Mr. Alexander (@OlegAlexander),

(and anyone else interested in the topic)

Thank you for your patience. The text ended up a bit longer than I anticipated. Of course, there's much room to improve, but I believe what's been achieved is enough to address our ongoing discussion regarding branching in Nodezator and its compliance with the dataflow paradigm.

Since the text has 19 pages, I'm also sharing a PDF version of it, in case you prefer the format (although the web version has the advantage of allowing the images to be closely inspected by opening them in a tab, etc.).

Without further ado, since we have a lot to discuss, let's begin right away!

Brief recap and introduction

So, 02 months ago you recommended me a chapter from this book, the chapter about Maya's dependency graph. On the occasion, you claimed to have done so for 02 reasons:

1. mainly to clarify my mind on the dataflow approach and the benefits of keeping Nodezator purely within that paradigm, providing evidence for why I should give up on my design of a branching feature for Nodezator.
2. it is a very interesting read per se, since it also explains the inner workings of the Maya DAG;

Now that I read the chapter twice, it is finally time to comment on its contents.

In addition to all of that, you have been providing ideas of how branching in Nodezator can be achieved using what's already available in Nodezator, as can be seen in the previous replies in this GitHub discussion. Everything based on principles and tools from both functional programming and the dataflow paradigm.

So my full reply will address everything, both the chapter and your replies. Here's the structure of this article:

1. On the contents of the recommended chapter
2. Brief comment on looping
3. My journey regarding branching in Nodezator
    i. Summarizing decision making in Nodezator
    ii. The problem and my original stance
    iii. Summarizing my solution to branching
    iv. My current stance and reviewing branching
4. Comment on Mr. Alexander latest replies/demonstrations

In the first section I'll address the chapter per se, without taking into account the discussion on my branching feature. However, I'll also provide extra info on how Nodezator and other features (not related to branching) relates to what I observed/learned from the chapter.

The rest of the articles is much longer, since it is an attempt to stitch many topics together into a concise piece by commenting on their importance/usefulness and the resulting influence on my stance regarding branching in Nodezator. That is, I'll comment on my journey regarding branching in Nodezator and also address your previous replies at the end. I'll also include more information regarding the actual approach I use to make decisions regarding features in Nodezator, my initial stance on branching for Nodezator (even before my proposed design), and my current stance.

As I said before I did give up on the implementation of my proposed design for branching (and for looping as well).

However, even though I gave up on my proposal, I'll still write about it a bit more, in order to walk you, and whoever reads this, through my decision-making approach and why despite giving up on my proposal I still think it is was valid. The reason for dropping it is simply due to the fact that the current design of Nodezator plus some of the features to be implemented (like the ability to reference the callable on the nodes and the subgraph feature) already provide all the versatility and flexibility needed to handle both branching and looping.

The purpose of explaining the validity of my original proposal though is not to pat myself on the back, but rather provide a deeper analysis of the decision-making approach behind it, which I use not only in Nodezator, but the entire Indie Python project.

This knowledge will surely be useful to guide future discussions. Because of this, I once more ask you and anyone reading this to read it with an open mind, specially those who didn't agree with my proposal from the beginning.

As always, take all the time you need to read and reply.

On the contents of the recommended chapter

The contents of the chapter are very accessible and straightforward. It is probably due to how simple, yet powerful, the design of Maya's dependency graph is. That caught me by surprise cause I never thought of a graph as a way to aid in the structure and operation of an app, much less a complex one like Maya.

In fact, the concepts are so simple that the entire chapter didn't even need to introduce any code, except for `attributeAffects()` and `compute()`, which were presented only to describe their roles in everything.

The author even went out of his way to explain how other apps, in contrast, organize/structure their functionality in different modules rather than benefiting from the atomicity, flexibility, extensibility and versatility of a node-based structure.

Most of the other concepts following these initial discoveries about Maya's structure based on the dependency graph are actually pretty common not only in the dataflow paradigm, but in other related/similar concepts/paradigms as well, like component-based software engineering/programming (CBSE/CBP) or flow-based programaming (FBP). Notions like the nodes being individual and independent black-boxes, the extensibility of their operations and how combining them provides versatility, extensibility and power for the environments/apps/features that use them. This is not to say that there was no benefit from revisiting those concepts. On the contrary, seeing how they apply in practice to Maya's design was relevant and enriching.

The fact that Maya's dependency graph serves as the foundation over which the graphical user interface operates (mediated by the MEL command engine) was also interesting, especially at the end of the chapter where it is explored how it allows independent features to coexist, like animations driven by the timeline that at the same time allows interactions from the user with the animated objects.

It is also interesting how the dataflow behaviour is partially ignored in favor of the push-pull approach. Of course, this is not to say that the dataflow approach was thrown away in the app. As you said before, the push-pull model is just a trick/tool to help optimizing the graph's execution. This particular decision plus other information laid along the chapter provides further insights into the inner workings of Maya and the very nature of the decision-making behind its design.

In summary, it was made clear that they understood the value of the data structure that a graph represents per se, independently of any paradigms laid upon it. That is, even before being considered a part of a paradigm, a graph is already useful on its own as a data structure. This perception is what allowed them to bend the dataflow paradigm a bit, for the sake of optimization.

They used the dependency graph for what it is above all else: a way to store data representing all attributes, values, behaviours and how they depend on each other. With that in mind, they had freedom to adopt the push-pull approach to efficiently update the app whenever needed, without needing to discard the dataflow paradigm.

Nodezator also makes a clear separation between the graph and its execution. So much so, that additional execution behaviours can be easily added. The app has full access to the data in the graph and can use it as it sees fit.

For instance, the `Graph` menu in the menubar has 02 execution behaviours, the `Execute graph` which is the default one, and `Execute with custom stdout`, which is actually just the default one wrapped in a with-block that temporarily replaces the standard output stream by a custom one during execution.

However, I intend to take advantage of the highly customizable nature of the execution to implement additional execution modes. The current execution algorithm just visits each node, one by one, without a particular order, and checks whether that node has all the data needed to execute. If it has, it is executed and the outputs are sent to the next nodes, if not, it is skipped and we visit the next node. Then we repeat the process revisiting the nodes we skipped earlier until all nodes are visited and executed.

During a visit, we also check whether the node is missing data in any of its parameters. If a node doesn't have a needed input from a connection or from a widget, it is deemed inexecutable and the execution is aborted with an error dialogue. There's nothing special about this algorithm, it was just the first thing that came to mind when I created Nodezator.

I actually have many written notes about possible improvements to the efficiency of this algorithm and also other execution modes that I want to implement. For instance, the current algorithm executes the node right away as soon as it is visited if it has all data required to execute. This is usually what we want, but if we had a large graph with many nodes and they took considerable time to execute, it would be desirable to first visit all nodes before executing any of them, to see if all their required parameters had connections or widgets providing data. Otherwise, we would only discover later during execution (possibly after a long time) that a node was missing data, thus wasting our time.

For now, Nodezator only executes the graph synchronously, one node at a time, but I also have notes describing an additional execution mode that has multiple workers executing nodes as soon as they have all arguments, that is, using concurrency/parallelism. Of course, this is something for the future, after I implement other critical features. I've been studying concurrency/parallelism in Python for quite some time but I feel like I only scratched the surface.

The chapter also gave me the idea for another mode where the app also keeps the output of each node cached, and whenever the graph is executed, only the node who have their inputs changed (and the ones downstream from it) are executed. That is, this is similar to what Maya does with the push-pull approach and the dirty flags to avoid executing nodes needlessly.

Speaking of the push-pull approach, I'm also glad for having learned about it, cause I was planning to do something similar for the "persistent viewer nodes" feature (it's been in the list of planned/requested features since last year).

A persistent viewer node is just a regular viewer node, but instead of just generating the visualization when the graph is executed, the visualization stays visible in the graph after execution. It is something that other node editors have that Nodezator is missing.

In addition to having all persistent viewer nodes update when the graph is executed, I also want each individual viewing node to have a button to trigger the update of that node alone, and that's when I plan to do something similar to the push-pull approach. That is, when requesting a single viewing node to update (by clicking in that special button in the node), only the nodes upstream from it will execute to produce the data

needed for it to update the visualization. So learning that a similar approach is used in Maya gives me a bit more confidence.

There was certainly more content that could be discussed like the implementation of the timeline, how the transform nodes are applied are organized and applied within an hierarchy, etc., but I assume this is beyond what you wanted me to analyze in the chapter.

My final conclusion is that the usage of a graph to model and represent all data and dependencies in the Maya gives the app all power and flexibility/versatility/extensibility that it needs. Also, by thinking outside the box, the developers of the app managed to guarantee a better performance for the app by focusing the execution of the graph only in the points where the corresponding data is requested, reducing execution only to the nodes that need to be updated.

Brief comment on looping

Despite the fact that my original proposal involved both branching and looping, in order to simplify this already large article, we'll be focusing entirely in comments/analysis of branching designs for Nodezator.

Additionally, even though all opposition to my original proposal only ever mention the design of branching, I also rejected its looping-related tools/implications, since they are also indirectly affected by the disapproval of the branching portion, due to also consisting of a deviation of the pure/conventional dataflow-conformant design of Nodezator.

My journey regarding branching in Nodezator

Summarizing decision making in Nodezator

When we last discussed about the branching approach in Nodezator, I mentioned an article I'm writing about the decision-making approach behind the birth of Nodezator, the current state of its source, challenges, possibilities, etc. and also provided a copy of its first draft. I'm still working on it, but I'll summarize the part of it which I think is relevant for our discussion: the decision-making approach.

First of all, software development is complex. I believe people usually underestimate how difficult it is to implement any kind of meaningful decision-making framework. The Nodezator project is even more difficult in this regard due to its inherent complexity. Another thing to keep in mind is that being an open-source project makes it subject to decision making and discussion by people with very different backgrounds.

This factor is actually good because it brings different useful perspectives to the decision making and discussions. It also brings new challenges though. That is, it may also be source of confusion, misunderstandings or just require more time and effort so people involved in a decision/discussion can at least understand the different opinions, regardless of their own positions on the matter in question. This is actually fine though. Good things require time and effort. The trade-offs are worth.

I believe in the potential of the whole IndiePython project and its flagship app, Nodezator, to bring more funds to the project over time. Slowly but surely, almost daily, more and more people are becoming aware of their existence and the value they bring to Python development and education. However, until I can fully fund my development time, I can't invest 100% of my time in it (though, fortunately I've been able to invest almost 100% of it), nor hire/outsource people to work full/part time on the projects' tasks.

It is okay, though: I'm happy with the slow growth because I'm already quite busy with its current size, so it is better that it keeps growing in small increments as I invest more and more time and resources in the project. My purpose in mentioning all of that is to point out that this situation also influences the decision-making process in the IndiePython project. Until we are big enough, we need to be careful not to adopt a decision-making approach too rigid or complex to the point where it slows down the actual decision making. After all,

we must take into consideration that we are a small community with busy people trying to contribute however little time and effort each of us have.

With all of that in mind, rather than rules, my approach actually consists of a few principles, some of which I list here:

1. decisions/discussions must be based on:
    i. evidence (when possible);
    ii. concise arguments (always);
2. proposed/requested features must solve real problems/use-cases;
3. Kanat-Alexander's equation of software design (make sure to consider the updated equation in the comment on the link, not the one in the body of the article) is a great tool for evaluating how worth a feature proposal is: `D = Vf / Em`; in summary, the desirability of implementation (D) is directly proportional to the value of implementation over time (Vf) and inversely proportional to the maintenance effort over time (Em);
4. it is fine to adopt paradigms, but they must prove their worth (following the first principle) in order to stay relevant; a solid partial adoption of a paradigm is better than a full sloppy adoption;
5. it is fine to change your stance on something or go back on a decision, as long as the change is based on the first principle;
6. it is better to ponder and discuss things first, but some problems can only be fully understood in practice;
7. if you are not ready to discuss/decide on something, it is fine to ask for time; no one knows everything or has time for everything;
8. unless something has to be decided or acted upon, postponing is actually great; it is better to postpone something than to decide or act too hastily;
9. if you cannot frankly and openly list/discuss the shortcomings of your proposed solutions, can you really say that you understand it? In other words, let's use proper scientific method and submit our solutions to thorough scrutiny.

I think this is all simple and loose enough to allow a healthy and efficient decision making considering all the constraints of the IndiePython project.

Now that this decision-making approach was presented, what is specially relevant to our discussion regarding branching in Nodezator is the point about the adoption of paradigms. A lot of node editing apps boldly claim to adopt paradigm X or Y. When subject to closer inspection, though, such claims are often proven inaccurate.

When I first studied FBP (flow-based programming) to gather insights for the implementation of Nodezator, the number of apps claiming to be flow-based that were actually proven not to be by the author and other researchers was astounding. This is why I decided not to follow any specific paradigm for Nodezator.

Instead, I implemented it based on simple and proven concepts and tools from some useful paradigms. And so, I decided to use a function/callable to represent a blueprint for a node, and a graph structure to store node instances and the relationship between the data that flows through them. I also tried to bring the design as closely related to Python as possible.

This approach resulted in many of the useful things that Nodezator has today and that some people take for granted. For instance, I'm not a genius that can develop a generalist node editor that can export a graph to Python code. This Nodezator feature is actually an accident (sweating-smiling emoji). Though I wanted to eventually implement it, I thought it would take some years for Nodezator to be able to do that, but it turns out the concept came to me when I was implementing the other exporting features (.png and .svg). The more one inspects the design, the less impressive it becomes. Nodes are already based in calls to Python callables, so a Python exporting feature is just a matter of "untangling" the nodes in the graph into individual calls and values in a script.

This kind of stuff is only possible because I kept the design as simple as possible by implementing stuff iteratively, rather than trying to faithfully reproduce an entire paradigm.

My point is not against paradigms, just against stubbornly clinging to them. It is (hard) evidence and/or concise arguments that must guide decision making and adoption/rejection of paradigms or parts of them.

Nodezator is not a dataflow or flow-based or functional programming/paradigm application. It is just an app to convert Python callables into nodes that can be arranged into a graph and executed/exported. However, it is true that it adopts dataflow and functional programming principles, concepts and tools. And I want it to keep doing this, but the importance of such principles/concepts/tools comes from their proven usefulness, not from blinding compliance to paradigms.

I wanted to make all of this clear just so you and anyone willing to contribute code and ideas to Nodezator have a firm grasp on the decision making behind it. I hope it didn't come off as arrogance on my part. Quite the opposite, it is precisely because I understand my limitations that I must strive to ensure a relevant decision-making approach is put to practice, one which can guide and empower people to discuss and decide on matters related to the project.

In addition to that, your feedback and stance against my proposed design to branching in Nodezator, despite always announced as attempts to fiercely defend the dataflow paradigm, have been backed up by concise arguments, specially your detailed demonstrations of leveraging the current design of Nodezator to achieve branching using pure dataflow/functional programming-conformant concepts. In other words, your approach has not been lacking in anything, I just wanted to highlight the importance and relevance of evidence and arguments above paradigm compliance.

I acknowledge the strenghts and usefulness of the dataflow paradigm and this is why want to keep Nodezator compatible with it, not of its prestige or "conceptual solidity".

With that explained, we can now explore the original problem using the decision-making approach I described. After that, we'll revisit my proposed design. Then, we'll finish by explaining my current stance (the rejection of my proposed design) and how my knowledge on functional programming as well as your comments here on GitHub contributed to it.

The problem and my original stance

Proposed/requested features must be driven by real problems/use-cases. It was no different in the case of branching and looping in Nodezator. The first person in the project to request such features was Mr. Adams (@WillAdams), which engaged in a lot of discussions since the very release of Nodezator last year. He's currently also a patron and a talented individuals who works with CNC (Computerized Numerical Control), which includes 3D modelling. Our first interactions can be seen in a post on reddit. It spans a lot of comments and even links to another discussion held in a youtube video comment section. Since all of this includes a lot of things unrelated to our discussion of branching and looping, I'll summarize the points discussed and conclusions reached in the following paragraphs. In case you are curious, the original post and comments can be seen here.

In summary, Mr. Adams was sharing his concerns regarding his first impressions of the software and was having some problems to perform some actions like looping. As we know, Nodezator currently has no specialized nodes to deal with neither branching nor looping. However, in addition to that, at the time I was not actually even planning on implementing any kind of looping or branching in the app. This is because I wanted to keep the app as simple as possible so it could be used similarly to the Blender app's compositor which also doesn't have branching nor looping.

Rather, I wanted branching and looping to be dealt outside Nodezator, by exporting the graphs to a script and doing the necessary modifications there, or just pasting it into another more complete script with all the needed branching/looping logic.

Because the Blender app's compositor deals with a finite set of operations and is focused in achieving specific results, like editing an image of image sequence, it doesn't need branching/looping. I was also convinced that, whenever needed, some of the branching and looping could be done inside the nodes instead of in the graph itself. All of this was to avoid making the graph needlessly complex.

However, Mr. Adams did present a simple problem which convinced me of the usefulness of branching in Nodezator (I'll present it further below). After pondering about the problem and the differences between Blender and Nodezator I also reached the conclusion that it is only natural for a generalist node editor for the Python language to have branching and looping implemented as well. From them on I've been trying to come up with a satisfactory solution for branching and looping to Nodezator. Then a few months ago I presented my proposal.

And that summarizes my conversations with Mr. Adams. Let's now just focus on the problem he presented, which is a simple yet representative example of how one would make use of branching within Nodezator.

In his own words, as can be seen in the reddit post previously linked:

> Having branches allows one program to do multiple things --- for example, I've worked up a BlockSCAD/OpenSCAD program to make a box:
>
> https://www.blockscad3d.com/community/projects/1385418
>
> which has 3 separate options for lids:
>
> - Hinged
> - Sawn
> - Sliding
>
> not having branches would require 3 separate files, one for each lid option.

My original stance of not implementing branching or nodes specialized for branching in Nodezator would indeed make the problem unnecessarily complex and even a bit less clear, since it would require him to replicate part of his graph across 3 different files. For a software like Blender this is fine, cause one usually have a single result in mind and works in the graph towards such result in mind, tweaking the values as needed. A generalist node editor like Nodezator however, has an infinite set of nodes that can be created and an infinite amount of purposes, depending only on each user's goals.

In other words, Nodezator needs a branching feature that allows different paths of execution to be chosen and such paths must be clearly represented in the graph.

So, to summarize this entire subsection, I was actually wanting to avoid including branching/looping nodes in Nodezator in order to avoid changing the execution flow. I wanted to keep the current dataflow-compliant execution flow because despite acknowledging the desirability of looping/branching, I didn't think they were strictly necessary inside Nodezator, i. e., they could be explored outside within an external Python script. Additionally, the current dataflow/fp-compliant execution flow already allows looping and branching with `map()` and if-blocks (both implementing them in custom nodes, or when I add the node representing the ternary operator (`a if condition else b`).

Summarizing my solution to branching

As I mentioned in past conversations, the Elixir language doesn't have for-loops. Instead it uses tools like higher-order functions, recursion, list comprehensions and more to iterate over items. For someone like me, who has only ever used PHP, Javascript and Python, this is understandably mind-boggling at first.

Nonetheless, after getting familiar with some functional programming concepts and practices, the usefulness of higher-order functions like `map` , `filter` and `reduce` for instance becomes much clearer.

Moreover, anyone who spends a bit of time learning about generator-functions and the usage of the `yield` and `yield from` statements, specially by studying the free materials from David Beazley on generators or the chapter from Luciano Ramalho's "Fluent Python" book on iterables, iterators and generators (the chapter is called "Iterables, iterators and generators" in the 1st edition and "Iterators, Generators, and Classic Coroutines" in the 2nd one, though this last one I didn't read yet) will realize how powerful, versatile and flexible the iteration machinery is in Python.

Now, what all of those tools from functional programming and Python-specific iteration tools have in common is that they require a bit of understanding of the underlying concepts and how to approach the problems and model the solutions according to those concepts.

Not everyone is willing to put the extra effort to understand and employ such concepts in their code/graphs. And quite honestly, some people probably shouldn't, because perhaps what they want to achieve simply doesn't need the extra power/versatility that concepts/tools from FP/dataflow/Python iteration machinery provides. The trade-offs might not be worth for them.

This is a serious problem that have potentially killed or hindered the progress of some technologies like the Haskell language and still threatens many others. It is important to be zealous supporters of great tools, but one must not ignore how they are approached by beginners and new adopters, lest they become burdens.

In my conversation with Mr. Adams, he jokingly shared this link that points to the anecdote of a programmer that used to write too many lines in his code due to not being aware of the existence of for-loops. Joking aside, the account clearly aims to educate people on the dangers of becoming too invested in specific designs to the detriment of simpler, practical constructs/solutions like the for-loop.

I'd like to add that all of those reservations are not about FP/dataflow/Python iteration tools themselves, nor do they imply my disapproval of them or that I found them inherently complex. Quite the contrary, they are awesome tools to bring power, versatility and flexibility to one's code, and many of its constructs/concepts are easy enough to grasp that understanding and using the simplest among them may be enough to significantly improve that quality of the code/systems using them.

It is just that, if there's an even simpler solution, we must consider/seek it first. Because of all that, I set to come up with a design as close as possible to branching with if-blocks and looping with for-loops. And thus the solution I proposed (and on which I recently gave up, as I explain further ahead) was born.

I believe it is not necessary to explain my proposal here, but I'd like to highlight the aspects that still make the it a valid solution for branching and looping in Nodezator. In addition to such aspects, for the sake of fairness, completeness, and to comply with one of the principles I listed before (the one about being aware of the shortcomings of a solution) I'll also point out my dissatisfactions/the shortcomings of the design.

Such highlighted aspects and shortcomings will be briefly presented in the next 02 subsections. After such subsections, we'll finally dive into why I came to gave up on my proposed solution and further comment on your recent replies about exploring the dataflow approach in Nodezator using its existing features.

Highlighted aspects that validate the design

There are 03 aspects about my proposal that I'd like to highlight and that I believe makes it a valid option.

First, the design for branching matches the analogy of the graph as a set of machines and conveyor belts between them quite accurately, making usage of the analogy to justify the choice of an specific branch in detriment of the others, by ceasing the flow of data into the ignored branches. Of course, the exact mechanism could still be refined, but the core is solid.

Second, the design for branching is compatible with the Python exporting feature, that is, it seamlessly exports to a Python script. The only thing that might be of concern is that the graph now contains sets of nodes that won't be executed, which is unusual for node editor in my experience. Nonetheless, considering

Nodezator aims to be a 100% match of a Python script, that concern quickly disappears since this is precisely what having if/elif/else-blocks in Python scripts (or really those of most other languages) mean.

That is, having if/elif/else-blocks in a script means that parts of the script will execute while other parts won't. The branch whose condition evaluates to True (or the else-block) is executed while the others are ignored. As such, it is not that unnatural for a node editor emulating a Python script to portray such mechanism, even if comes at the expense of a full compliance to the dataflow paradigm. If it is within reason, paradigms can and should be fully or partially put aside.

Finally, the design can coexist with a pure dataflow approach. Nodezator is flexible/versatile enough that one can still use a purely dataflow-compliant branching mechanism like the ones you've been demonstrating.

Shortcomings/dissatisfactions with the design

There are 03 main shortcomings when it comes to my proposal. They are largely responsible for my ultimate decision of rejecting it.

First, though fully or partially parting with a paradigm can be acceptable, it doesn't mean it is encouraged/welcome. Paradigms are not built over irresponsible assumptions. Rather they are usually subject to many tests and scrutiny over time. That means that while blind compliance is a bad thing, justified compliance shields the design of systems against many problems tackled by the paradigm and provide power and versatility inherent to it.

Concepts/tools from the dataflow paradigm and functional programming are specially powerful for understanding and greatly simplifying many complex problems and ensure programs/systems remain simple, powerful and flexible at the same time.

Second, despite the close resemblance to how if/elif/else-blocks work in Python scripts, there is also some cognitive load to the usage of the specialized nodes that support the design. In short, rather than completely eliminating the cognitive load required to use dataflow/FP concepts and tools, it just replaces it with its own, specially since such design, to my knowledge, is quite unorthodox when it comes to node editors. Just to clarify, I use the term cognitive load rather loosely in this text, meaning an amount of energy and concentration to learn something that is higher than normal.

Last but not least, making use of Kanat-Alexander's equation, my proposal has quite the maintenance cost over time. While keeping Nodezator compliant and reliant on dataflow and FP concepts/tools only requires the addition of some feature to fully take advantage of branching and looping in Nodezator, my proposed design would require the creation and maintenance of a different algorithms to handle edition and execution of the graph.

Even though the value over time would consist in the full ability to tackle branching and looping in Nodezator for all use-cases/purposes and in a "relatively simple" way, the maintenance cost over time will probably not be worth it. Such cost would be multiplied manifold, whenever implementing new features that influence how the graph is managed/executed.

My current stance and reviewing branching

Requirements of a solution to branching/looping

More or less 05 months since the last update on my proposal and now my stance is not favorable to it anymore. Since the last update in November 2022 I have not been giving my proposal much thought, because I've been working in other unrelated stuff since them. In December I worked on the first version of automated GUI testing for Nodezator, then in January I worked exclusively on the Bionic Blue game and since February I've been further developing automated GUI testing while I refine its design, which I only recently was able to finish.

Of course, opportunities to rethink the design were brought up by people disagreeing with it, like Mr. Tom (@Tom-99) in December and you, Mr. Alexander, at the end of March. However, even though both of you expressed your disapproval with sound arguments, I still couldn't abandon my proposal, not because of some personal/emotional attachment to it, but because I still wasn't able to find what I believed to be a better replacement.

That is, I wanted an alternative design that could fulfill these requirements as closely as possible:

1. satisfy all the branching and looping requirements, i. e., be sufficient for any looping/branching use-case;
2. be as simple as possible to explain to beginners, in order to avoid a difficult learning curve and/or high cognitive load;

Of course, although not listed, it must also be 100% translatable into pure Python code, for the Python exporting feature.

Problems to achieve the requirements

Achieving the requirements listed above present 02 final problems. I'll present them in the following paragraphs.

Since even before publishing my proposal, I was already 100% sure that it was possible to solve any use-case involving looping and branching using only Python's `map()` for iteration, `filter()` for filtering and the ternary operator `a if condition else b` for branching. Of course, there are many other functions to make the solutions even easier to achieve or more versatile, but just the mentioned tools already provide us with all we need.

This means I never doubted that keeping Nodezator FP-compliant would allow us to achieve that (and, as a result, we could keep the current dataflow-compliant design as well, which greatly simplifies and empowers the handling the flow and processing of the data). However, here's the first problem: **I was still struggling to visualize what Nodezator was lacking in order to take full advantage of such FP tools**. That is, I know the FP tools to use, but what does Nodezator lacks so they can be used without any limitations?

The second and final problem has to do with the cognitive load of some of the mentioned FP tools. In other words, **How to teach and encourage people to model and solve their problems purely using FP tools?**. That is, for instance, how do I ask people used to simple for-loops to model all of their problems using `map()`?

This is not a problem inherent to FP, but perhaps it stems from observing other people question the simplicity of FP tools/concepts and struggle to adopt them in their workflow. The links presented earlier about the failure/underachievement of Haskell teach us that we must not underestimate how challenging new tools/concepts are to beginners.

I myself, despite my limited knowledge, experienced no particular problems when learning and using concept/tools from FP and from Python iteration machinery in my daily programming. Nodezator's source is filled with examples of usage of higher-order functions like map or reduce, identity functions and recursion (with the `yield_subgraph_nodes()` function).

However, we must not be tempted to believe other people presented to such concepts will have the same ease of understanding and applying such concepts to their code, much less within a node-based interface, which is yet another unusual factor to take into account.

Put simply, though, this second problem is just the obstacle preventing us to meet the second requirement mentioned before.

Meeting the requirements with FP tools

Only recently I was finally able to close the gap preventing me to achieve a solution to the problems and, as a consequence, meet the requirements presented before.

What contributed to such achievement was my knowledge of functional programming (limited as it may be), extra research/pondering over the time, and our discussions here on GitHub, Mr. Alexander (not only in this post, but in others as well, like when we discussed subgraphs).

All those sources of information, inspiration and discussions were important because the final solution achieved consists of a combination of various ideas, concepts and features. It isn't a feature proposal, cause it doesn't represent a new feature.

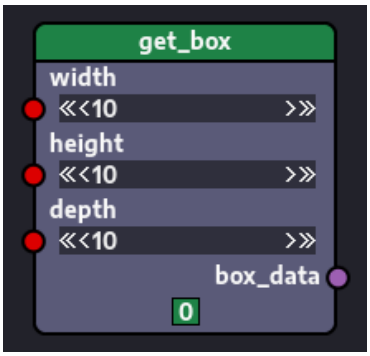
It is just the usage of features we already know, either already implemented, being implemented right now (the callable mode, for instance) or things already listed for implementation (or at least mentioned in discussions, like group nodes/subgraphs). In other words, the solution is achieved within Nodezator's current design. The features used also represent various FP tools/concepts.

Analyzing a problem and presenting our solution

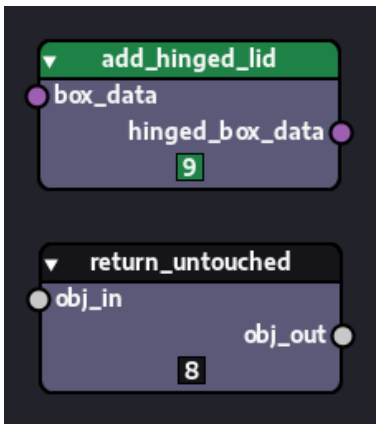
Let's jump straight into action by analyzing how a real problem could be solved with Nodezator and how our method solves the problems and meets the requirements presented earlier.

The problem in question is the one originally presented by Mr. Adams: **He has data representing a 3D box model and wants the model to be changed depending on the requested kind of lid (hinged, sawn, sliding,)**. To keep the problem as fundamental/representative as possible, let's reduce the possibilities to 02 options: either "hinged" or "no lid" (the box without a lid could be used as a prop, for instance). Once we solve the problem, we can then extend our solution to include more options as needed.

The image below depicts a dummy node that returns data representing a box. At this point we still didn't decide whether the box will have lids or not, the node just returns general data describing a box.

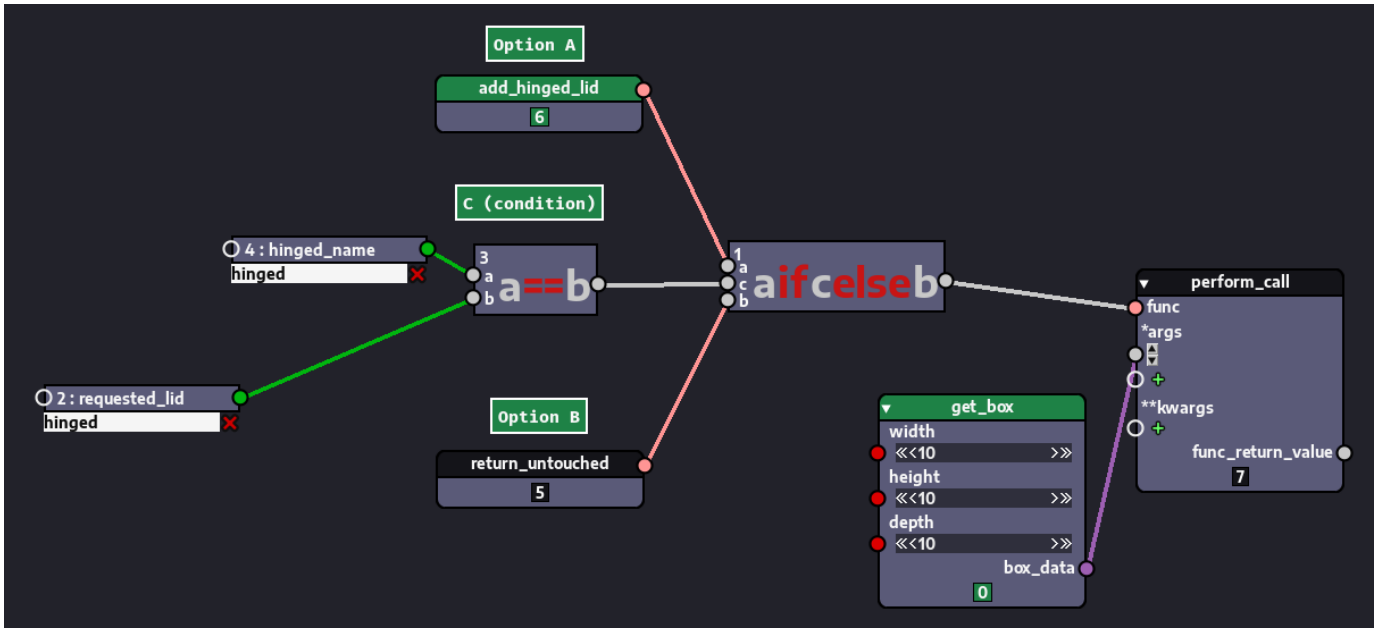


The image below depicts 02 different dummy nodes that represent operations that result in the simplified possibilities: a box with a hinged lid or the same initial box with no lid added to it. As you can see, they take exactly one input and return a single value.



The `add_hinged_lid` node takes the box data and returns the box data changed to include data describing the hinged lid. The `return_untouched` node doesn't change the input at all, returning it exactly as it is. That is, the same object that goes in, goes out as well. In mathematics, this kind of function is called an **identity function**. It is a node I want to add as an app-defined node to Nodezator, that is, one that is available by default (it would probably be added in the "Useful encapsulations" menu).

Finally, the image below shows a viable solution to our branching problem.



At the center of our solution is the `a if c else b` operator node. A node that I'm also planning to add to Nodezator. It just represents a conditional expression or ternary operator in Python (it is also called an "inline if-else statement" and other similar names). `a` and `b` represent the alternatives and `c` represents a condition. So, in plain English, what the operation represents is clear: use **a** if **c**ondition is True, else use **b**.

The usage of the ternary operator was first mentioned by Mr. Alexander in his many demonstrations/explorations of how branching could be achieved with FP (check this comment and the ones that follow it). However, the usage of if/else blocks had been explored before by Mr. Tom, though in a different way.

I'll provide more comments on Mr. Alexander replies in a dedicated subsection after we are done walking through this solution and the next few subsections that extend and solves the problem in other important ways. For now, let's keep focusing on the solution presented above.

Everything to the left of the `a if c else b` node represents the alternatives and condition being passed to the node. The leftmost node, the `requested_lid` variable just holds a string representing the kind of node

that is requested. Then, this `requested_lid` variable is compared to the `hinged_name` variable using equality (the `a==b` node) and the result is fed to the `c` parameter of the `a if c else b` node.

If the requested lid equals 'hinged', then **a** is used, that is, the `add_hinged_lid` node in callable mode, that is, a reference to the callable that receives the box data and returns the changed data describing a box with a hinged lid. If otherwise, the requested lid is different, than **b** is used, that is, the `return_untouched` node in callable mode, which receives the box data and returns it untouched.

The output of `a if c else b` is then passed to the `perform_call` node as the argument for the `func` parameter. The `perform_call` node just performs a call with the given callable object and additional arguments given to it and returns the return-value of the call. It's been available in Nodezator since a long time ago (since its release, if I recall correctly). The `perform_call` node also takes the `box_data` output from our `get_box` node as an argument. In other words, whichever callable comes from the `a if c else b` node will be called with the box data as an argument and will then return either the box data describing a box with a hinged lid or our original box.

This solved problem represents the most atomic/fundamental/basic problem in branching, choosing between an option or doing nothing. In text-based programming and when not using pure FP principles, this would appear in the script as a single if-block without an else-block accompanying it. Something like this:

```
box_data = get_box(...)

if requested_lid == 'hinged':
    box_data = add_hinged_lid(box_data)
```

The exported Python code from our solution will instead represent a text-based script using FP. The code below is a simplified representation of how the graph from our solution would be exported as Python code:

```
chosen_func = add_hinged_lid if requested_lid == 'hinged' else return_untouched
box_data = get_box()
box_data = perform_call(chosen_func, box_data)
```

We are showing a representation, rather than actual code generated within Nodezator, because I still need to update the Python exporting feature to take the callable mode into account. Doing that should be pretty quick and straightforward though, because nodes in callable mode are much simpler to handle than regular nodes. This is so because regular nodes have many more elements to export, that is, all the arguments. Nodes in callable node, on the other hand, just represent a reference to a callable.

Quick note on the efficiency of our solution

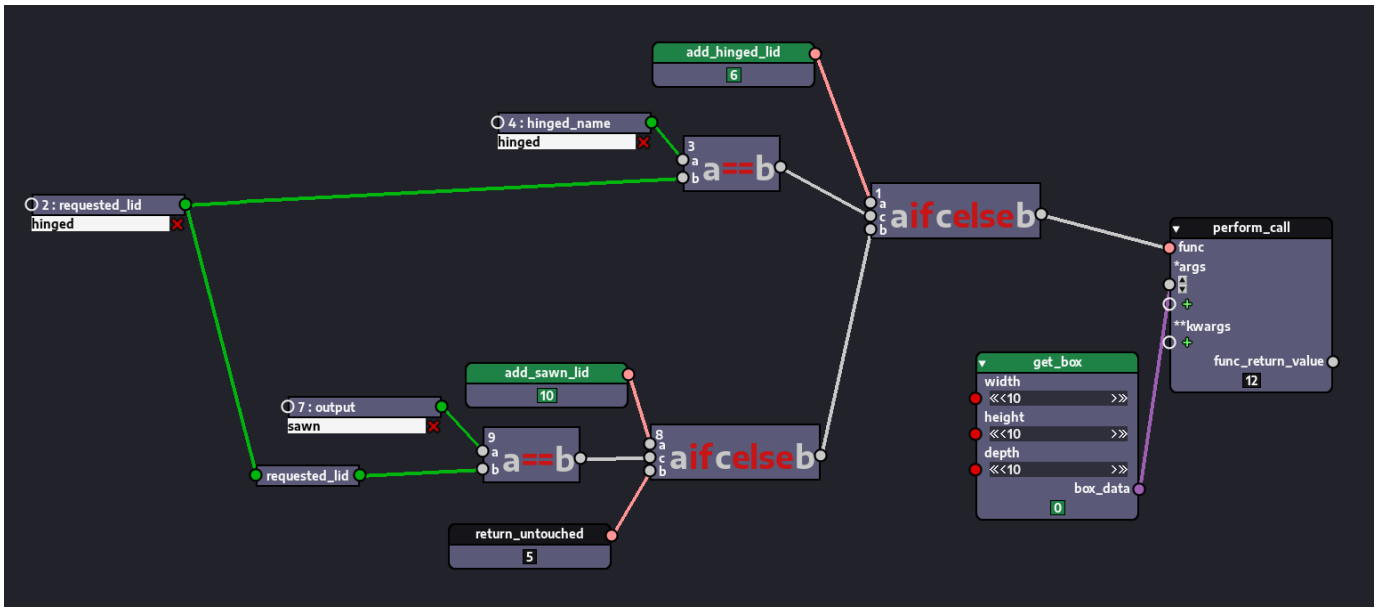
Despite the fact that within a node-based interface like Nodezator the `a if c else b` node doesn't short-circuit because the alternatives must be evaluated before being passed to the node, the solution still doesn't waste any resources. This is so for 02 reasons:

1. only references to the callables are passed to the ternary operator, so there's actually nothing to be evaluated;
2. only the reference to the chosen callable reaches the `perform_call` node and only then the callable is executed.

In other words, our solution is efficient.

Extending the problem and our solution

Now let's extend this problem to work with an additional option: a box with a sawn lid. The image below depicts the solution:



The solution above shows how versatile the ternary operator (the `a if c else b` node) is. Just like shown before by Mr. Alexander, in this comment here on GitHub, we can use multiple instances of it to add as many options as we want.

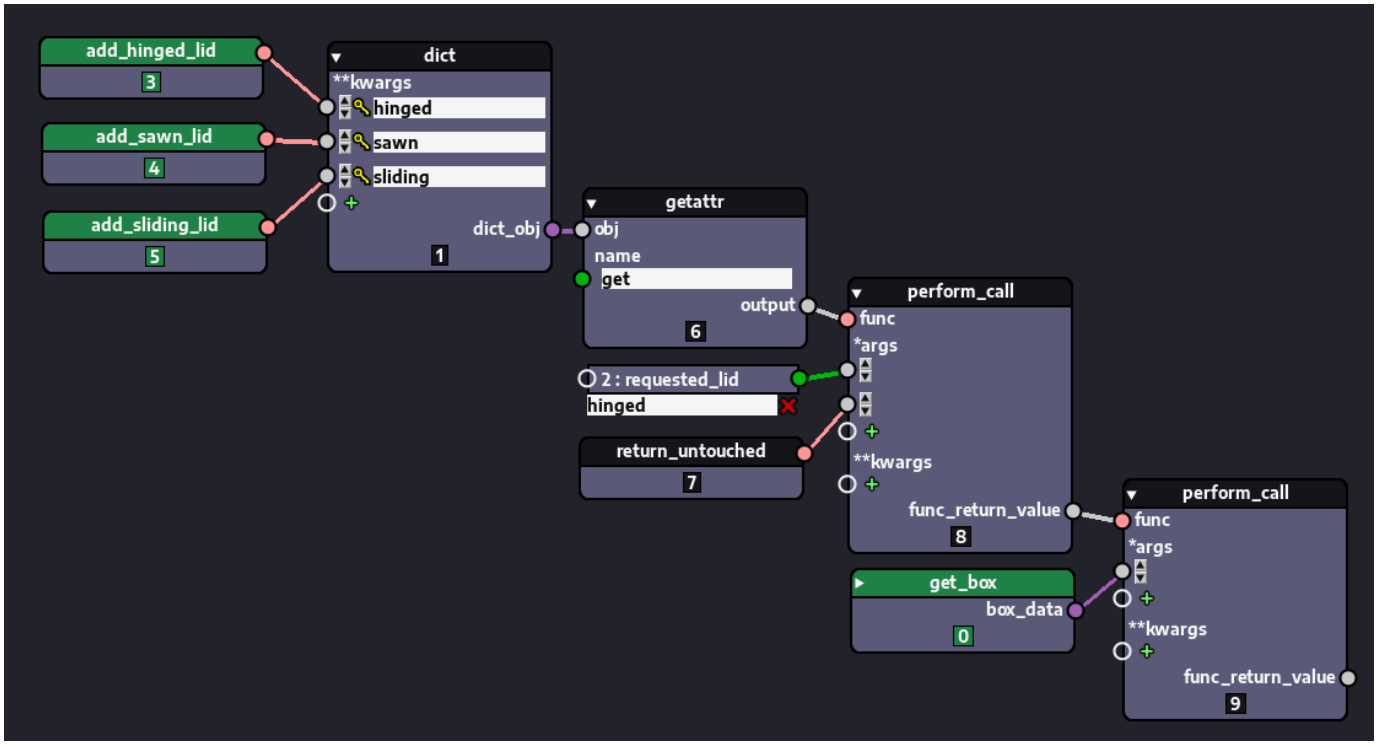
In this solution, the first ternary operator (the one closer to the center, in the bottom half of the image) executes and, if the requested lid was a sawed-off one, it passes a reference to the `add_sawn_lid` callable to the next ternary operator. Otherwise, it passes a reference to our identify function, the `return_untouched` node.

The next ternary operator checks whether the requested lid was a hinged one and, if it is, it passes a reference to `add_hinged_lid` to the `perform_call` node. Otherwise, whichever reference is received in its `b` parameter is passed on, that is, either `add_sawn_lid` or `return_untouched`. The call is then performed in the `perform_call` node with the box data from the `get_box` node.

Just as shown in the image, this solution can be extended indefinitely, regardless of how much alternatives there are. We just need to chain as much ternary operators as needed.

A dict-based simplification

Our solution can still be further simplified in some cases. Whenever the conditions to be evaluated are simple values that correspond to specific options, such different values and respective options can be stored in a dictionary, like demonstrated in the image below:



As seen in the image, the dict is populated with our options, using the respective lid type name. We then retrieve it's `get` method and call it with the `perform_call` node, passing the name of our requested lid to it and our identify function (`return_untouched`). If the name of the requested lid corresponds to one of the alternatives stored in the dict (the callable objects), it is returned, otherwise `return_untouched` is returned. Finally, the returned callable is executed in the next `perform_call` node with our `box_data` .

This solution is actually not innovative. It works similarly to `match/case` statements and in fact people have been using it in Python before `match/case` was implemented and some (maybe many) still use it instead. Speaking of which, a node representing a `match/case` statement is not out of question for Nodezator as well, but something to which I didn't give much thought, due to other features of higher priority.

Completing our solution: different signatures/arguments

There is one case that our solution still needs to address. What if...

1. the alternatives had different signatures?
2. regardless of the signatures, we wanted to pass different arguments to the chosen callable, depending on which one is chosen?

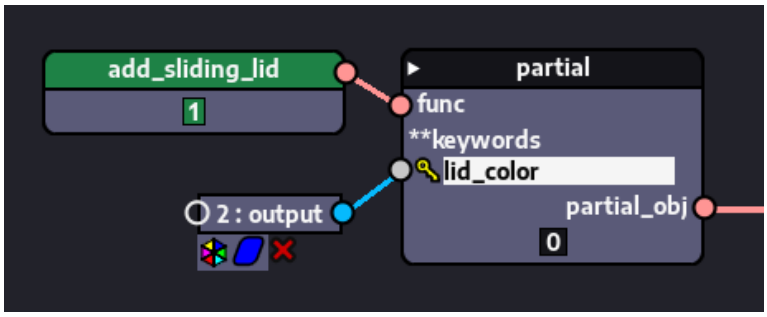
In the problems we just explored, the alternatives had the same signature and we only wanted to pass our `box_data` to them. That is, both our `return_untouched` identity function and the other alternatives like `add_hinged_lid` , etc., all accepted a single argument.

However, what if each of the callables required different arguments or we just wanted to pass different arguments to them on our own volition?

The answer is actually simple: we would only need to pass each reference of the callables through a `partial` node along with the additional arguments. Such node is a representation of the standard library's `functools.partial()` function.

This way we could feed the required/desired arguments to the callables even before the chosen one reaches the `perform_call` node. The `partial` node, like the `perform_call` node is also available since Nodezator's release. It can be found among the standard library nodes under the `functools` submenu.

As an example, let's pretend our `add_sliding_lid` accepts an optional `lid_color` parameter and we wanted our call to `add_sliding_lid` (in case it is the requested lid), to include such an argument for the `lid_color` parameter. All we'd have to do is to pass the reference to our `add_sliding_lid` callable through the `partial` node along with the `lid_color` argument, as demonstrated in the image below:

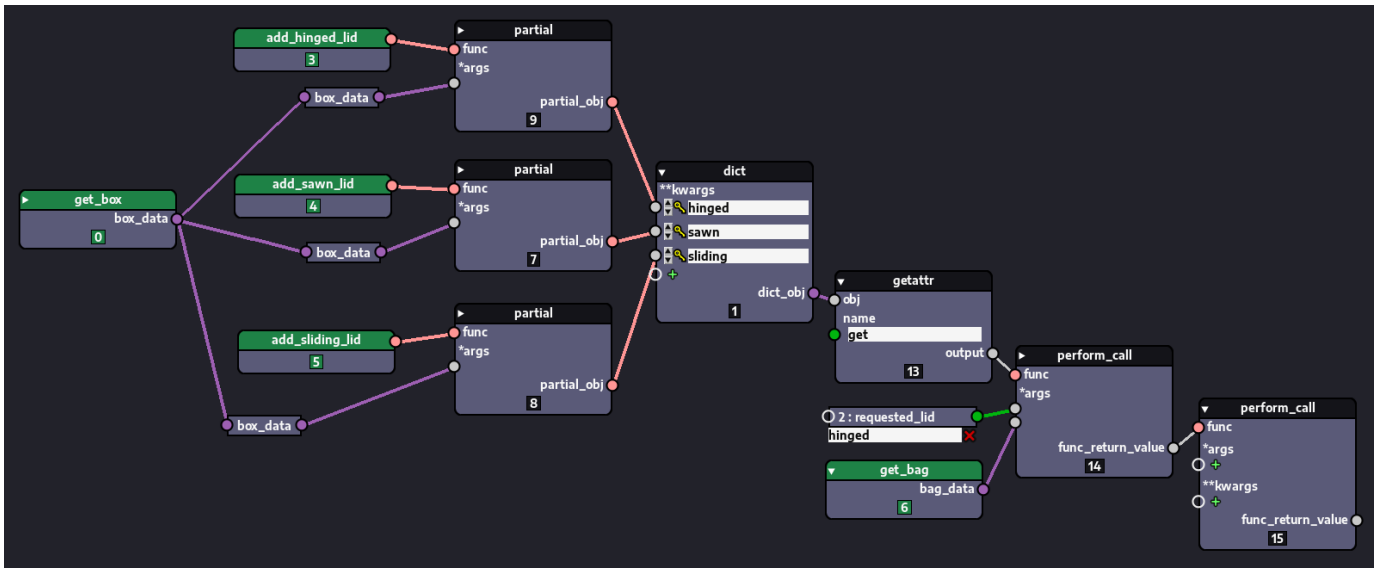


The resulting `partial_obj` returned by the `partial` node would then be passed to the ternary operator as usual and, if it were to reach the `perform_call` node, where it would receive the `box_data` argument, it would be properly executed with both the `box_data` and `lid_color`.

Alternatively, if the order of the arguments was important, we could feed both the `box_data` and `lid_color` arguments to the `partial` node as well. In this case, we'd also need to add `partial` nodes to the other alternatives and also feed the `box_data` to it. Also, since we'd be feeding the `box_data` argument to all alternatives from the very beginning, we wouldn't need to pass the `box_data` to the `perform_call` node at the end.

This ability to work with alternatives that have different signatures is actually crucial, because it allow us not only to use alternatives that require different sets of data, but also alternatives that require no data at all. Our solution so far has been addressing a problem with similar alternatives, that is, different kinds of lids (or no lid at all). All of them require some data to be passed to the chosen alternative. Even in the alternative where no lid is added, the identity function used receives the `box_data`.

However, there are also cases when no data is passed on at all. For instance, if no lid is required, you might instead want to do something completely unrelated, like sending an email or creating an entirely new 3D model. In such cases, we'll be ignoring the existence of the box data entirely. Here's the resulting graph:



In other words, the `get_bag` alternative doesn't care about the box data. Also notice that any additional argument needed was feed to the alternative beforehand when turning than into partial objects. All of what we demonstrated guarantees that the alternatives, though sometimes similar to each other, can also represent completely different operations/paths of execution. As was writing this, I also remembered that

requiring data to be passed on was one of the faults in my (now rejected) original design pointed out by Mr. Tom on discord:

> I am afraid, I am not convinced this is the right way to implement it. While I understand your reasoning and think it would work, I must say the solution does not seem to be a clean representation of an if statement. It feels like a quick hack. The main reason is the need for an end node. And also the need for a data input. It is perfectly valid to have an if statement with no `data_to_forward` in Python. But that is not possible with this solution.

Fortunately, as demonstrated until this point, keeping Nodezator FP/dataflow-compliant allows us to perform branching with none of the constraints mentioned by Mr. Tom. No signatures enforced, no special nodes required.

Completing our solution: subgraphs as a needed tool to handle complexity

There's still a final crucial piece for the completion of our solution: subgraphs (group nodes).

Subgraphs are actually an integral part of the solution. Notice that the nodes representing the different alternatives (`add_hinged_lid` , `add_sawn_lid` , etc.) we presented in the explored problems are only dummy nodes that represent an atomic operation: adding a lid to a node (or other atomic operation). However, in practice, each different alternative/branch from which we'll be choosing can't always be represented by a single node.

In fact, it often won't, which is precisely why we use a node-based interface to program, so we can combine different nodes to achieve a certain result. For instance, the alternative to create a 3D bag instead would probably be followed by other operations related to that alternative. As such, we'd probably use multiple nodes, not only the one to just create the bag. The options where a lid is added and we keep using the box data, would, in practice, maybe require many nodes to be combined and executed in order to add such lid to our box.

That's where subgraphs (like group nodes in Blender3D) are useful. This is something that will still take a while to land on Nodezator, but is something indispensable in order to be able to fully tap into branching/looping in Nodezator.

Once subgraphs/group nodes are implemented, whenever we need to represent a branch/alternative that requires the usage of multiple nodes, all we'll have to do is to create the nodes, group them together, select which inputs and outputs we want to expose, and use the resulting group node as demonstrated in our solutions: in callable mode, and with help of ternary operators or dictionaries (and `partial` nodes, to provide dedicated arguments).

I won't offer more comments on subgraphs here, but you can check this comment where I provided my recent thoughts on their design.

Wrapping it all up: requirements met, problems solved

In this quick subsection we'll revisit the requirements and problems we presented earlier, as I comment on how the previous demonstrations met/solved them.

The first half of the requirements/problems are related to the efficacy of the branching/looping tools:

- requirement: satisfy all the branching and looping requirements, i. e., be sufficient for any looping/branching use-case;
- problem: visualizing what Nodezator was lacking in order to take full advantage of FP tools.

To my knowledge, considering all that was demonstrated, there's no doubt anymore that all current and planned features met the requirement and solved the problem related to the efficacy of branching and looping in Nodezator. All the presented features, currently implemented or not, contribute to solve all possible

use-cases, that is, all that I could think of. The tools presented give us all the flexibility needed to handle multiple similar or completely different alternatives and with varying degrees of signatures and complexity.

The second half of requirements/problems are related to the simplicity and ease of learning of performing branching/looping in Nodezator:

- requirement: be as simple as possible to explain to beginners, in order to avoid a difficult learning curve and/or high cognitive load;
- problem: How to teach and encourage people to model and solve their problems purely using FP tools?

To be completely honest, the demonstrated solutions are only relatively easy to understand, not absolutely easy. I think the demonstrations represent the best available solution to the specific problem and an initial effort to meet the specific requirement. Of course, the demonstrations must be translated into more fundamental step-by-step guides to help beginners grasp the individual concepts and model their problems accordingly.

Even so, and in order to be able to begin tackling the challenge, we must admit that for a complete Python beginner, or someone who never relied/got used to FP concepts/tools, it still takes time and effort to grasp the concepts. It just can't compare in terms of simplicity to how straightforward `if/elif/else` statements work in text-based non-FP Python code and that is something we must acknowledge in order to properly address the proper.

Nonetheless, each day I'm more and more convinced that this cognitive load is inherent to FP thinking and its tools. It is not something that can simply labeled as bad in itself. Rather than thinking only in terms of cost, we must also acknowledge that despite the initial learning curve, the outcome of relying in FP tools is still 100% worth. It gives our solutions modeled and solved in a node-based interface like Nodezator all the power and flexibility needed with a relatively simple design and structure.

Because of that, rather than analyzing the usage of FP thinking/tools/modelling to solve our problems from the perspective of its costs, it is more relevant to analyse such usage from the perspective of the trade-offs, that is, the costs and the benefits. Beginners trade a bit of time and effort for unlimited power, flexibility, versatility and a fundamentally simple design and structure that fits a node-based interface perfectly, ensuring high extensibility capabilities and low maintenance cost for Nodezator.

My rejected proposal, in comparison to FP tools/concepts, also had a learning curve/cognitive load anyway (and other problems), and with far fewer benefits.

Maybe the acknowledgement and disposition to address the beginner's struggles (whether regarding FP thinking/tools/modelling or other subject) is what has been missing in the many failed attempts to popularize technologies like node-based programming and functional programming languages.

If we want to do something different, it is only natural that some effort is required. It is just how the natural world works. As such, effort must not be demonized. Another piece of software that has a difficult learning curve is Git. And yet, people go through the difficult process of learning its many features in order to be awarded with the ability to manager their development work effectively and efficiently. Thus, we must accept the effort. That is, as long as the outcome is worth of the effort. And I believe this is the case with FP thinking/modelling for branching in Nodezator.

Comment on Mr. Alexander latest replies/demonstrations

Mr. Alexander, as I pointed before in the text, the core of my demonstrations was the ternary operator that you had already explored in your replies here on GitHub. When comparing what I demonstrated here with what you have been showing in your own replies my conclusion is that this also applies to many other concepts presented here. Let's go over these more explicitly.

Your most recent reply, as the ones before it, already used the ternary operator. You also included a mechanism in your `if_then_else_fp` to return the operation representing the evaluation of the ternary operator when the last parameter is `None`, rather than performing the evaluation on the spot. In other words, you make it so the condition can be evaluated lazily if desired, which shows the efficiency of your solution.

This other reply in particular, shows the usage of partial objects to customize the behaviour of other functions like `constant()` and `divisible_by()`. The `apply()` function also portrayed in the reply is used in a similar way to how I used the `perform_call` node in my demonstrations.

All of them also explore how ternary operators can be chained.

I believe that all of such similarities, despite the different ways in which our demonstrations were represented and different kind of problems tackled, just show how versatile the FP tools/concepts are. When combined with Nodezator's dataflow-compliant design, such tools/concepts allow the same problem to be modeled and solved in many different ways.

The fact that my demonstrations explored problems in a broader scope is only due to a few situational factors like:

- my early access to the callable mode for nodes, since I'm implementing it;
- my higher familiarity and experience with Nodezator and its many features, current and planned, which:
  - allowed me to take advantage of existing nodes like `perform_call` and `partial`;
  - helped me incorporate the concept of subgraphs in my solutions;
  - provided me more insight on specific problems like the one from Mr. Adams that we explored;
- the fact that you only recently became aware of and started using Nodezator.

Given time, and once the few planned features in synergy with FP are implemented (like the callable mode, subgraphs, etc.), there's no doubt that your deeper knowledge and familiarity with FP tools will result in solutions/demonstrations even more sophisticated than the ones I shown here. I say this not as flattery, but as an objective assessment of your capabilities considering our discussions since you've joined the Indie Python community and the proficiency with which you've been exploring its features with your FP knowledge alone.

It is precisely because I know a bit of FP and am aware of your knowledge that I must take extra caution when assessing how fit our demonstrations are for beginners. That is, my challenge is to provide this same proficiency to people who are not familiar with FP, so they can easily grasp and start using the most fundamental FP concepts/tools to model and solve their programming problems in Nodezator.

Fortunately, I believe the fact that your FP/dataflow knowledge resulted in a proficient usage of Nodezator after such short time means Nodezator is being steered in the right direction, one that provides power and flexibility in problem solving within a node-based interface, using well-established paradigms like FP and dataflow (as long a such paradigms are adopted not for their name alone, but for the practical benefits they bring to our software development).

With this I conclude my analysis regarding branching in Nodezator, specially in response to Mr. Alexander's replies/contributions to the topic.

As always, Mr. Alexander and anyone reading this, if you think any point here needs extra clarification, feel free to ask me anything. If you have anything to add or feedback on what's been discussed, it is also welcome.

Peace.