OpenDF – Application Developer's Tutorial

About this document:

In this tutorial, we will go over the dataflow implementation of a toy application.
This document is written for a developer of an OpenDF **application**, whose primary interest is to implement a new application using the OpenDF library (rather than a core OpenDF developer wanting to modify and extend the OpenDF library itself). Ideally, as an OpenDF application developer, you will not have to look at any of the library code during this tutorial.

Note, that the concepts at the heart of OpenDF are basically simple, and hopefully this tutorial will clarify that. There are, however, a lot of details on top of the core concepts, which may be confusing, but hopefully are beyond the concerns of the typical user.

Rather than starting with detailed explanations of the library, we'll go through the implementation of a toy example, and introduce OpenDF elements the user is likely to need, as they come up, trying to avoid unnecessary details.

We'll cover the basic functions you are likely to need for implementing simple applications. From time to time we'll introduce some advanced details, which you're unlikely to use, and can be skipped, but may be interesting for some.

Implementing an application on top of OpenDF requires making multiple design decisions on how to structure the application. We will look at some examples of these design decisions and their implications.

Starting point:

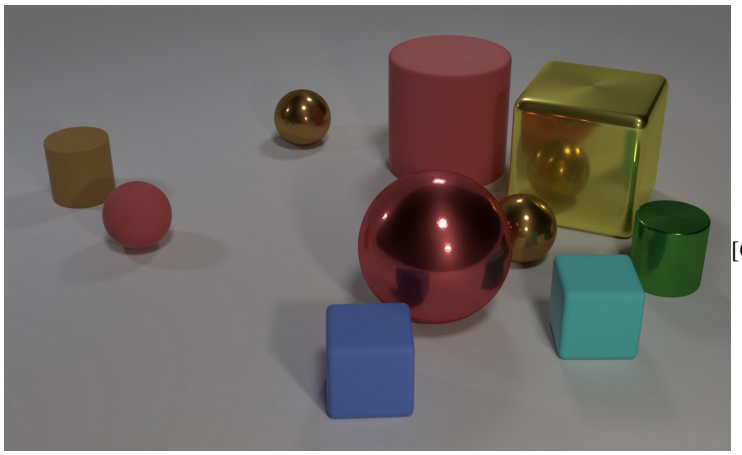
You should be able to understand this tutorial just by reading it, but a better way is to run the examples as we go through them, experimenting with variations, and verifying your understanding. If you want to take the hands-on approach, you should install the OpenDF package, and make sure it's running correctly on your system. See the README.md file for the explanation how to install and run the system.

The code for the implementation is in the "tutorial" directory, in the `blockWorld_V`*`n`*`.py` file(s) (and `block_examples.py`). The different versions track the progressive changes we'll make to the implementation.
To run, use:

```
PYTHONPATH=$(pwd) python opendf/main.py  \
                  -n tutorial.blockWorld_Vn  \
                  -ef tutorial/block_examples.py \
                  [-d dialog_id#]
```

In order for this tutorial to make sense, you should be familiar with the basic concepts of dataflow dialogue (presented in the original Semantic Machines paper), and should have looked at the (several) README files in OpenDF.

Toy application - BlockWorld:



[CLEVR]

As an example application, we will use BlockWorld, which uses as inspiration Winograd's SHRDLU (https://en.wikipedia.org/wiki/SHRDLU), and the example in the CLEVR paper (https://arxiv.org/pdf/1612.06890.pdf). (We will not assume you are familiar with them).
We will pick some elements of these systems, and add some new ones, in order to demonstrate different aspects of implementing an OpenDF application.
Please remember that there are may ways to implement an OpenDF application. We'll see a couple of designs, but there are many other ways.
Let's start with a simple application, inspired by the image above: we want the application to have building blocks of different shapes (cube, pyramid, ball), sizes (small, big), colors (red, blue, yellow), and materials (wood, metal, plastic).
On top of this we may add some of the following functionalities, allowing the user to:
- add/remove objects
- move the objects
- stack objects on top of each other
- ask questions about properties of objects and relationships between objects
- try to implement a simple game on top of this setting

Let's start!

First Object:

Let's start with implementing a simple object, for example a cube. We add a new class `Cube()`.
All objects in an OpenDF are ultimately derived from the base class `Node()` (therefore we often refer to these classes as *node types*).

`Node()` takes care of the basic functionality of the dataflow graph (like keeping track of which nodes are connected to which node, access to node properties, evaluation of nodes etc.), so that when we define a new node type, we only have to implement the specific logic unique to the new node type, and we can rely on the base `Node` to handle the common behaviors.

Let's define Cube (we'll start with Cube1, refine it to Cube2, and so on). For our application, a cube needs to have the attributes: size, color, material. The following code defines the new class Cube1 (derived from Node).

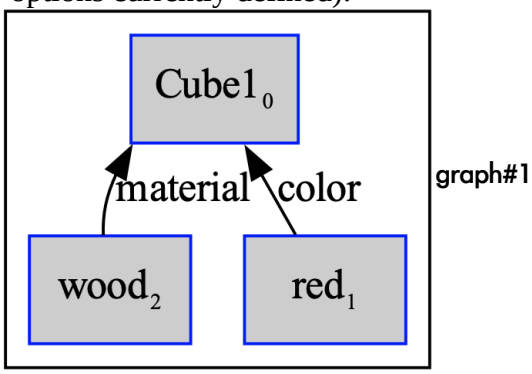
```
1 class Cube1(Node):
2    def __init__(self):
3        super().__init__(type(self))
4        self.signature.add_sig('color', Str)
5        self.signature.add_sig('material', Str)
6        self.signature.add_sig('size', Str)
```

The `__init__` method initializes a new instance of `Cube1` in the following way:
- Does the initialization needed for the base `Node` class (line 3)
- Defines the `Signature` of the node (lines 4-6).

Each node has a `signature`, which defines which inputs it can **potentially** accept. For each input, we define:
- obligatory:
  - the name of the input
  - the type of the input
- optional:
  - additional features governing the behavior of the specific input

In this example, we define `Cube` to allow an input named "`color`", whose type is `Str` (a string).

We're now ready to run our first DF program (Pexp)!
[1] `Cube1(color=red, material=wood)`
(the notation "*[1]*" indicates this is expression #1 in block_examples.py, so to run it, add: "`-d 1`" to the command line.
Note that this looks quite similar to a python expression.
Running this produces the following output (slight changes are possible, depending on the draw options currently defined):



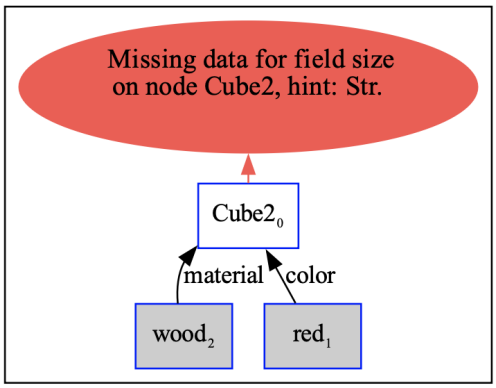
We can see the dataflow graph that was generated. Each block represents one node, with either the node type or its value shown, along with the node id shown as subscript.
An instance of `Cube1` was created (node id=0), which has two inputs (incoming arrows). One input is named "`color`", and is a node of type `Str`, having the value "`red`", and the node id=1). The other input's name is "`material`", is of type `Str`, node id=2, and has value of "`wood`".

Behind the scenes, this is what `main.py` did:
- Parsed the Pexp (making sure it's syntactically correct)
- Constructed the DF graph (created the requested nodes, connected inputs nodes to their parents)
- Verified that input nodes have allowed names and types
- Evaluated the graph (in the current example, nothing to do)

In addition to `Node.signature` (describing **potential** inputs), there is also `Node.inputs`, which holds the **actual** inputs which are connected to this node.
Note that the example expression did not specify the "`size`" attribute for the cube – there is no "`size`" entry in the inputs of the created `Cube()`.

If we wanted to ensure that Cube gets a size input, we could add the parameter `oblig=True` to the signature (we add this to `Cube2`):

```
6  self.signature.add_sig('size', Str, oblig=True)
```
Which causes an exception to be raised if the ‘size’ input to a Cube is missing. Now, running the same expression (*[2]*) will produce:



graph#2

The red balloon shows the exception. It is attached to a node (typically the one which raised it), and shows the text message.
Exceptions are our friends - They do not imply there is a bug in the implementation. They are the way the application interacts with the user.
In this case, we told the user that an input is missing, so the user can provide additional input, if they want.

After the graph is successfully constructed and checked (syntactic and type correctness), the graph is evaluated. The evaluation process proceeds in a **bottom up** order. For each node, the evaluation consists of some (optional) custom checks on the input, and then running of (optional) custom execution logic (both custom checks and custom execution logic, if they exist, are defined for each specific node type). If either the check or the execution do not complete successfully (by raising an exception), the bottom up evaluation of the entire graph is halted.
This setup guarantees that if we’re evaluating a node, all of its input nodes have already been evaluated successfully.

Note in **graph#2**, the blocks for nodes #1 and #2 (“red” and “wood”) are colored gray, while node #0 (`Cube2`) is white. This is an indication of the evaluation status (successfully evaluated / not successfully evaluated yet) of the different nodes. “red” and “wood” have been successfully evaluated (no exception thrown), so in the drawing they are colored. `Cube2` threw an exception during its evaluation, therefore it has not been successfully evaluated.

Adding `oblig=True` to the signature had the right effect, but the error message which was generated is very generic, and not very user friendly.

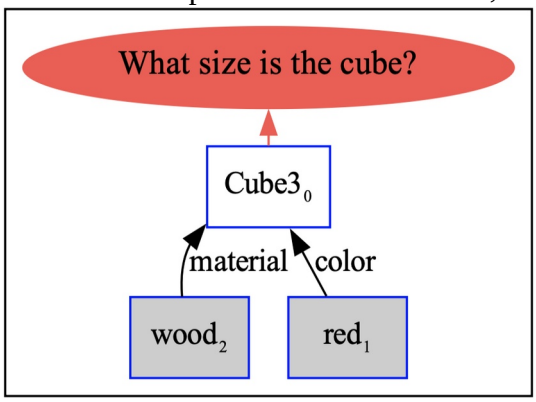
Another way to achieve the same effect is to use the previously mentioned custom check function which is called during the evaluation of a node.

This would look like this:

```
1 class Cube3(Node):
2    def __init__(self):
3        super().__init__(type(self))
4        self.signature.add_sig('color', Str)
5        self.signature.add_sig('material', Str)
6        self.signature.add_sig('size', Str)

7  def valid_input(self):
8        if 'size' not in self.inputs:
9            raise DFException('What size if the cube?', self)
```

Which would produce a similar result, but this time the exception message is customized. *[3]*



Graph#3

Custom checks could be used not just to check if an input is missing, but also to check that the values of several inputs are consistent with each other, etc.
So far, we have defined the cube attributes (size, color, material) to be of type Str (string).

A better design may be to create a separate type (node type) for each of these attributes, which would encapsulate the logic for each of them. For example, the type for size will know to accept only ‘big’ and ‘small’. Otherwise, `Cube` (as well as `Pyramid`, `Ball`…) will have to check for each attribute if it got a valid value. (if `Cube`, `Pyramid`,... allow different sets of values, such encapsulation may be less useful)

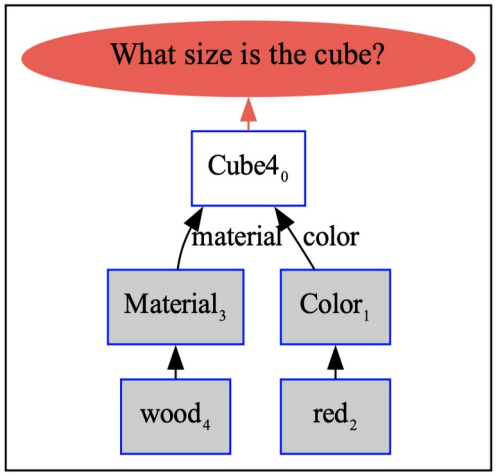
```
1 class Color(Node):
2    def __init__(self):
3        super().__init__(type(self))
4        self.signature.add_sig(posname(1), Str)

5    def valid_input(self):
6        valid_colors = ['red', 'yellow', 'blue']
7        dt = self.dat
8        if dt is not None:
9            if dt.lower() not in valid_colors:
10               raise DFException(dt + " is not a valid color!", self)
11       else:
12           raise DFException(message="Please specify a color", self)
```

With this definition, the following Pexp’s will produce these results:
- `Color()` → will raise exception: “`Please specify a color`”
- `Color(green)` → will raise exception: “`green is not valid color`”
- `Color(red)` → will evaluate successfully

Note that when calling e.g. `Color(red)`, we did not specify a name for the parameter of `Color()` - we used a positional parameter (similar to python).
In order to specify a positional parameter in the signature of the node, we use the notation `posname(n)` to refer to the name of the positional parameter at position n. See line 4.

Note: OpenDF currently has only 4 types of basic data: `Int, Float, Str, Bool` (they are called – *base types*).
In order to help simplify Pexp's, we allow shorthand notations when it comes to base types.
The full expression for the color red should be: `Color(Str(red))`. However, we also allow the shorter expression – `Color(red)`. During graph construction, the missing node (Str in this case) will be automatically added. When drawing the graph, we do not show the added base type node, in order to simplify the drawing (in **Graph#4** – we don't draw separate `Str` nodes for `Color` and `Material`).



Graph#4

After changing the signature of `Cube4` to the following:

```
4         self.signature.add_sig('color', Color)
5         self.signature.add_sig('material', Material)
6         self.signature.add_sig('size', BlockSize)
```

Running **Pexp#1** now generates **Graph#4.** *[4]*

`Note: Node().data` is where the actual value of a base type is kept: e.g. for a node `Int(4)`, the field "`data`" will contain the integer 4.
The data field is used **only** for *base types*. For all other types it is `None`. For example – for `Color(Str(red))`, the data field for the `Str` node is "`red`", but for `Color` it is `None`.
Practically, if we wanted the value of a Color node, how would we get it?
In **Graph#4**, looking at node#1 (`Color`), how do we get the value "`red`"?
we would have to start from node1 (`Color`), get the node which is its first positional input (node2 `Str`), and then get the `data` field for that node:
`node.inputs[posname(1)].data` **(1)**
Note: we use the term *leaf node* to refer to nodes which allow only one input, and that input's type is a base type (e.g. Color is a leaf type). For leaf types, it is clear what the value of the node is, as opposed to, e.g. asking the value of a Cube node, which is an ambiguous question.
In addition to `Node.data` (an attribute of Node), we also have `Node.dat` (a property of Node), which returns **(1)** in case of a leaf node, and `Node.data` otherwise.

Line 7 in the code snippet for `Color` uses `self.dat` to get the data of `Color` (in case no value was given, `dat` returns `None`)

Note: regarding exceptions, in this tutorial we'll be using the generic `DFException` when we raise an exception. In practice there is a whole set of different exception types, but for this exercise, we will keep with one exception type, for simplicity.

Multiple Shapes:

Now that we have defined a Cube type, let's add more shapes – Pyramid and Ball.
There are several ways we could do this:

1. Simply duplicate the `Cube` code for each shape
2. Define a base class (`Block`), and inherit each of the shapes from it
3. Define a `Block` type, but instead of inheriting from it, simply add a "shape" attribute to it

Option 1 will result in code duplication, and could be inconvenient e.g. in case we want to refer to a red block, not caring about its shape.

Option 2 is much nicer from an Object Oriented Design (OOD).

Option 3 avoids the need to have separate types for each shape, but then if different shapes have custom behaviors, the code for `Block` will get messy.

Let's go with option 2:

```
1  class Block(Node):
2    def __init__(self, out_type=None):
3        out_type = out_type if out_type else type(self)
4        super().__init__(out_type)
5        self.signature.add_sig('id', Int)
6        self.signature.add_sig('color', Color)
7        self.signature.add_sig('material', Material)
8        self.signature.add_sig('size', BlockSize)

9  class Cube(Block):
10   def __init__(self):
11       super().__init__(type(self))

12 class Pyramid(Block):
13   def __init__(self):
14       super().__init__(type(self))
```

For now `Cube` and `Pyramid` share the behaviors of `Block`, so their definition is very short – they just call the `__init__` method of their base type (`Block`).
Note that the parameter given to the __init__ of the base class is the output type of the node. For "objects" the output type of a node is typically the same as its own type (e.g. the output type for `Cube` is `Cube`). (it gets more interesting for "function" nodes).

Describe a graph:

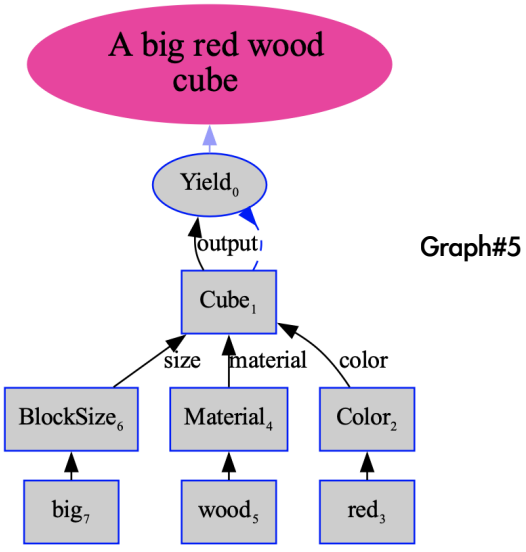
`Node.describe()` is used to describe a graph (or sub graph), typically in a recursive way – the parent node calls the describe function of its inputs, and then composes the results of the calls to describe into a new message.

For `Block`, we can define:

```
1   def describe(self, params=None):
2        dats_dict = self.get_dats_dict(['size', 'color', 'material'])
3        dats_dict.update({'shape': self.typename().lower()})
4        key_value_strings = []
5        text = ''
6        for i in ['size', 'color', 'material', 'shape']:
7            if dats_dict[i] is not None:
8                key_value_strings.append(dats_dict[i])
9        text = ('' if params and 'no_art' in params else ' A ') +
10                   (' '.join(key_value_strings))
11       return Message(text=text)
```

In line 2, we collect the values for the different inputs of the block (some may be `None`, if the corresponding input does not exist).
In line 3, we add the attribute shape. Remember that `Block` has derived classes, so this code should work for `Cube, Pyramid,`...
To get the shape name, we use self.typename(), which returns the name of the type of the node (e.g. "Cube" for a node of type `Cube`).
We collect the non `None` values into a list (lines 6-8). The returned message is composed by joining these values (line 10), resulting in e.g. "big red plastic pyramid".
Note that the description we need for a node type may change, depending on where we use describe(). To allow variations in the generated description, we can pass `describe()` additional parameters as a list (`params`). In this example, We add the indefinite article "A" to the description ("**A** big red plastic pyramid"), unless `params` indicate no article should be added ('no_art' in `params`).

So far all the messages we've seen have been exceptions – the system complains about an unsuccessful evaluation. If the evaluation is successful no exception is raised, and no message is returned.
Using the function Yield() allows a node to generate a success message. This is done by calling that node's describe(). (skipping some details). For example:
`Yield(Cube(color=red, material=wood, size=big))` *[5]*
produces the following graph and message (where the message is generated by `Cube.describe()`):



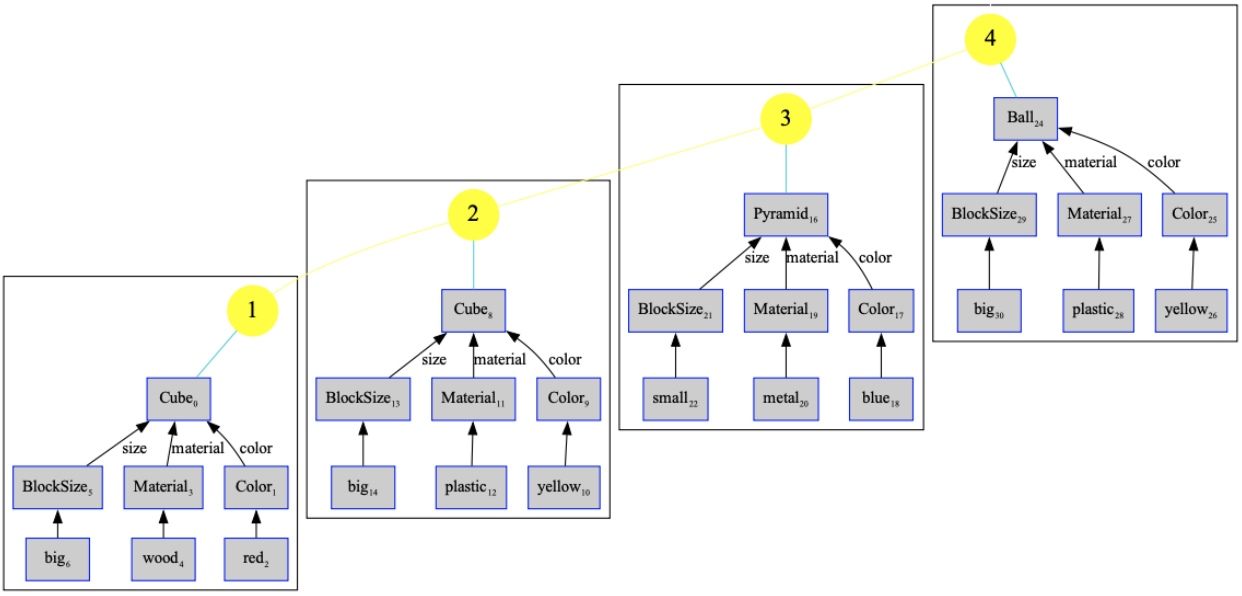
Graph#5

Multiple Objects:

Now that we've defined `Cube`, `Pyramid` and `Ball`, we can instantiate multiple objects:

```
Cube(color=red, material=wood, size=big),
Cube(color=yellow, material=plastic, size=big),
Pyramid(color=blue, material=metal, size=small),
Ball(color=yellow, material=plastic, size=big),
```

Running this produces: *[6]*



Graph#6

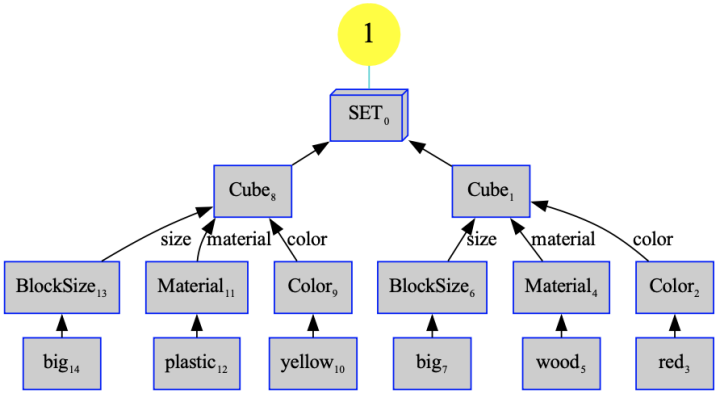
We can see that 4 separate dataflow graphs were created – one per block.
The nodes (and graphs) are actually held in a class (`DialogContext`), along with additional parameters describing the current state of the interaction.
We use the term *goal* to refer to the top node of a computation (a graph) (the name makes more sense for "function" nodes).
`DialogContext.goals` holds the top nodes of the different graphs which were created during the interaction (dialog).
In the example in **Graph#6**, `DialogContext.goals` consists of the top node of each of the 4 blocks which were created.
The number in the yellow circles point to entries in `DialogContext.goals`.
As the interaction continues, more goals are added, with the most recent goal being the last one in `DialogContext.goals`.

Collection of Objects:

We use the node `SET()` to hold a collection of objects.
Actually, this is like a python list, not a set, as `SET` can have the same object more than once.
Note: we freely interchange between "node" / "graph" / "object" / "calculation": for `Node` N, the "graph" are all the nodes under N. The graph may correspond to an object or a calculation (depending on the type of N), so we use these terms accordingly.
More precisely, a `SET` allows an unlimited number of positional parameters, each of them can be a node of any type.

For example, a collection consisting of two cubes:

```
SET(Cube(color=red, material=wood, size=big),
    Cube(color=yellow, material=plastic, size=big))
```
[7]



Note that `SET` is drawn using a "box" icon. This is the first example of an `Operator` – we'll see more soon.

Querying for nodes:

Now that we have multiple blocks we can use the `refer()` function to select blocks which satisfy specific conditions.

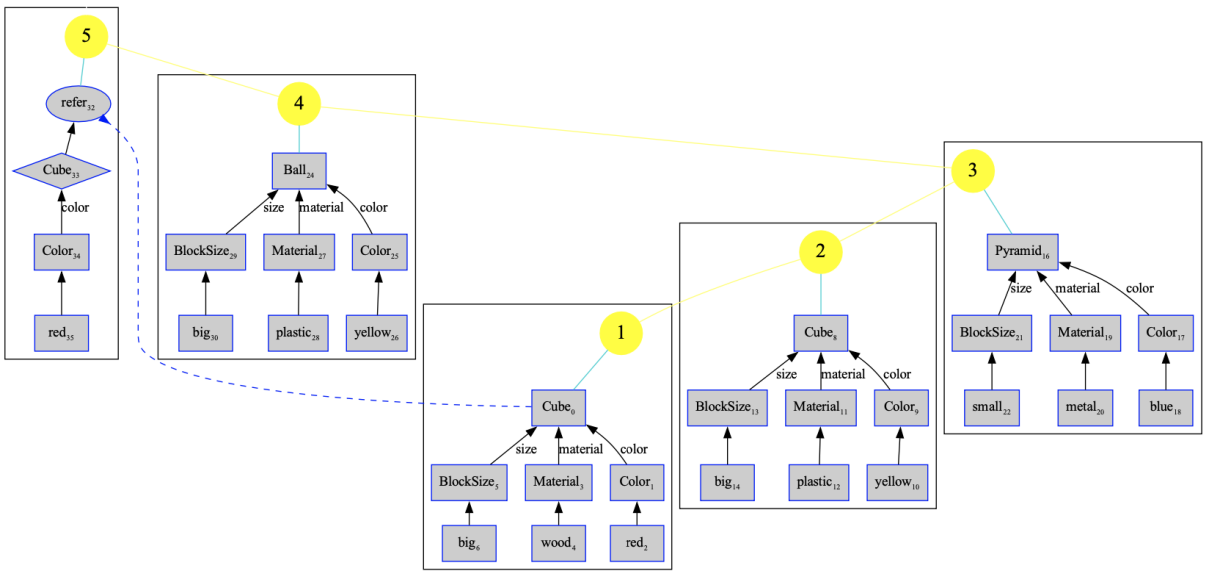
SMCalFlow uses the term *constraint* for these conditions. Since they are used exclusively for searching for nodes, we also use the term *query* interchangeably.

Note the difference between an object (e.g. a specific big red plastic cube) and a query (e.g. look for a red cube). Given the query, the search engine goes over all of the existing nodes, collecting only those nodes which match the query condition. The result could be either one, many or no matching nodes.

In OpenDF, the syntax to define a query is very similar to that of describing an object. For example, looking for a red cube:

```
refer(Cube?(color=red))
```
[8]

Graph#7



Note the "`?`" after `Cube` in the above Pexp – it indicates that the node is a query (constraint).

Note: we do not force the same checks (`valid_input`) on the query node, as typically it **is** ok to give only a partial description when searching for objects.

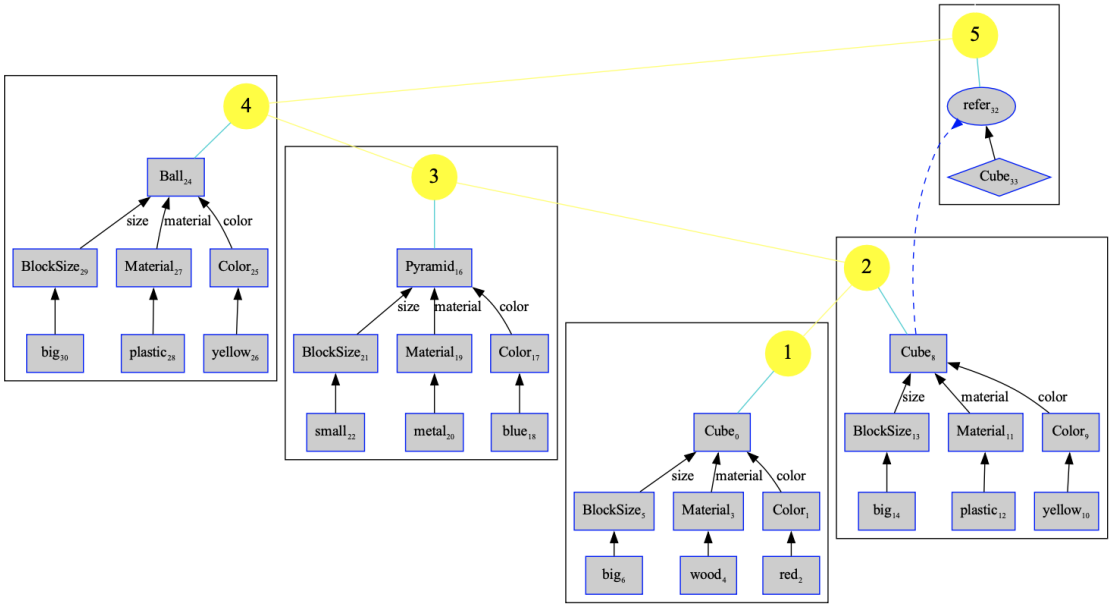
`refer()` is an example of a "function" node – it actually does something, and has a **result** (indicated by the dashed blue arrow). The result is kept in `Node.result` (for object nodes, `Node.result` points to the node itself). `Node.res` is a property of Node, which is similar to `Node.result`, except that it returns the transitive result in case of a chain of results.

We can see that the result indeed points to a red cube.

Note that in the drawing, the node of `refer()` is shown as an ellipse, to indicate it's a "function" node.
(The distinction between "function" and "object" nodes is fuzzy, and for conceptual use only. For the purpose of drawing, any node whose declared output type is the same is its own type is considered conceptually as an "object", otherwise – a "function").
Note that the constraint node (the `Cube?()` node, under goal#5) is drawn as a diamond to indicate that it's a constraint.

Here are a few other queries – you can verify that they indeed work correctly:
`refer(Pyramid?())` *[9]*
This looks for a Pyramid, with no specification on its attributes.

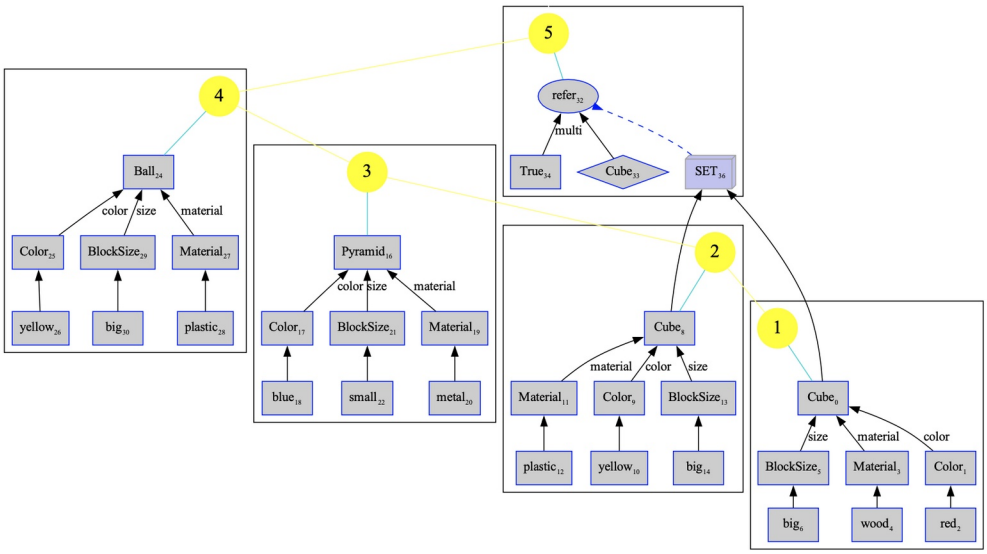
`refer(Cube?())` *[10]*



Graph#8

Similarly, this looks for a `Cube`, without any conditions on its attributes.
Note that we actually have defined 2 `Cubes`, but refer returns the `Cube` of goal #2. Why is that? By default, refer returns only one result. This corresponds to a common behavior in natural dialogs, where if at some point in the dialog, participant A asks e.g. "what is the phone of the customer?", the common interpretation (by participant B) would be to assume participant A meant the phone number of the **last** customer that came up in the conversation, even though other customers may have been mentioned previously in the conversation. More generally, it does not have to be the last object of the searched type, there could be some other pragmatics to select what is the "natural" choice in case several options are available. (SMCalFlow uses the term *saliency function* to denote this pragmatics). In practice it's mostly the last (most recent) matching node which is selected. In Graph#8, you can see that refer selected the second `Cube` (since it's more recent than the first `cube`).
`refer()` actually found the two `Cubes`, sorted them by the *saliency function*, and returned the `Cube` with the best saliency score.

We can tell `refer` to return multiple results (e.g. if the user explicitly indicated the result is plural – e.g. “what are the phone numbers of these customers?”), by adding the parameter `multi`:
`refer(Cube?(), multi=True)` *[11]*

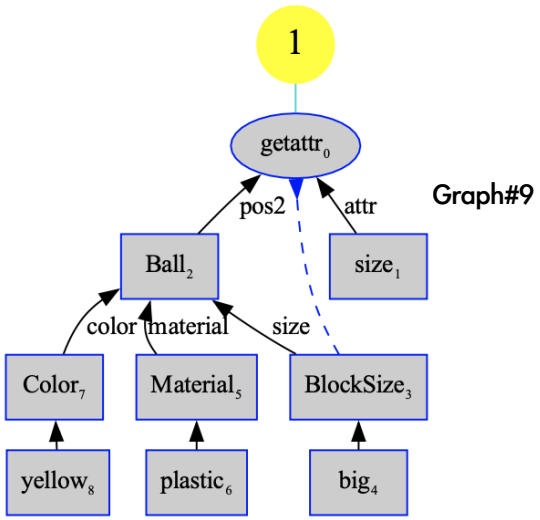


`refer` now returns a `SET` consisting of the two `Cubes`.
The query:
`refer(Block?(material=wood))`
looks for a `Block` made of wood, and could find any of the subtypes of `Block` (`Cube`, `Pyramid` and `Ball`) with the required attributes (made of wood).

Getting node attributes:

The library function node `getattr()` can be used to get an input (attribute) of a node. It has two positional parameters: pos1- the name of the attribute; pos2 – the node for which we want to get the named attribute. In our current example, if we wanted to get the size of a given `Ball`:
`getattr(size, Ball(size=big, material=plastic, color=yellow))` *[12]*

will produce:



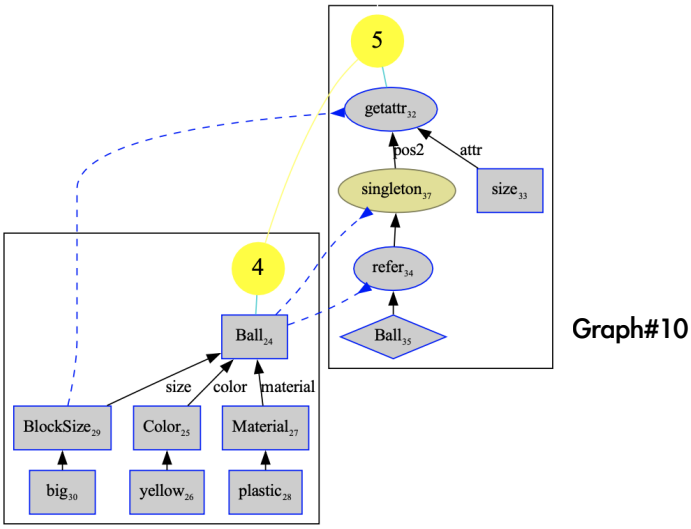
As can be seen, the result of `getattr` is the `BlockSize` node of the `Ball`.

There is a shorthand notation to `getattr` (inherited from SMCalFlow, called *sugared get*):
`:size(Ball(size=big, material=plastic, color=yellow))`

(During the construction of the graph, it is converted to `getattr`, so we end up with exactly the same graph).

We've asked: "what is the size of the big, plastic, yellow ball?".
Now, let's ask: "what is the size of the ball?" - first we need to find the ball among all the blocks that were declared, and then get the size of the ball we found.
We use `refer` to find the `Ball` among the 4 `Block`s which we have created already, and then apply `getattr` to that:

```
:size(refer(Ball?()))
```
[13]



Graph#10

This is an example of function composition: `getattr()` operates on the result of `refer()`.
Note that we omitted drawing the other 3 `Block`s in **Graph#10** (by adding `-env show_only_n=2` to the command line, drawing only the last two goals), but they still exist as before in `DialogContext`.
(The `singleton` node was automatically added to handle the case `refer` returns multiple results – ignore this for now).

More queries:

Let's write Pexp's to execute slightly more complicated requests.
As a reminder, here are the blocks we have:

```
Cube(color=red, material=wood, size=big),
Cube(color=yellow, material=plastic, size=big),
Pyramid(color=blue, material=metal, size=small),
Ball(color=yellow, material=plastic, size=big),
```

"find a cube with the same color as the ball":
Let's break this down:

1. find the Ball → `refer(Ball?())`
2. get its color → `:color(...)`
3. find a Cube with that color → `refer(Cube?(color=…))`

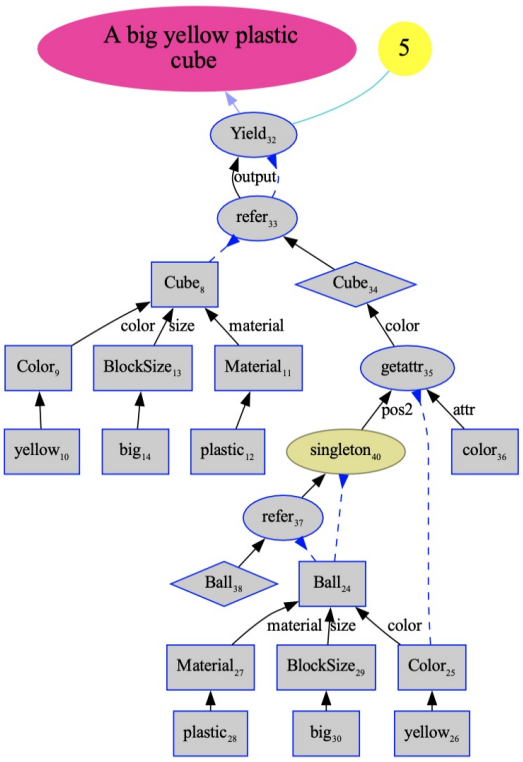
put together:

```
refer(Cube?(color=:color(refer(Ball?()))))
```

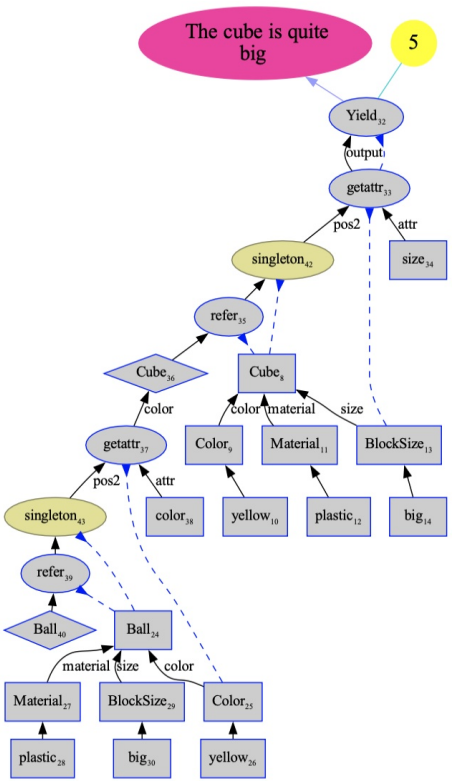
and, as always, we can add a Yield around this to get a text description of the result: **Graph#11**

```
Yield(refer(Cube?(color=:color(refer(Ball?())))))
```
[14]

Graph#11



Graph#12

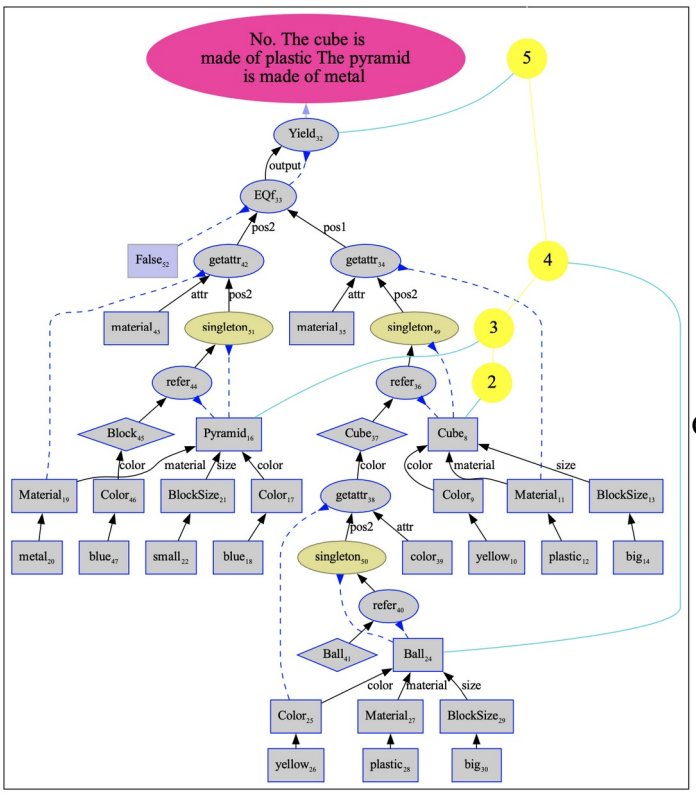


"what's the size of the cube with the same color as the ball": **Graph#12**

```
Yield(:size(refer(Cube?(color=:color(refer(Ball?()))))))
```
[15]

"is the material of the cube, with the same color as the ball, the same as that of the blue block?"

```
EQf(:material(refer(Cube?(color=:color(refer(Ball?()))))),
    :material(refer(Block?(color=blue))))
```
[16]



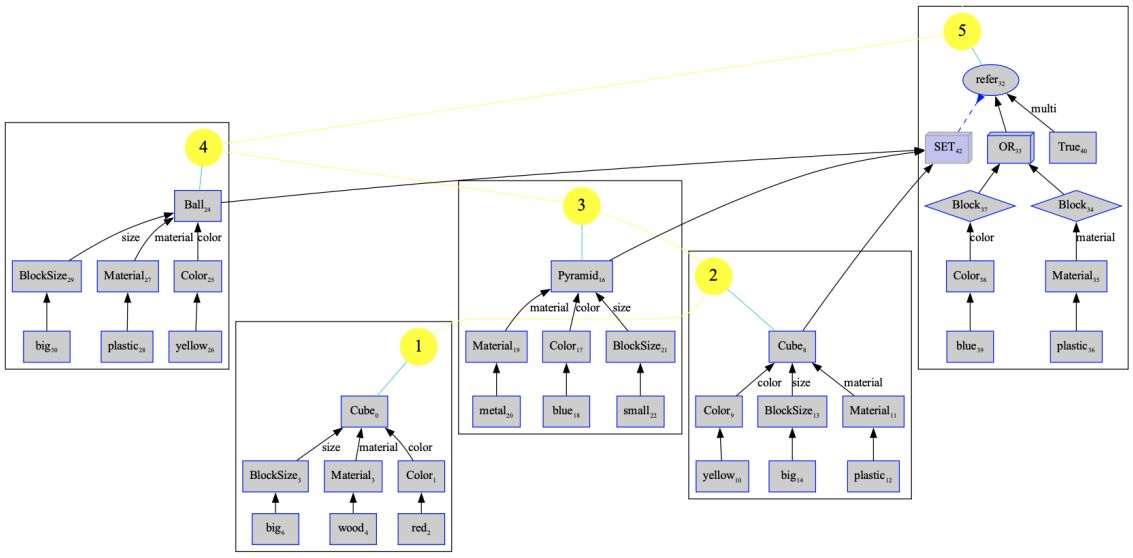
Graph#13

The function `EQf()` checks if its inputs are equal to each other.
In our example, the answer is `False`, as the specified `Cube` is plastic, and the blue `Block` is metal.

Queries with logic operators:

We can combine queries by using logical the operators `AND`, `OR`, `NOT` (and others).
For example: find the `Block`s which are plastic or blue: *[17]*

```
refer(OR(Block?(material=plastic),Block?(color=blue)), multi=True)
```



`refer` found the plastic `Cube`, plastic `Ball`, and blue `Pyramid`.

Revise:

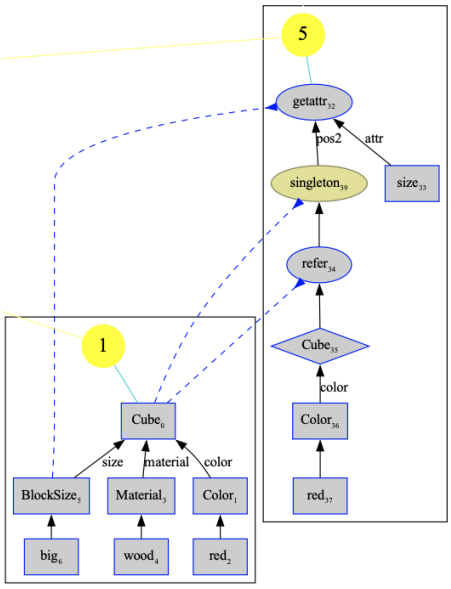
`revise()` allows us to reuse and modify a previous computation (graph), and then reevaluate it.
This is a very common strategy in human dialogs – instead of tediously re-stating the full modified computation, only the modification is specified.
For example – Q: "where did Dan have breakfast yesterday?" A: … Q: "And John?"
Conceptually, in order to be able to revise a previous computation, we need to know the following:

- What is the computation we want to modify → *RootLoc* in SMCalFlow
- What in that computation we want to modify → *OldLoc* in SMCalFlow
- What is the modification → *new* in SMCalFlow

Let's see a simple example:
question: "what is the size of the red `Cube`?"

```
:size(refer(Cube?(color=red)))
```

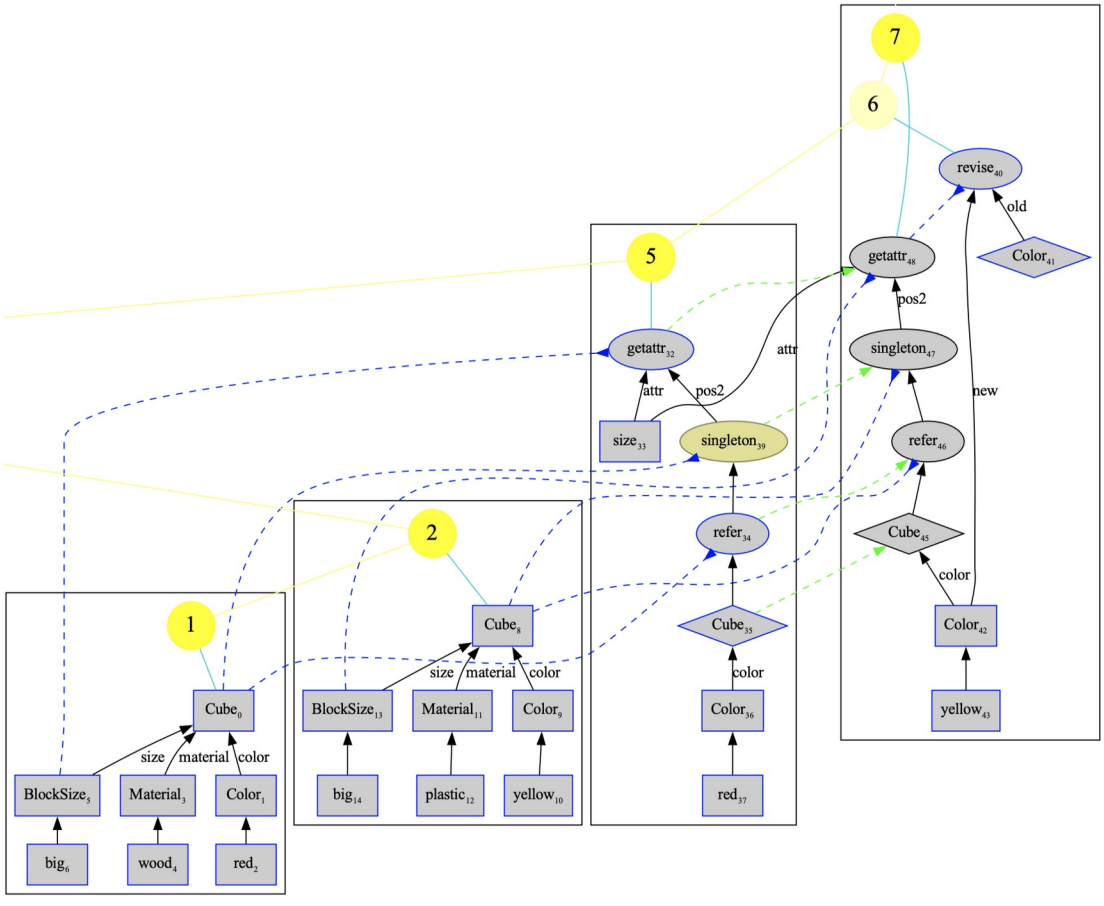
Pexp#4 *[18]*



Graph#15

follow-up: “and the yellow one?”
The user wants to know the size of the **yellow** `Cube`, instead of the **red** `Cube`, i.e. the user wants to execute the following:
`:size(refer(Cube?(color=yellow)))` **Pexp#5**
by revising **Pexp#4**.

The user just said that the new value should be yellow, but did not explicitly mention that the old value was red – so translating the user request to “revise the old value red to the new value yellow” is more difficult than translating it to “revise the color to have the new value yellow”.
Note that the user did not explicitly mention “color” – the translation will still need to understand that since the new value (yellow) is a color, we are trying to modify an existing color node.
The `revise` Pexp to modify **Pexp#4** to **Pexp#5** is:
`revise(old=Color?(), new=Color(yellow))` *[19]*
Which produces: **Graph#16**

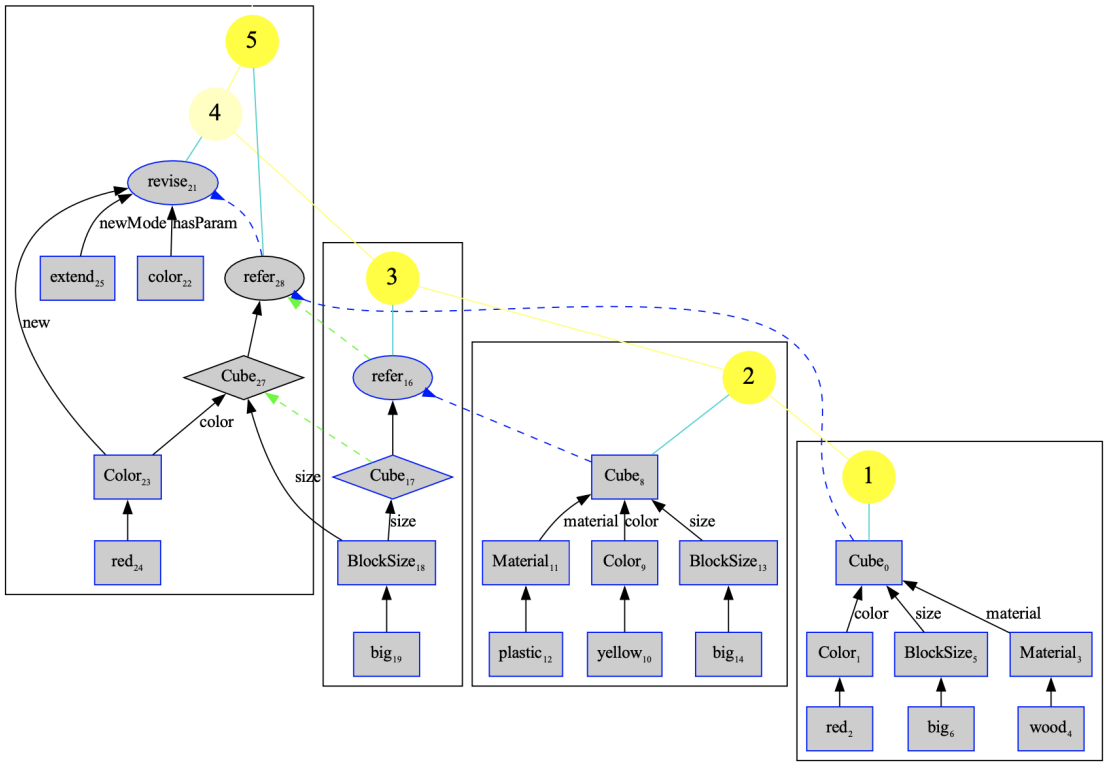


In **Graph#16,** we can see the `revise` function at work – it takes goal#5 from **Graph#15** (`getattr(size,...)`) and creates a modified copy of it (goal#7), with the `Color` now being yellow instead or red.
Looking closer we can see:

- revise identified node#36 (`Color(red)`) as *OldLoc*
- since no RootLoc was given to revise, it inferred that the root is the top node (goal) of the computation containing *OldLoc* (i.e. node#32 – `getattr`)
- revise then duplicated the computation of *RootLoc*:
  - it made a copy of all the node starting from *RootLoc*, down to (but not including) *OldLoc*
    - The green arrows show which nodes are the duplication of nodes of the original computation.
  - It then added the "new" input of `revise` (node#42 `Color(yellow)`) to the duplicated graph, at the place *OldLoc* used to be.
- The newly duplicated computation (node#48 `getattr`) is added as a new goal (goal#7), and then gets evaluated.

Note that in order to reduce the graph size, the duplication process **reuses** (rather than duplicates) any node which is not on the direct path between *OldLoc* and *RootLoc* - e.g. node#33 (Str(size)) is reused by the new graph, rather than being duplicated.

In the example above, revise has effectively replaced node#36 (`Color(red)`), by node#42 (`Color(yellow)`).
There are other options to revise a computation. For example, if we have a query for a big `Cube`: `Cube?(size=big)`, and we want to change it to a query for a big **red** `Cube`, i.e. we want to "extend" the graph, not completely replace it.
Note that in t`Cube?(size=big)`, there is no `color` input (and no `Color` node), which means `revise(old=Color?(),new=…)` will not work.
Instead, we can use:
`revise(hasParam=color, new=Color(red), newMode=extend)` *[20]*
Using `hasParam` tells `revise` to look for a node whose signature accepts the given input name. `newMode` specifies how to use the node `new`, in this case – extend the node (which accepts the requested input name) – we "extend" the (duplicated) `Cube?(size=big)` by adding a new input (with the given input name `color`) connected to the `new` node (`Color(red)`).



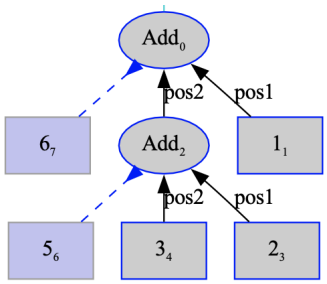
We can see how node#27, which is a duplication of node#17 (`Cube?(size=big)`), copies the `size` input, and adds the `color` input.

If `newMode` is omitted (as in the first example we've seen for `revise`), then `revise` does a replacement of `OldLoc` by `new`.

Other variations of `revise` are available.

View:

Let's look at an example of function composition. It's something we're so used to do, that we may miss the finer details.
Let's say we have a function `Add()` which adds two `Int`s and creates an `Int` result.
Now let's look at `Add(1, Add(2,3))`:



Node#2 (the bottom `Add`) has two inputs (`Int(2)` and `Int(3)`). It adds them, to get the value 5, creates node#6 to hold this value (`Int(5)`), and sets the result of node#2 to point to node#6.
Node#0 (the top `Add`) does the same, but notice that its second input is an `Add` node (node#2), not an `Int`. What we have to do (obviously) is to use the **result** of node#2 as input to node#0.

If `a` is node#0, then `a.inputs[posname(1)]` is node#1 (`Int(1)`), and
`a.inputs[posname(2)]` is node#2 (`Add(2,3)`).
`a.inputs[posname(2)].res` (node#6, `Int(5)`) is the value we actually want to use for the top addition.

In OpenDF, we use the term *view modes* for these two ways of "consuming" a node as input node – using the node itself (`VIEW_INT`), or using its result (`VIEW_EXT`).

In principle it's possible that some nodes would use an input using the `VIEW_INT` view mode (i.e. use the node itself, and not its result), but in practice almost always we use `VIEW_EXT` (i.e. use the result of the node).
The good news is that there is a way to automatically take care of the view mode, so you don't have to think about which mode to use.
All you have to do is use `a.input_view(inp_name)` instead `of a.inputs[inp_name]` (where inp_name is a name of the input). `input_view` will know which node to use.

Note: in SMCalFlow, *intension* and *extension* are explicitly (and frequently) mentioned in their dataflow expressions. It turned out that for the use cases we've seen so far in OpenDF, this is hardly ever an issue, but still it's best to use `input_view` to be on the safe side.

Time for some design decisions…

This discussion is added here not because it's really important which decisions we take for our toy application, but because it gives some examples of the considerations and trade-offs of different dataflow application designs.

Who owns the nodes?

In the previous example, we've let the user create their own blocks at will.
This may be ok in some cases, but very often it is not what we want.
Consider an application implementing the game chess. We would not want the user to be able to add or remove chess pieces at will, or move them without any restrictions.
Similarly, in a bank application, we will not allow a user to just declare they deposited 1M$ to their account.
We want the system to act as a gate-keeper, or a mediator: treat user input as requests, not as facts, check if they are valid, and execute them when appropriate.

To achieve this, we need to design the interaction accordingly, by

- controlling the translation of user request to Pexp's, e.g.:
  - instead of translating user request "create a red… cube" to `Cube(color=red…)`,
  - translate it into e.g. `request_add_block(Cube(color=red…))`
- implementing the gate keeping logic
  - For the example above – `request_add_block` checks that it's ok to create the `Cube` (no game rule is violated). If so – create, else raise an exception

Suppose we wanted to implement a game "**Griddy**", where some given blocks are placed on a grid, with each grid point having a cost, and we define a (hopefully not too trivial) total cost function as a combination of the blocks' attributes and the costs of the grid points they are placed on (and maybe some interaction between neighboring blocks/grid points).
The goal of the game is for the user to move the blocks so that the total cost is minimal.
Obviously, we would not want the user to decide on the location/attributes/costs of the initial configuration, or to allow the user to remove blocks – these operations should be controlled and executed by the system.

Adding a grid:

In order to implement Griddy, we'll need to have a grid (with associated costs), and a way to indicate that a specific block is at a specific position.

There are multiple ways we could do this (there always are…), each one with its own implications.

We could have a design which is "pure" dataflow – i.e. all objects are directly accessible and modifiable by simple `refer / revise`.
Alternatively, we could have a design which does not support direct `refer / revise`. For example, some of the data may be kept internally inside a node, or stored in an external database (similar to the way events are kept in SMCalFlow) (this makes sense in cases where the information is dynamic – e.g. it may be changed by other users on other systems, so we can not rely on our dataflow graph to be up to date). This internal data is then not part of the dataflow graph, and thus will not appear in the graph drawing.

Another consideration regards `revise`: the original SMCalFlow work emphasized non-destructive revision, which has the benefit of keeping the history of the conversation intact, with the (maybe not always practical) possibility to go back to any point in the conversation.

Let's say we add position information to `Block` (e.g. as {x, y} coordinates on the grid).
Say we have a `Cube` at position {1,1}, and then move it to position {2,2} by using a `revise`. This would mean that we will now have two copies of the `Cube` – the older one at position {1,1}, and the newer at {2,2}. If we now want to check if there is a B`lock` at position {1,1}, we will find the older version of the `Cube`, which is probably not what we wanted.

A way around this could be to have a `GameBoard` class, holding all the current `Block`s, and in case of moving a block, we duplicate the `GameBoard`, i.e. all of the B`lock`s, and modify just the one which moved. Then, if we restricted `refer` to look only at the new `GameBoard`, it will not find a `Block` at {1,1}. This is not very memory efficient, but it could work.

Another question is how to keep the location information. We could add two attributes (*x_position*, *y_position*) to Block, or, alternatively define a Position type, which has these attributes (*x_position*, *y_position*). The second option seems more in line with OOD principles (e.g. in case not all {x,y} combinations are valid), as well as allowing easier translation of user requests such as "put it there" (which combine x and y coordinate values).

Another question is if position information should really be part of the Block, or put outside of it. Conceptually, the `Block` itself, once created, does not change – it retains its shape, size, color, and material – so making duplicate `Block` objects may be questionable. Instead we could have a new object type (`PlacedBlock`) which has a `Block` and a `Position` as inputs. This way, we duplicate `PlacedBlock`s, but each individual `Block` gets reused, rather than duplicated (reflecting the intuition "the block did not change, it just moved"). On the other hand, `PlacedBlock` feels awkward...

We may want to give each `Block` a unique id. This can be convenient e.g. when manually writing Pexp's during debugging to directly specify which `Block` we want to move (avoiding the need for a refer expression).

Let's make some design decisions, and see how far we get. If we get stuck, we'll try again…
As said before, the point in this tutorial is not to find the best implementation, but to demonstrate the use of different OpenDF elements, and to go through the design process.

Putting together the ideas from above, let's try ta design having the following elements:

- We'll use the same definition for `Block` (and derived classes) that we've used so far
  - but we'll add an attribute "id"
- `Position` – a class to represent a grid position, with attributes "x", "y"
- `PBlock` – a "placed block", with attributes "block", "position"
- `Board` – the game board, holding the current SET of `PBlock`s, with attribute "pblocks"
  - `Board` will not be purely dataflow-y:
    - it will also keep the grid costs as an internal parameter (not directly accessible to `refer`/`revise`)
    - Instead, modification of the `Board` will be done by calling specific functions (`add_block`, `move_block`, …)
    - We'll design the interaction so that only one Board exists, and will not allow duplication

Let's see where this takes us...

We'll now go over parts of the implementation of these classes (we'll skip the more obvious ones), and point out some practical details. We'll switch to `blockWorld_V2.py` from this point on.

```
1 class PBlock(Node):
2      def __init__(self):
3         super().__init__(type(self))
4         self.signature.add_sig('block', Block)
5         self.signature.add_sig('position', Position)

6      def describe(self, params=None):
7          msg = ''
8          if 'block' in self.inputs:
9              m = self.input_view('block').describe(params=params)
10             msg += m.text
11         if 'position' in self.inputs:
12             m = self.input_view('position').describe()
13             if m.text:
14                 msg += ' at ' + m.text
15         return Message(msg)
```

The `describe` method: we use a common pattern – composing the description of PBlock from the description of its inputs (`Block` and `Position`).
Note: `describe` returns a `Message` type (not plain text), which has the attributes "text" (the text of the message), as well as other optional attributes. In this example, we take the text attribute (e.g. `m.text` in line 10) from the `Message`s returned from the call to describe on `PBlock` and `Position` (if they exist), combine the two texts to a new text (string), and return a new `Message` containing this new text.
Some common patterns:

- Line 8: `if 'block' in self.inputs` - check if the required input was given (exists in the graph)
- Line 9, `self.input_view` – remember to use `n.input_view()` and not `n.inputs[]` (see previous discussion about *view*)

For Board, the node holding the game board information:

```
1 class Board(Node):
2      def __init__(self):
3          super().__init__(type(self))
4          self.signature.add_sig('blocks', PBlock)
5          self.costs = None  # dictionary: {(x,y): cost}
6          self.size = None  # size of the game board

7      def get_pblocks(self):
8          if 'blocks' in self.inputs:
9              return self.input_view('blocks').get_op_objects()
10         return []

11     def init_grid(self,  size, rand=False):
12         self.size = size
13         self.costs = {}
14         for x in range(size):
15             for y in range(size):
16                self.costs[(x, y)] = randint(1,4) if rand else 1 + (x + y) % 4
```

Following our design decisions, `Board` will hold the `SET` of the current `PBlocks` (`Block+Position`). The `PBlock`s are accessible directly through `refer/revise`.
`Board` also holds the costs of the grid points – this is inside an internal dictionary, which is not directly accessible by `refer/revise`.

We define a utility function for Board, which returns a Python list with all the PBlocks currently in Board.
This function (in line 10) uses the utility function `Node.get_op_objects()`. This function is used in order to collect "real" objects directly under an operator (such as `SET`). For example, applying this function to the to node of the expression:
`SET(SET(Int(1), Int(2)), Int(3), Int(4))`, will return a Python list with the nodes `Int(1), Int(2), Int(3), Int(4)`. (note that it recursively unrolled the `SET`s).
Similar utility functions exist in `Node`:
`get_op_object(n)` – get just the first object node under n
`get_op_types(n)` – returns a list of all the different types of the objects nodes under n
`get_op_type(n)` – similar, but return just the first type
and others.

We can create a Board: `Board()` , but for now it still does not have any blocks or grid information.

Let's set the costs of the grid.
We've defined `Board.init_grid` (line 11) – to set the size of the grid, and then fill in the costs for the grid points – either randomly or with a fixed pattern (controlled by the flag `rand`).
Note that we have not actually called this in `Board.__init__()`. Instead, we'll do it by defining a new function node `InitGrid()`.

```
1 class InitGrid(Node):
2      def __init__(self):
3          super().__init__(type(self))
4          self.signature.add_sig('rand', Bool)
5          self.signature.add_sig('size', Int)

6      def exec(self, all_nodes=None, goals=None):
7          size = self.get_dat('size')
8          if not size:
9              size = 7
10         rand = self.inp_equals('rand', True)
11         boards = [g for g in self.context.goals if g.typename()=='Board']
12         # boards = get_refer_match(self.context, all_nodes, \
                                    goals, pos1='Board?()')
13         if not boards:
14             raise DFException('Please create a Board first!', self)
15         board = boards[-1]
16         board.init_grid(size, rand)
```

Remember that our design was to have the grid costs stored internally in Board, and not accessible through `refer/revise` (since the grid itself is not a dataflow graph). By defining `InitGrid()`, we're able to use a Pexp to manipulate this internal variable. In this case, `InitGrid()` calls `Board`'s own `init_grid()`.

Let's look at the patterns used here:
In line 7 we use `Node.get_dat(inp_name)`. This is a utility function, which gets the data for a given input name. This function combines both the logic of `Node.dat` and `Node.input_view`, so you do not have to worry about which *view* mode to use, or if the input is a *base type* or a *leaf type* (we've discussed all this previously). In case there is no data, or the requested input does not exist, it returns `None` (so no need to check first if the input exists).

In line 10, we use the function Node.inp_equals – this is another utility function which can save a couple of code lines.. Given an input name and a value, it gets the value for the input (same logic as get_dat) and checks if it's equal to the given value (return value is True/False).

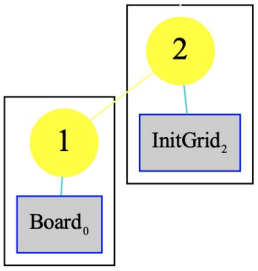
In line 11, we find all the top level nodes (goals) which are of type `Board`. To do this, we loop over `context.goals` (the list of goals defined in the context). For each goal, we use `Node.typename()` to get the type-name (a string) of the node, and then check if it equals "Board".

Line 12 is a different way to get the same result – instead of looping over goals, we execute a `refer(Board?())`, but we use the programmatic interface to execute the refer, by calling `get_refer_match(…, pos1=Board?())`, where the pos1 parameter is used to give the query.
Either way we choose, `boards` is now a list of all goals of type `Board`. We pick `board` to be the last one (the newest one) (line 15)
(in our design, we decided that we will have only one `Board` node existing, so we expect exactly one node).

Finally, in line 16 we call `board.init_grid`, to initialize the grid of the `Board`.
We can now run:

```
Board()
InitGrid(size=7,rand=False)
```

Which will create the Board and then initialize the grid, but another result of grid being an internal variable of Board (and not a dataflow graph), is that it is not displayed in the graph drawing.



Note also that we have not set a result pointer for InitGrid. Not that we have to set a result, but if we did, what should the result be? The grid itself is not part of the dataflow graph, so we can not point to it. Maybe the Board? Or we could make an `Str` node describing the current grid?

We'll use a different solution – add a message to the graph.
First, we add to `Board` a describe function:

```
1    def describe(self, params=None):
2          pos = {}  # {(x,y): block.id}
3          pblocks = self.get_pblocks()
4          for pb in pblocks:
5              x, y = pb.get_pos()  # get the position (x,y) of a PBlock
6              bid = pb.get_ext_dat('block.id')
7              pos[(x, y)] = bid
8          ss = []
9          for y in range(self.size-1, -1, -1):
10             s = []
11             for x in range(self.size):
12                 if (x, y) in pos:
13                     s.append('%s*%s' % (self.costs[(x,y)], pos[(x,y)]))
14                 else:
15                     s.append('_%s_' % self.costs[(x,y)])
16             ss.append(s)
17         s = '  NL  '.join([' '.join(i) for i in ss])
18         # calculate and append total board costs
19         s = s + '  NL  ' + 'total cost: ' + str(self.calculate_total_cost())
20         return Message(s)
```

In lines 2-7 we collect pos – a dictionary mapping, for each current `PBlock` a position (x,y) to a `block.id`.

In line 6 we use the utility function Node.get_ext_dat(), which is a convenient shorthand when we need to traverse multiple inputs in the graph. `self.get_ext_dat('a.b.c')` is equivalent to `self.input_view(a).input_view(b).get_dat(c)`, with the added convenience that we don't have to check first that all of these inputs actually exist. Like `Node.get_dat`, it will return the right value if it exists, else: `None`.

In lines 8-16, we create a text line for each row in the grid. For each element in a row, we add either `"c*i"` (if the grid point has a block on it) or `"_c_"` if not ; where `c` is the cost of that grid point, and `i` is the id of the block (if such exists).

In line 17, we combine the lines into one string. Note that to indicate a line break, we use the special string `'  NL  '` (two spaces before and after `NL`).

Finally, in line 19, we call `Board.calculate_total_cost()` and add that to the message.

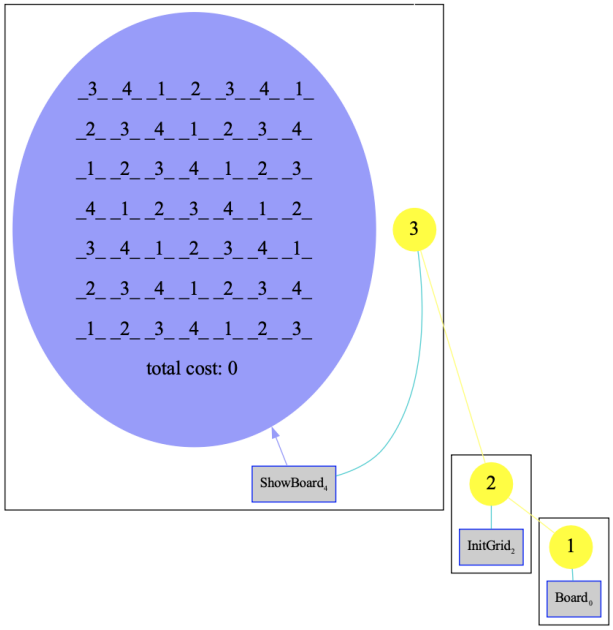
Now, we define a new function node – `ShowBoard`:

```
1  class ShowBoard(Node):
2      def __init__(self):
3          super().__init__(type(self))

4      def exec(self, all_nodes=None, goals=None):
5          txt = ''
6          boards = get_refer_match(self.context, all_nodes,
7                                   goals, pos1='Board?()')
8          if not boards:
9              txt = 'No Board exists'
10         else:
11             board = boards[-1]
12             if board.size is None:
13                 txt = 'The board is uninitialized'
14             else:
15                 txt = board.describe().text
16         self.context.add_message(Message(txt, node=self))
```

We check that the Board exists, and that it's initialized, and if so, we get the message from `Board.describe`.
Finally (line 16), we attach the message to the `ShowBoard` node in `context` (`DialogContext`).

Ugly, but it will do for now.



[21]

To add blocks to the board, we'll use a similar strategy to `InitGrid` – we'll define an internal function `Board.add_block()`, and call it from a separate function node `AddBlock()`.
Note that the blocks in `Board` are regular dataflow nodes (which will be drawn as part the graph), but in our design we chose to use `AddBlock` rather than `revise`, in order to avoid having duplicate versions of the `Board`.

```
1    def add_block(self, new_block, position_x, position_y):
           # check that the requested position is within the board
2          if position_x < 0 or position_x > self.size-1:
3              raise DFException("This position_x is invalid!", self)
4          if position_y < 0 or position_y > self.size-1:
5              raise DFException("This position_y is invalid!", self)
6          new_id = new_block.get_dat('id')
           # check that the requested position and id are not taken already
7          pblocks = self.get_pblocks()
8          for pb in pblocks:
9              x, y = pb.get_pos()
10             if (position_x, position_y) == (x,y):
11                 raise DFException("This position is already taken!", self)
12             bid = pb.get_ext_dat('block.id')
13             if bid==new_id:
14                 raise DFException("This ID already exists!", self)
           # position and id are ok - create the PBlock
15         new_pblock, _ = self.call_construct_eval(
16             'PBlock(block=%s, position=Position(x=%d, y=%d))' %
17             (id_sexp(new_block), position_x, position_y), self.context)
           # add the new PBlock to the SET of PBlocks
18         if not pblocks:  # if no blocks so far - create SET to hold blocks
19             pblocks, _ = self.call_construct_eval('SET(%s)' %
20                                   id_sexp(new_pblock), self.context)
21             pblocks.connect_in_out('blocks', self)
22         else:
23             self.input_view('blocks').add_pos_input(new_pblock)
24         return new_block
```

After verifying that the requested position is within the grid, that the position and id are not taken already by another block, we proceed to create a `PBlock` by combining the requested `Block` and `Position`.

This is the first time we see how to construct a graph programmatically.

One way to do this is by explicitly constructing each node individually, and then manually connecting parent nodes with their input nodes. In this case, that's not too difficult, since we need to create only the `PBlock` and Position nodes (the `Block` node is already given as input).
In practice, we hardly ever do this. Instead, we use the "normal" construction mechanism, which converts a Pexp (string) to a dataflow graph. To use this mechanism, we first need to create the (string) Pexp, and then use that to create a graph.
We do this in line 15, by calling `Node.call_construct_eval()`, which constructs a graph and then evaluates it. (evaluation is not strictly necessary in this case, but we do it anyway. `Node.call_construct()` can be used to do construction only).
In lines 16 -17 we create the Pexp for `PBlock`.
Since part of the new graph need to use existing nodes (`add_block` got as input `new_block` – the top node of the (already existing) `Block` to be added), we have to use special syntax to tell the graph construction to use this existing node.
This is done by using the function `id_sexp()` - in line 17.

After the new PBlock is created, we need to add it to Board.blocks. According to our design, Board.blocks is a SET node, under which are one or more PBlocks.
If no nodes have been added yet, we have to create the SET, with the new PBlock under it : lines 19-20, where we use call_construct_eval() again (we also use id_sexp() to connect an existing node)
If nodes have already been added, then we need to connect the new PBlock to the existing SET node: line 23, where we use the function Node.add_pos_input() to add a new object to a SET (or other operators).
Finally, we return the newly created PBlock.

We now define AddBlock:

```
1 class AddBlock(Node):
2      def __init__(self):
3          super().__init__(type(self))
4          self.signature.add_sig('block', Block)
5          self.signature.add_sig('x', Int)
6          self.signature.add_sig('y', Int)

7      def exec(self, all_nodes=None, goals=None):
8          block = self.input_view('block')
9          if not block:
10             raise DFException('Please specify the block to add', self)
11         x, y = self.get_dats(['x','y'])
12         if x is None:
13             raise DFException('Please specify the x position', self)
14         if y is None:
15             raise DFException('Please specify the y position', self)
16         boards = get_refer_match(self.context, all_nodes,
17                                  goals, pos1='Board?()')
18         if not boards:
19             raise DFException('Please create a Board first!', self)
20         board = boards[-1]
21         if board.size is None:
22             raise DFException('Please init the grid first!', self)
23         try:
24             new_block = board.add_block(block, x, y)
25         except Exception as ex:
26             re_raise_exc(ex, self)
27         self.set_result(new_block)
```

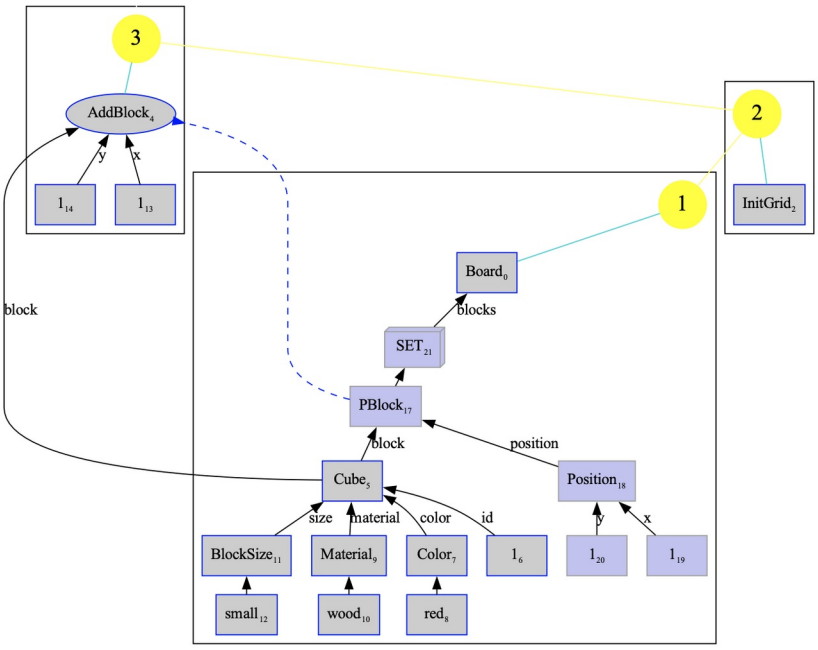
In line 11, we use a utility function Node.get_dats() - which is similar to Node.get_dat(), which we've seen before, but allows to retrieve at once several values.

Lines 16-20 are get the Board (same as in InitGrid). Lines 21-22 make sure that the grid was already initialized.

In line 24, we call Board.add_block(), letting Board handle the adding.
Note that we've wrapped the call to Board.add_block() by a try.
If we didn't do this, if Board.add_block() raised an exception, it will be attached to the Board node (by default, an exception is attached to the node which raised it).
In this case, we prefer that the exception will be attached to the AddBlock node, since conceptually that's the function which had problems.
If Board.add_block() raises an exception, we catch it with try-except, and use the function re_raise_exc() to raise that exception again, but this time attached to self (AddBlock).

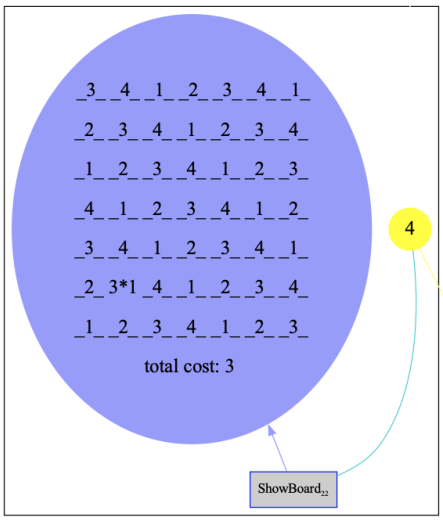
In line 27, we use `Node.set_result()` to set the result of `AddBlock`, so it points to the newly created `PBlock`.

Running `AddBlock` now (after creating the Board and initializing the grid) gives:
`AddBlock(block=Cube(id=1, color=red, material=wood, size=small), x=1, y=1)` *[22]*



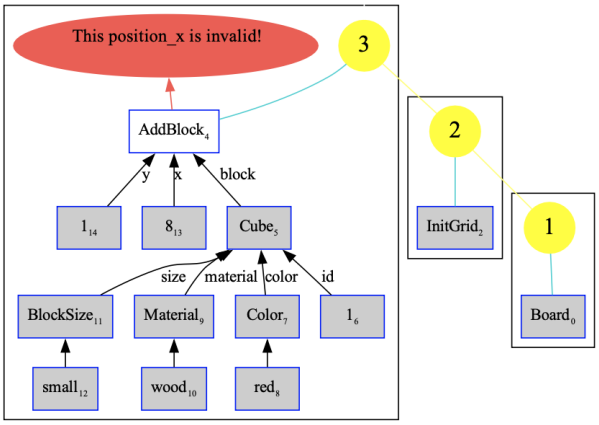
Note how the `Cube` (node#5) , which was given as input to AddBlock, is connected to the new `PBlock` (node#17), and that `AddBlock`'s result now points to the new `PBlock`.

Using `ShowBoard`:



"3*1" (in position {1,1}) is the way we indicate that for position {1,1} the cost is 3, and that the block with id#1 is at that position.

If we try to add a block at an invalid position (the grid size is 7):
`AddBlock(block=Cube(id=1, color=red, material=wood, size=small), x=8, y=1)` *[23]*



We see that the exception, although originally raised by Board.add_block(), was indeed reattached to AddBlock.

Next, we'll add a function MoveBlock(), to move an existing block to a new position. Like the case of adding a block, we'll define Board.move_block() to do the actual moving.

```
1     def move_block(self, id, position_x, position_y):
2         pblocks = self.get_pblocks()
3         if id not in [i.get_ext_dat('block.id') for i in pblocks]:
4             raise DFException("No block with ID %s" % id, self)
5         for i in pblocks:
6             x,y = i.get_ext_dat('position.x'), i.get_ext_dat('position.y')
7             if (x, y) == (position_x, position_y):
8                 raise DFException("Position (%s,%s) is already taken!" %
9                                   (position_x, position_y), self)
10        if position_x < 0 or position_x > self.size-1:
11            raise DFException("This position_x is invalid!", self)
12        if position_y < 0 or position_y > self.size-1:
13            raise DFException("This position_y is invalid!", self)
14        # move block
15        for i in pblocks:
16            bid = i.get_ext_dat('block.id')
17            if bid==id:
18                posx, posy = i.get_ext_view('position.x'),
19                             i.get_ext_view('position.y')
20                posx.data = position_x
21                posy.data = position_y
```

After validating that the board contains a block with the requested id, that the requested position is within the board, and that the requested position is free, we go on to actually move the block.

The utility function `Node.get_ext_view()` is similar to `Node.get_ext_dat()` which we've already seen. `node.get_ext_view('a.b.c')` is equivalent to `node.input_view('a').input_view('b').input_view('c')`, allowing us to traverse the graph at one go.

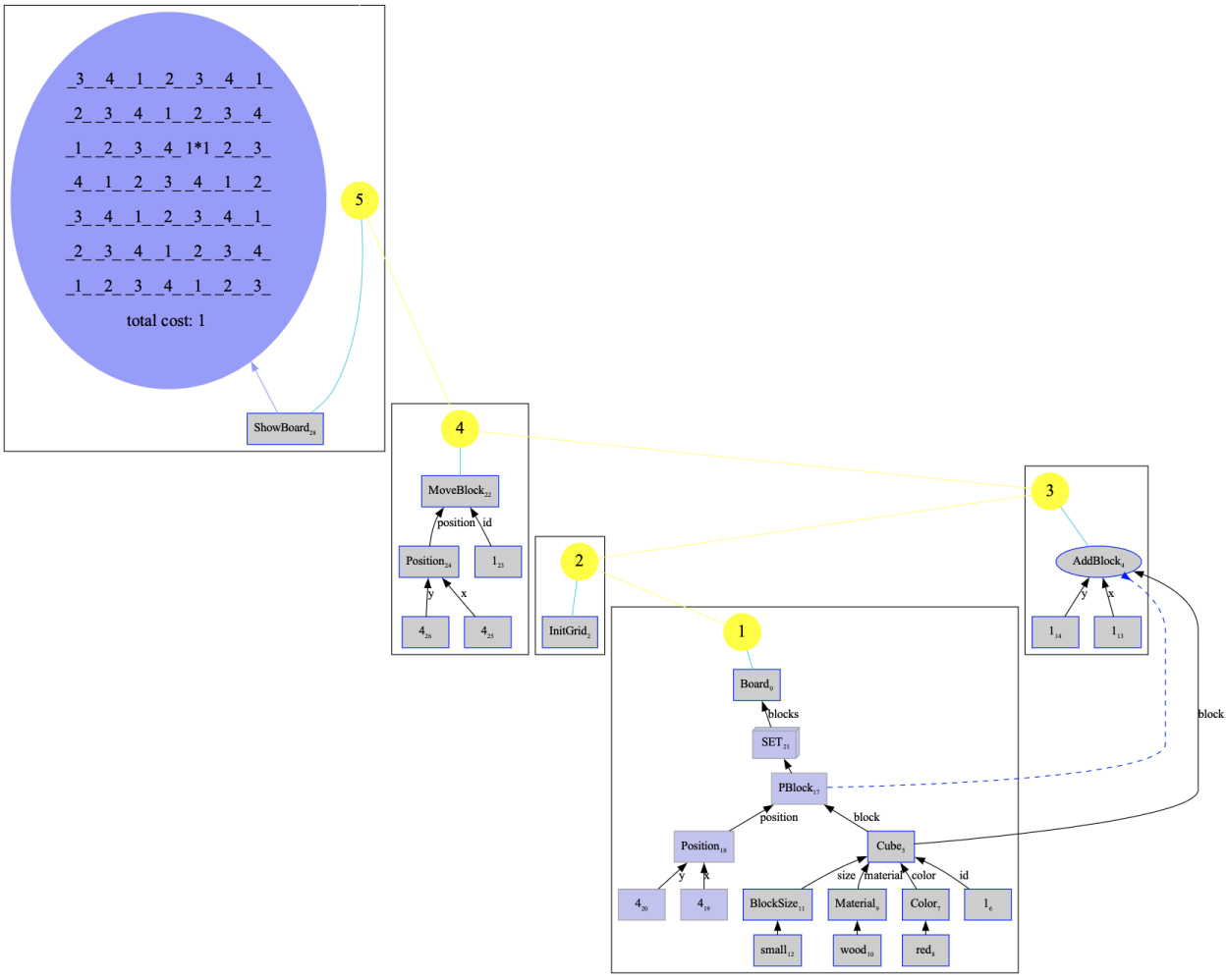
In line 18 we get the **nodes** (not the values!) which hold the x and y position of the block. We then overwrite the **value** of these nodes by directly setting their `.data` attribute to the requested value.

This is not the typical way we do things in dataflow, as this is a **destructive** operation.
As mentioned previously, the common way is to use revise, which is non destructive and creates duplicate graphs.

`MoveBlock()` is very similar to `AddBlock()`, except that the position is given as a `Position` and not separately as `Int` x and `Int` y (either way is ok), so we don't show the code here.

Moving the block we've previously added (at position {1,1}) to position {4,4}:
`MoveBlock(id=1, position=Position(x=4, y=4))` *[24]*

Let's look at the graph now (including all the steps which got us here):



The block has indeed moved to the requested position, as seen in the purple message, as well as in the `Position` of the `PBlock` under `Board`.
Note however that according to the graph it looks like the result of `AddBlock(x=1,y=1)` (goal#3) points to a `PBlock` with position {4,4}!
This is the result of having destructive modifications of the graph – some calculations may be "out of sync", and the graph may contain inconsistencies.

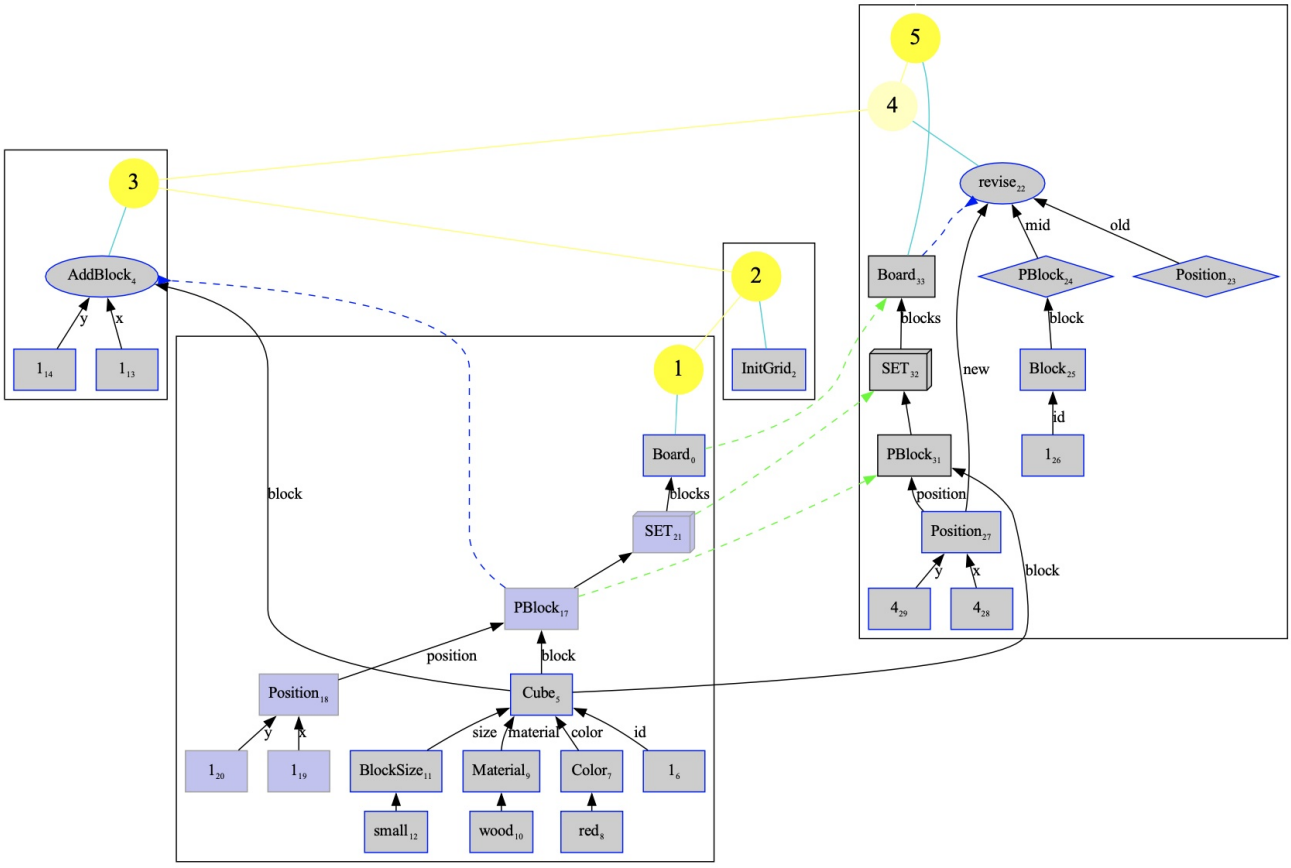
Design review:

Now that we have most of the basic building blocks to implement Griddy (a board, a grid with costs, the ability to add blocks to the board and move them), let's look back and see if the design we chose needs any changes.

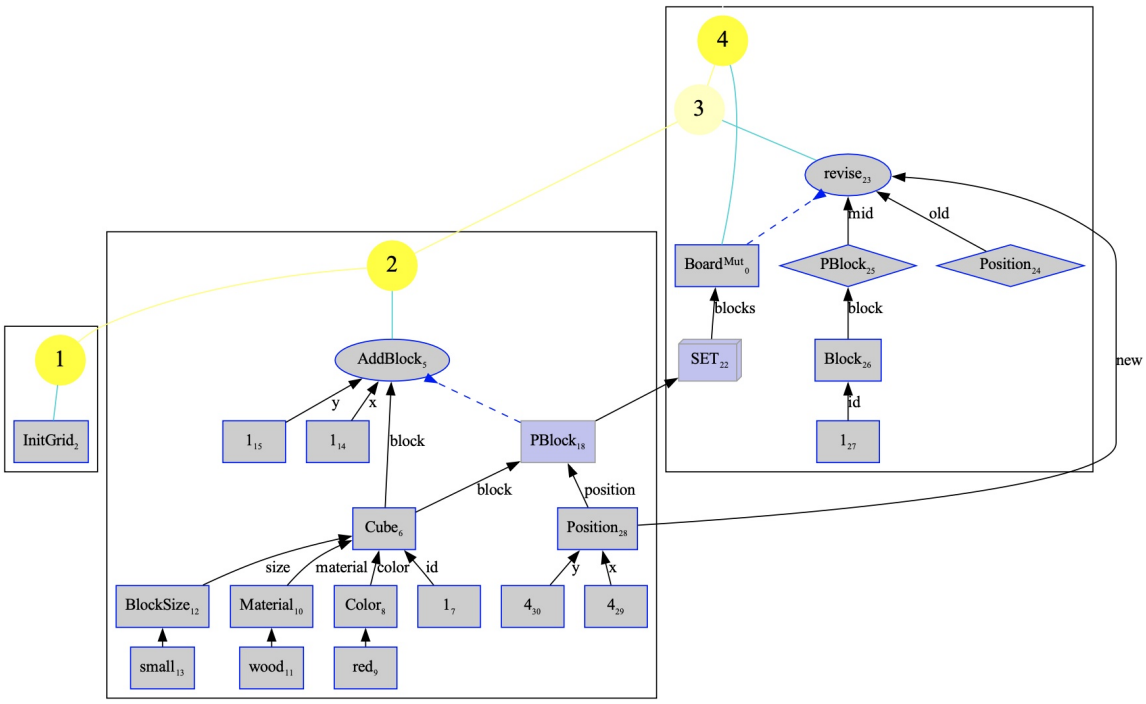
If we used a `revise` to move the block (non destructive), we would get: **Graph#17** *[25]*
Now `AddBlock(x=1,y=1)` intuitively points to a `PBlock` at position {1,1},.
But, as mentioned in the discussion about the design, there is a price to pay for this:

- The graph grows as we have to duplicate Board after every MoveBlock
- Querying for at the old position ({1,1}) will find the old version of the PBlock (there are ways to solve that)
- When duplicating the Board, we'd have to add extra code to make sure its internal (non-dataflow) variables (grid size and costs) get copied as well (there are ways to solve this too).

Graph#17



[Advanced feature]: Another alternative is to use revise, but to mark the initial Board as *mutable*, which causes revise to mutate the graph - in-place (destructive) modifications. To mark a node as *mutable*, we can use the *special feature* syntax: `<&>Board()`. No other change is needed. The resulting graph is (note the superscript "`Mut`" in node#0 (the `Board` node)):



Which is similar to the result with `MoveBlock`. However, `MoveBlock` performed some custom sanity checks (e.g. that the requested new position exists and is free), which `revise` (being generic) does not do.

How about `PBlock`? Was it really necessary? Or could we have just added `Position` to `Block`? Since we ended up doing all the modifications (`AddBlock`, `MoveBlock`) with custom functions rather than `revise`, we probably could have managed without PBlock.
On the other hand, so far it did not cause too much inconvenience (e.g. significantly longer Pexps or more complex implementation code).

Restricting user operations:

We mentioned already the need for the application to control the interaction, play the role of the "gate keeper" and restrict the user from performing unwanted operations.

How does this work in practice?
One part, which we've already seen, is implementing checks in the node logic to prevent operations which are "illegal" for anyone (either user or system) – e.g. like placing a block at invalid positions. But how can we prevent the user from performing an operation which is allowable for the system?

Remember that in a dialog system, the user generates a natural language (NL) request, which is converted (directly, or through intermediate steps, such as an intent/entities representation) to a structured representation (Pexp in our case), which is then executed.

This means we are in control of generating the Pexp to be used for executing the user's request, so can ensure no unwanted Pexps are generated. E.g if the user asked to add or remove a block from the board, we could translate that e.g. to the Pexp `InvalidRequest()` (which would generate some error message for the user).

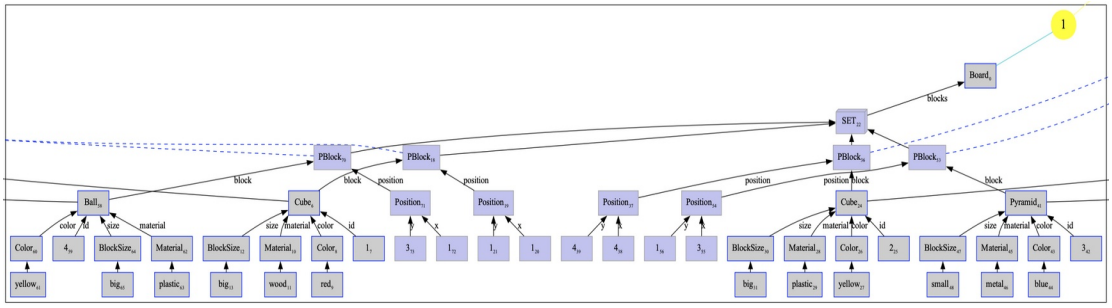
In fact, `AddBlock` was useful for development and debugging, but in run-time we will not use it. If we (the system) want to add blocks, we would do this programmatically by calling `Board.add_block()`.

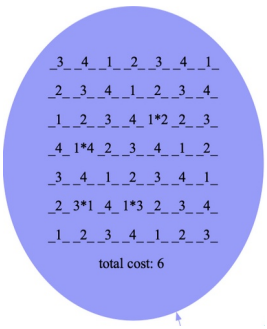
More Queries:

Let's return to some more queries.

First, let's verify that the queries that we've seen previously still work in the new design (with `Board` and `PBlocks`). We'll add the same blocks we've used for the previous queries (but this time we'll also add `id` and `position`):

```
AddBlock(block=Cube(id=1, color=red, material=wood, size=big),x=1, y=1),
AddBlock(block=Cube(id=2, color=yellow, material=plastic, size=big),x=4, y=4),
AddBlock(block=Pyramid(id=3, color=blue, material=metal, size=small),x=3, y=1),
AddBlock(block=Ball(id=4, color=yellow, material=plastic, size=big),x=1, y=3),
```

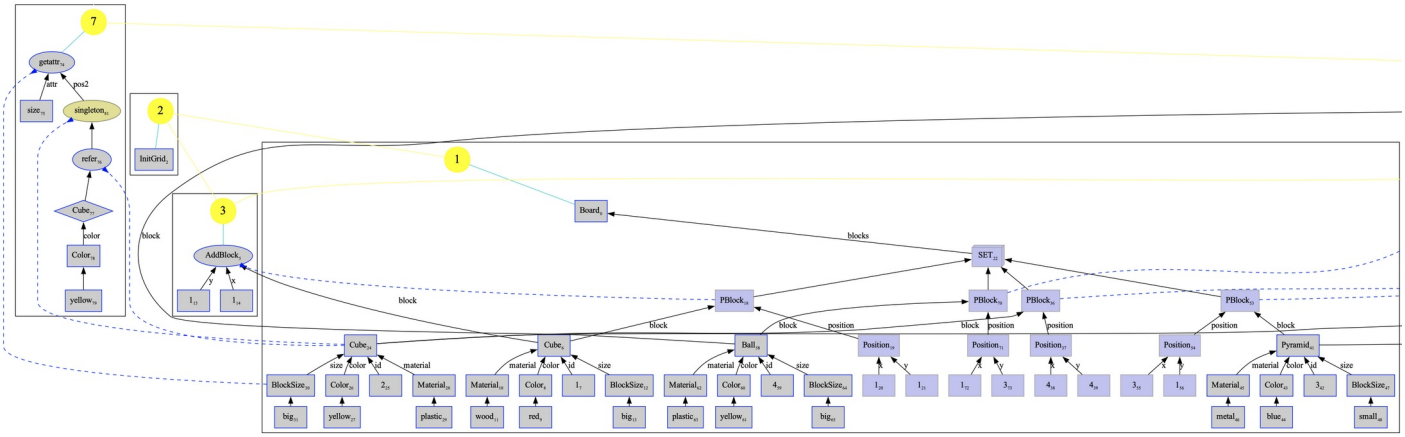




Running the query:

```
:size(refer(Cube?(color=yellow)))
```

[27]



(showing only part of the graph – you can verify this).

The query still runs. This should not be a surprise – refer looped over all the nodes in the graph, looking for nodes of type `Cube`, which fulfill the condition `color=yellow`. The addition of `PBlock` and `Board` on top of the `Block`s did not affect that.

However, note that if our implementation of `AddBlock` would have attached a new copy of its input block to the newly created `PBlock`, rather than reusing the input block and attaching it directly to the new `PBlock`, then we would end up with two copies of the input block, which would affect the result of queries.

Adding position queries:

Now that we have position in PBlock, let's use that for some positional queries.

First, let's look for a PBlock with position.x =4.

```
refer(PBlock?(position=Position(x=4)))
```

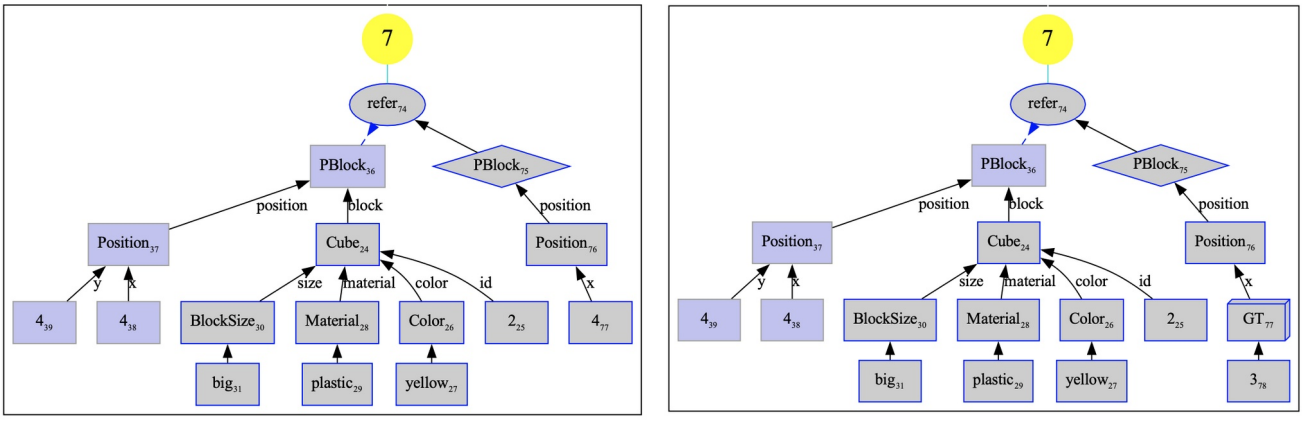
[28]

Then, look for a PBlock with position.x > 3.

```
refer(PBlock?(position=Position(x=GT(3))))
```

[29]

In the second query, we've used the qualifier GT (greater than) in order to express the condition that the x position of the candidate object (the one refer is trying to decide if satisfies the condition or not) must be greater than 3. (In addition to GT, we also have LT, EQ, GE, LE)



[Advanced] Internally, at comparison time, GT actually calls Node.func_GT (respectively, func_LT, func_EQ, …) to do the actual comparison between condition value and candidate value. This function can be overridden per node type, which allows further flexibility.

Another example – "is block1 left of block2?"
(Note that this is not exactly a query, since it's not a search. We'll extend it later to a query.)
We'll start by implementing a simple function, `IsLeftOf`, which gets two `PBlock`s (we have to use `PBlock`s, since `Block` does not have the position) and gives True if the first one is left of the second one, and False otherwise.
(We assume our grid is laid out with the origin {0,0} at the bottom left of the grid).

```
1  class IsLeftOf1(Node):
2      def __init__(self):
3          super().__init__(out_type=Bool)
4          self.signature.add_sig(posname(1), PBlock, alias='pblock1')
5          self.signature.add_sig(posname(2), PBlock, alias='pblock2')
6      def exec(self, all_nodes=None, goals=None):
7          pblock1 = self.input_view('pblock1')
8          pblock2 = self.input_view('pblock2')
9          answer = 'False'
10         if pblock1 and pblock2:
11             x1 = pblock1.get_ext_dat('position.x')
12             x2 = pblock2.get_ext_dat('position.x')
13             answer = x1 < x2
14         return_node, _ = self.call_construct_eval(
15             'Bool(%s)' % answer, self.context, add_goal=False)
16         self.set_result(return_node)
```

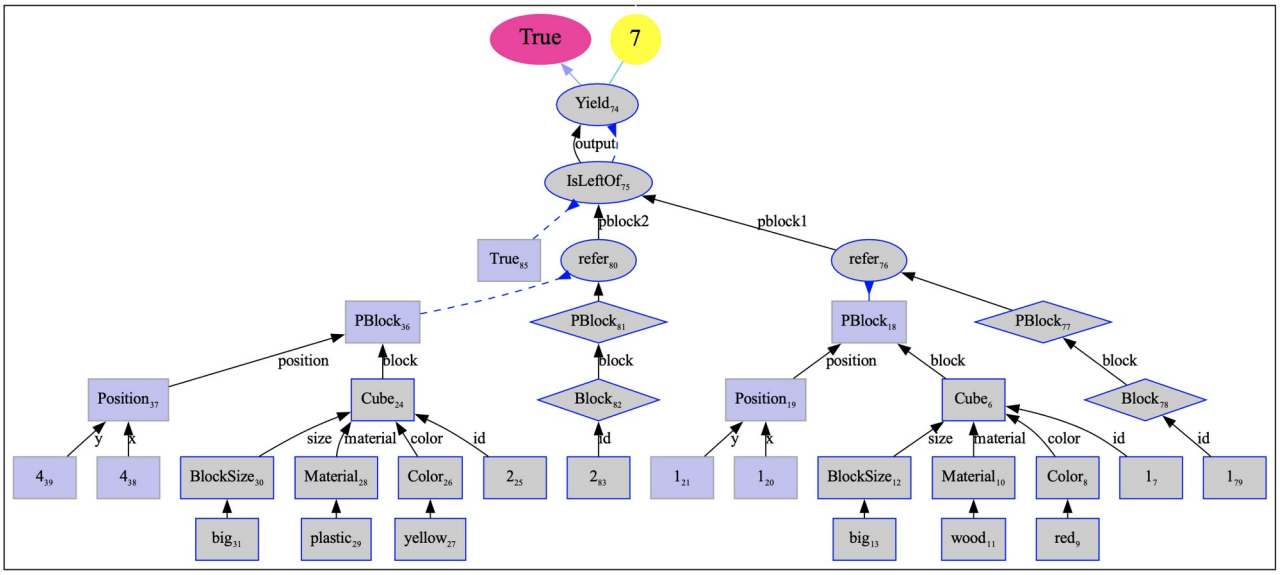
In lines 4,5 we use `alias=` in the definition of a signature entry. This is a convenience convention, which allows us to use both positional **and** named parameter syntax: we could use either of `IsLeftOf1(node1, node2)` / `IsLeftOf1(pblock1=node1,pblock2=node2)` / `IsLeftOf1(pblock1=node1, node2)` / `IsLeftOf1(node1,pblock2=node2)` / `...`
And we can also use either e.g. `posname(1)` or `'pblock1'` as input for e.g. `self.inputs[], self.input_view() ...`

We get the two input `PBlock`s (lines 7,8), get their x positions (lines 11,12), compare them to get the answer, and then construct a new `Bool` node to hold the answer, and set it to be the result of `IsLeftOf1` – nothing new here.

Let's ask if the block with id#1 is left of the block with id#2:

```
IsLeftOf1(refer(PBlock?(block=Block?(id=1))),
          refer(PBlock?(block=Block?(id=2))))     [30]
```

Graph#18



This works, but we may start to feel some pain from having the position in PBlock and not in Block – the Pexp gets longer (we could have had just `IsLeftOf(refer1(Block?(id=1)),…)`.

Can we do anything about this (without getting rid of PBlock)?

- We could define a function to directly get a PBlock given an id – PBlockFromId(), then we'll have `IsLeftOf(PblockFromId(1), PblockFromId(2))`
- We could allow `IsLeftOf` to accept `id1,id2` as additional inputs, and then at evaluation time either use the given `PBlock` or the id (in which case, we can use the programmatic refer to find the `PBlock`) . Then we'd have `IsLeftOf(id1=1, id2=2)`

With pros and cons for each one. Let's stay with the current design for now.

In **Graph#18** we've added a Yield around the request, in order to get a text answer. We got "True", using `Yield`'s generic answer generation logic.

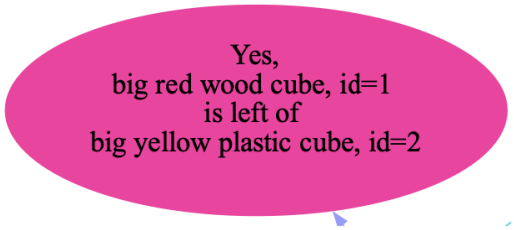
We can write custom logic for `IsLeftOf2`, by adding the function `yield_msg()`.

For example:

```
1   def yield_msg(self, params=None):
2         block1 = self.get_ext_view('pblock1.block')
3         block2 = self.get_ext_view('pblock2.block')
4         if block1 and block2:
5             txt1 = block1.describe(params=['no_art']).text
6             txt2 = block2.describe(params=['no_art']).text
7             if self.res.dat == True:
8                 txt = 'Yes,  NL  %s NL is left of  NL  %s' % (txt1, txt2)
9             else:
10                txt = 'No,  NL  %s NL is NOT left of  NL  %s' % (txt1, txt2)
11        else:
12            txt = "No"
13        return Message(txt)
```

We use recursively call `Block.describe()` to get the text description of the blocks (if they exist).

We used the option '`no_art`' which we defined for `Block.describe()` to avoid the text for each block have the article 'A' (maybe we should have an option, `def_art`, to replace 'A' with 'The' ?).We now get: *[31]*



What else can we do?
We could add `IsRightOf, IsInFrontOf, IsBehind` – these would have the same logic as `IsLeftOf`, so we could generalize, and have a
`IsRelativePosOf(pblock1, pblock2, direction)`
Where `direction` is a string, having the value of `left`, `right`, `infront`, or `behind`.

What if we wanted to ask: "is the `Ball` left of **a** `Cube`?", as opposed to:
"is the `Ball` left of **the** `Cube`?"
This would mean the second argument may be a list (`SET`) of `Cubes`.
Further, we could ask: "is there **a** `Ball` left of **a** `Cube`?"
In this case `IsLeftOf` should be able to handle inputs which may be either one or a `SET` of `PBlocks`.

The new definition of `IsLeftOf3.exec()` is:

```
1     def exec(self, all_nodes=None, goals=None):
2         pblocks1 = self.input_view('pblock1').get_op_objects()
3         pblocks2 = self.input_view('pblock2').get_op_objects()
4         answer = 'False'
5         for item1 in pblocks1:
6             for item2 in pblocks2:
7                 x1 = item1.get_ext_dat('position.x')
8                 x2 = item2.get_ext_dat('position.x')
9                 if x1 < x2:
10                    answer = 'True'
11        return_node, _ = self.call_construct_eval(
12            'Bool(%s)' % answer, self.context, add_goal=False)
13        self.set_result(return_node)
```

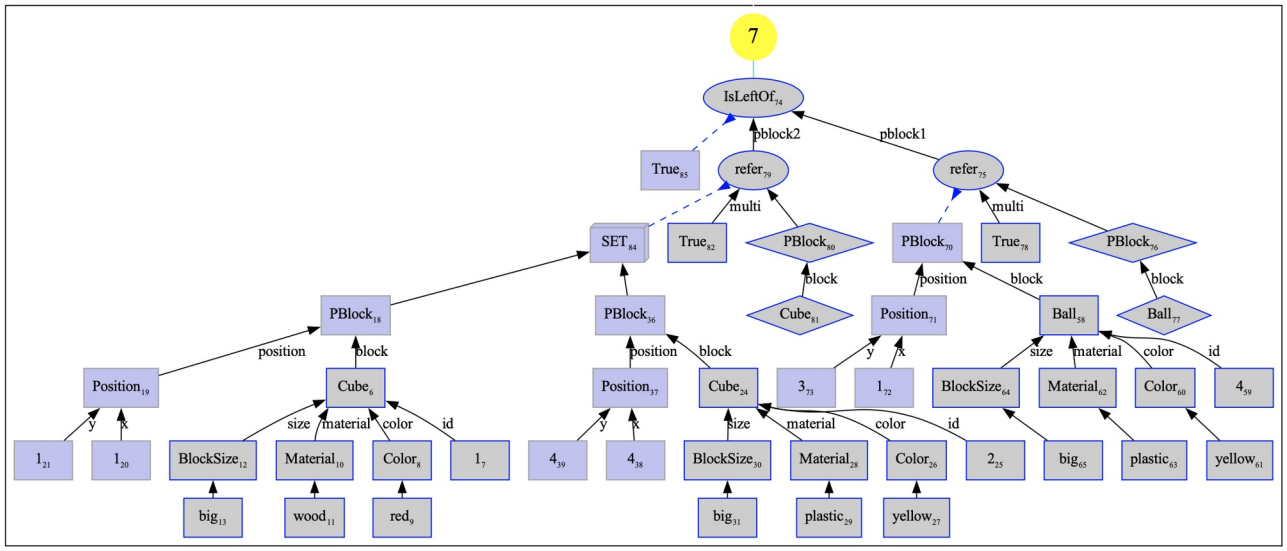
Where we now use `get_op_objects()` (in lines2-3) to get either one or a list of `PBlocks`, and then loop to see if any of the items in pblocks1 is left of any of the items in pblocks2.

To call this, we use: ("is there **a** `Ball` left of **a** `Cube`?")

```
IsLeftOf3(refer(PBlock?(block=Ball?()), multi=True),
          refer(PBlock?(block=Cube?()), multi=True))
```

[32]

Graph#19



To get **a** `Cube` (as opposed to **the** `Cube`), we add `multi=True` to `refer`, indicating that there may be more than one result.
We can see in **Graph#19** that the second `refer` indeed found the two `Cubes`, and that the Ball is left of one of them.

[Advanced topic] – `func_FN`
How do we now run a query (search) like: “find a node which is left of node1” ?
Remember that queries are executed as refer expressions, and that refer loops on the graph nodes, and checks for each of them if the conditions specified in the query are satisfied or not.
In the examples we’ve seen so far, the conditions were given as object constraints (e.g. `Block?(color=red)`), optionally with added operators: e.g. the aggregators (such as `AND,OR`), and qualifiers (such as `GT`, `EQ`). For qualifiers, we’ve seen that refer calls a function (e.g. `Func_GT`) to compare the query and the candidate nodes.
For this case, we can use the qualifier `FN` to indicate that we will run a custom comparison function – in this case, a function which will check if the candidate’s position is left of the position specified in the constraint.

We’ll define the actual function – `pos_is_left_of()`

```
1 def pos_is_left_of(block1, block2):
2     x1 = block1.get_ext_dat('position.x')
3     if x1 is not None:
4         blocks2 = block2.get_op_objects()
5         for b2 in blocks2:
6             x2 = b2.get_ext_dat('position.x')
7             if x2 is not None:
8                 if x1 < x2:
9                     return True
10    return False
```

where block1 is a (single) `PBlock`, and block2 may be a single `PBlock` or a set of `PBlocks`. (we probably want to refactor IsLeftOf() to call pos_is_left_of(), to avoid duplication)

We then define `Pblock.Func_FN()` to override the base implementation as:

```
1    def func_FN(self, obj, fname=None, farg=None, op=None, mode=None):
2        if fname == 'left_of':
3            return pos_is_left_of(obj, farg)
4        return False
```

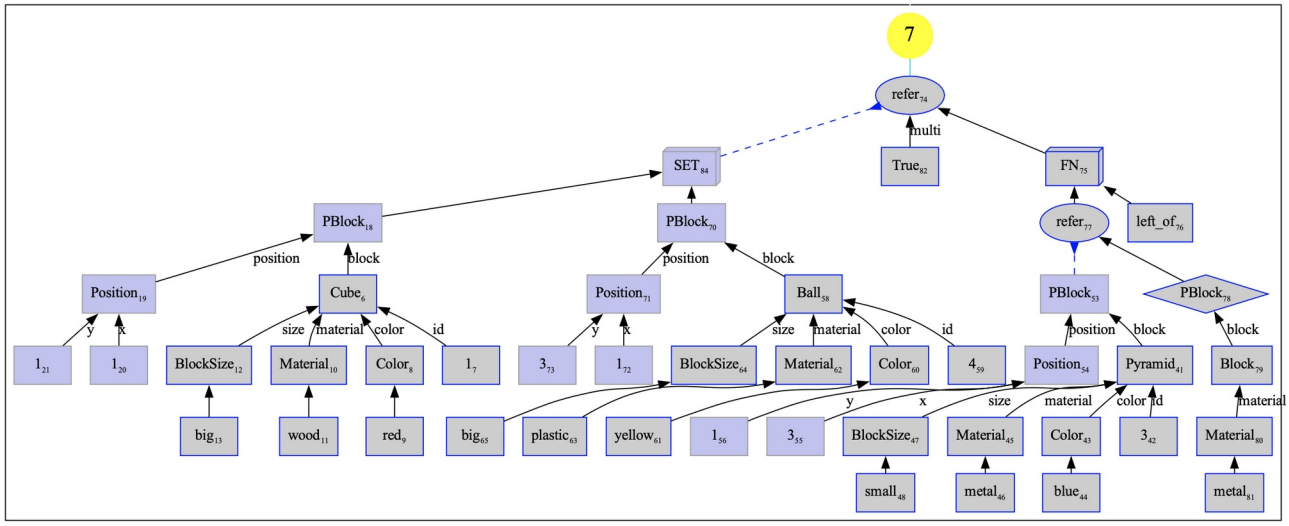
This function returns either True (if the constraint matches the candidate object), or False (if not).

We can define more than one custom functions per node type. The parameter fname is used for selecting which of them to run. In the code above: if `fname=='left_of'` (line 2), we execute `pos_is_left_of(obj, farg)` (line3), where `obj` is the candidate object (node), and `farg` is the constraint value (in this case, what is it that the candidate node needs to be left of).

Putting this together: to run the query "which blocks are left of the metal block?":

```
refer(FN(fname=left_of,
         farg=refer(PBlock?(block=Block(material=metal)))),
       multi=True)
```

[34]



Again – if we wanted "… **a** metal block" instead of "...**the** metal block", we'd add `multi=True` to the inner `refer`.

Note that this method allows us to have unlimited recursive queries: e.g.
find a ball which is left of a red block which is in front of a metal ball which is right of a pyramid…
which makes good use of the compositional nature of dataflow graphs.

Excercises:

This is where we expect you to try to do everything we were too lazy to do:

1) Refactor `IsLeftOf.exec` to use `pos_is_left_of()`

2) Implement the `IsRelativePositionOf(pblock1, pblock2, direction)` node as described above

- implement the corresponding function which can be used by refer (like '`is_left`' in `Func_FN`)

3) Implement a StackBlock() node, to put one one block over another. You'll need to

- Make sure the requested blocks/positions are valid, and stackable (e.g. don't allow to stack a cube on a pyramid, or a a ball on a cube…
- Change the design to keep track of which block is below/on-top-of which node, and to allow multiple blocks in on one grid point
- Add implementation of IsBelow() and IsOnTop() / add 'below', 'on_top' for `PBlock.FuncFN()`

4) Write some interesting queries – see CLEVR for ideas of queries. You may want to add more blocks to the board, to allow for more complex queries.

5) Change the design to use only `Blocks` (and no more `PBlocks`)

And any other task we've put off so far.

Going Conversational:

Our game, Griddy, is, in a way, complete – we can create a board, add blocks, move them, and calculate the cost for each block placement combination.
Currently, we simply sum up the costs of the grid points with blocks on them, so it's a pretty trivial game, but with a more complex score function (and maybe other twists, such as changing the grid costs conditioned on some board configurations, and adding to the score the number of turns used...), the task may be a bit more challenging for the user. Or we may play the "lights off" version, where the user can't see the blocks – the system tells the user what blocks there are, and where they are only once, at the beginning, but does not display them afterwards. The users need to use their memory, but can ask the system (with point penalty) e.g. where a specific block is...

However, the game is still not completely conversational: there is no cohesive high level concept of a conversation with a clear purpose.
We haven't told the user what they can do (in general, and in each turn), we don't provide text feedback, and there is no attempt to direct/encourage the user to bring the conversation (game) to a "successful" end. The current implementation is a "user initiative" design: the user drives the interaction, and the system is reactive. If the user stops moving blocks after two turns, the system will just wait silently for the next input.

To be more precise, actually we *have* seen some examples of the system taking initiative and giving feedback to the user. We've done that when we raised exceptions complaining about bad input – we explained the problem, and let the users know how they can correct the problem (although the user may decide not to follow the advice of the system – e.g. they may want to ask a question). But for successful requests, the system is silent.

One mixed initiative design could be:
1) We initialize the game – initialize and populate the board, and then give an introductory message to the user.
2) After the user moved a block, we inform what is the current score, ask the user what is their next move (potentially suggesting some moves)
3) Allow the user to decide to end the game
4) If the user says they want to end the game, we ask if they really want to stop
5) If so – calculate final score, and say goodbye.
6) If the user is silent for a "long" time give a sign of life.

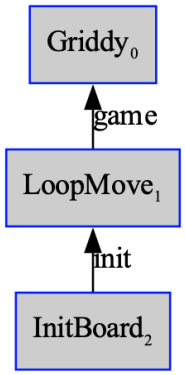
Let's try to design such a system.

Creating a Session:

The way we'll address the issue of a cohesive session (we interchangeably use the terms: *interaction*, *process*, *session*) is by creating (at init time) a dataflow graph to represent the entire session. The inputs in a dataflow graph correspond to logical dependencies. We create a graph to represent the dependencies between the sub tasks of the session.
Remember that the graph is dynamic – we can add and remove subtasks as we proceed.

Once we have the process (session) graph in place, we run an evaluation on this graph (bottom-up pass, respecting the logical dependency between the sub tasks). Typically, the evaluation will fail on lower nodes when we start the interaction, which will cause an error message to be shown to the user, explaining what needs to be corrected. The user will try to correct the error by issuing a corresponding request. Typically, these corrections are executed by a `revise` to the graph (a missing input is added, or a wrong input is replaced), which results (as we've seen before) in the creation of a new version of the session graph (non-destructive duplication). As the interaction proceeds, the errors in the lower parts of the graph are corrected, so the evaluation moves on to higher nodes (which would have their own errors), and so on until the top node is successfully evaluated.

The overall design of the process looks like this:



At the bottom, we have `InitBoard`, which is responsible for creating the grid, setting its costs, and adding blocks to the board. (From this point on, we'll use `blockWorld_V3.py`)

```
1  class InitBoard(Node):
2      def __init__(self):
3          super().__init__(type(self))
4          self.signature.add_sig('rand', Bool)
5          self.signature.add_sig('size', Int)

6      def exec(self, all_nodes=None, goals=None):
7          rand = self.inp_equals('rand', True)
8          if 'size' not in self.inputs:
9              raise DFException('Please specify the size of the board', self)
10         else:
11             size = self.get_dat('size')

12         board, _ = self.call_construct_eval(
13             'Board()', self.context, add_goal=True)
14         board.init_grid(size=size, rand=rand)
15         d, _ = self.call_construct_eval(
16                 'Cube(id=1,color=red, material=wood, size=big)',
17                  self.context)
18         board.add_block(d, 1, 1)
           # omitted - creating and adding the other blocks
19         self.context.add_message(Message(board.describe().text, node=self))
```

`InitBoard` requires a grid size, and will complain if it is not given.
In line 12, we create a `Board` by calling `Node.call_construct_eval()`. Note the parameter `add_goal=True`, which causes the node at the top of the newly created graph to be added as a goal to `DialogContext`.
In lines 15-17 we create the first block, and in line 18 we add it to the board. (we omitted the other blocks here).
Finally – we create a message to show the initialized board.

At the heart of the interaction is `LoopMove`: the process of repeatedly moving the blocks. In addition to the dependence on InitBoard, `LoopMove` dependencies (inputs):
'`move`' – a `move_spec` (specifying which node should be moved where) , and
'`end`' – a `Bool`, which (when has a `True` input) indicates the user asked to end the game.

In more detail: if the user requests a move, we will `revise` the `LoopMove` node to have a new '`move`' input. This will duplicate the whole session graph, which will then be evaluated. The (newly duplicated) `LoopMove` will call `Board.move_block`, and then will raise an exception requesting another move.
This is how we make the loop!

If the user requests to end the game, we'll use `revise` to put a `True` input into the '`end`' input of `LoopMove`. The (newly duplicated) `LoopMove` checks if it has a `True` as input to end, and if so, the evaluation of `LoopMove` succeeds (and moves on to evaluate its parent) – this is how we exit from the loop.

```
1  class LoopMove(Node):
2      def __init__(self):
3          super().__init__(type(self))
4          self.signature.add_sig('init', InitBoard)
5          self.signature.add_sig('move', move_spec)
6          self.signature.add_sig('end', Bool)

7      def exec(self, all_nodes=None, goals=None):
8          done = self.get_dat('end')
9          if not done:
10             msg = ''
11             mspec = self.input_view('move')
12             if mspec:
13                 id, x, y = mspec.get_dats(['id', 'x', 'y'])
14                 board = get_last_board(self.context)
15                 try:
16                     board.move_block(id, x, y)
17                     cost = board.calculate_total_cost()
18                     msg = 'Moved block #%d to {%d,%d}  NL  ' +
19                           'The cost is now %d  NL  ' % (id, x, y, cost)
20                 except Exception as ex:
21                     msg = ex.message + '  NL  '
22             raise DFException(msg + 'What is your next move?', self)
```

If end is not True, we check if there is a requested move, and if so try to execute it by calling `board.move_block()` in line 16.

If `board.move_block()` fails, we catch the exception, and extract its message (line 21)– we will create a new exception, which will report the error, but which will also ask what is the new move (there is no point asking the user to correct a wrong request in this context).
If `board.move_block()` succeeds, we report that the move was executed, and the new cost (lines 17-19). This, also, will be added to the exception we'll create, requesting for the next move.

Above `LoopMove` we have our top node, `Griddy`, which corresponds to the entire session (game).
`Griddy`'s signature allows two inputs: '`game`' – to which we'll connect `LoopMove`, and '`confirm`' – a `Bool`, which (when has a `True` as input) indicates the user confirmed they want to end the game.
Remember that the evaluation is a bottom-up process, so that if we are evaluating `Griddy` it means that all its (existing) inputs have been successfully evaluated - i.e. the code in `Griddy.exec()` can rely on this assumption. In this case, we know that the user already requested to end the game. If `confirm` is not given, then we'll raise an exception asking for confirmation. If it's `True`, we'll say goodbye and finish the evaluation successfully. Since this is the top node, we're done!

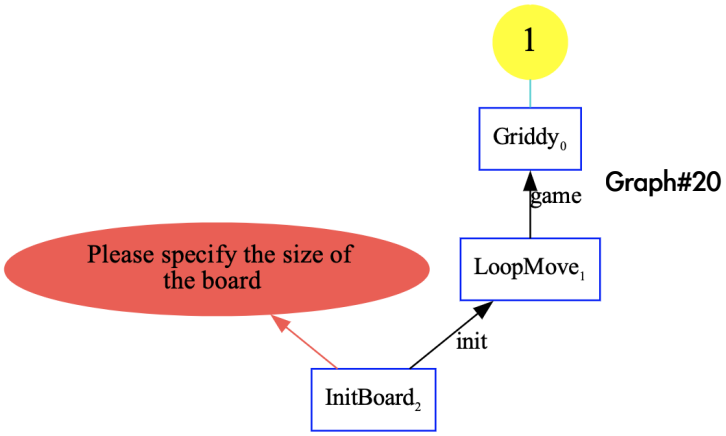
```
1  class Griddy(Node):
2      def __init__(self):
3          super().__init__(type(self))
4          self.signature.add_sig('game', LoopMove)
5          self.signature.add_sig('confirm', Bool)

6      def exec(self, all_nodes=None, goals=None):
7          done = self.get_dat('confirm')
8          if done is None:
9              raise DFException('Are you sure you want to quit?', self)
10         elif done:
11             self.context.add_message(Message('Goodbye', node=self))
12         else:  # confirm = False
13             # todo - resume game
```

Running the game:

Let's step through one session of Griddy, to see how the interaction works.

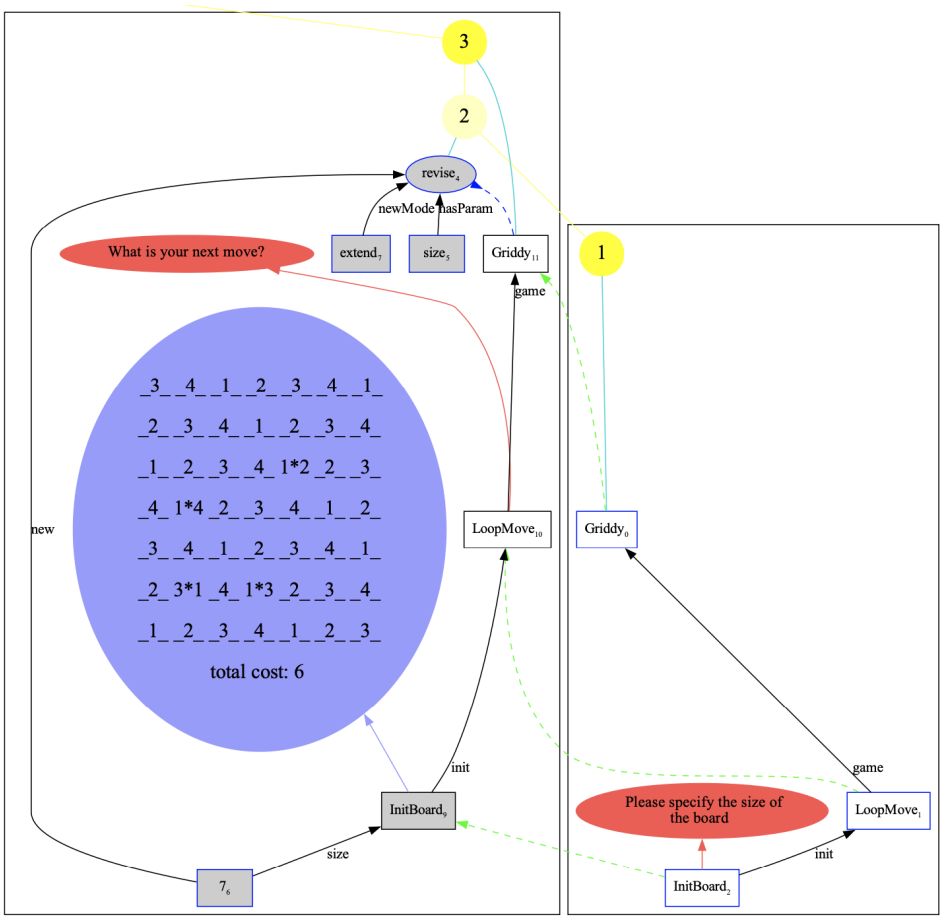
Starting the session, we run the Pexp:
`Griddy(game=LoopMove(init=InitBoard()))` *[35]*



The expression is constructed, and evaluated (bottom up).

In this case, `InitBoard` (at the bottom of the computation) requires a board size as input, so it raises an exception, telling the user what the problem is.

To fix the problem, the user requests size 7. We express this as:
`revise(hasParam=size, new=7, newMode=extend)` *[36]*
(For now, we'll just look at the Pexps representing the user requests, without worrying how the user's requests are translated from natural language to Pexps).

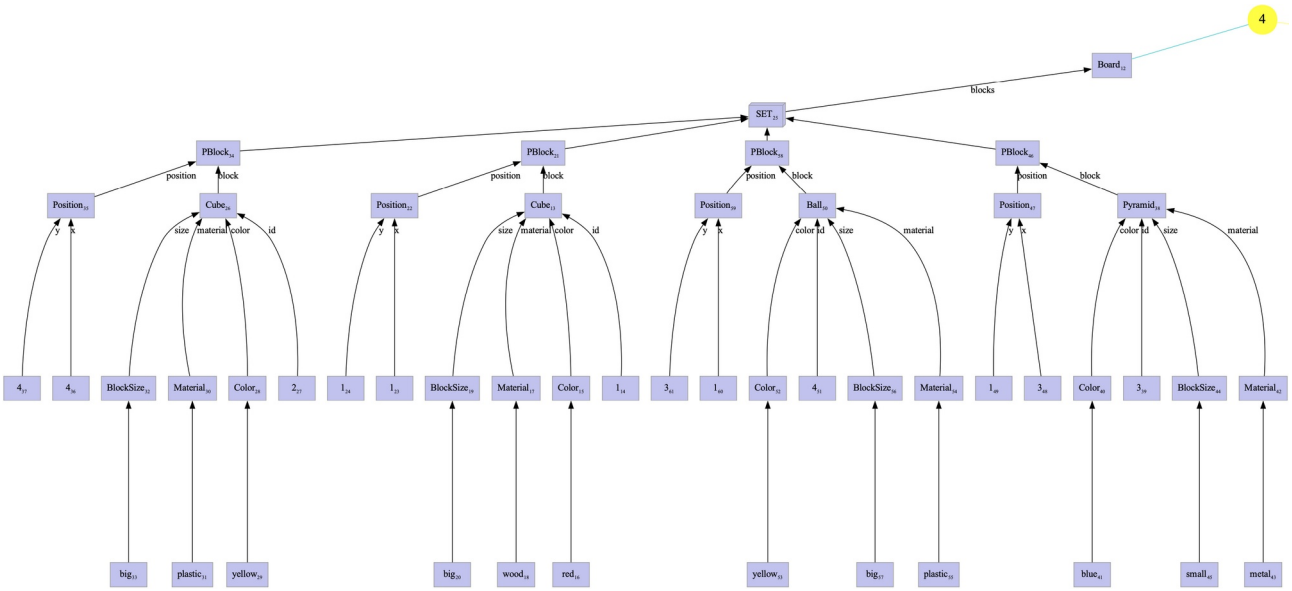


Graph#21

`revise` added the missing input to (the duplicated) `InitBoard`, which now is able to create and initialize the `Board`, and to successfully be evaluated. Moving to the next node, `LoopMove` is now evaluated, and it raises as exception to ask for a move.

We also get (which we won't show anymore):

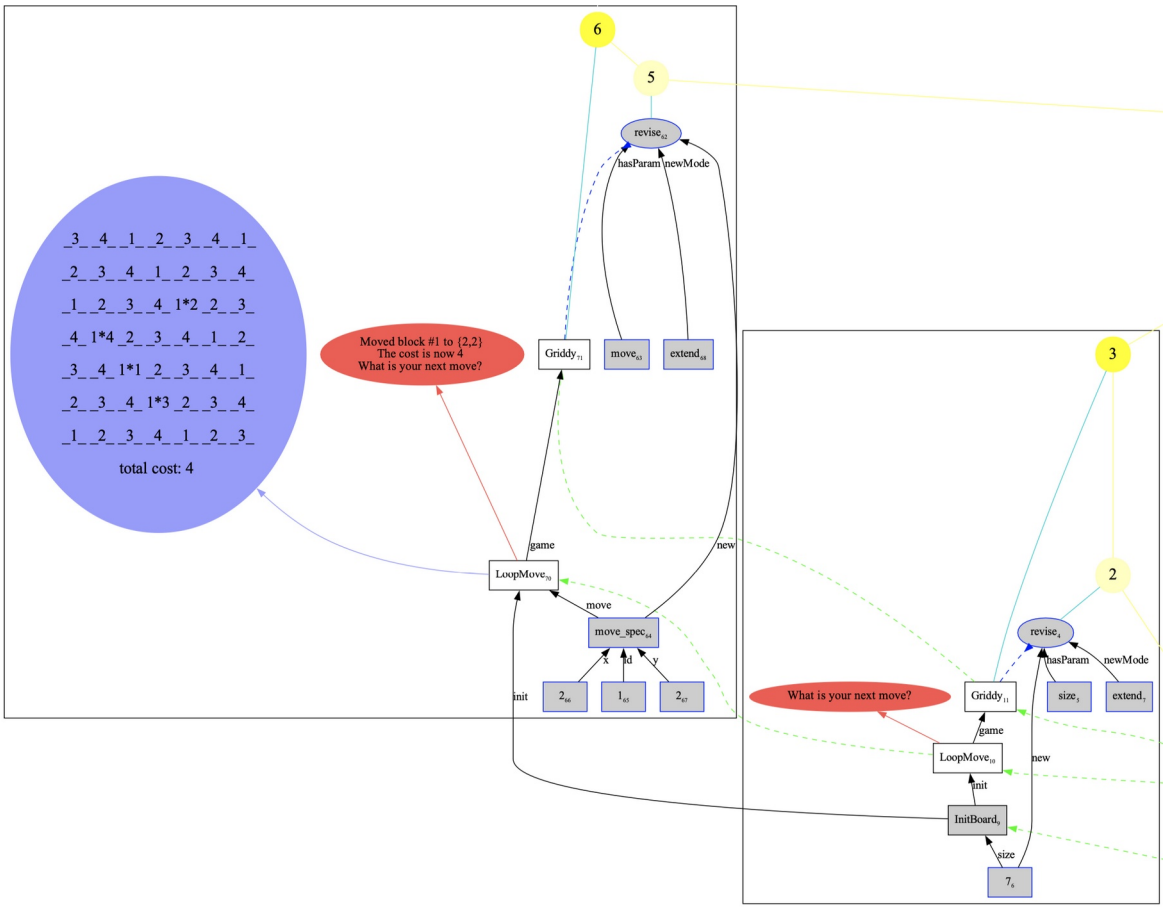
Graph#22



The user now specifies a move – move block id#1 to position {2,2}

```
revise(hasParam=move, new=move_spec(id=1, x=2,y=2),
       newMode=extend)    [37]
```

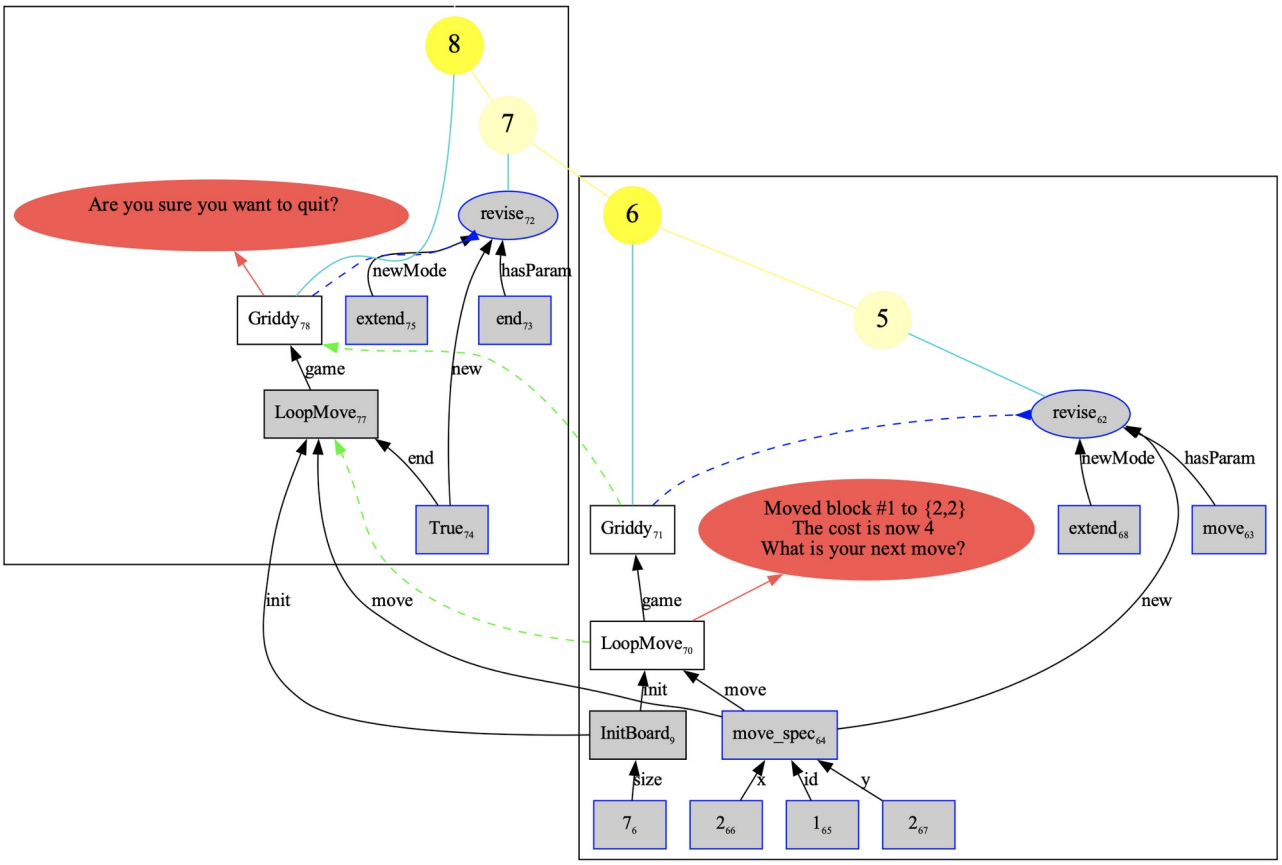
Graph#23 (the graph was altered to graph to display the Board after the new move instead of after init)



This adds a `move` input to (the duplicated) `LoopMove`, which is executed (the block is successfully moved). It reports the success, and the new cost, and asks for another move (by raising an exception).

Additional moves are possible, and finally the user would request to end the game:
`revise(hasParam=end, new=True, newMode=extend)` *[38]*

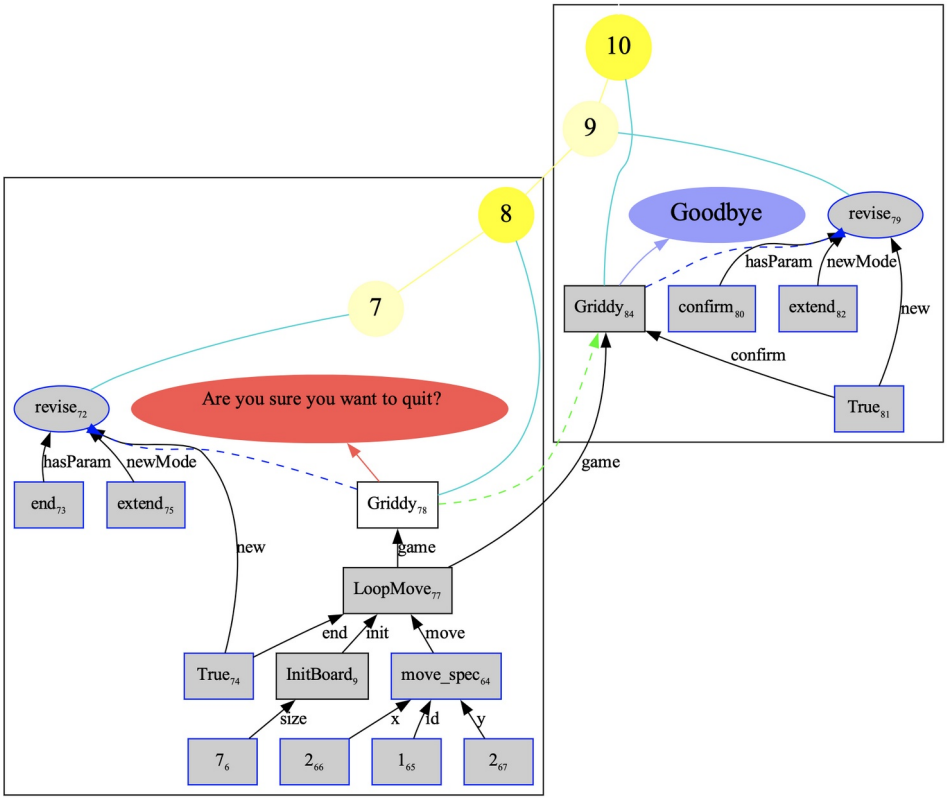
Graph#24 (partial graph, omitting display of Board)



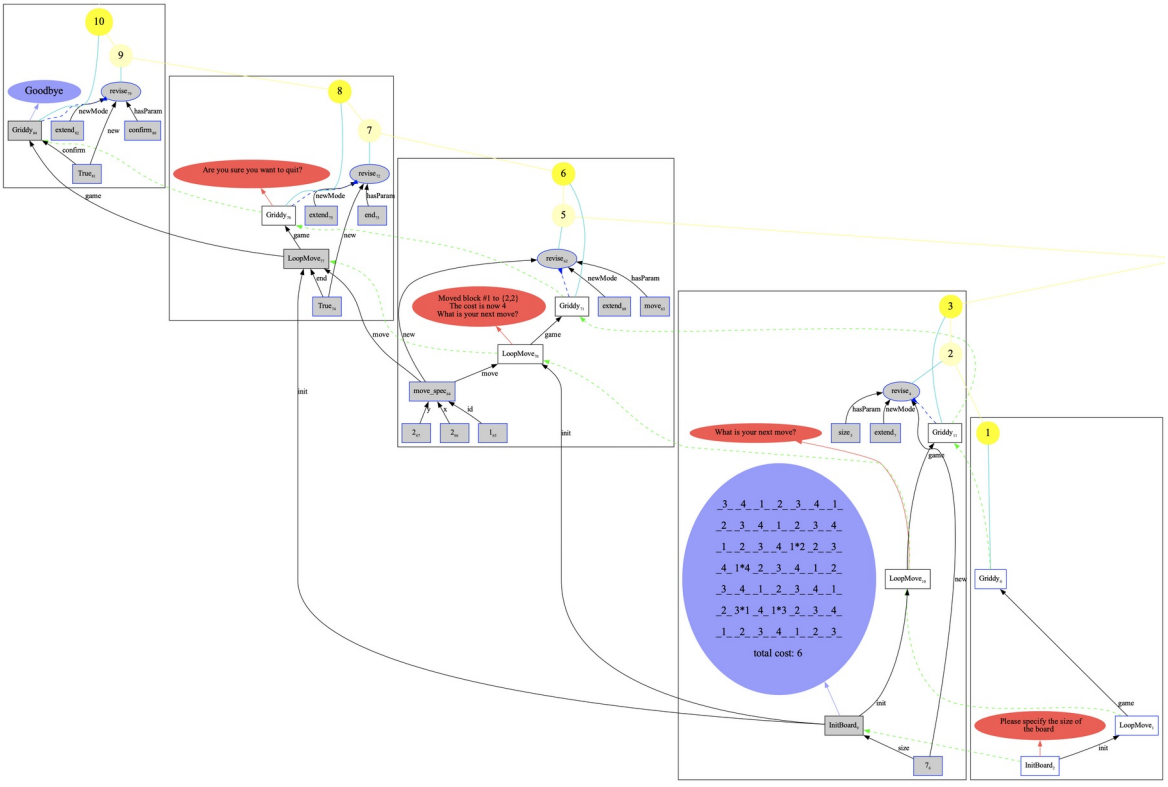
Given the input `end=True`, (the duplicated) `LoopMove` is now successfully evaluated, and the evaluation proceeds to the next node – `Griddy`, which in turn raises an exception asking the user to verify ending the game.

The user confirms:
`revise(hasParam=confirm, new=True, newMode=extend)` *[39]*
which adds the input `confirm=True` to (the duplicted) `Griddy`, which then proceeds to say goodbye and successfully finish the evaluation.

Graph#25



The entire session looks like this:



Initializing the process:

We've seen in the example, that we initialized the session by explicitly running:
`Griddy(game=LoopMove(init=InitBoard()))`

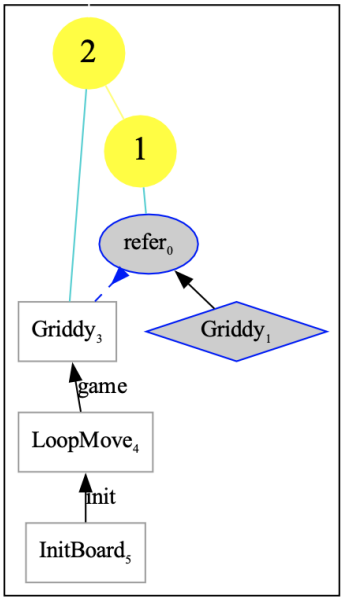
Another common pattern is to use `refer` for initializing processes.
This is convenient e.g. in settings where you may have several processes running in parallel (e.g. the user is buying a pizza, and buying movie tickets at the same time, switching back and forth between them). For these, we do not have to differentiate between the case where the process has already been previously created (and the user wants to resume them (context switching)), and the case where they have not been created yet. For both cases, we can use the same Pexp.

To do this, we use *fallback search* – in case refer fails to find a result, it can call the method Node.fallback_search() to execute custom designed to deal with failure to find search results (in SMCalFlow, failed queries on the graph often result in a call to an API to search external databases).
In case no results are found, refer will call the custom fallback_search() matching the node type of the query; e.g. `refer(Griddy?())` would call `Griddy.fallback_search()`.
Instead of a database call, we can use fallback_search to initialize a new process, so that `refer(Griddy?())` would return a process if one exists, otherwise, it would create a new process graph.

For our case, we need to define Griddy.fallback_Search():

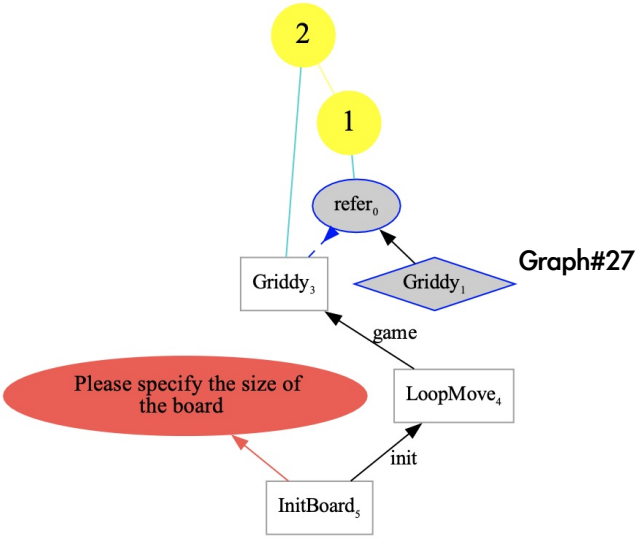
```
1   def fallback_search(self, parent, all_nodes=None,
2                       goals=None, do_eval=True, params=None):
3       g, _ = self.call_construct(
4                   'Griddy(game=LoopMove(init=InitBoard()))',
5                   self.context, add_goal=True)
6       return [g]
```

Which creates the process, adds it as a new goal.
`refer(Griddy?())` *[40]*



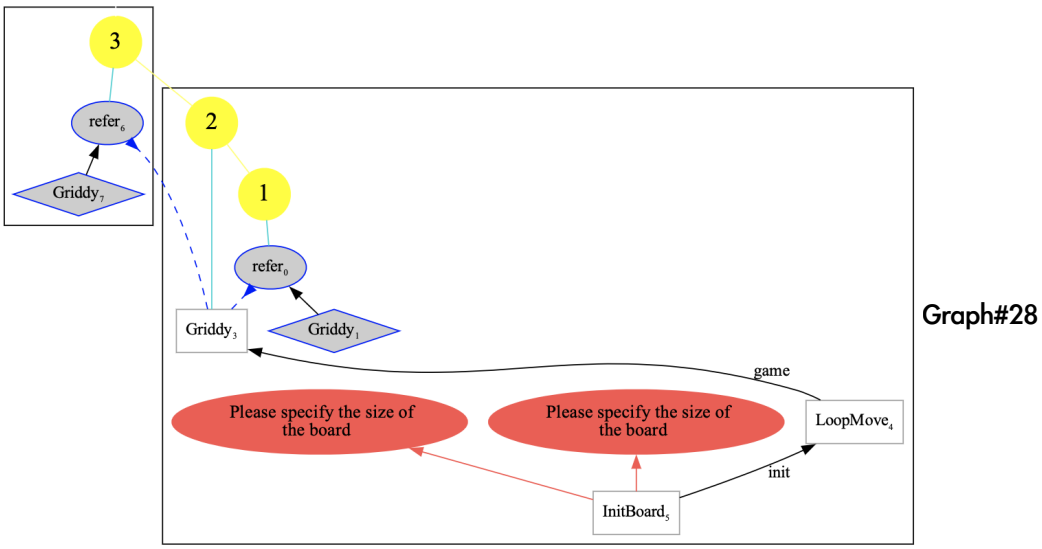
Graph#26

We don't just want to find (or create) the Griddy node/process, we also want to evaluate it.
For this, we can use the *special feature* notation `<!>`:
`<!>refer(Griddy?())` *[41]*
which will evaluate the result of the node annotated with `<!>` (in this case – the result of `refer`)



If we run this same Pexp once more:

```
<!>refer(Griddy?())
<!>refer(Griddy?())
```



The second `refer` finds the process (`Griddy`) which was created by the first `refer`, and the process is evaluated again (which causes the same exception to be raised again for `InitBoard`).

[Advanced feature] Suggestions:

In natural dialogues, a common strategy is for one person to make a suggestion: either a binary yes/no ("do you want to buy it?"), or multiple choice ("we have the shirt in red, blue and yellow"). By making the suggestion, the speaker sets up the context within which to interpret the other participant's answer: in the example above, "yes" means "I want to buy it".

Let's say we wanted to add to `LoopMove` the possibility to suggest a few moves to the user. The user can either accept one of the suggestions, or specify a different move.
`LoopMove`'s message would then be:
"*What's your next move? You can specify your own move, or choose one of the following: 1) move Block#1 to {5,5}, 2) move Block#3 to {…}, …*"

OpenDF has a mechanism to deal with this:
For each of the suggestions, we attach a Pexp, whose execution will result in the intended effect of the suggestion.
E.g., in the example above, we'll attach the following Pexp (which uses the same format as the one we used to represent a user request to move a block) to the first suggestion:
`revise(hasParam=move, new=move_spec(id=1,x=5,y=5), newMode=extend)`

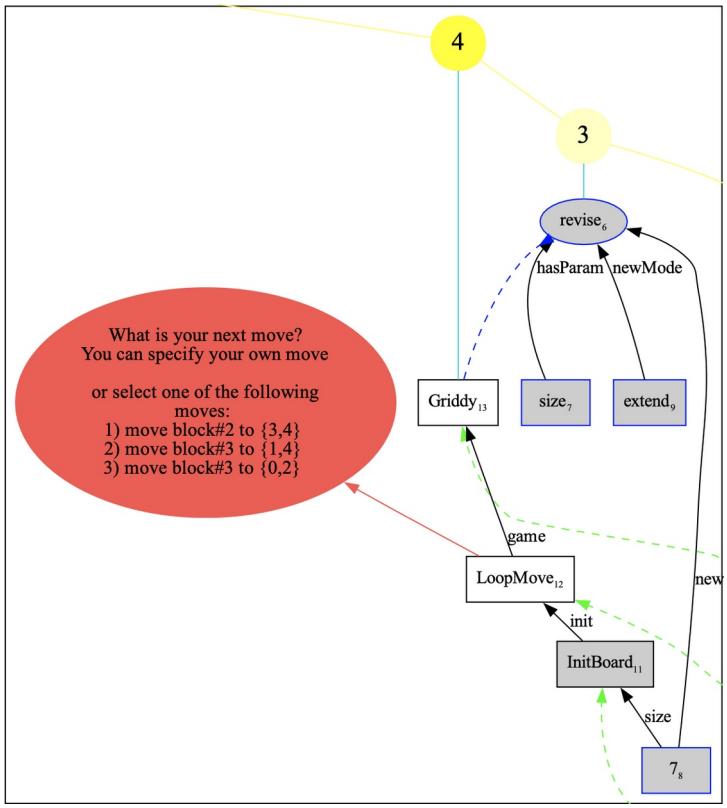
Adding the suggestions, `LoopMove.exec()` now looks like this:
(From here on, we use blockWorld_V4.py)

```
1     def exec(self, all_nodes=None, goals=None):
2         done = self.get_dat('end')
3         if not done:
4             msg = ''
5             mspec = self.input_view('move')
6             board = get_last_board(self.context)
7             if mspec:
8                 id, x, y = mspec.get_dats(['id', 'x', 'y'])
9                 if id is not None:  # non-empty move request
10                    try:
11                        board.move_block(id, x, y)
12                        cost = board.calculate_total_cost()
13                        msg = 'Moved block #%d to {%d,%d}  NL  ' +
14                              'The cost is now %d  NL  ' % (id, x, y, cost)
15                    except Exception as ex:
16                        msg = ex[0].message.text + '  NL  '
17            # Add suggestions
18            txt = []
19            sugg = ['revise(hasParam=move, new=move_spec(), newMode=extend)']
20            for i in range(3):
21                id, x, y = … #
22                txt.append('%d) move block#%d to {%d,%d}' % (i+1, id, x, y))
23                sugg.append('revise(hasParam=move, new=move_spec(
24                            id=%d, x=%d, y=%d), newMode=extend)' % (id, x, y))
25            msg += 'What is your next move?  NL  '
26            msg += 'You can specify your own move  NL  '
27            msg += 'or select one of the following moves:  NL  '
28            msg += '  NL  '.join(txt)
29            raise DFException(msg, self, suggestions=sugg)
```

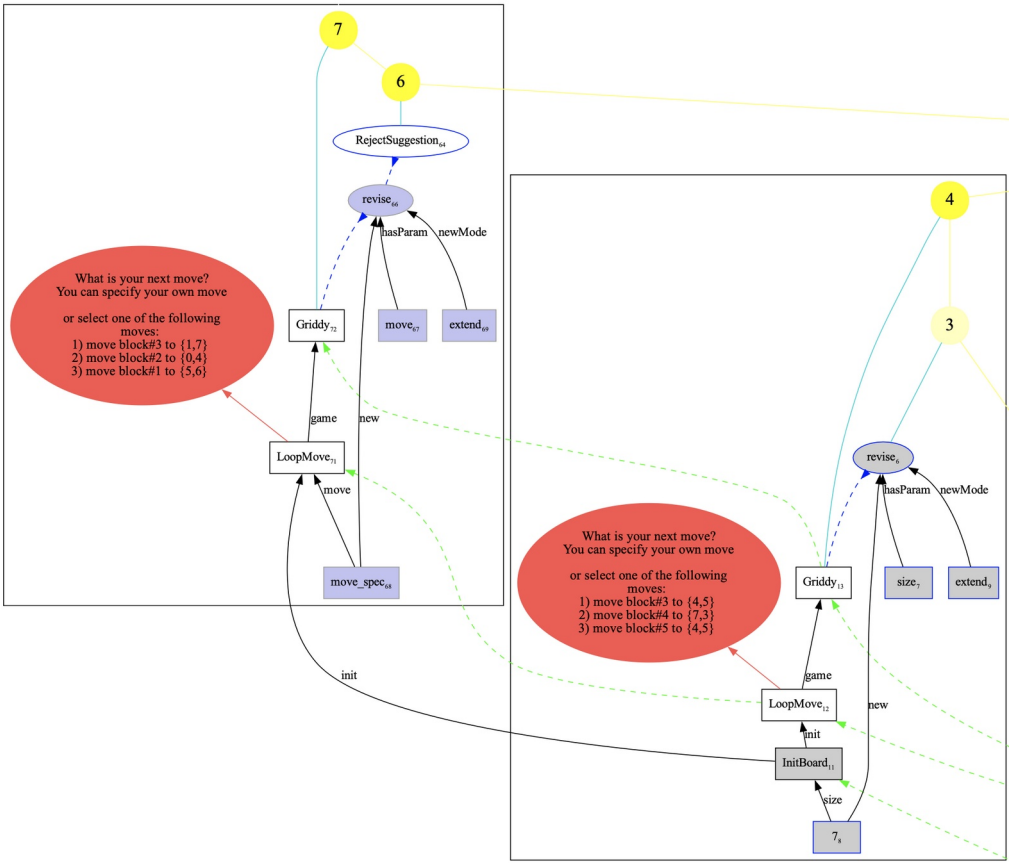
We define txt (line 18) to hold the text messages per suggestion, and sugg (line 19) to hold the Pexp expression corresponding to each suggestion.
Note that by convention, the first suggestion (`sugg[0]`) is the one which corresponds to the user (explicitly) rejecting all the given suggestions. Here, we initialize `sugg[0]` to be `'revise(hasParam=move, new=move_spec(), newMode=extend)'` i.e. make a new copy of `LoopMove`, with an empty move request.

In lines 20-24 we create 3 suggestions, by selecting `id`, `x`, `y` values (randomly, or with some strategy), and then generating a text description (line 21) and Pexp (lines 23-24) for it.
We create the message to the user, containing the suggestions, and finally raise an exception with the message, and the suggestions (line 29).

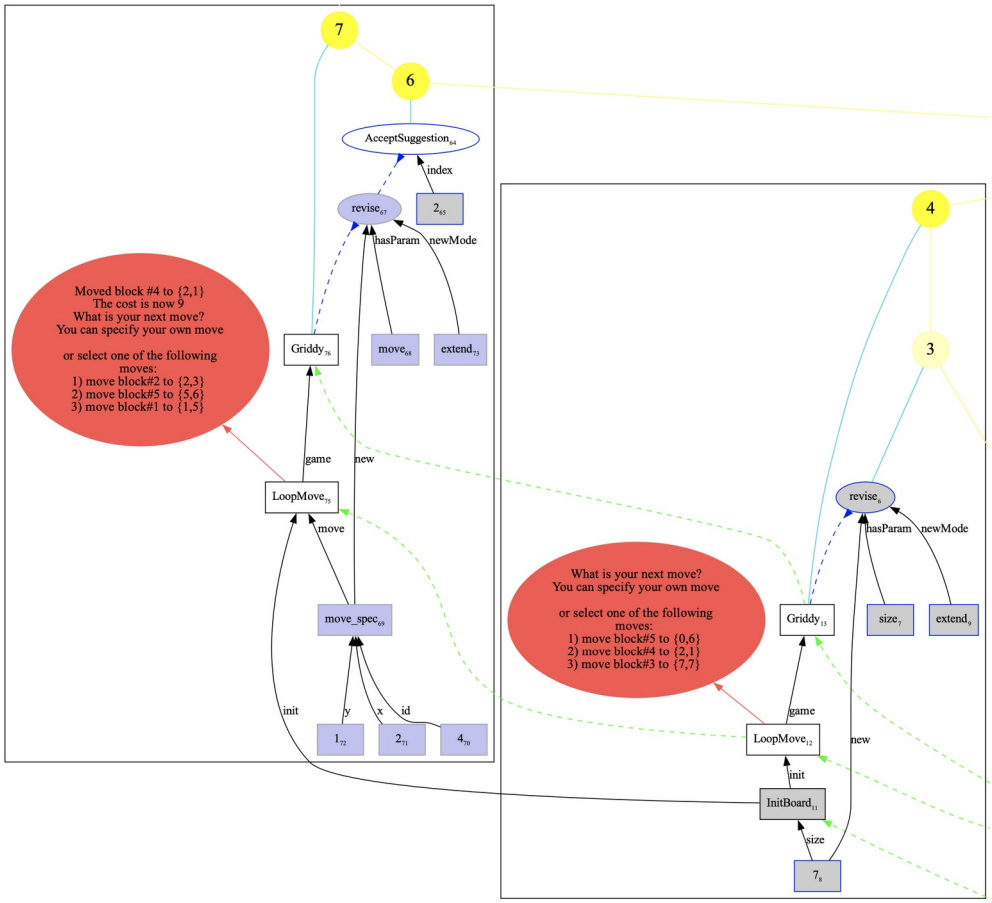
With the suggestions, the interaction now looks like this: (we select id,x,y randomly, so the values change between the graphs)



The user can now accept one of the suggestions, reject all suggestions, specify a move (or do something else).
To reject the suggestions, we execute
`RejectSuggestion()` *[44]*
Which would then run the Pexp in `suggestion[0]`. In our case – it will revise `LoopMove` with an empty `move_spec`, which will cause new suggestions to be made:



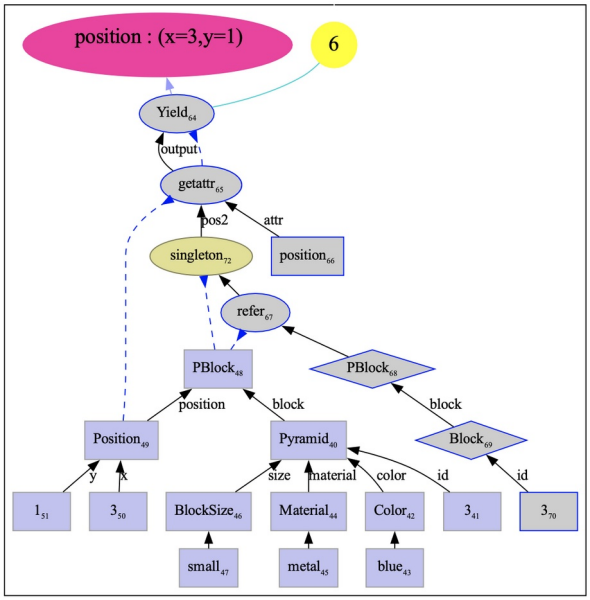
If the user wants to accept one of the suggestions, e.g. suggestion #2:
`AcceptSuggestion(2)` *[45]*
which will produce:



Suggestion#2 (move block#4 to {2,1}) was indeed selected and executed.

[Advanced topic]Side dialogs:

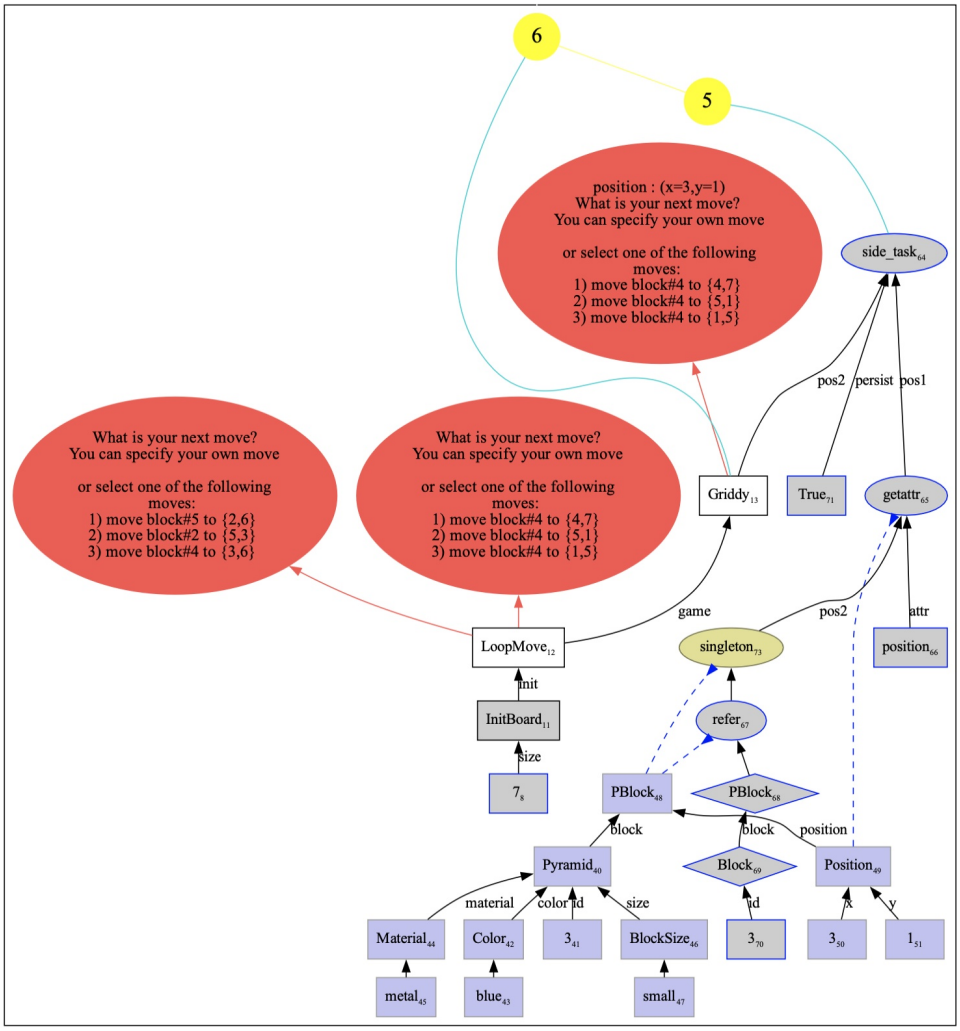
Let's say that while the user is moving blocks (in `LoopMove`) we wanted to allow the user to ask some queries about the blocks on the board (e.g. in the "lights-off" mode, where we don't show the position of the blocks after each move, the user can ask about the position of blocks).
If we simply evaluate the Pexp for the query (e.g. "*what is the position of block#3*"), we will get the right answer:
`Yield(:position(refer(PBlock?(block=Block?(id=3)))))` *[46]*



but we have now shifted the goal of the conversation: from evaluating `Griddy` to evaluating the query: the system responds: “position is {3,1}”, but now it does not ask for, or suggest, the next move – we’ve dropped the thread of the conversation.
What we want is to “tie” the query to the conversation: after evaluating the query, we continue with the evaluation of `Griddy` (asking the user for the next move).

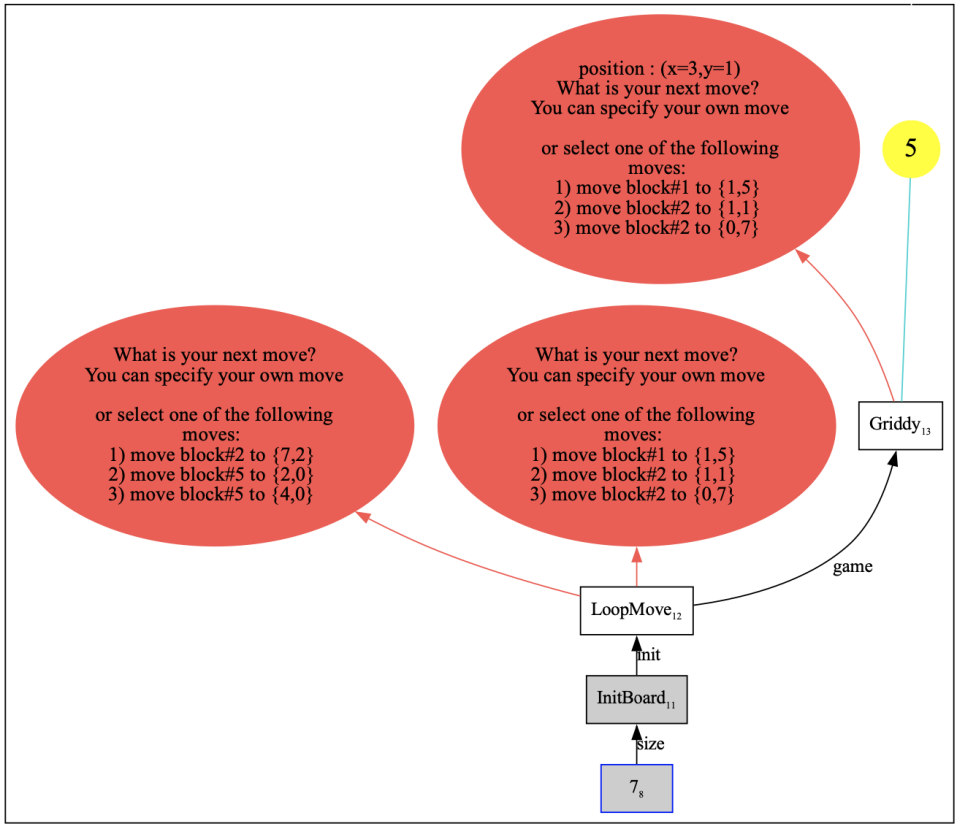
We can use `side_task` to do this. side_task ties the new Pexp to the previous goal: it evaluates the new Pexp, and after that returns to the previous goal.

```
side_task(task=:position(refer(PBlock?(block=Block?(id=3)))),
          persist=True)    [47]
```



`side_task` evaluated the query, re-evaluated `Griddy` (in the graph we see the old and new exceptions from `Griddy`), and combines their messages into a new exception (with the position **and** the suggestions).

If we omit the parameter `persist=True` from side_task, we get the default behavior, where side task is removed from the graph. In that case we’re just left with:



`side_task` is just one (simplistic) example of dealing with more complex dialogue structures. We hope more and better patterns will be added in the future!