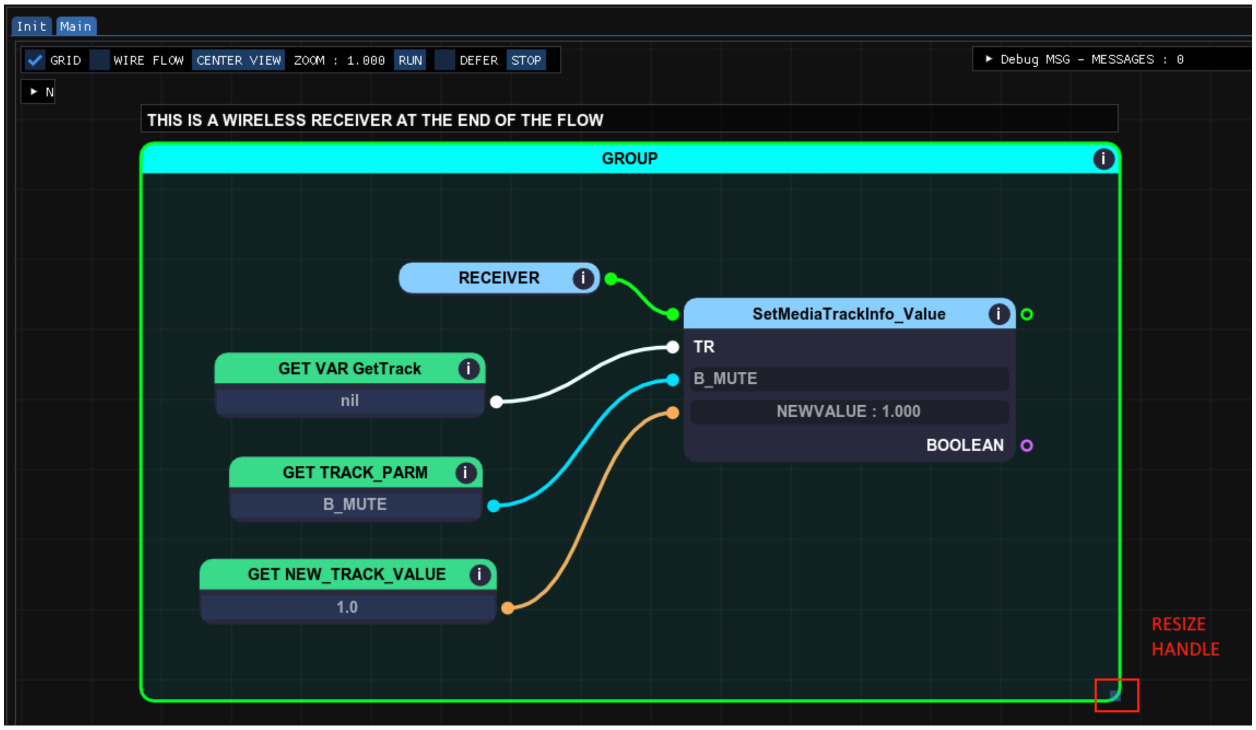
As you see you made a dynamic function which you can now reuse.

UTILITY NODES

Wireless nodes is self explanatory, you connect **SENDER** to one end and **RECEIVER** at other end. It only works for **RUN** pins. Wireless nodes come in pair and are also deleted in pairs



Group node makes a container for easier managing or commenting parts of the flow. All nodes inside will move together:

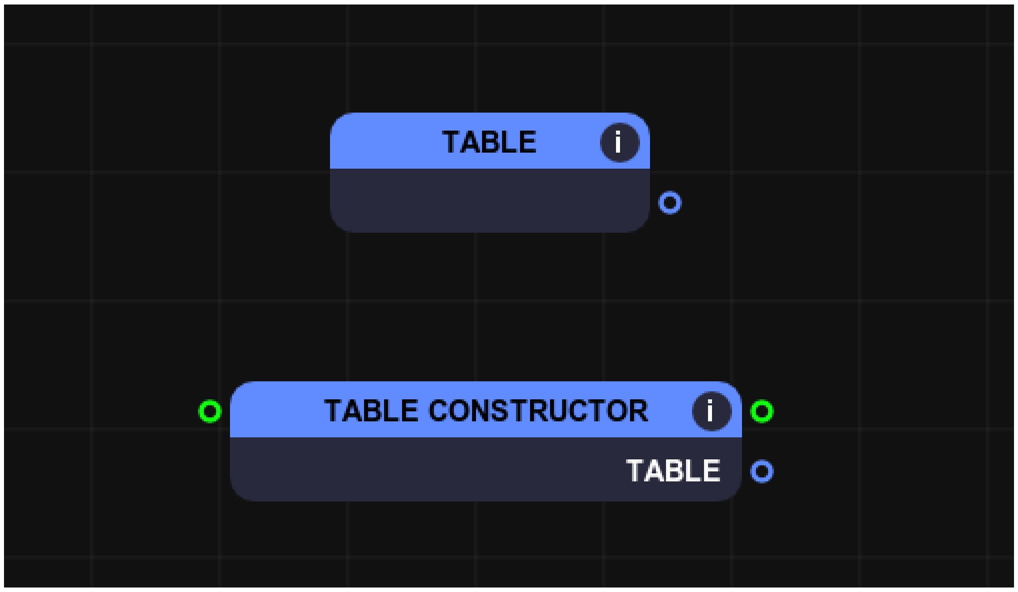


Every node has its own comment section which can be toggled ON or OFF with "**i**" button in the top right

TABLE NODES

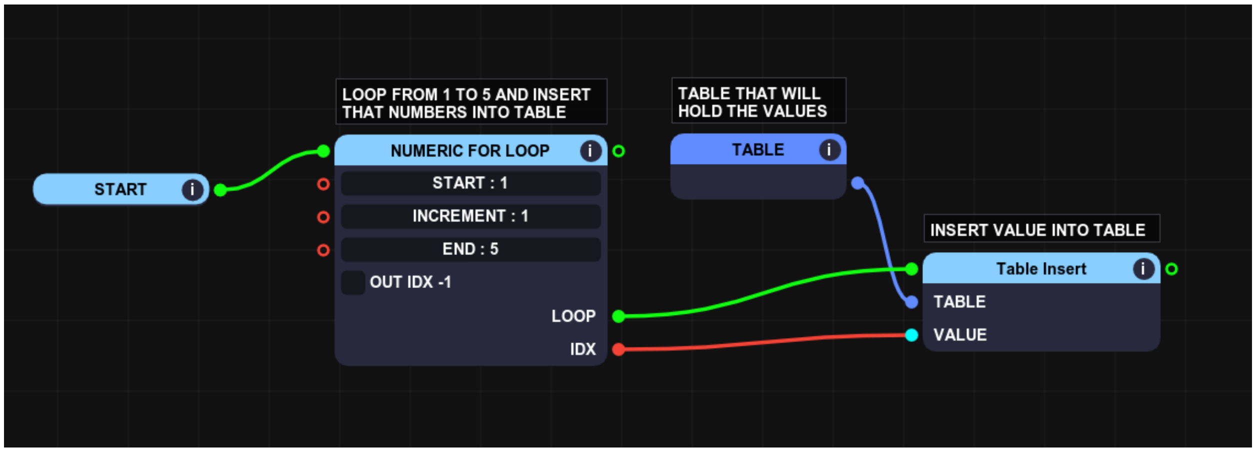
There are two types of **Table** nodes:

1. Pure table
2. Table Constructor



The difference between them is **Pure Table** on its own cannot have fields. That's why **Table Constructor** is there to help.
Lets try inserting, accessing and removing something from **Table**:

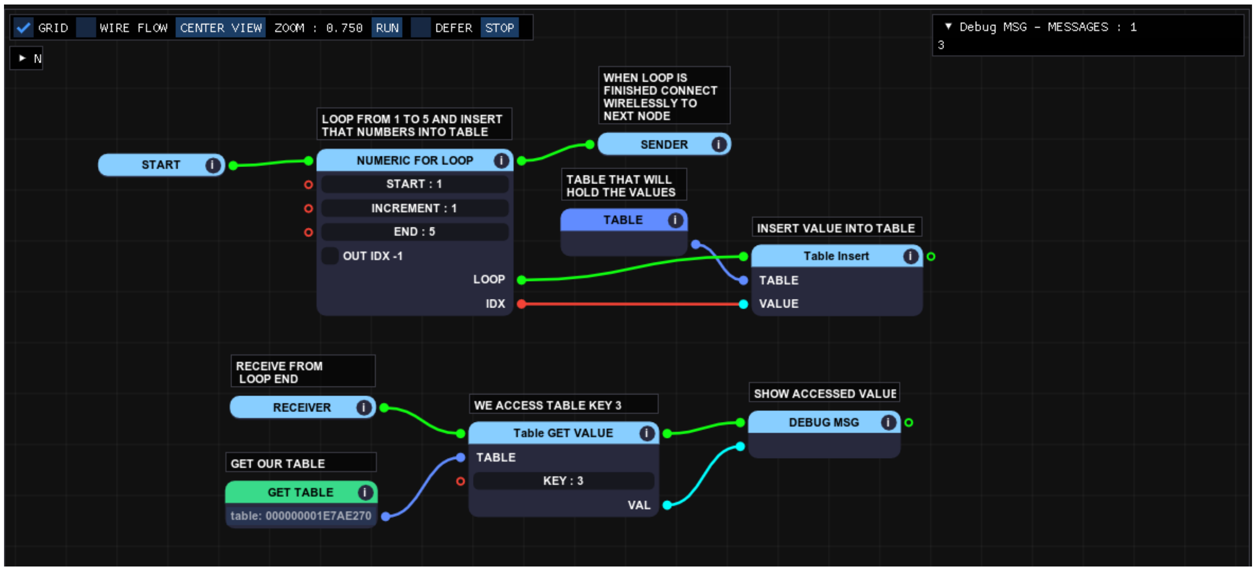
Inserting:



NOTICE : We unchecked OUT IDX-1 since we want to index table lua preferred way from 1

Accessing:

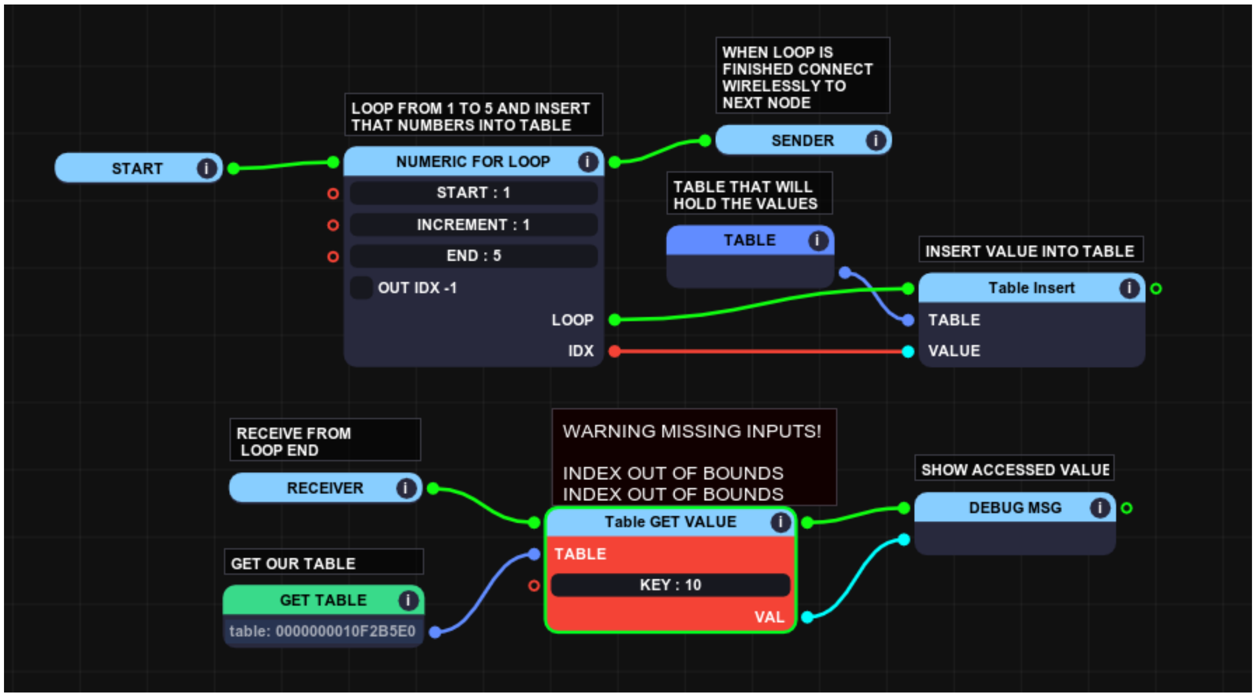
To retrieve values from the table it can be done in few ways from accessing single field to iterating every key



```
▼ Debug MSG - MESSAGES : 1
3
```

As you can see in **DEBUG MSG** View the **Value** in the **Key 3** is **3** which is correct.

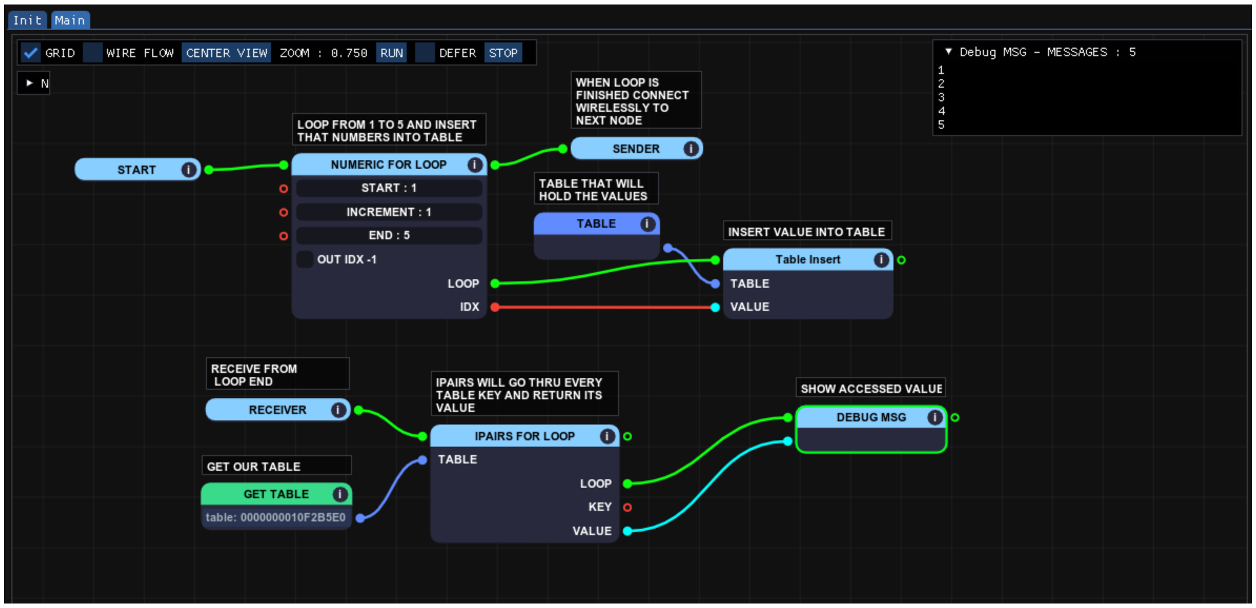
Scripts Error Catching Mechanism prevents you from accessing Keys which do not exits. Lets try to **Access Key 10:**



```
WARNING MISSING INPUTS!

INDEX OUT OF BOUNDS
INDEX OUT OF BOUNDS
```

Second way to access ALL keys is to iterate them all:

IPAIRS:



```
▼ Debug MSG - MESSAGES : 5
1
2
3
4
5
```

LOOP FROM 1 TO 5 AND INSERT THAT NUMBERS INTO TABLE

WHEN LOOP IS FINISHED CONNECT WIRELESSLY TO NEXT NODE

TABLE THAT WILL HOLD THE VALUES

INSERT VALUE INTO TABLE

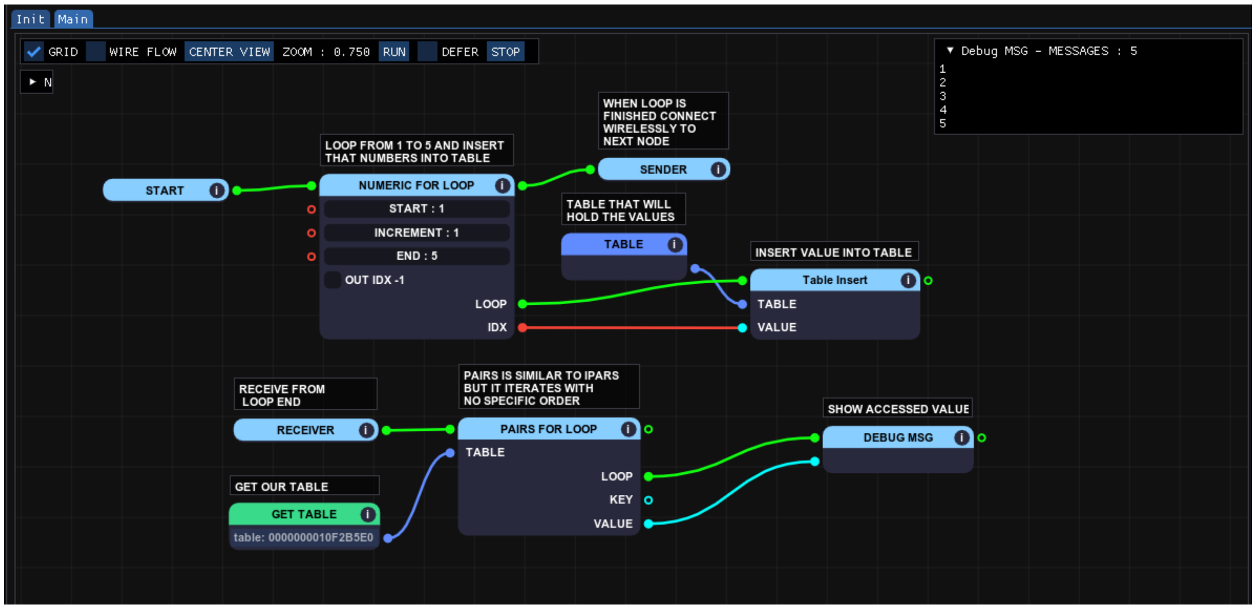
RECEIVE FROM LOOP END

IPAIRS WILL GO THRU EVERY TABLE KEY AND RETURN ITS VALUE

GET OUR TABLE

SHOW ACCESSED VALUE

PAIRS:



```
▼ Debug MSG - MESSAGES : 5
1
2
3
4
5
```

LOOP FROM 1 TO 5 AND INSERT THAT NUMBERS INTO TABLE

WHEN LOOP IS FINISHED CONNECT WIRELESSLY TO NEXT NODE

TABLE THAT WILL HOLD THE VALUES

INSERT VALUE INTO TABLE

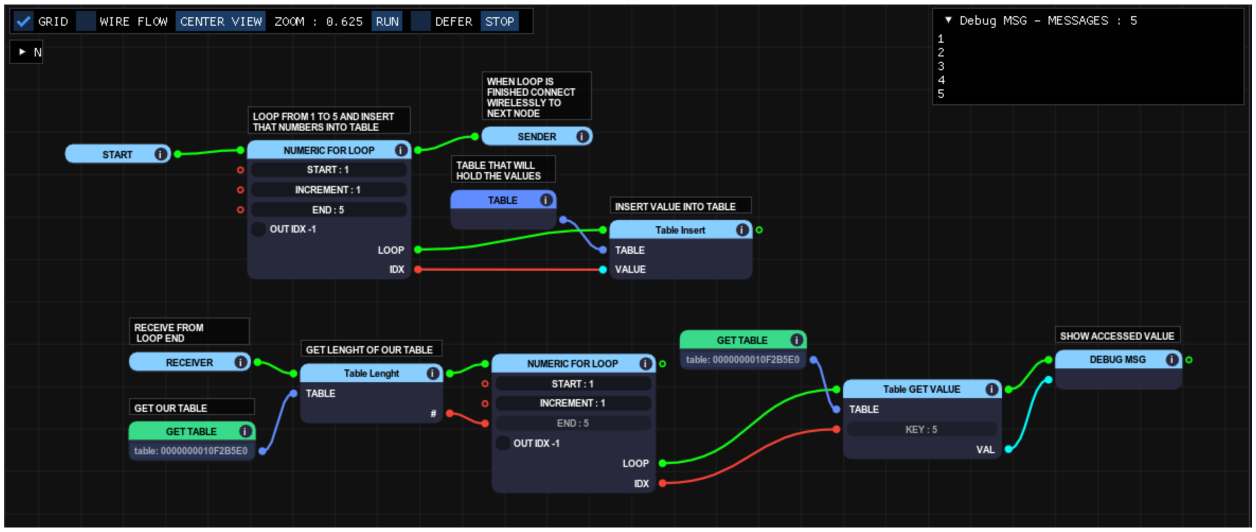
RECEIVE FROM LOOP END

PAIRS IS SIMILAR TO IPARS BUT IT ITERATES WITH NO SPECIFIC ORDER

GET OUR TABLE

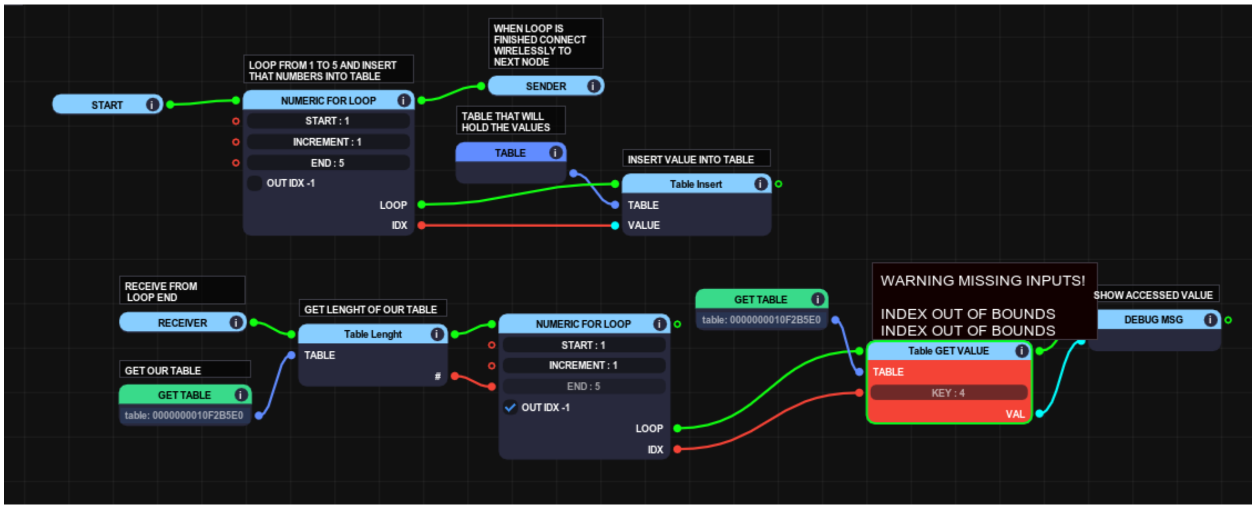
SHOW ACCESSED VALUE

FOR LOOP -> TABLE LENGTH:



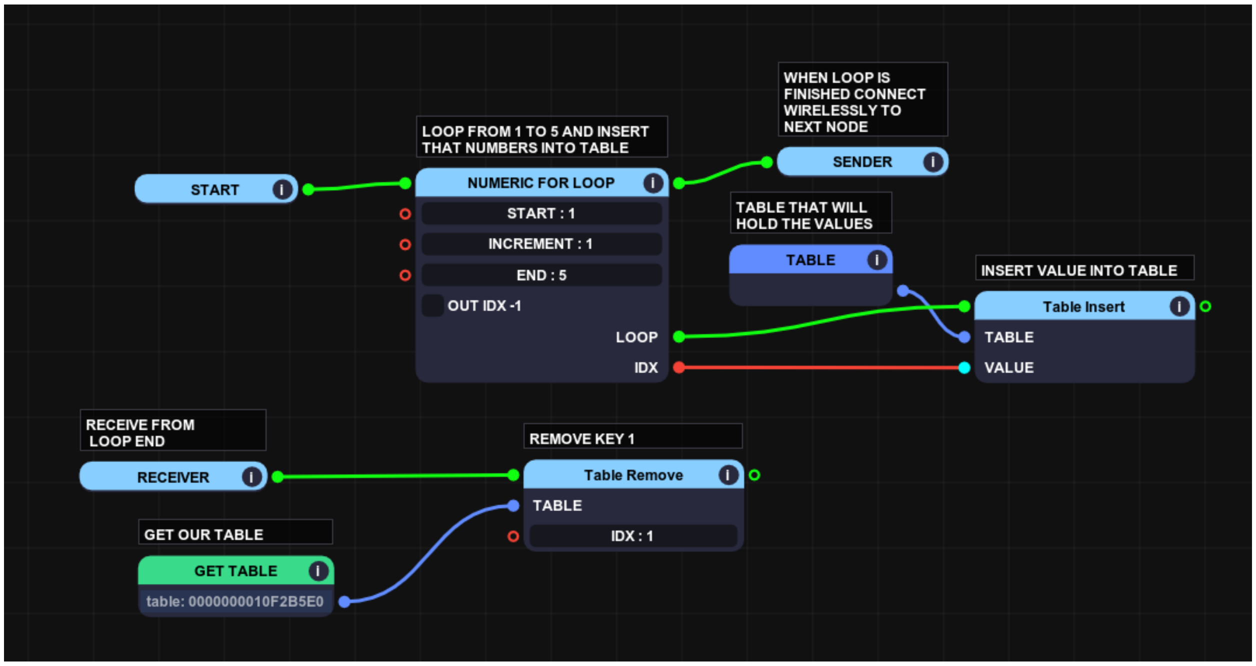
```
▼ Debug MSG - MESSAGES : 5
1
2
3
4
5
```

NOTICE : HERE WE ALSO UNCHECKED OUT IDX-1 SO WE CAN INDEX THE TABLE FROM 1. LEAVING IT CHECKED WILL TRIGGER ERROR CATCHING SINCE THE FIRST INDEX WILL BE 0 THAT DOES NOT EXIST IN THE TABLE!

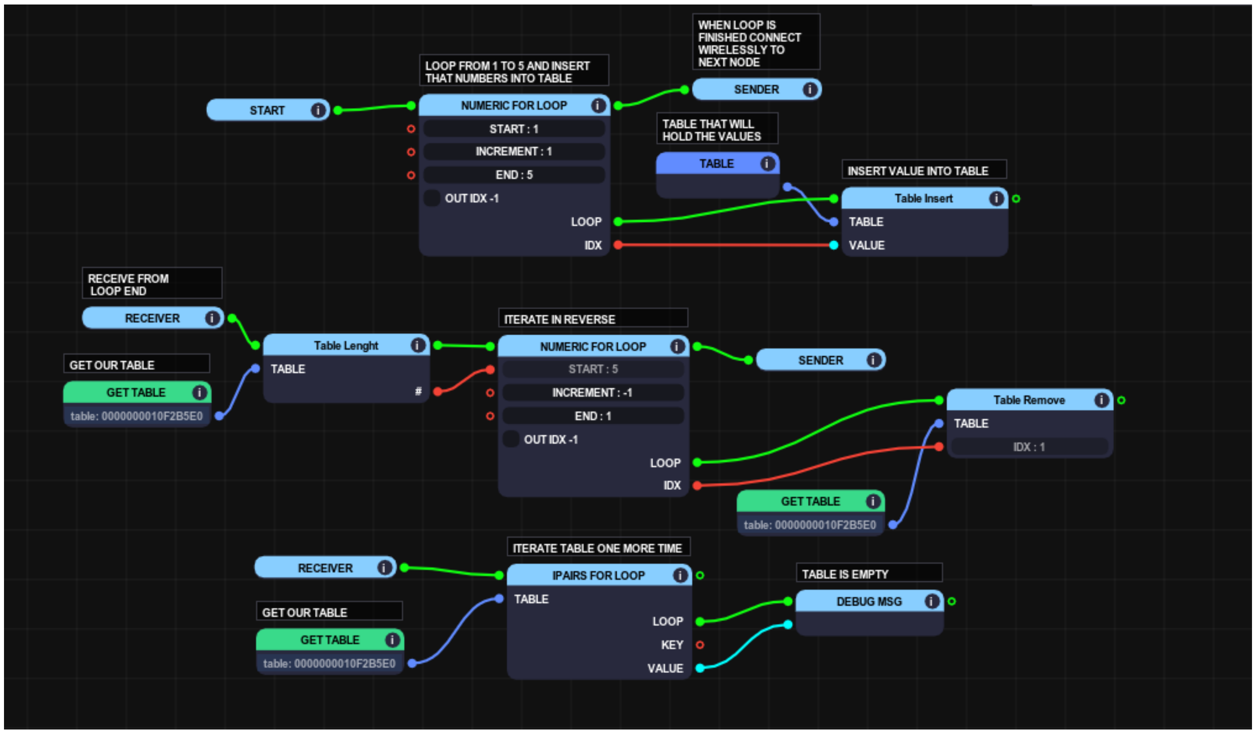


```
WARNING MISSING INPUTS!

INDEX OUT OF BOUNDS
INDEX OUT OF BOUNDS
```

Removing is simple as Inserting and Error Catching will prevent removing keys that don't exist:



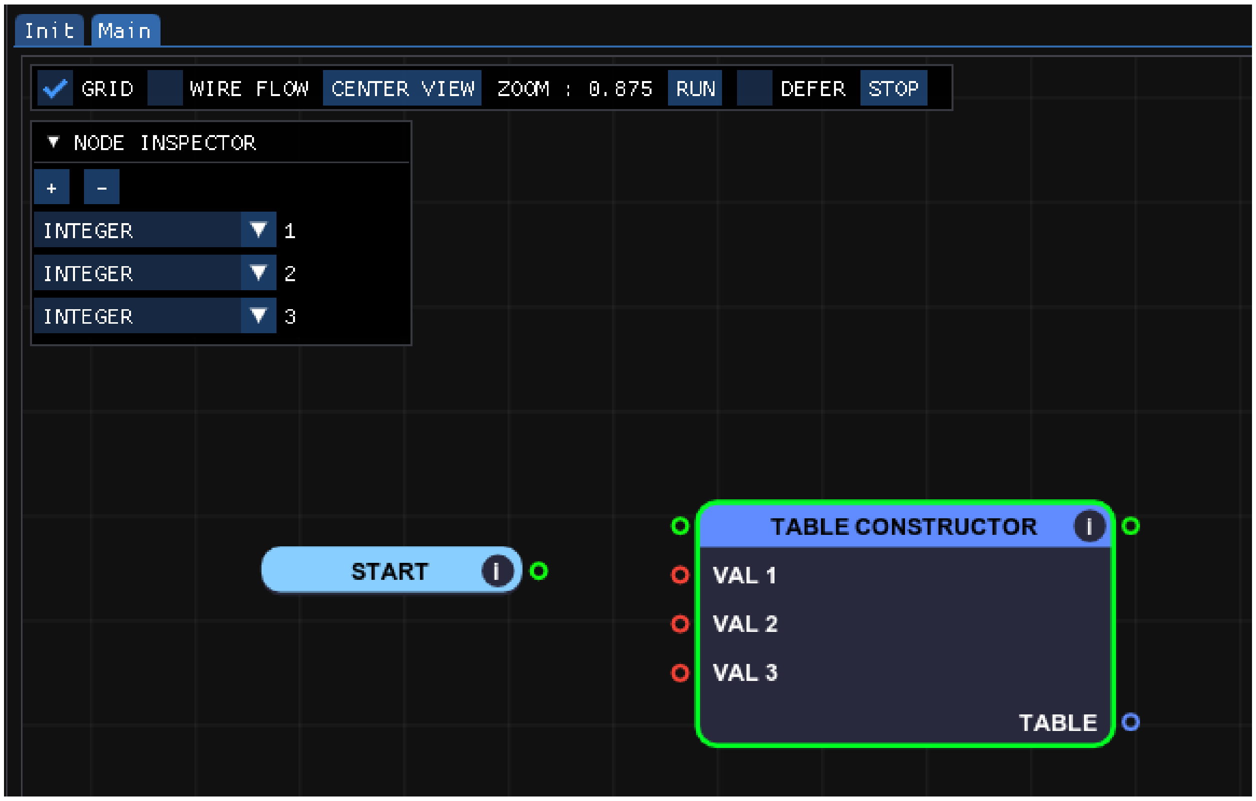
Or looping thru whole table and remove all keys **(WARNING LITTLE ADVANCED OPERATION)**



NOTICE: ONE LUA TIP IS WHEN YOU WANT TO DELETE THE TABLE VALUES YOU DO IT IN REVERSE ORDER. DOING IT IN NORMAL ORDER WILL TRIGGER ERROR SINCE YOU ARE REMOVING KEY AND DECREASING TABLE LENGTH BUT YOU STILL LOOK FOR INDEXES WHICH ARE REMOVED LATER

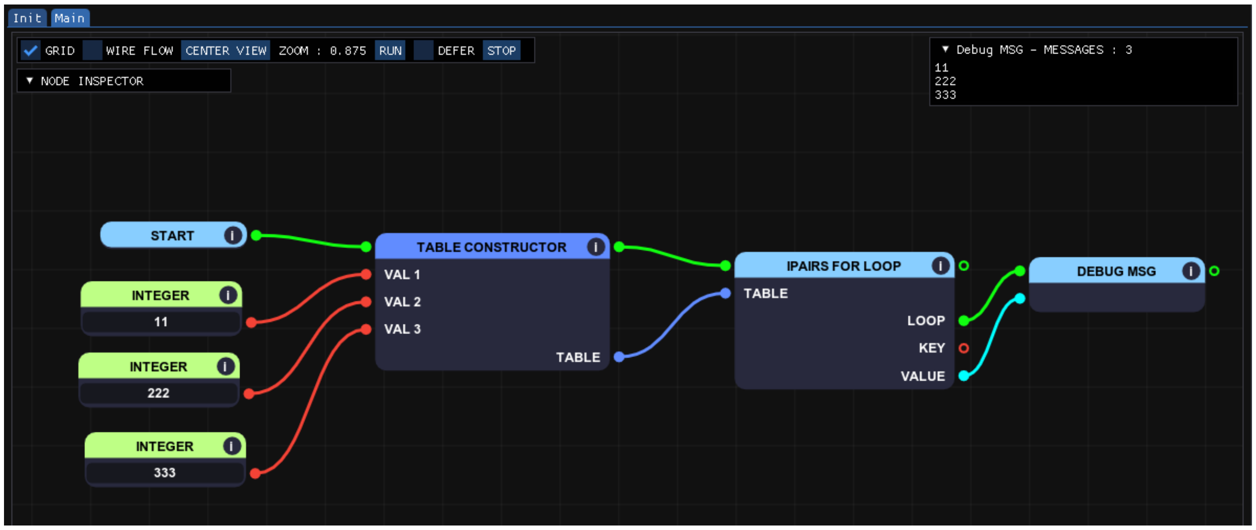
TABLE CONSTRUCTOR

To have some predefined keys we use TABLE CONSTRUCTOR. Adding Keys to it is done via **Node Inspector** in Top Left corner. Select the node and open the Inspector:



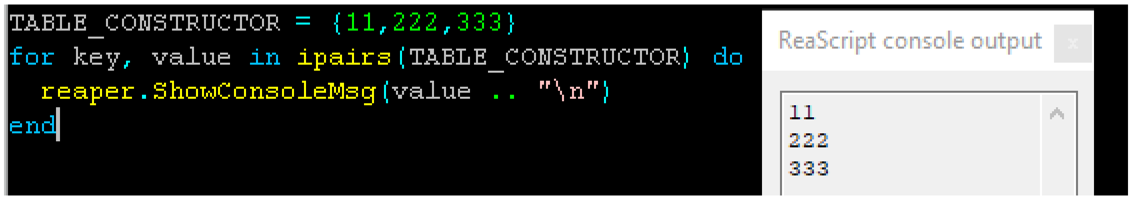
We added 3 keys which will hold **INTEGER** values. You can Change the type in the drop down list.

Now you can use Table constructor as a **Table on its own:**



```
▼ Debug MSG - MESSAGES : 3
11
222
333
```

This is equivalent to next LUA code:

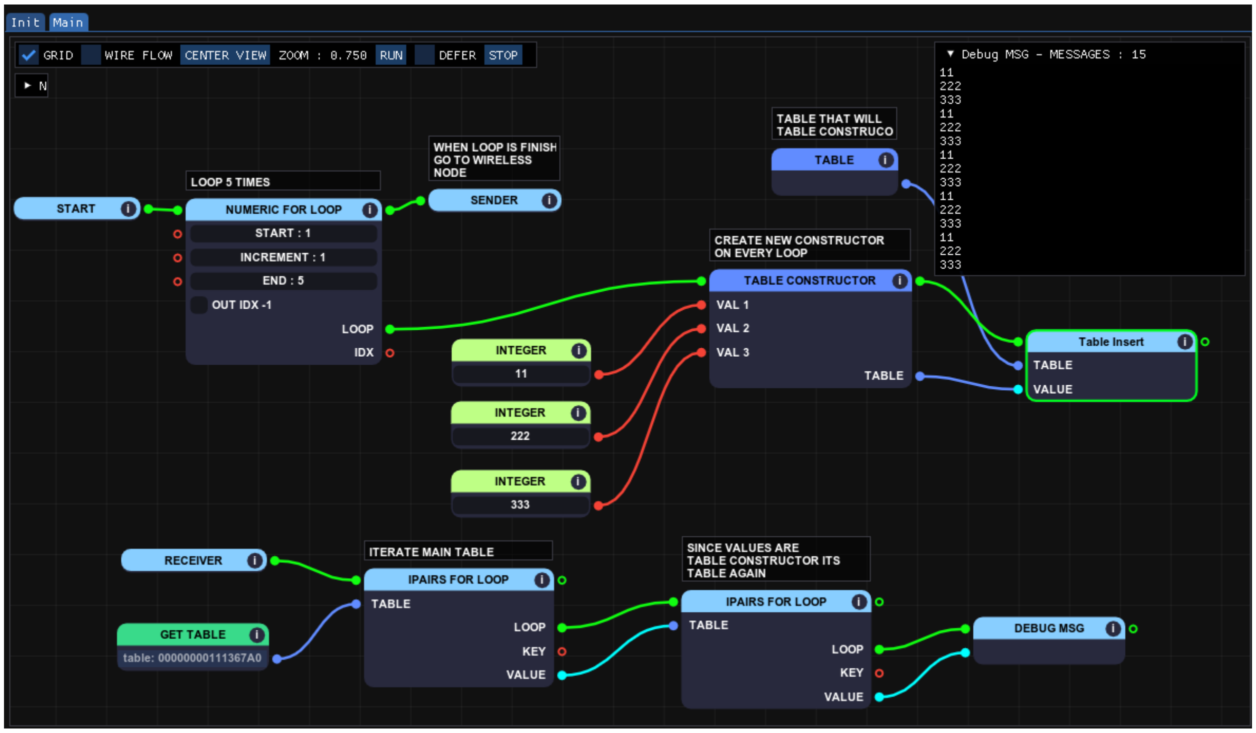


```
TABLE_CONSTRUCTOR = {11,222,333}
for key, value in ipairs(TABLE_CONSTRUCTOR) do
  reaper.ShowConsoleMsg(value .. "\n")
end
```

ReaScript console output

```
11
222
333
```

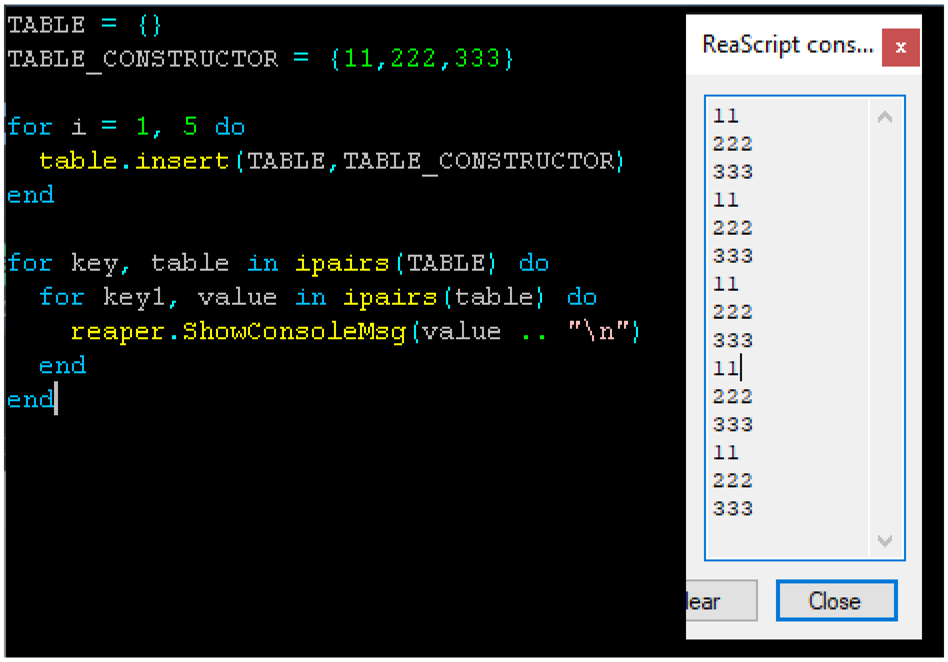
For more advance uses it is used for nested tables which you need **TABLE** node also:



```
▼ Debug MSG - MESSAGES : 15
11
222
333
11
222
333
11
222
333
11
222
333
11
222
333
```

We inserted **5 Table Constructors** each containing **3 values**. Result is **5 tables** with **3** values which can be confirmed by **DEBUG MSG** View

This is equivalent to next LUA code:



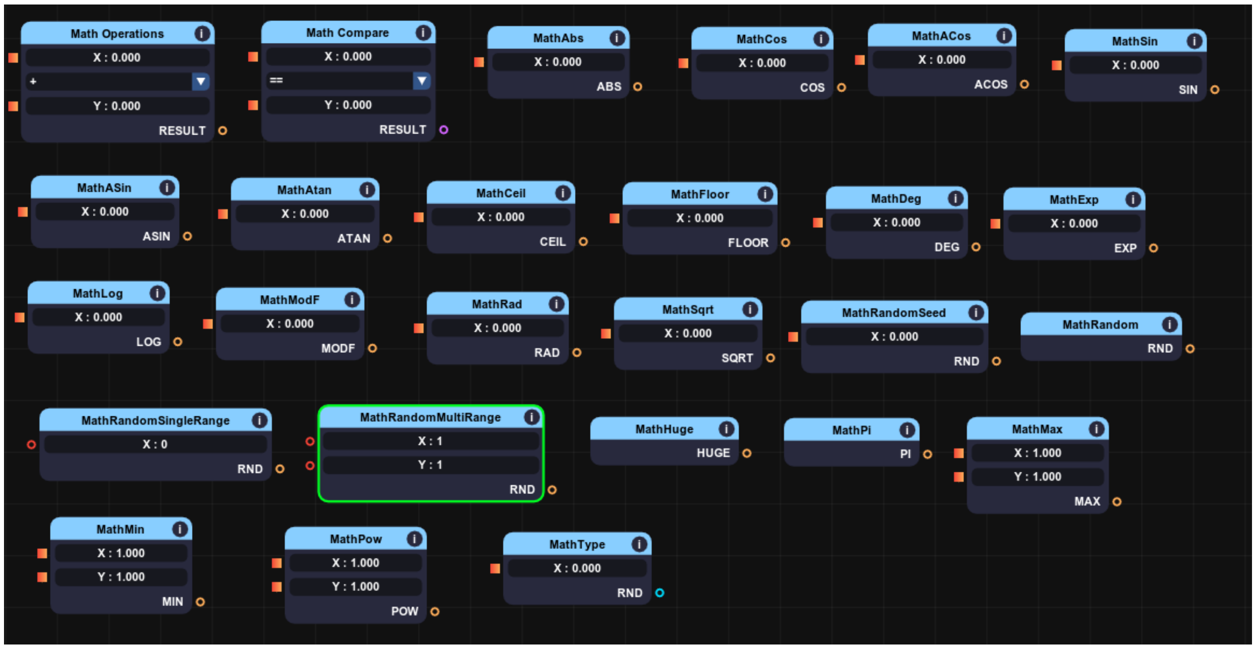
```
TABLE = {}
TABLE_CONSTRUCTOR = {11,222,333}

for i = 1, 5 do
  table.insert(TABLE,TABLE_CONSTRUCTOR)
end

for key, table in ipairs(TABLE) do
  for key1, value in ipairs(table) do
    reaper.ShowConsoleMsg(value .. "\n")
  end
end
```

```
11
222
333
11
222
333
11
222
333
11
222
333
11
222
333
```

As you can see the results are identical

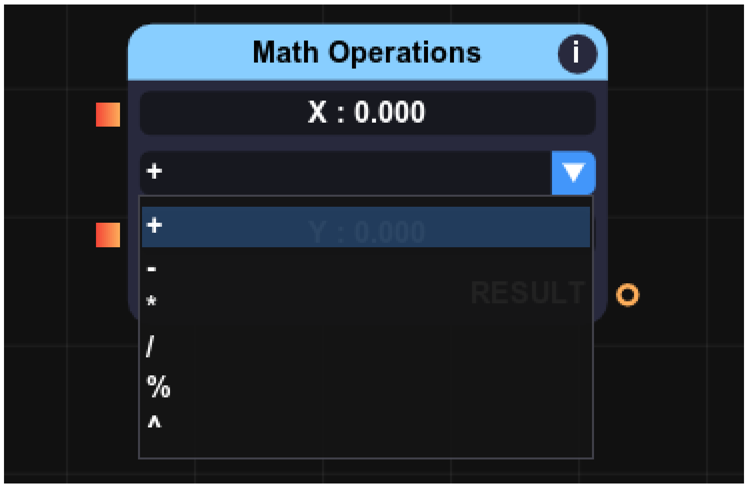
MATH NODES

Script provides (I think) whole LUA standard library. Unlike most of the nodes these ones don’t have **RUN** outputs.

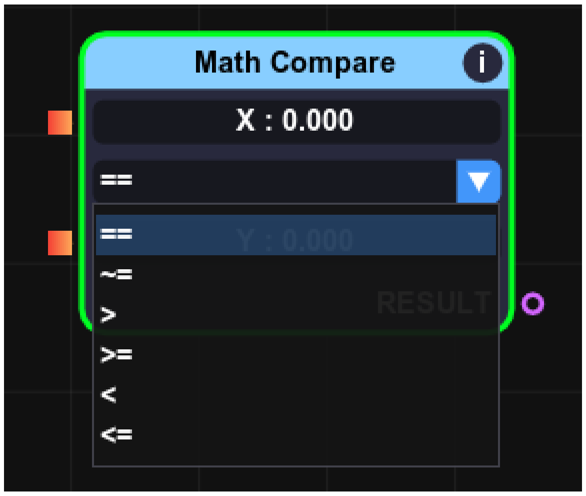


Math nodes accept both **INTEGER** and **FLOAT/NUMBER** values. Just few of them are exclusively **INTEGER (Math Random Single and Multi Range).**

Also little red/orange pin indicates that it can receive multiple inputs. **Math Operation** is used to do standard math operations:

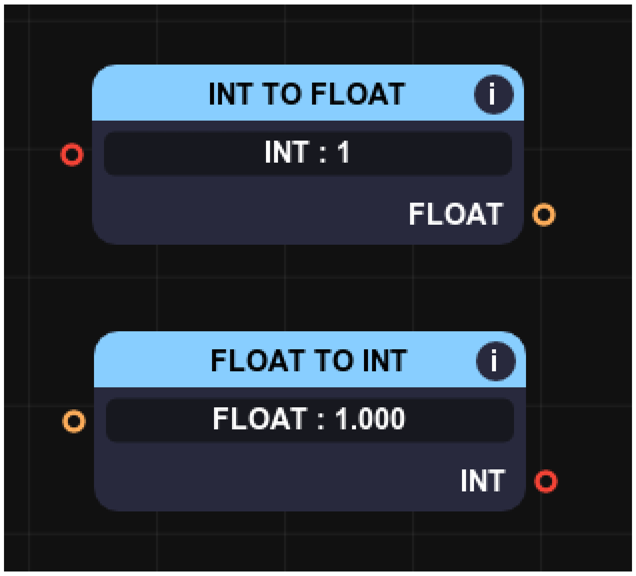


Math Compare outputs **BOOLEAN** to use it with **IF-ELSE** nodes



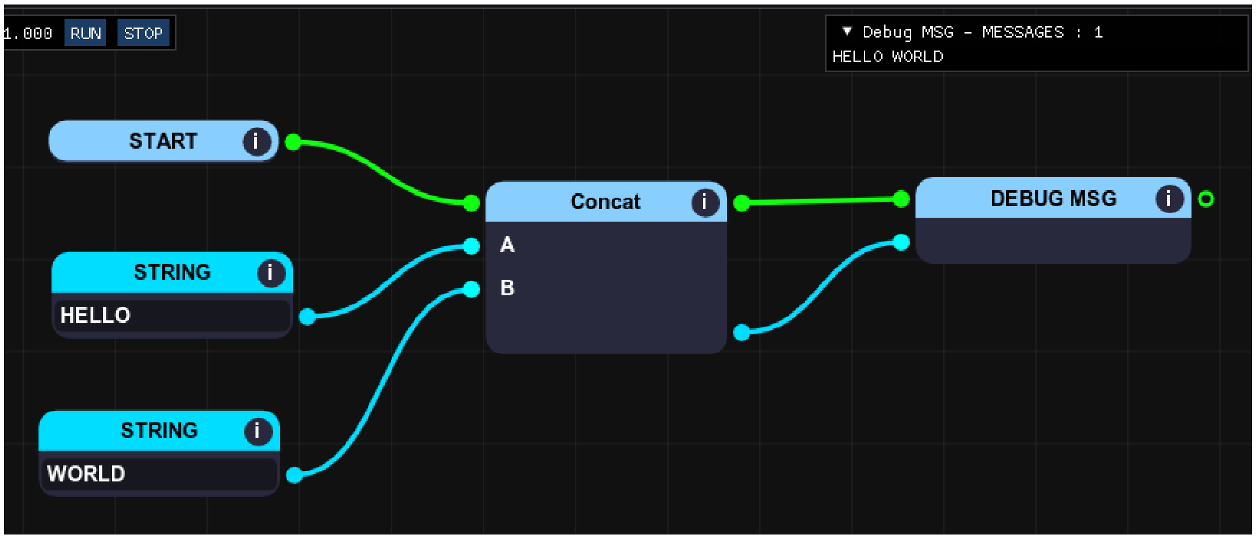
Since they don't have **RUN** pins like other nodes MATH nodes are auto-calculating meaning what ever you connect to them it will automatically calculate the output (API nodes need to run first in order to output the value).

If you need the result to be the **INTEGER** type you use **FLOAT TO INT** node or vice versa:



OTHER NODES IN STANDARD LIBRARY

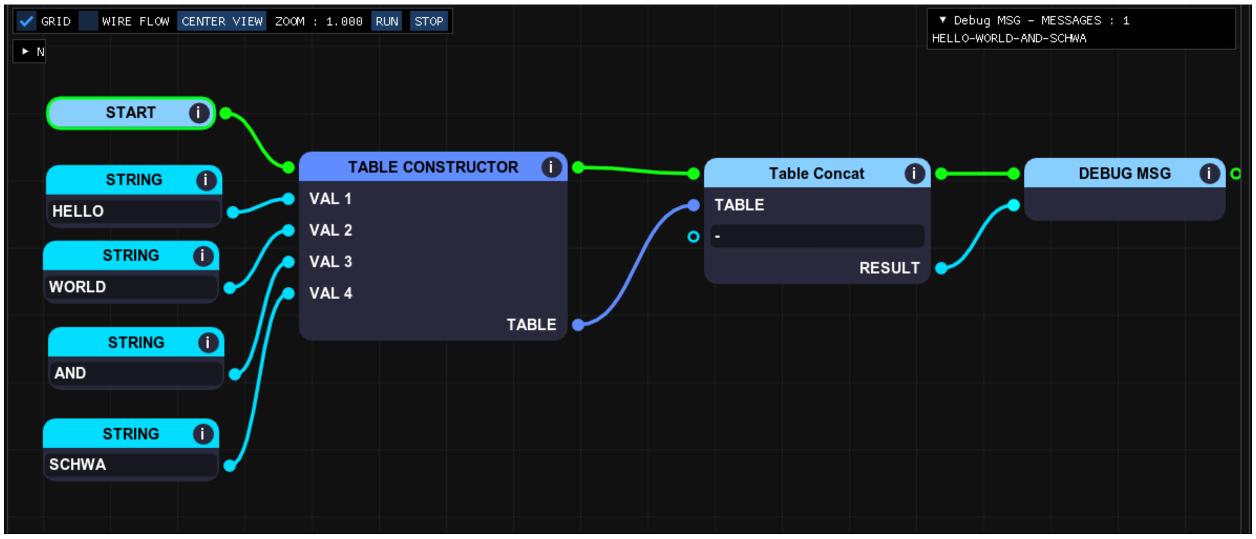
There are other nodes in the library like **CONCAT** to **MERGE/CONCATENATE** two strings for example:



```
▼ Debug MSG - MESSAGES : 1
HELLO WORLD
```

NOTICE : YOU CANNOT SEE IT UNFORTUNATLY BUT MY HELLO STRING HAS WHITESPACE THAT'S WHY THE OUTPUT IS SEPARATED. THIS NODE HAS NO DELIMITER (SYMBOL FOR SEPARATING STRINGS)

TABLE CONCAT does the same but with tables and has delimiter of choice:



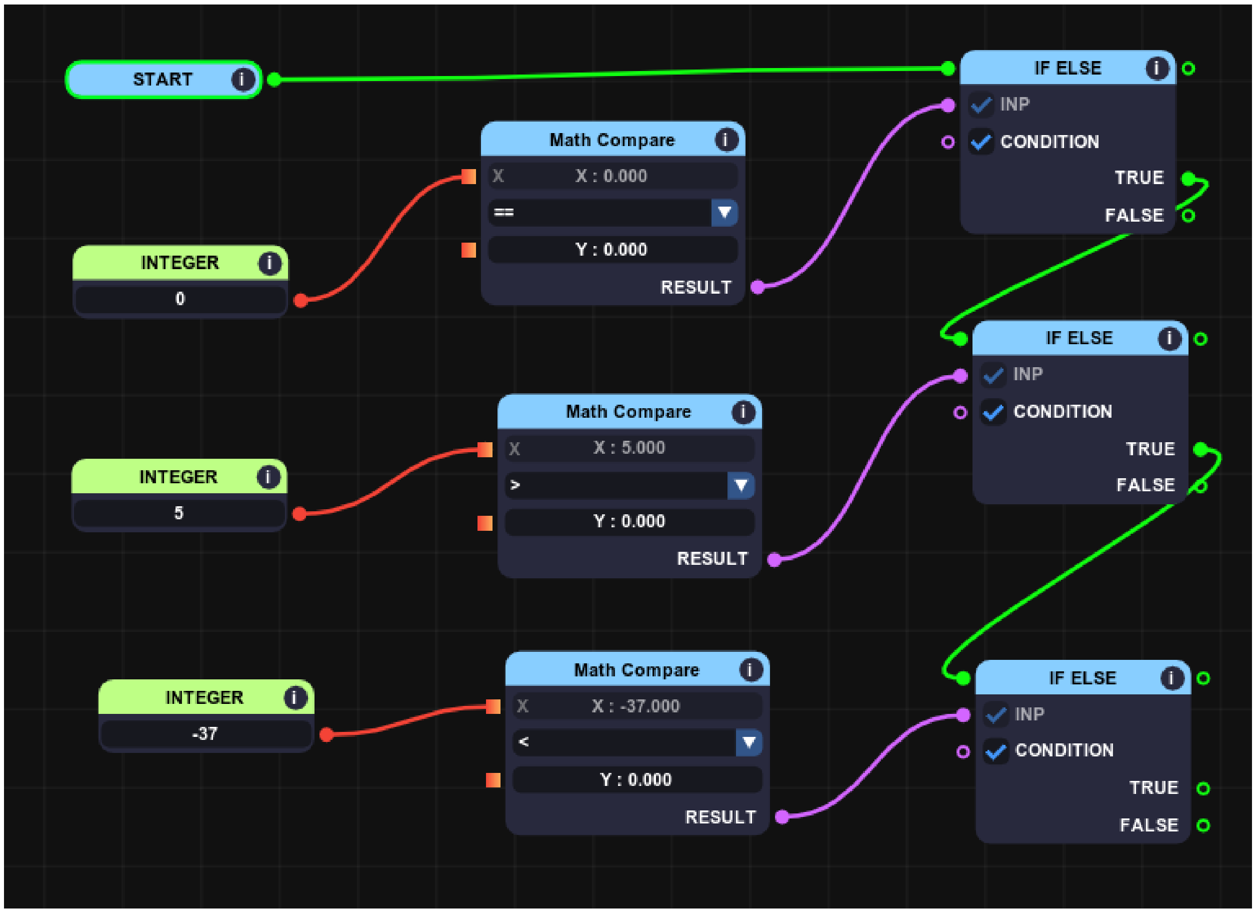
```
▼ Debug MSG - MESSAGES : 1
HELLO-WORLD-AND-SCHWA
```

There are also BITWISE operations, etc. etc.

SPECIAL NODES

I've forgot to show them in with **IF-ELSE** nodes but we will do them here now.

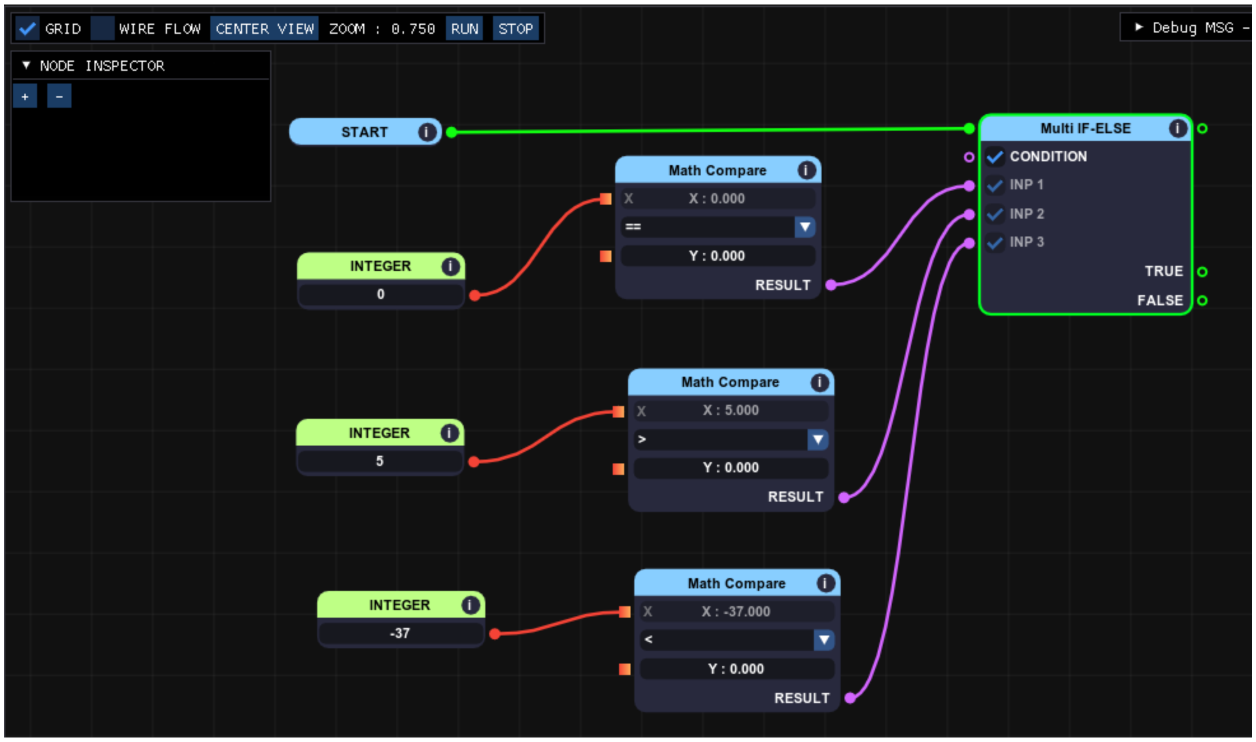
You will mostly run in a case where you will need to check multiple conditions in one go. This can be little awful to do with normal **IF-ELSE** because you would need a lot of them for example:



This is awful to work with and there are only 3 what if there are 5 or more?

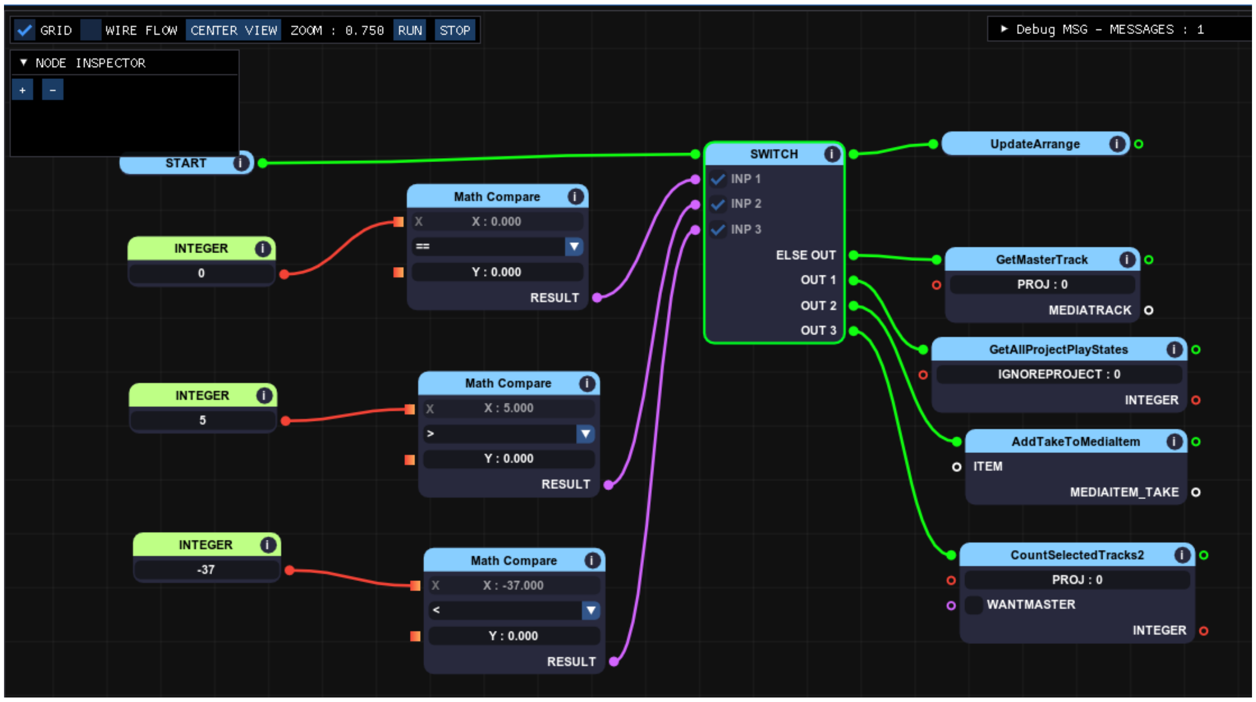
This is where **MULTI-IF-ELSE** comes into help.

You add/remove number of its inputs in **Node INSPECTOR**



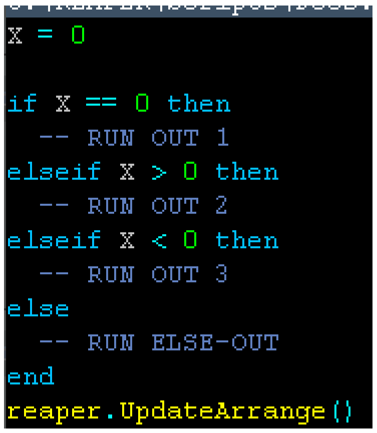
And everything looks a lot cleaner.

There is also a **SWITCH** if you need to **RUN MULTIPLE OUTPUTS:**



Like previous node this one also adds/removes inputs/outputs in **NODE INSPECTOR**

It is equivalent to next LUA CODE:



```
X = 0

if X == 0 then
  -- RUN OUT 1
elseif X > 0 then
  -- RUN OUT 2
elseif X < 0 then
  -- RUN OUT 3
else
  -- RUN ELSE-OUT
end
reaper.UpdateArrange()
```