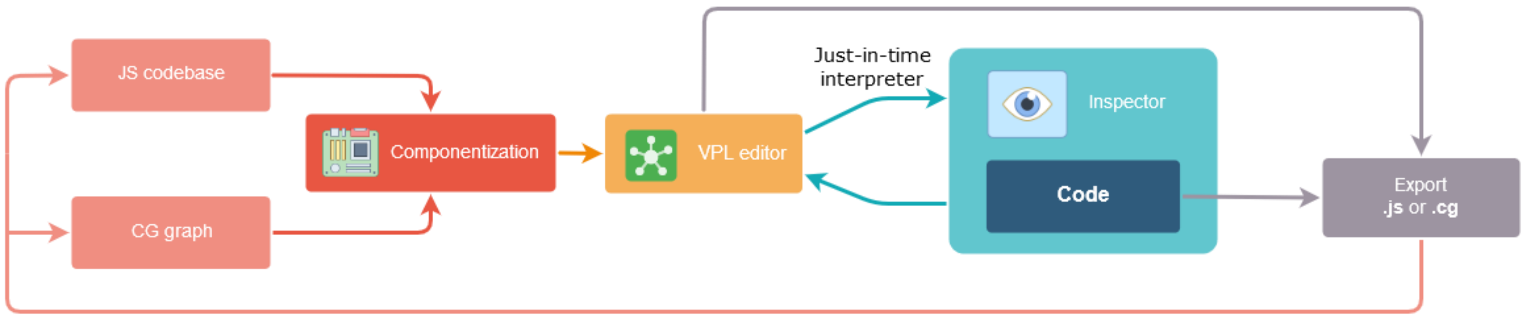
CODEGRAPH *2020* SPECS

This document aims to describe the architecture and mechanism behind Codegraph: both the core aspects of the visual programming interface as well as its interpreter.
It acts as functional specifications for the project.

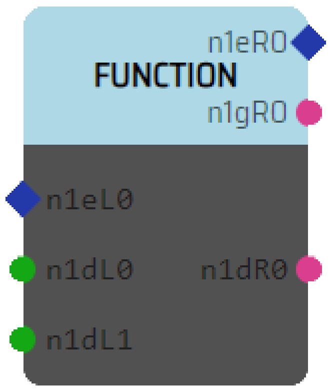
Contents

CG PHILOSOPHY: code reuse through componentization

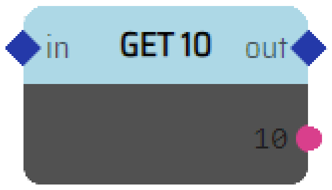
Codegraph can work with an existing code base, the objective is not to translate this code into a complete graph. Instead the aim is to rather to create high level components based on this code base. This allows us to use the codebase in the graph to build new components. Codegraph is not a dependency, it should allow anyone to get in and out very easily. Therefore, it needs to be able to compile node graphs into code.



THE BASICS: nodes, docks, execution flow

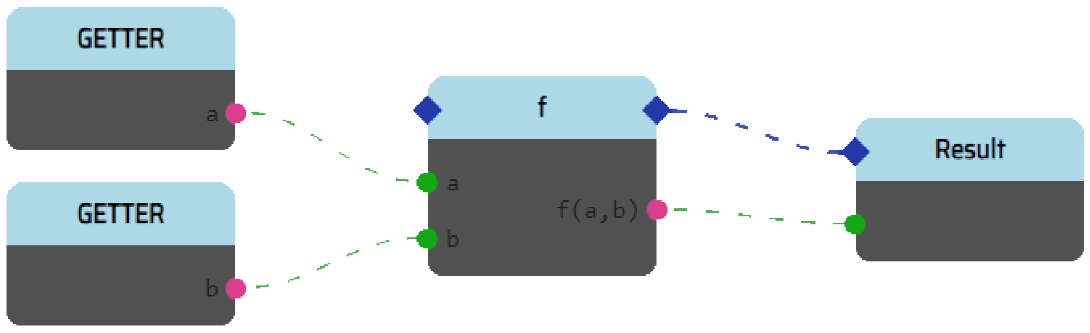


A **node** is a block which represents a function or a process. It has a head and body, but these sections are only a visual separation, they do not hold any semantic sense. Nodes can have **docks** which are essentially sockets from which we can link the node to other nodes. The side on which seats the dock, however, has its importance. Information comes from the left inside the node, then leaves from the right side. There are two fundamental types of docks **execution docks** (squared docks) and **data docks** (rounded docks). These are two types of information flows which are used in Codegraph. Execution controls the execution of the program while data is the actual information passed to feed the nodes.



Let's take the most basic node as an example. We will represent a simple function which has no arguments and returns a constant: 10. It will have a single data dock as output. It will also have two execution docks that will allow us to indicate when the node needs to be executed. We can link this node with other nodes to pass information through.

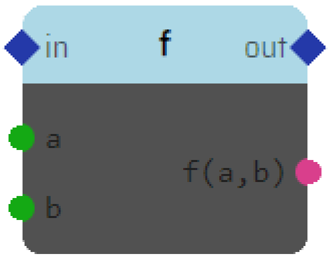
Below is an example of such graph, where we have two getter nodes passing their values to the node in the center which processes the two arguments and returns a value. The value is then relayed to a fourth node on the right, the end of the graph. Green links are data links. They link two data docks: an output data dock with one or more input data docks. Blue links on the other hand allow us to define the order of execution of the graph: literally setting which nodes are executed first, second, etc...



SOME FLEXIBILITY: dynamic node output

Note that some nodes *do not* have execution docks. This is because they have no process, i.e. the value they hold is retrieved from the current scope, not computed. If the result of a node is computed, it needs to have input and output execution docks in order to tell *when* it needs to compute it. For getters, the value will always be the last known value for that variable. Let's imagine we have a getter node that is used twice: for instance, by a first function then a second further down the graph. Now, imagine that the value of this variable has changed between the two accesses. In this case the getter node will have provided two different values because it was accessed the getter at two different moments in time. This will open to another discussion on **data propagation** and **resolving dependencies**.

COMMON NODES: function-nodes



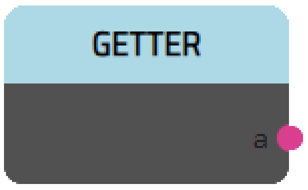
We can assert that every node has at least one exiting data dock because every function as a returning value (in some languages if no value is returned there will be a default value). Therefore, when defining a node, we need a function definition. This is the core component. A function can have arguments and will always return a value. The corresponding node will have as many entering data docks as there are arguments.

The node's process will be defined as `` `*process: (a,b) => f(a,b)*` `` where we imagine *f* is a function declared explicitly somewhere else. NB: No support for rest or spread operators.

The internal process of a given node has a list of references to the data docks and their values in order to execute and calculate its result.

A good proposal would be to have a base node which has no docks (no data docks nor execution docks). We can attach a process to it. This means adding data input docks for arguments and the **process' definition** which will have two layers. The first layer will compute the result of the function while the second layer returns a string function called on those arguments.

SPECIAL NODES: getter-nodes



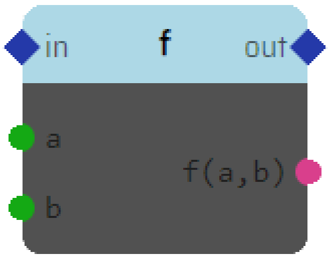
Another commonly used type of node is one which acts as a getter to a variable. It has no data input dock and a single data output dock. They don't work the same way function-nodes do: the value of `` `*a*` ``, will be fetched when it is accessed.

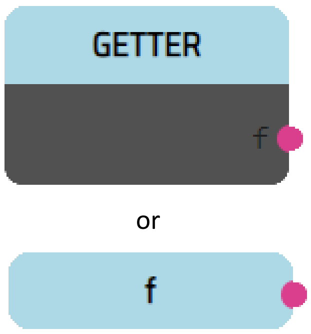
VERY SPECIAL NODES: control-flow-nodes

In some cases, nodes need to make some variables available to the outside scope. This is only the case for special nodes which have control on graph's execution. They're called **control flow nodes**. In this case, we will need more than one output data dock. However, this is an exception.
Also, control-flow-nodes have one input execution dock and can have more than one output execution dock. As they are usually able to redirect the flow of execution on a specific **route**.

FUNCTION vs FUNCTION CALL

Once defined, a function can either be used or passed:

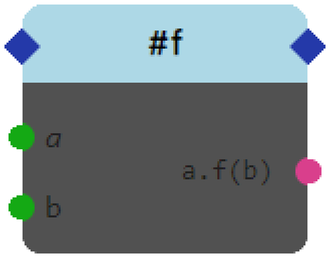




When passed, the function acts as a value. This feature is not required right now but might be in the future if want to pass a function as parameter like a *callback*. For now, an *if-then* approach is preferred.

We already have a proposal to implement *anonymous functions*. It will introduce a new kind of dock used to plug a function node directly into another node's input data dock.

PROPOSAL FOR OOP



To allow OOP, calling methods is like applying filters on an object. We can use the pythonic approach which consists of adding the *self* object in the argument list. This translates to adding an additional data input dock to set the value of this inside the method's scope. On the right, the node is applying method *f* on object *a* with parameter *b*.

This structure can be used to allow **function binding:** either on-call (an argument allows us to set the value of the bind) or by applying a bind before-hand on the function node.

NODE PROCESS: the API

We will describe the attributes that are given to the Node constructor. The most important part of a node's definition is its process. The two-layered function declaration and the input and output docks definitions, here named respectively params and the unique result dock:

```
process: {
   params: DockDefinition[],
   result: DockDefinition,
   function: Function,
   string: Function,
}
```

We call *DockDefinition* the set of attributes that are used to define a dock.

A proxy is created on the node to access an instance of **Process**, a class which is responsible with computing the node's result. The dock will be instantiated and passed to this Process object. At the same time, we must also add these docks in a *Set* inside the Node instance. It will be used to update the graph. We end with two attributes: *docks* and a proxy named *process*.

EXECUTION CONTROL: ROUTERS

We now need to deal with the execution docks. For a process there are two by default. So, these two need to be created, added to the list of docks and relayed to an instance of **Router**. This instance is attached to the node with a proxy, the same way we did with the *Process* instance.

How do we use the Router instance: how do Control-flow nodes manipulate it?

We might as well declare a function for the router with a special API to access information about the execution of the route. Ultimately, this function would return the output execution dock that needs activating. By default, i.e. for basic function-node, a default router will be used.

Let's see study the router for a conditional block. We need to think about what the function needs to have access to in order to decide: there are two possible output routes (the ''true'' OED and the ''false'' OED), which one to activate? The answer depends on one of the input data docks, the one called ''condition''. The router's function needs to check if it's truthy or falsy and choose the dock accordingly. For node loops, it is trickier. The node's router saves its state and decides depending on the ''end'' input data dock whether to execute the ''body'' or the default OED.

The router might also need to make some variables available. This is generally the case for loop nodes where an index value will be provided as an ODD. This will simply be specified with a *getter* attribute which will list all the variable.

```
router: {
   params: DockDefinition[],
   getters: GetterDefinition[],
   function: Function,
}
```

GETTERS: the API

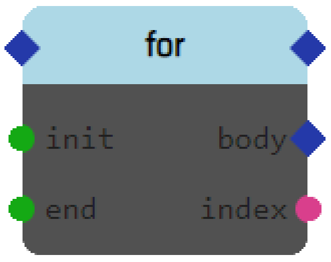
Lastly there are getters, which unsurprisingly will have the same definition we used for routers:

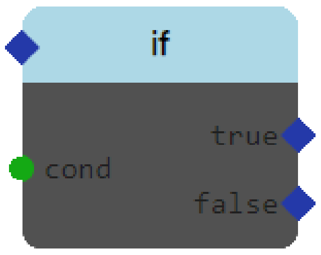
```
getters: GetterDefinition[]
```

Every ODD is connected to either a process module or a getter module. This module allows it to find its value: the former will try to compute it with the node's parameters and the latter will retrieve the value in the current scope's context.

CONTROL FLOW NODES: the routers

Some more details on control-flow nodes:





These types of nodes have multiple output execution docks. This means there's a decided to be made. The *if* node will look at the condition pick the right dock and redirect the execution to the connected node.

The *for* node on the other hand will need to keep track of the loop's progress and execute the dock ''body''. When executions of this path end up on a leaf (the last node of the loop's body), the information is relayed back to the *for* node to proceed with the loop. If the condition last index hasn't been reached, the body will be executed again. When the last index is reached, execution is redirected to the upper output execution dock (the default execution dock). Looping sections of the graph require the execution links to have a special payload describing the process that initiated its execution. This way, we can jump back to the initiator when the leaf is met.
Also, we notice we have an output data dock *index* which can be used by the nodes in the body. The value of *index* will be set and updated by the *for* node before each iteration. This variable will be scoped, meaning that nodes outside of the body won't have access to this value.

Every process, i.e. dock, has a scope attached to it. It allows to store and retrieve variables as well as create sub-scopes. This allows a given node to know what variables it has access to. Getter will retrieve variables; setters will store, or override variables and function will create new scopes.

DOCK VALUES: where are they coming from?

How do data docks get their values? An Input Data Dock (IDD) gets its value directly from the dock it is connected to. That will, of course, be an Output Data Dock (ODD). An ODD will get its value differently depending on its type. A *getter dock* will access the node's scope to retrieve its value. A *return dock* works differently because it will first look in the cache to see if the value ready for use.

Links do not hold specific information. It might be useful down the road to provide additional information for the user.

QUICK RECAP: node types

In total, there are five types of nodes:

Cq \ **Type**	**function-nodes**	**control-flow-nodes**	**getter-nodes**	**setter-nodes**	**operator-nodes**
	f-nodes	cf-nodes	g-nodes	s-nodes	o-nodes
Description	Functions which are dependent on execution sequence.	Can change the flow of execution based on params given.	Gives access to scope variables.	Gives ability to modify a variable's value inside scope	Function with no side effects i.e. independent on execution seq.
Router	**Default** router.	Has a function to determine which OED to activate.	No router.	**Default** router.	No router.
Process	Has a process.	Control-flow *specific* **default** process.	Getter-**default** process.	Setter-**default** process	Has a process.
Scope	**No access to scope**. Works as a module.	The router and process must have access to the *Scope*.	No access to scope.	No access to scope.	No access to scope.
Data docks	IDD: n ODD: 1	IDD: n ODD: n g-docks	IDD: 0 ODD: 1	IDD: 1 ODD: 0	IDD: n ODD: 1

🟧 Internally implemented
🟦 User implemented
🟥 Unused node feature

LEXICAL SCOPE: variable's reach

We need an object *Scope* which tracks variables in each context and gives tools to get and set their values as well as create new variables. When a getter dock is accessed it will get its value in the scope that is attached to it. Same with setter docks (equal to setter nodes) which will modify a variable's associated value.

Does everything inside the same canvas share the same scope? A single scope? No: if we have a loop that defines a new variable *i* then, only the body of the loop will have access to it. So definitely: scopes have a parent/child structure. If we want to access a specific variable, we need to navigate up the chain of parents until we find one of the same names. Therefore, **control-flow-nodes are the only nodes capable of creating a new scope**.

DATA PROPAGATION AND EXECUTION: the interpreter

The idea is to allow the user to **interact with the graph and get immediate feedback**. That includes possible errors in interpretation and each node's output. Here's how it works:

(1) Update triggers

Whenever a node is interacted with (including *f-node*, *g-node*, *s-node* or *o-node*): it will get all the required arguments from its IDDs. For a given IDD, the value can be cached or not available at all. (a) If at least one is not available it will abort execution; (b) else the process is executed and a value is cached in the ODD, displayed on the node and propagated. Propagation allows this new value to update the rest of the graph: this process is called **direct data propagation**.

(2) Direct data propagation

The current node needs to propagate the value to its ODD. For each connected node, we recalculate their result: (a) check all IDD, (b) get values, (c) calculate result, (d) propagate. Execution nodes can, potentially, be connected to other nodes independently from data links. If propagation hits an execution node, it might be part of a larger execution tree (e.g. a loop) which means each execution node of that tree must be updated as well. Therefore, we need to keep track of this execution tree from the start, whenever a given node of this tree is triggered, then an **Execution Tree update** must happen.

(3) Execution Tree update (ET update)

First, the root node of that tree must be found. It will get updated (all argument values are considered already cached), a direct data propagation will happen from this node. On the execution tree, if a node has ODD, a direct data propagation will occur, but it won't be able to trigger yet another ET update to prevent infinite loops. By propagating, the execution will end up reaching a leaf, and will eventually end.

(4) Back data propagation

In some cases, updates are nonlinear, when something affects nodes in another part of the graph. Set-nodes have this effect. If at one point in the ET, a variable is set/changed with a set-node, this change needs to affect all nodes, using this variable after this point. However, we are not refreshing all cached values. We want to recalculate only the necessary nodes.

➔ Dependency tracking

To achieve this, we must keep track of dependencies: is node *x* dependent on variable *a*? Therefore, each node **including get-nodes** (which are the source of the dependency) **and set-nodes** (which depend on a specific variable) must have a ***Dependency*** object on each of their IDD argument. Whenever a process or router is executed it must update this dependency set by combining the dependencies of each of the nodes it's connected to and provide it to its ODD.

➔ Setters trigger updates

When a setter-node performs a change on the current Scope i.e. its IDD value changes, this information needs to be carried over down the execution tree. A special object will keep track of variables that have been updated.

> **Unnecessary updates (scope confusion)**: we need to keep track of scope too e.g *i = 1* then in a child scope *i = 10* therefore then we go up a scope, *i* hasn't changed (it has changed in the child scope but not in the main scope). Therefore, no update is required on nodes as *i* remains equal to 1. If we don't check that, all nodes dependent on *i* will get updated.

➔ Dependency resolver

When updating an execution tree, we need to determine which nodes need to be recalculated. For each IDD, we compare the changed variables with the dock's dependencies. If a dock depends on a variable that has changed, then a back propagation is executed to resolve this dependency. Note that this propagation is promised-based: for a given node, each IDD creates a promise. When **all** promises have been resolved the result of that node is calculated. It will reach the previous node still carrying over the list of changed variables, check for dependency updates and promise a result. When a node with no dependencies is reached the promise is resolved and the previous promise can then resolve itself until, consequently, the first promise created resolves.

(5) Concurrent updates

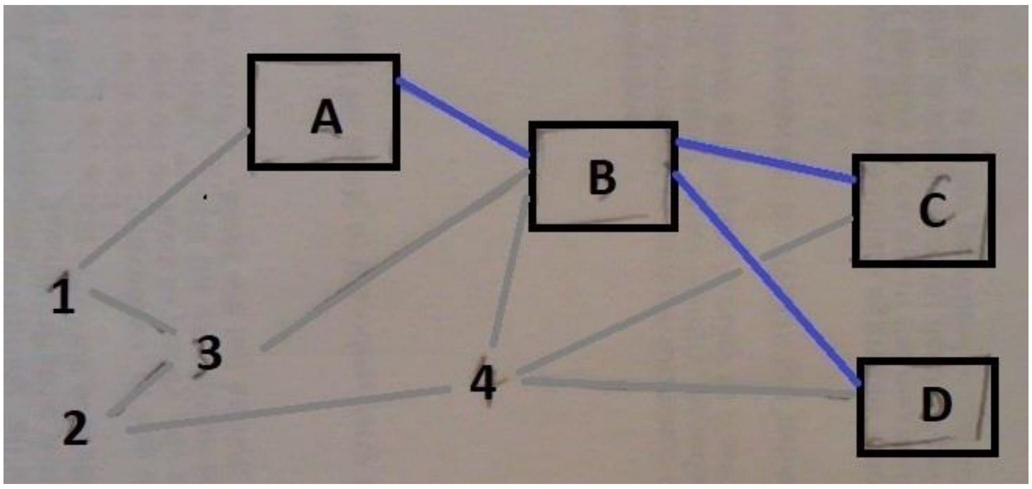
When a node interacts with two or more nodes from the same execution tree. In other words, when it is connected to two nodes on two different points of the ET. Updating that first node will result in two concurrent updates of the execution tree. Considering those two updates will not happen at the same time and not in the correct order, we need to ensure every access point has been updated before updating the ET entirely.

➔ ET access mapping

Each node of the execution tree will collect the identifiers of all data nodes it's connected to. This mapping will be inverted resulting in a new object, mapping each ascending data node to an array of execution nodes it's connected to. With that in memory, we are ready to supervise propagation.

When a node is being updated, the propagation will eventually lead down to the execution tree. If so, it will look at the origin of that update and look at the associated set of entries on the ET (described above). It will remember that this execution node has been accessed and wait for other accesses on the ET if the set contains other nodes. As long as all accesses on the ET haven't been made, the ET cannot update.

Here is a minimal example showing the construction of the inverted access mapping.



The access points to the ET are *A*, *B*, *C* and *D*. The connected data nodes are:

$A \rightarrow \{1\}$; $B \rightarrow \{1,2,3,4\}$; $C \rightarrow \{2,4\}$; $D \rightarrow \{2,4\}$

The inverted mapping looks like this:

$1 \rightarrow \{A, B\}$; $2 \rightarrow \{B, C, D\}$; $3 \rightarrow \{B\}$; $4 \rightarrow \{B, C, D\}$

➔ <u>Access buffer</u>

The ET will have an access buffer in memory which allows it to know whether it is waiting for an access or not. On each ET access we can determine the origin of the update. If it's the first update, we copy the associated set of execution nodes from the inverted access mapping.

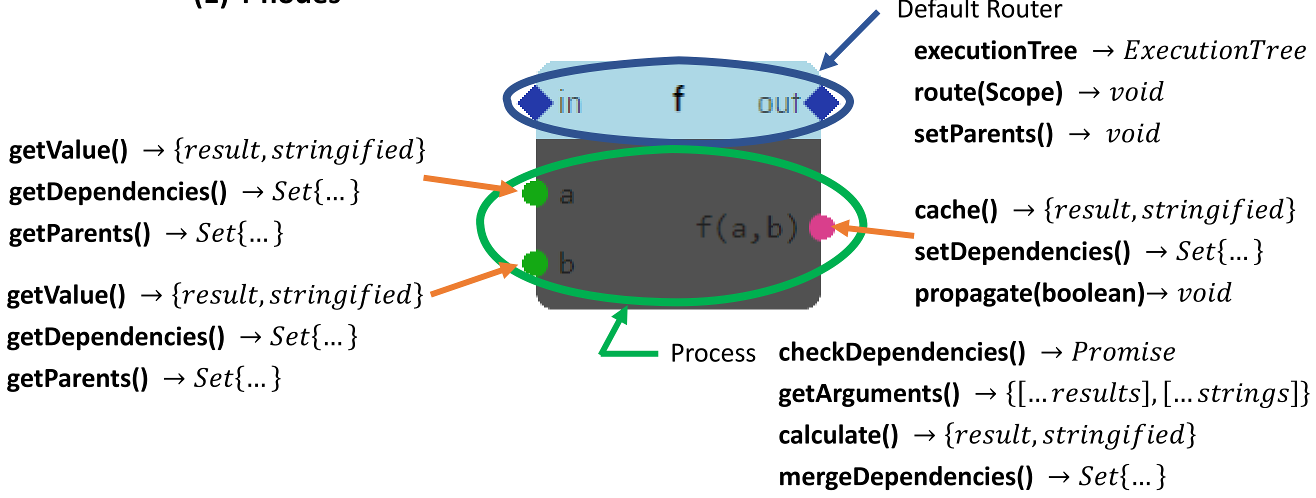
For example, *1* is the origin and it has propagated to *B*. Then the buffer will have to be updated to $1 \rightarrow \{A\}$. This means, the ET is waiting for an access to *A* from *1*. On each ET access, we will look at the access buffer, if there is a set saved, it needs to validate the access and update the set, if the set is empty the ET can be updated from the root. Else, it will pass i.e. wait for an access on another exe node.

In the example, if *2* triggers an update: either B, C or D will be accessed first (let's say C). The access buffer will be $2 \rightarrow \{B, D\}$. Then either B or D will be updated, the buffer will become $2 \rightarrow \{D\}$. Then lastly $2 \rightarrow \{\}$. The buffer is empty, and the ET can be updated: $A \rightarrow B \rightarrow C|D$.

If for some reason the access is not validated, i.e. there are concurrent updates from two different origin sources. The most recent one will be ignored.

Triggers and propagation: detailed breakdown

(1) f-nodes


Default Router
executionTree $\rightarrow ExecutionTree$
route(Scope) $\rightarrow void$
setParents() $\rightarrow void$
in
f
out
a
b
f(a,b)
getValue() $\rightarrow \{result, stringified\}$
getDependencies() $\rightarrow Set\{\ldots\}$
getParents() $\rightarrow Set\{\ldots\}$
getValue() $\rightarrow \{result, stringified\}$
getDependencies() $\rightarrow Set\{\ldots\}$
getParents() $\rightarrow Set\{\ldots\}$
cache() $\rightarrow \{result, stringified\}$
setDependencies() $\rightarrow Set\{\ldots\}$
propagate(boolean)$\rightarrow void$
Process
checkDependencies() $\rightarrow Promise$
getArguments() $\rightarrow \{[\ldots results], [\ldots strings]\}$
calculate() $\rightarrow \{result, stringified\}$
mergeDependencies() $\rightarrow Set\{\ldots\}$

We need to be able to determine that the **node is executable**. Designated by the ***type*** property holding the value ***EXECUTABLE***. Also, it needs to have a scope property containing the local *Scope*.

The node is said to be triggered when one of following actions occur:

(a) Directly edited by the user (e.g. a link is attached/detached to/from the node).

(b) Triggered by calling one of the IDD's ***propagate(boolean)*** function[1]. If its **1st argument is false**, the *Process* will only run through part (1) and (2). This *boolean* represents whether the propagation can trigger an ET update.

(c) The node's IED is triggered.

And here's what happens when it is triggered:

(1) Resolve dependencies: on each IDD, we need to look at the dependencies with ***getDependencies()*** and check if the dock's state needs to be updated. We call ***checkDependencies()*** which will return a *Promise*. This function will compare each dependency set with the local *Scope* which is provided by the ET. If some docks need updating further up, this will make sure they are. When the initial *Promise* is resolved, the *Process* can proceed.

(2) Update parents: the set of parent node associated to this executable node needs to provide its access mapping to the *Execution Tree*. This is used to supervise the ET propagation.

If case (b) && *boolean* = false, stop process here. If case (c) do (4), (5) and (6). Else only do (3).

(3) Validate the access on the ET with the ET access buffer. If validated, update the access buffer and update the ET starting from its root node with**.** Otherwise, if not validated, stop process here.

(4) Evaluate: *Process* gets the arguments with ***getArguments()*** then injects them in ***calculate()*** in order to get the result and string expression. Both results are cached inside the output dock with ***cache()***. Then we are combining each IDD dependencies with ***mergeDependencies()***. This

[1] IDDs don't have a ***propagate()*** function, here we are referring to the ***propagate()*** function of the ODD(s) connected to this IDD.

dependency set is then attached to the ODD with ***setDependencies()***.

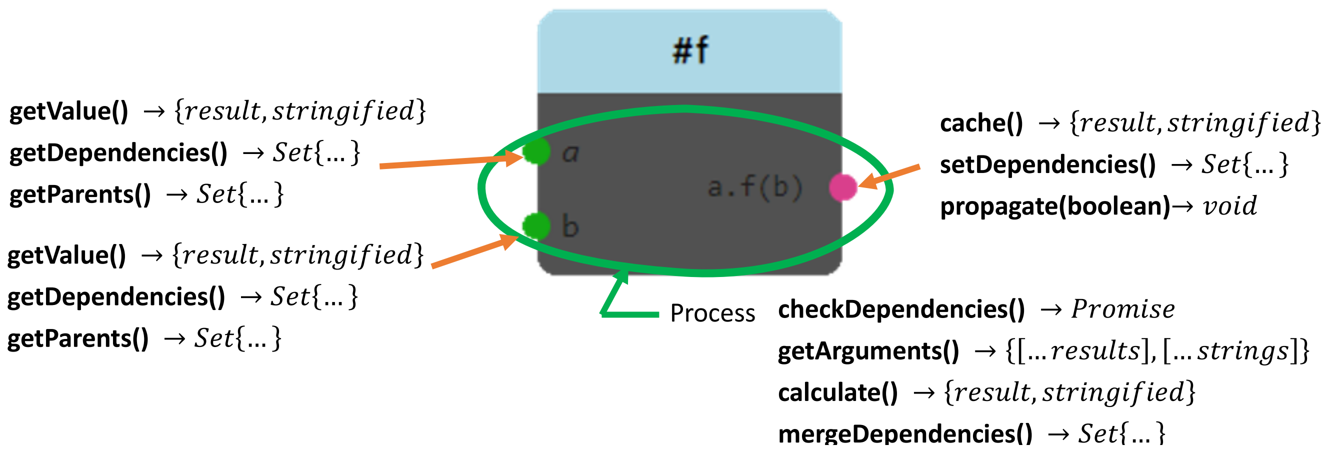
(5) Propagate: Since this is an executable node, direct data propagation **will not trigger ET updates**. Function ***propagate()*** will access each connected node and update their result. This will eventually call other ***propagate()*** functions. If one of these nodes is an execute node, propagation must abort on that branch. Therefore, we need to pass this as argument: ***propagate(false)*** will identify the propagation as originating from an execute node.

(6) Route: Since this is an executable node, the execution must resume by triggering the next executable node. To do so, ***route(Scope)*** will simply get called with the *Scope* it received.

> It is important to note that the ODD also has the methods ***getValue()***, ***getDependencies()*** and ***getParents()***. Which would be used by the nodes following this node. Essentially calling one of those three functions on an IDD would result in calling the same function on the connected ODD.

(2) o-nodes

Here we assume that calling ***#f*** on a has no effect on object ***a*** and simply returns a value.



#f
a
b
a.f(b)
getValue() $\rightarrow \{result, stringified\}$
getDependencies() $\rightarrow Set\{\dots\}$
getParents() $\rightarrow Set\{\dots\}$
getValue() $\rightarrow \{result, stringified\}$
getDependencies() $\rightarrow Set\{\dots\}$
getParents() $\rightarrow Set\{\dots\}$
cache() $\rightarrow \{result, stringified\}$
setDependencies() $\rightarrow Set\{\dots\}$
propagate(boolean)$\rightarrow void$
Process
checkDependencies() $\rightarrow Promise$
getArguments() $\rightarrow \{[\dots results], [\dots strings]\}$
calculate() $\rightarrow \{result, stringified\}$
mergeDependencies() $\rightarrow Set\{\dots\}$

Operator nodes have a similar behavior to executable nodes. One importance difference though is that they don't have a router. There are two ways to trigger an o-node:

(a) Directly edited by the user (e.g. a link is attached/detached to/from the node).
(b) Triggered by calling one of the IDD's ***propagate(boolean)*** function.

And the sequence of steps is:

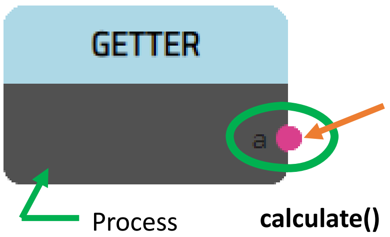
(1) Resolve dependencies: same as f-node.

(2) Update set of parents: same as f-node.

(3) Evaluate: same as f-node.

(4) Propagate: calling ***propagate(boolean)*** with the same *boolean* as the one received. In case (a) use ***boolean = true*** by default as we are looking to update an ET if any are encountered.

(3) g-node



cache() $\rightarrow \{result, stringified\}$
setDependencies() $\rightarrow Set\{\dots\}$
propagate(boolean)$\rightarrow void$

Process
getter-default

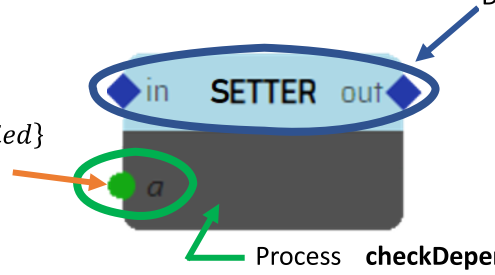
calculate() $\rightarrow \{result, stringified\}$

The node needs to have a access to a scope!
g-nodes are triggered the same o-nodes are. The execution steps are as follows:

(1) <u>Evaluate:</u> *Process* executes the default getter ***calculate()*** function and the result is cached with ***cache()***. Then *Process* calls ***setDependencies()*** to set itself – the node – as dependency.

(2) <u>Propagate:</u> We propagate allowing ET updates: ***propagate(true)***.

(4) s-node



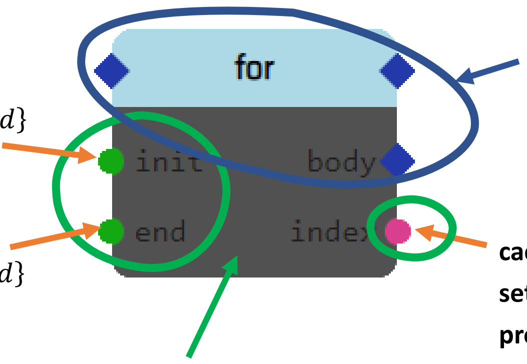
Default Router
executionTree $\rightarrow ExecutionTree$
route(Scope) $\rightarrow void$
setParents() $\rightarrow void$

getValue() $\rightarrow \{result, stringified\}$
getDependencies() $\rightarrow Set\{\dots\}$
getParents() $\rightarrow Set\{\dots\}$

Process
checkDependencies() $\rightarrow Promise$
getArguments() $\rightarrow \{[\dots results], [\dots strings]\}$
calculate() $\rightarrow \{result, stringified\}$
mergeDependencies() $\rightarrow Set\{\dots\}$

An s-node has the same triggers and behavior than an f-node. It has a default ***calculate()*** function.

(5) cf-node



Custom Router
executionTree $\rightarrow ExecutionTree$
route(Scope) $\rightarrow void$
setParents() $\rightarrow void$

getValue() $\rightarrow \{result, stringified\}$
getDependencies() $\rightarrow Set\{\dots\}$
getParents() $\rightarrow Set\{\dots\}$

getValue() $\rightarrow \{result, stringified\}$
getDependencies() $\rightarrow Set\{\dots\}$
getParents() $\rightarrow Set\{\dots\}$

cache() $\rightarrow \{result, stringified\}$
setDependencies() $\rightarrow Set\{\dots\}$
propagate(boolean)$\rightarrow void$

cf-node-specific default *Process*
checkDependencies() $\rightarrow Promise$
getArguments() $\rightarrow \{[\dots results], [\dots strings]\}$
calculate(Scope, ...args) $\rightarrow \{result, stringified\}$
mergeDependencies() $\rightarrow Set\{\dots\}$

It triggers itself the same way as f-nodes: (a), (b) or (c). The execution steps are:

(1) Resolve dependencies: same as f-node.

(2) Update parents: same as f-node.

If case (b) && *boolean* = false, stop process here. If case (c) do (4), (5) and (6). Else only do (3).

(3) Validate ET access: same as f-node (execute ET from root node).

(4) Evaluate: same as f-node. However in this case, the scope is passed as well.

(5) Propagate: same as f-node.

(6) Route: the execution must resume by triggering the next executable node. ***route(Scope)*** will determine which OED to activate. It's able to activate a special option which allows the execution to go back to (1). For instance, when the main execution branch (i.e. the loop's body) has reached a leaf, it can go back to execute again its process and router.

Annexes

(1) To-do / To-discuss

- How do back propagation updates work?
- How to pass the scope to a data node?
- How about instancing a canvas and a node?
- The view and canvas class?
- Key input manager?
- Rename exe dock to **sockets** and leave data docks as **docks?**
- Node instances (cf. Component is the class; Node is the instance)

(2) Abbreviations

- IDD - Input Data Dock
- ODD - Output Data Dock
- IED - Input Execution Dock
- OED - Output Execution Dock
- ET – Execution Tree

(3) Pain points (as of the 24th of Feb.)

- How does one access the Scope object ?

The scope object is stored on **Executable** nodes – i.e. nodes that have a router different than the **NullRouter.** The scope is passed on direct execution on the Execution Tree. Solving dependencies when hitting an E-node will work (back propagation -> do we need promises?).

- Unclear how to make Control flow work.

The string version is wrapping the body. Also, *func* and *stringFunc* are defined for the node's output not it's *"control-flow capability"*. Therefore, there needs to be another function to calculate the wrapper's string. We have not implemented router's functions yet: how to bind exe-docks to the execution? Activate one out-exe-dock or?