Progress

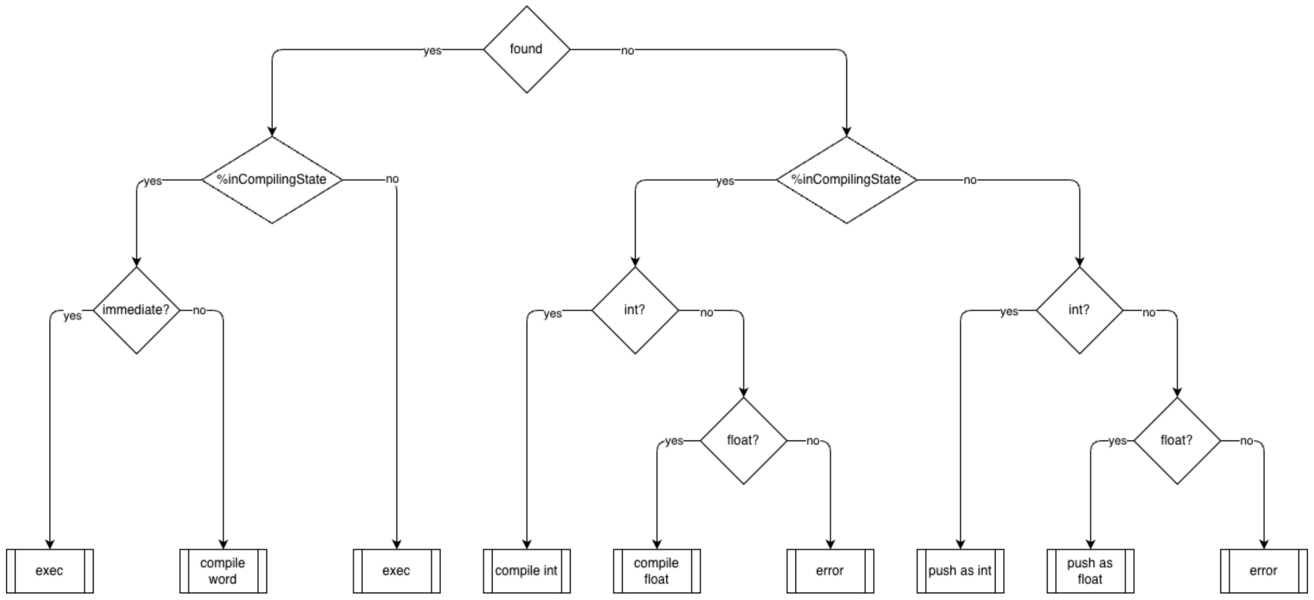
Begin Dec. 2, 2025

Goal: transmogrify a simple decision tree diagram to `.frish` then to `.py`

[*These diagrams a PNG files, hence not scalable. Open* `progress.drawio` *using* `draw.io` *for the actual source code (diagrams) and use draw.io's ability to zoom in and out.*]

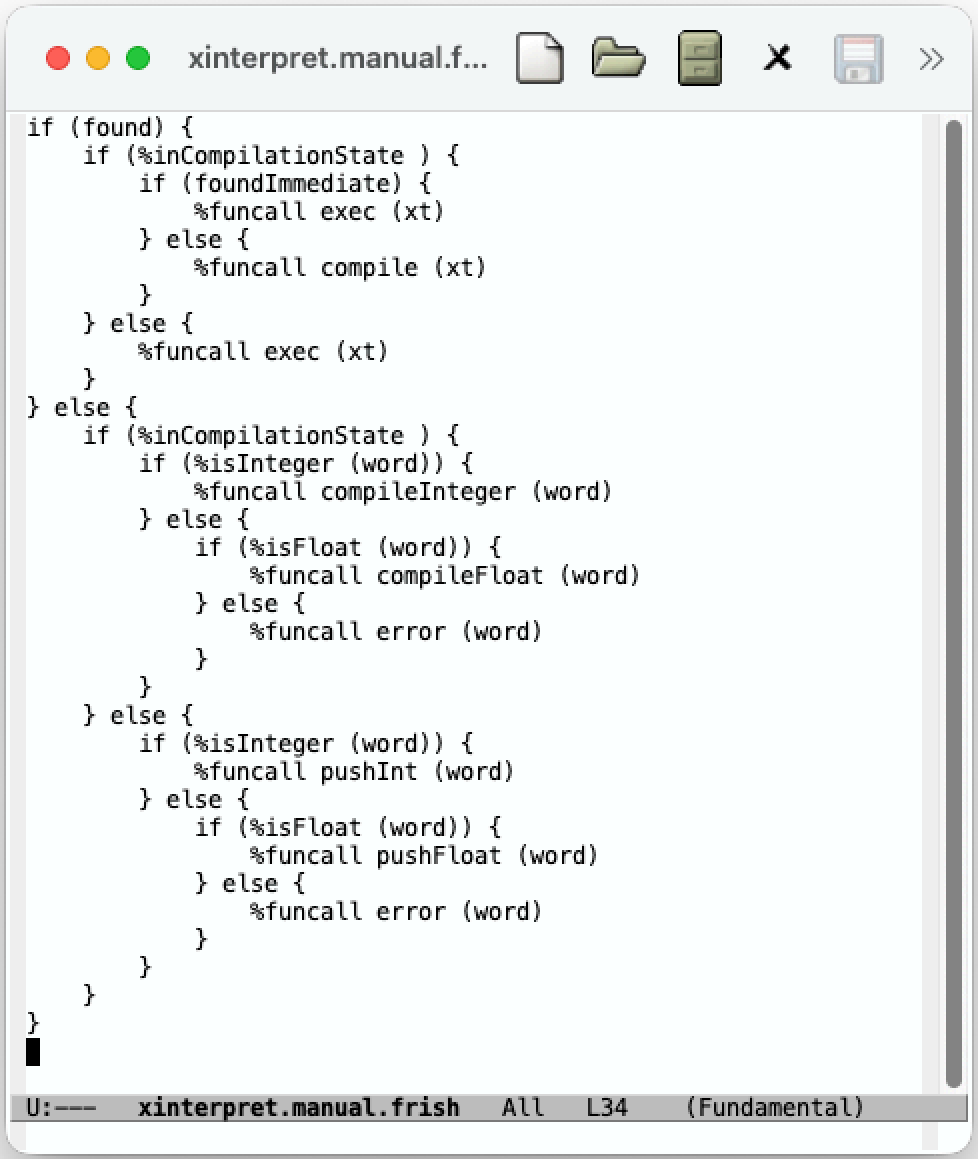
Decision Tree Diagram

- drew the decision tree diagram



Mocked Up Code

- mocked up manually



```
if (found) {
    if (%inCompilationState ) {
        if (foundImmediate) {
            %funcall exec (xt)
        } else {
            %funcall compile (xt)
        }
    } else {
        %funcall exec (xt)
    }
} else {
    if (%inCompilationState ) {
        if (%isInteger (word)) {
            %funcall compileInteger (word)
        } else {
            if (%isFloat (word)) {
                %funcall compileFloat (word)
            } else {
                %funcall error (word)
            }
        }
    } else {
        if (%isInteger (word)) {
            %funcall pushInt (word)
        } else {
            if (%isFloat (word)) {
                %funcall pushFloat (word)
            } else {
                %funcall error (word)
            }
        }
    }
}

U:---   xinterpret.manual.frish    All    L34    (Fundamental)
```

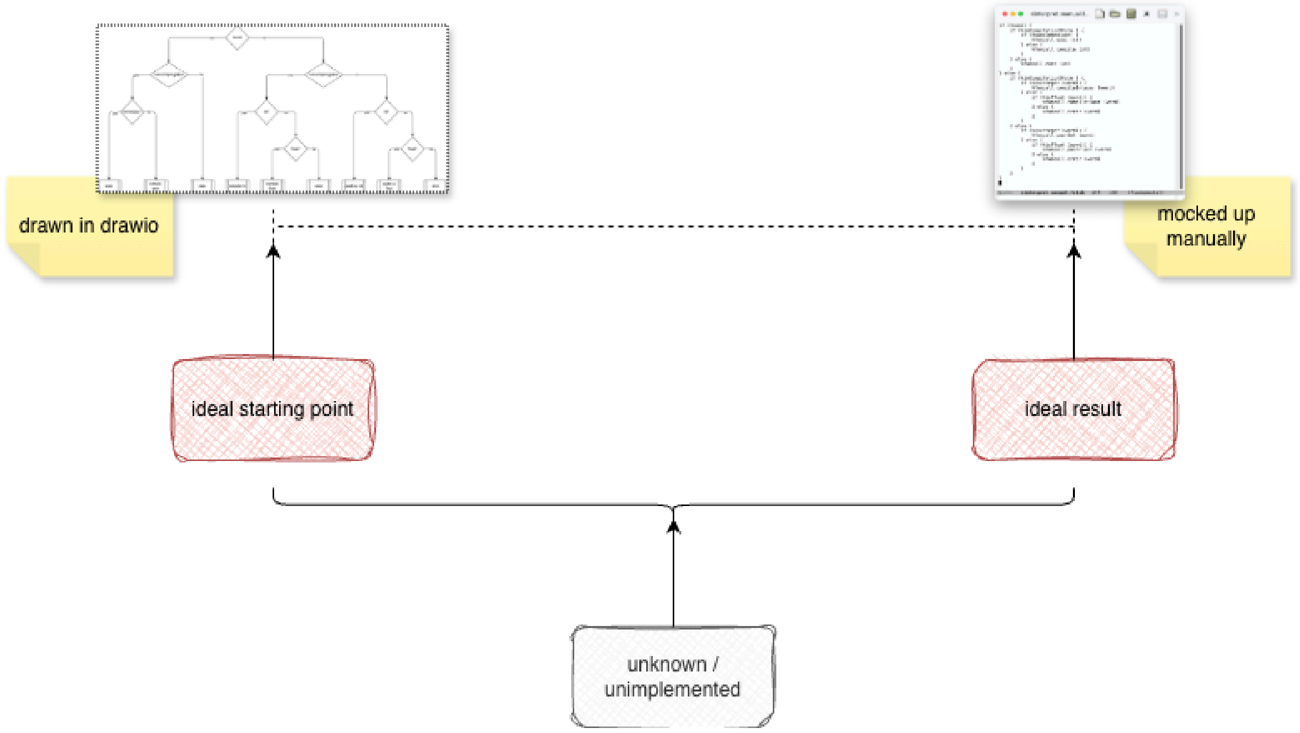
This `.frish` code looks a lot like Python with braces instead of indentation for parsing. The `%` items are calls to Python functions or Python variables, external to the diagram. I expect them to be found in some `.py` file. I use the `%` character for my eyes during debugging, so that I can see what I think I want to accomplish.

The diagram and mocked up code were generated manually and may contain errors. I will revise as I proceed.

Beginning

sketch of the two extremes

- left = what I want to draw
- right = the `.frish` code that I want the diagram to be transmogrified to
- I haven't included a mockup of `.frish` -> `.py`



Design decision: I plan to transmogrify the diagram to a meta-syntax that I call `.frish`. This will let me generate code in many different languages, e.g. Python, JS, C, etc. I've already built a frish transmogrifier, related to the Forth-ish project. I will just steal that code for this small project.

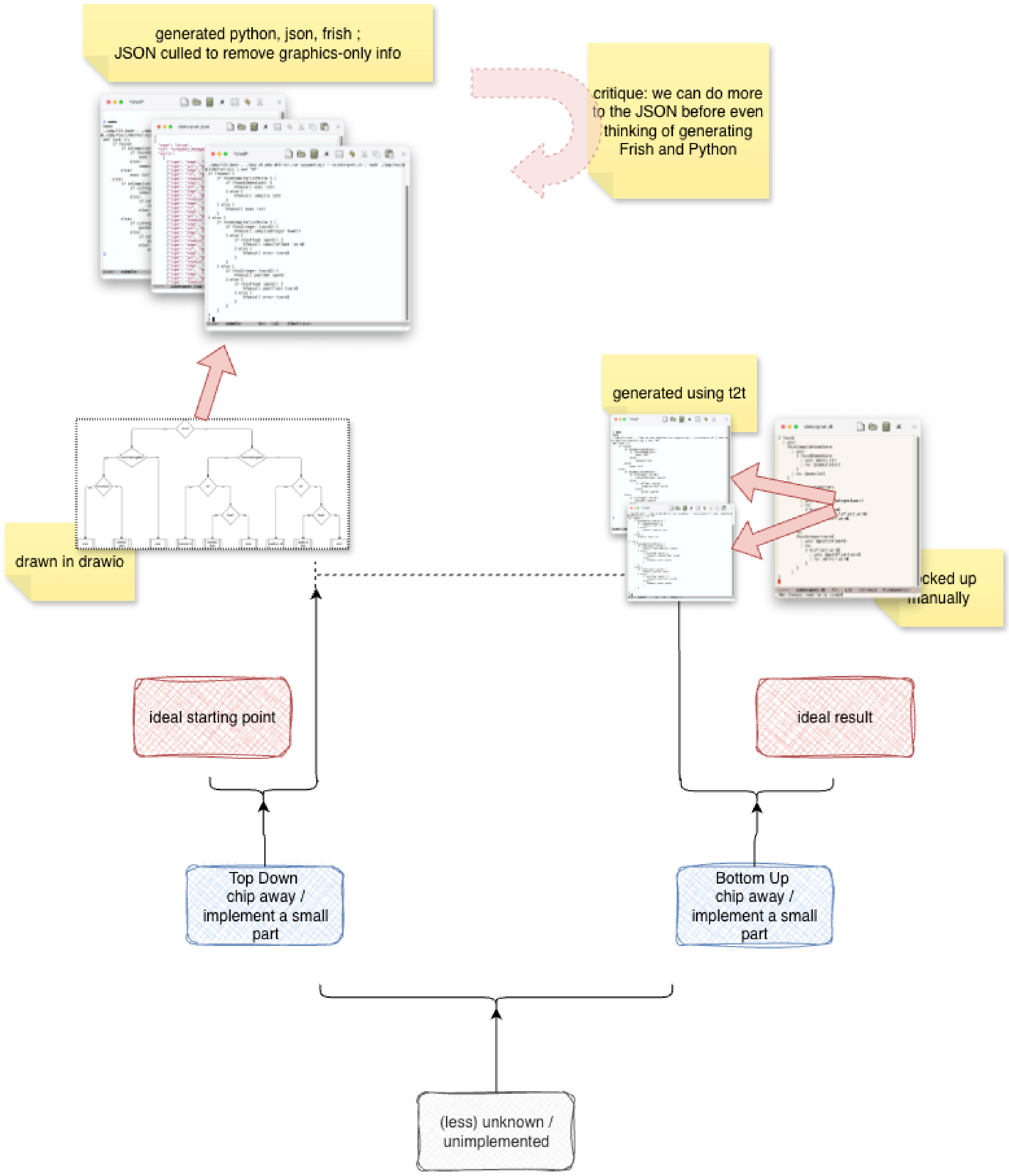
Dec. 2, 2025

I began chipping away at the problem from both ends - the diagram and the code generator.

I started on Dec. 2 using only a `Makefile` and `t2t.bash` scripts. I didn't draw the process using PBP at this point (I drew the transmogrifier code on Dec. 7, see below).

On Dec. 2, I used `das2json.mjs` to convert the graphML to JSON. I drew the diagram using draw.io, it saved the diagram as graphML.

I wrote `dt.ohm` and `dt.rwr` scripts to transmogrify
I was thinking that I would dive into Prolog code ASAP. On inspecting the JSON code, I saw opportunities to reduce the information further using `jq`

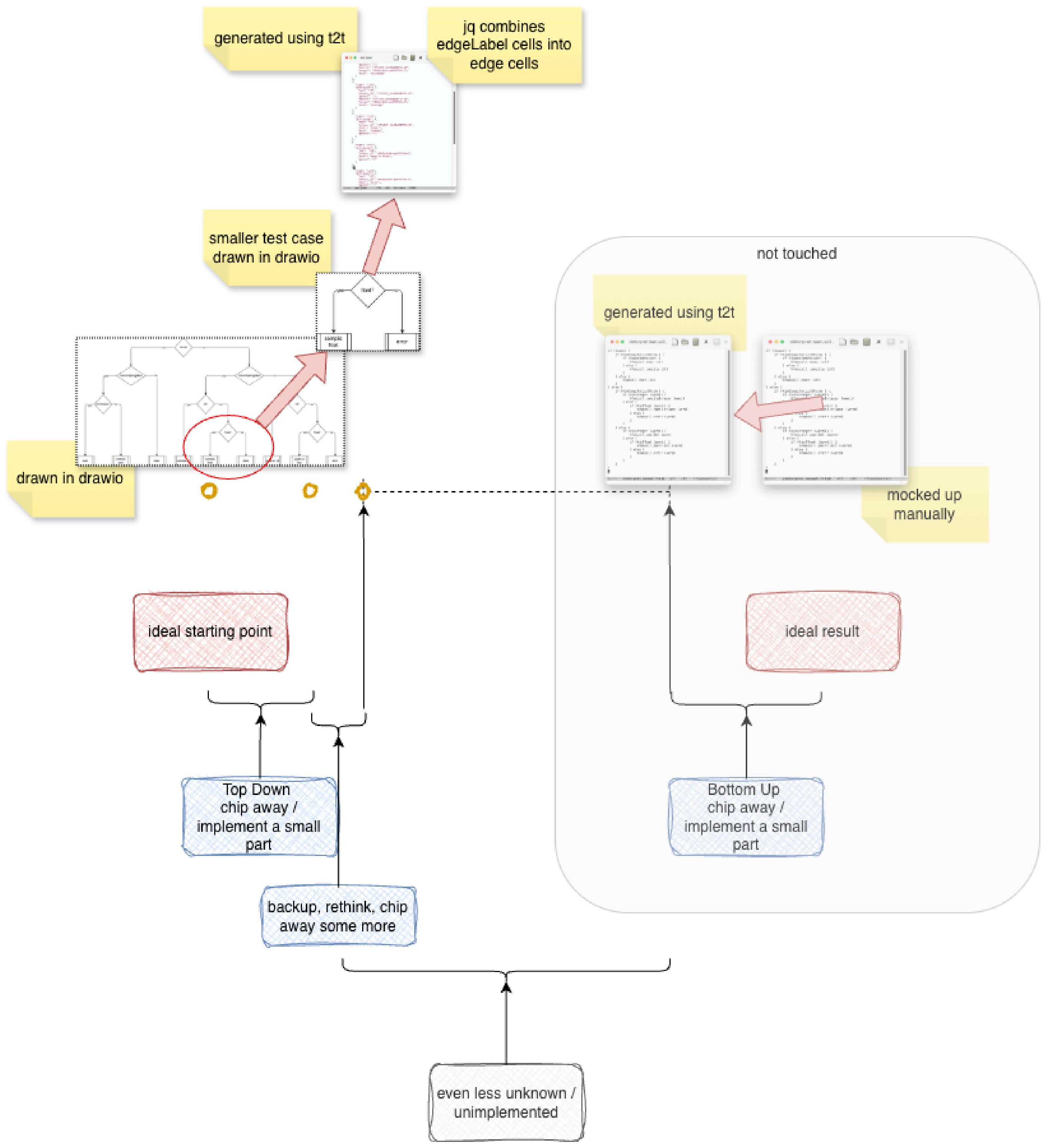

generated python, json, frish ;
JSON culled to remove graphics-only info
critique: we can do more to the JSON before even thinking of generating Frish and Python
generated using t2t
drawn in drawio
...cked up manually
ideal starting point
ideal result
Top Down chip away / implement a small part
Bottom Up chip away / implement a small part
(less) unknown / unimplemented

Dec 6, 2025

I created a smaller test diagram `test.drawio` to reduce the amount of noise I had to look at.

I followed up on the observation that I could reduce the JSON further by simply using `jq`.

I don't know much about how to write `jq` scripts, so I asked Claude to write the `jq` script.

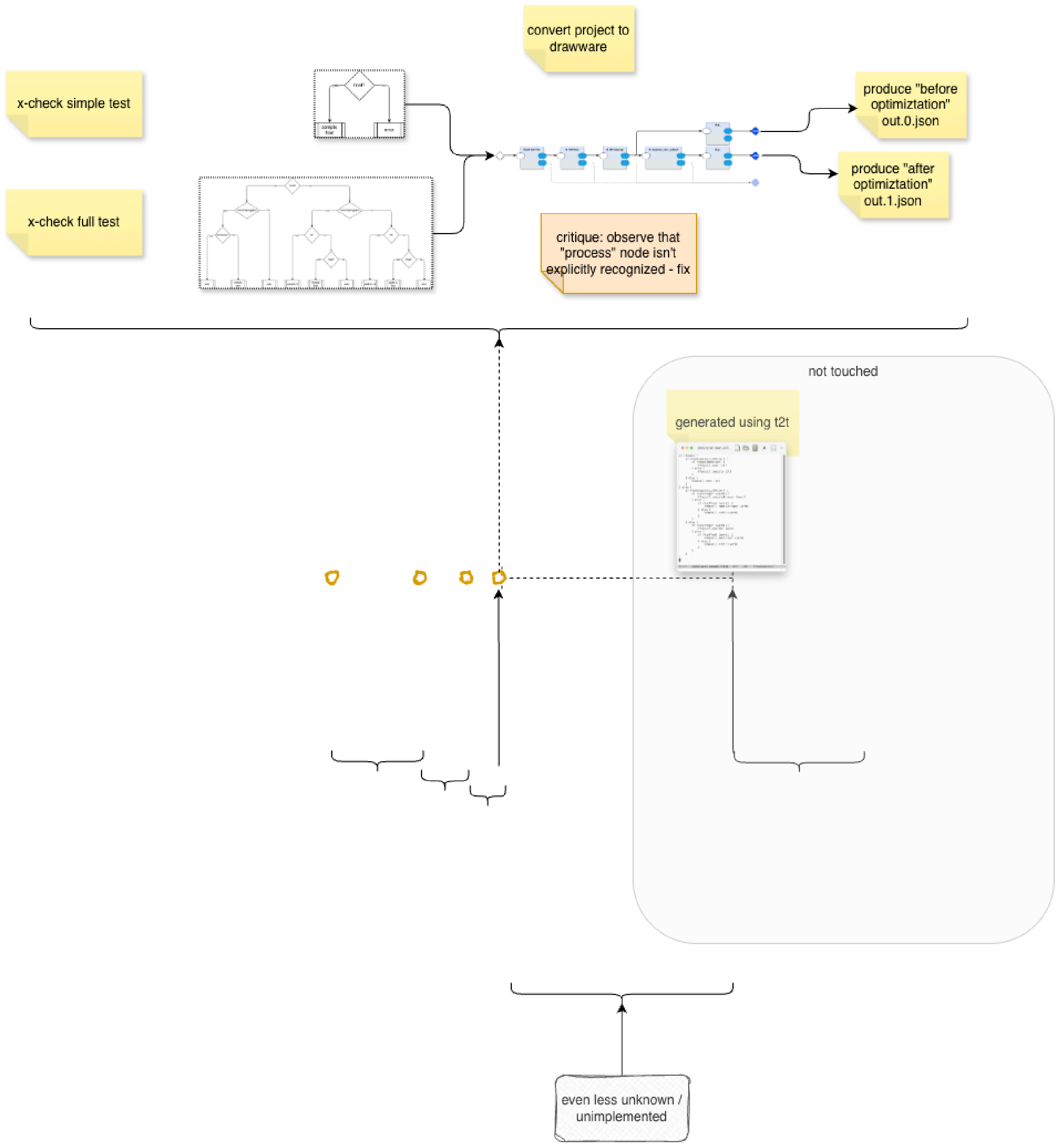

generated using t2t
jq combines edgeLabel cells into edge cells
smaller test case drawn in drawio
drawn in drawio
not touched
generated using t2t
mocked up manually
ideal starting point
ideal result
Top Down chip away / implement a small part
backup, rethink, chip away some more
Bottom Up chip away / implement a small part
even less unknown / unimplemented

Dec. 7 2025

I decided to switch from using just `.bash` scripts, to doing the whole thing in PBP. This immediately made it "obvious" what intermediate results I wanted to examine.

I stuck the pre-optimized output into `out.0.json` and the optimized output to `out.1.json` and (rapdily) eye-balled the resulting files.

I noticed a problem - the "process" node wasn't being recognized and emitted in the JSON output. This required a quick fix in the `.ohm` grammar and `.rwr` rewrite files (`dtree.ohm` and `dtree.rwr` respectively. Note that the t2t grammars created on Dec. 2 were called `dt.ohm` and `dt.rwr` . Similar, but different names, 2 sets of grammars.)


convert project to drawware
x-check simple test
produce "before optimiztation" out.0.json
produce "after optimiztation" out.1.json
x-check full test
critique: observe that "process" node isn't explicitly recognized - fix
not touched
generated using t2t
even less unknown / unimplemented

Dec 27, 2025

On Dec. 27, I switched to remembering what my progress was and I simply converted the IR code generator to PBP. The grammars `dt.ohm` and `dt.rwr` were already written (Dec. 2), so the job was "easy".

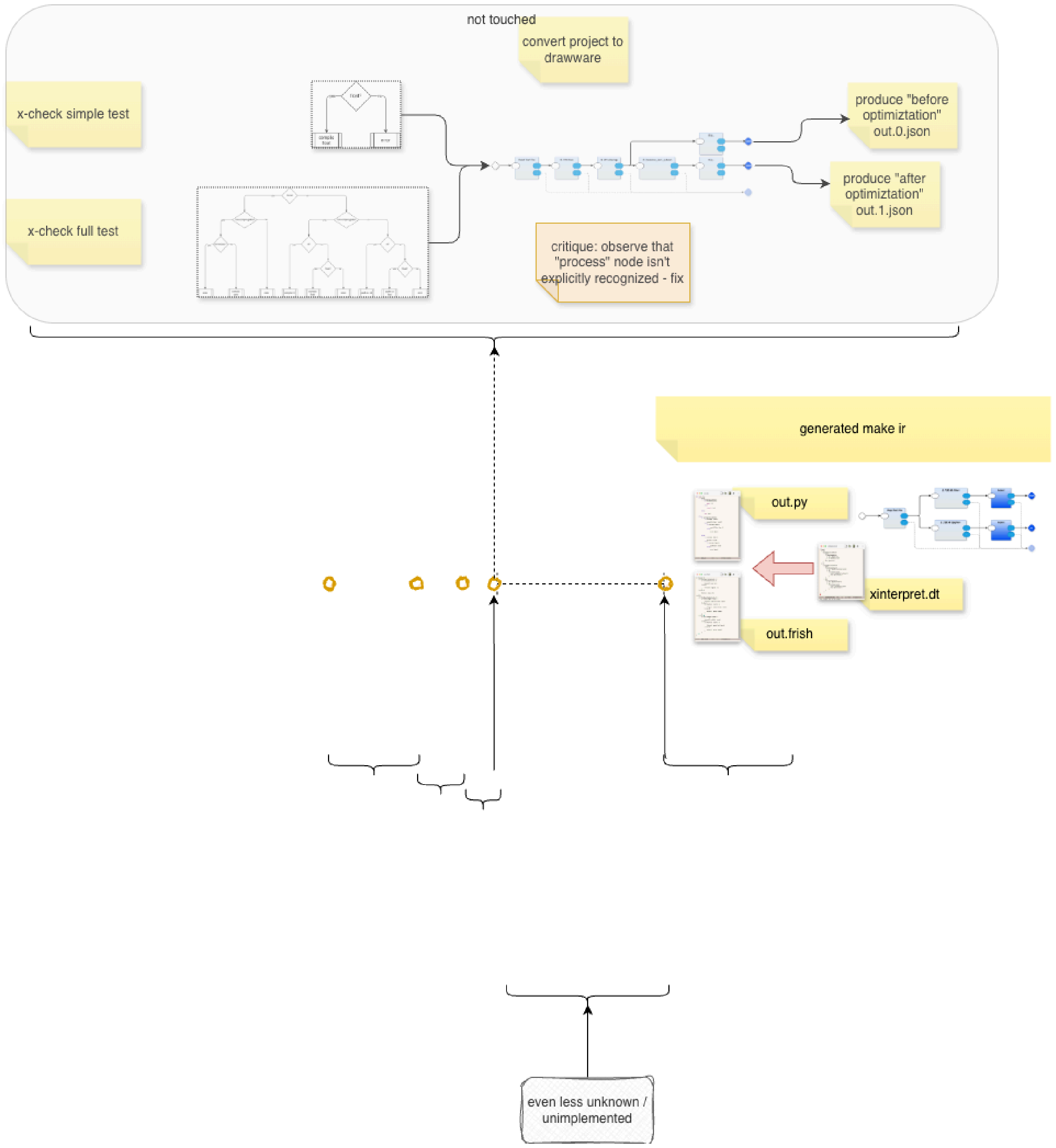
Question: why did I start out using `.bash` scripts then switch to PBP? Why didn't I start with PBP? Bash is good for expressing sequential pipelines. But, as soon as I wanted to start probing intermediate results, `.bash` gets kind of clumsy and inconvenient (not impossible, just inconvenient and slows me down). Draw.io is "acceptable" for creating PBP diagrams, but, leaves a lot to be desired. My knee-jerk reaction was to avoid draw.io for as long as possible.

By this time, I began realizing that I could use `jq` for a lot more of the transmogrification than I had expected. I was expecting to dive into Prolog (SWIPL) right away, but I was getting away with doing more and more with just `jq`. Part of the "beauty" of this approach is that `das2json` converts `.drawio` files into JSON which makes them much more accessible to simpler tools like `jq`. I still think that I'm going to need Prolog to do some inferencing. `jq` isn't powerful enough to do inferencing. I *could* use raw Python or raw Javascript to do the inferencing, but, I know from experience that Prolog lets me express my ideas with fewer errors.

In general, the goal is to spend time *thinking* instead of writing code and dealing with niggly details required to appease type checkers and the syntaxes of the existing PLs. Most of my time is spent staring at the JSON and trying to "see" what patterns emerge.

I've been incrementally chipping away at the JSON files. I write a bit of PBP code to simplify the JSON, then I spot "new" patterns in the JSON that I hadn't seen earlier. I know from experience that this is going to continue until there is no more "problem" left (the right side and left side meet in the middle).

I'm doing top-down-bottom-up development. My tools (PBP, Claude) are letting me do little bits of work that progressively suggest new ideas about how to get to what I wanted. I didn't predict that the design was going to go this way, but, I knew from experience that I could rely on this iterative approach to inch my way towards a solution.


not touched
convert project to drawware
x-check simple test
x-check full test
produce "before optimiztation" out.0.json
produce "after optimiztation" out.1.json
critique: observe that "process" node isn't explicitly recognized - fix
generated make ir
out.py
xinterpret.dt
out.frish
even less unknown / unimplemented

Dec. 29, 2025

On Dec. 29, I simply added on more Block to the transmogrifier diagram. This reduces the JSON even further and gets me even closer to the end goal.

I expect that I'm going to use SWIPL soon. I expect that the inferencing I need will be only a few (10-20) lines of Prolog. Claude is generating over 100 lines. I think that I can do better.

So, I'm just chipping away at the JSON, making it flatter and easier to process with Prolog rules.

I'm going to need to need to use inferencing to recognize how the decision tree branches are recursively connected all the way down. I'm expecting the I need to tree-walk the data and emit code as I go. Again, I could do this in Python or JS, but, Prolog syntax makes it easier to express this kind of exhaustive search (aka tree-walk) thing.

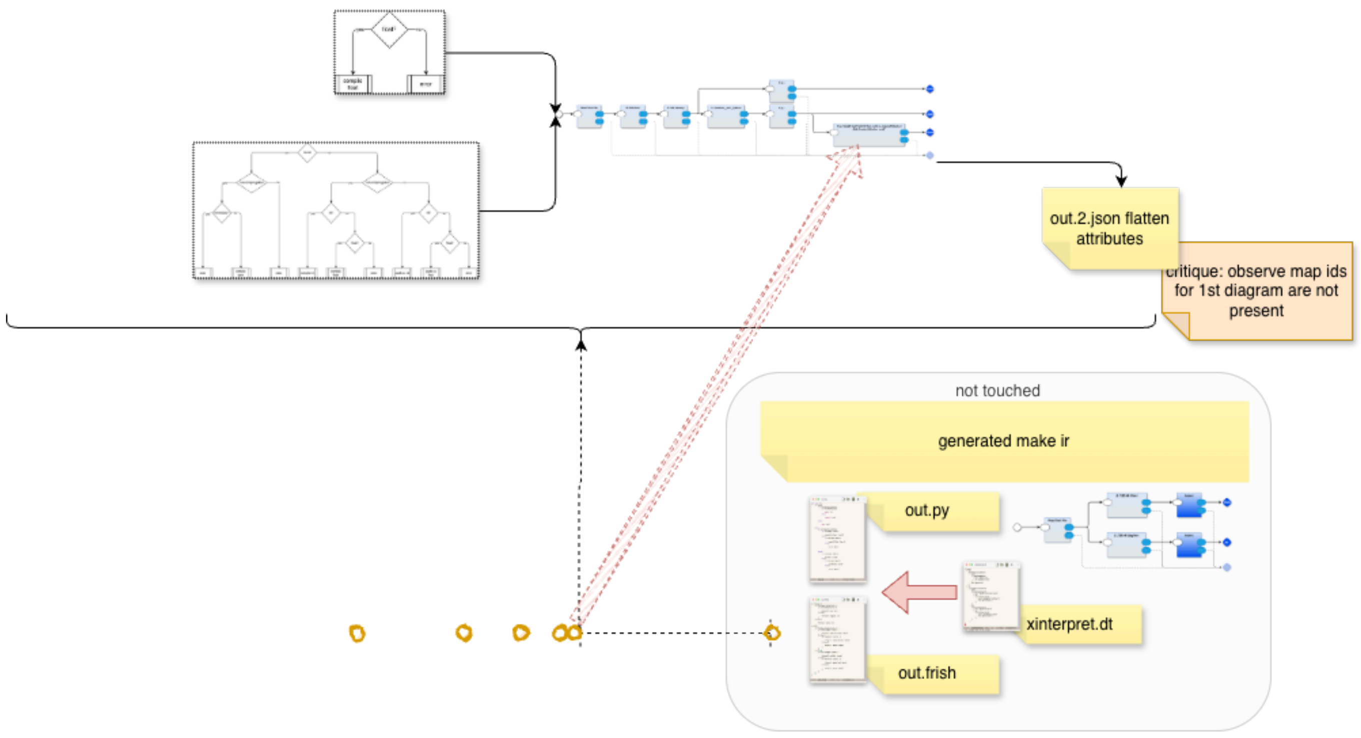
The short-term goal is to make the JSON *so* simple that the inferencing rules become ridiculously easy to write. We'll see...

My biggest blocker is that the JSON data is structured and nested. I know how to use Prolog to search through flat, unstructured data. I need to learn how to use modern SWIPL syntax to write relational rules for structured JSON patterns.

I might give up trying to learn how to write relations for structured JSON and just write some code the flattens the JSON. This would need to generate one fact for each attribute in the form `attr(id,...)` which lets Prolog pattern match rules that have the same `id`s. Maybe I won't need to do that if I learn how to write relations that pattern-match SWIPL dictionaries. We'll see...

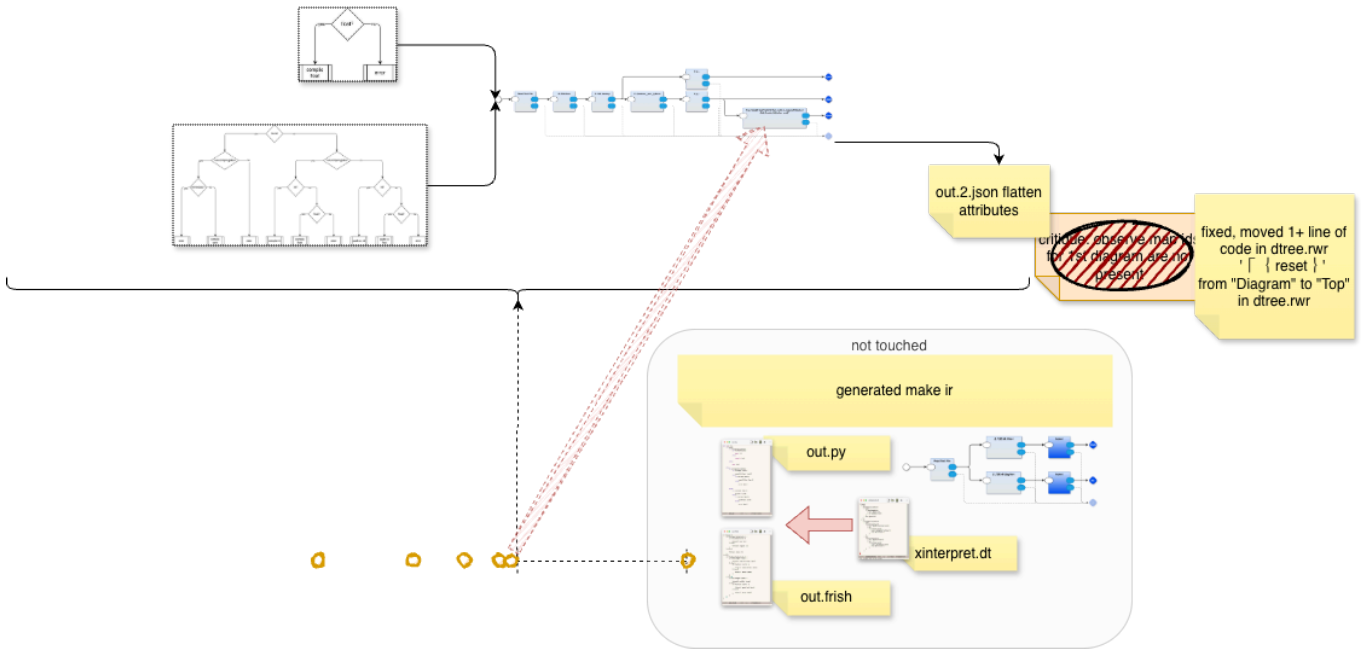
In the end I don't really care if the code is amazingly beautiful or not. I just need it to transmogrify the diagram to legal python code. There isn't a lot of code in this tool, so maintaining it is not going to be hampered if I use flat rules instead of dictionaries.

Learning to write relations with dictionaries is an "investment" in the future. If I find out that it's easy, I will be encouraged to build more of these kinds of little tools. (I already think I want a state diagram to code transmogrifier, for example).


out.2.json flatten attributes
critique: observe map ids for 1st diagram are not present
not touched
generated make ir
out.py
xinterpret.dt
out.frish

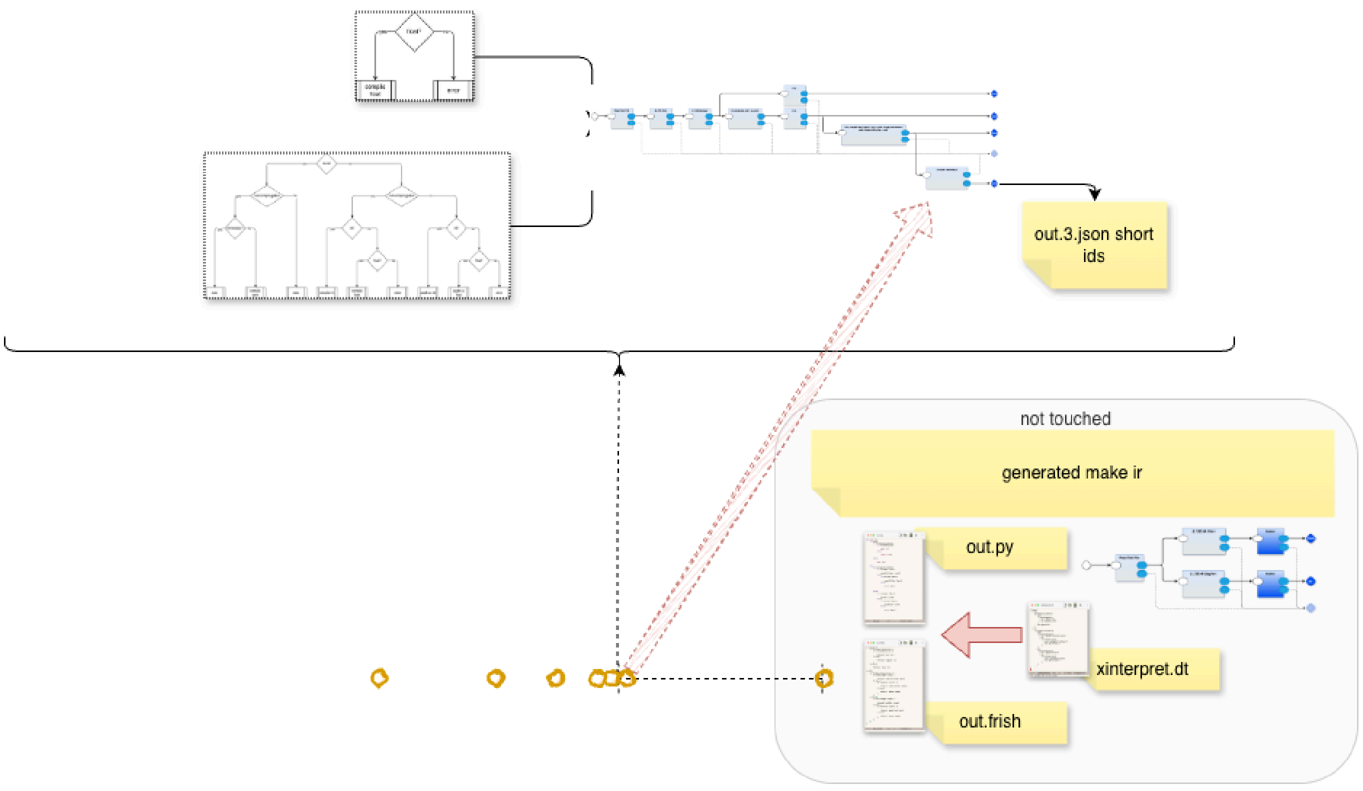
Dec 29 a, 2025

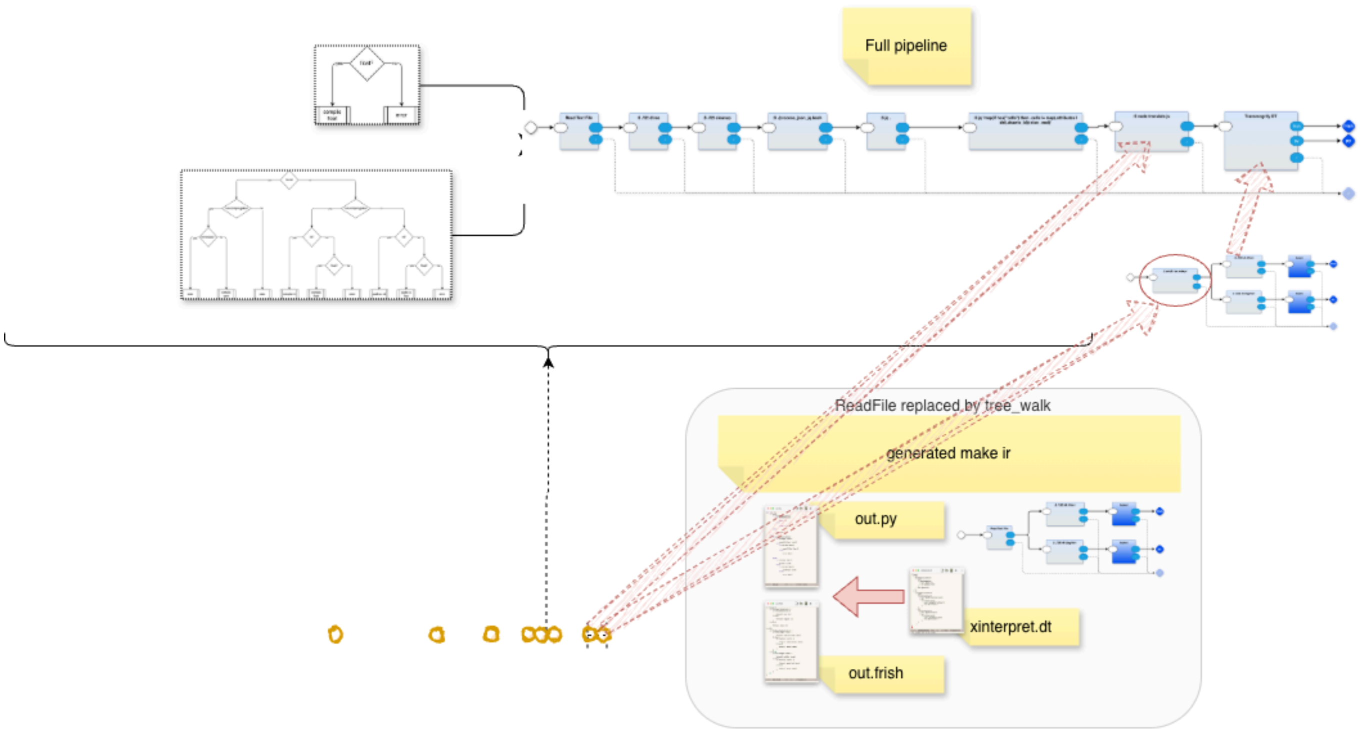
fixed the missing map entries by moving one line (plus closing bracket) in `dtree.rwr` from the `Diagram` rule up to the `Top` rule


out.2.json flatten attributes
critique: observe map ids for 1st diagram are not present
fixed, moved 1+ line of code in dtree.rwr
' ⎡ { reset } '
from "Diagram" to "Top" in dtree.rwr
not touched
generated make ir
out.py
xinterpret.dt
out.frish

Jan 1, 2026

added call to `translate.js` which rewrites all drawio long ids into shorter ids.





Added translate.js to pipeline. Not strictly necessary, but makes the ids shorter and more readable for debugging.

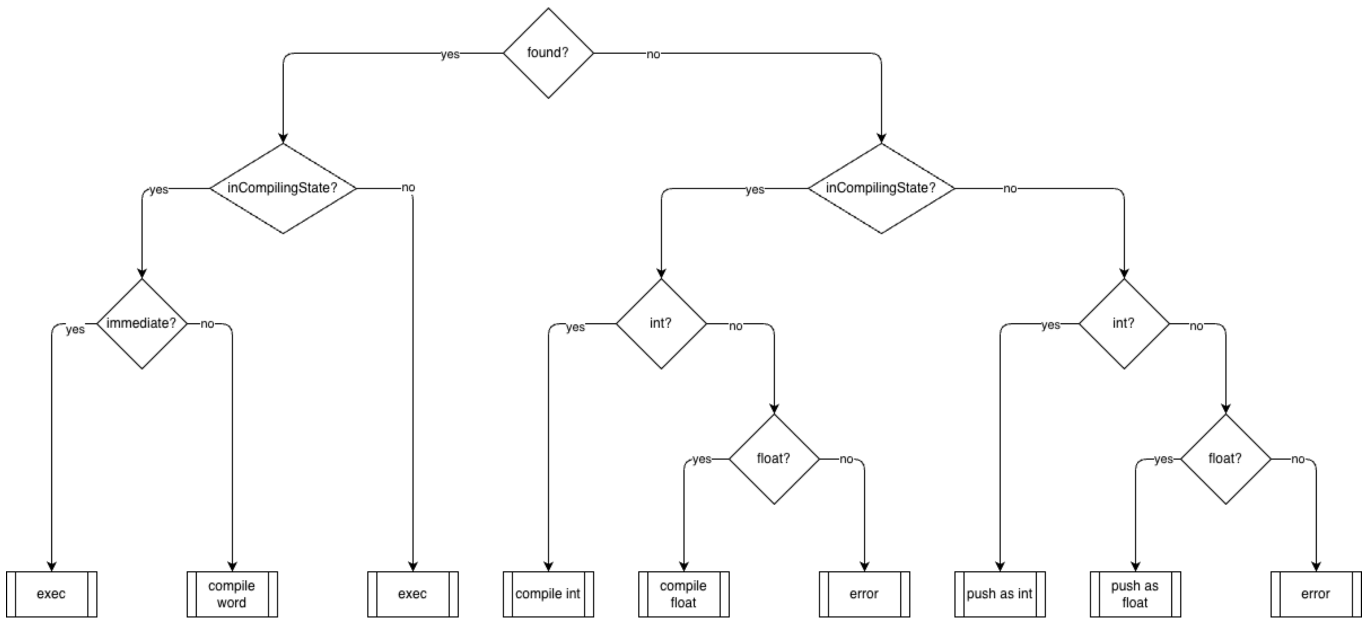
Cloned `ir` part to `Translate DT` part. Replaced `Read File` with `tree_walk`. Tree_walk accepts input from pipeline in form of JSON, then produces `.dt` code (square bracket decision tree syntax). This is fed into the `t2t2` parts to produce `.frish` and `.py` code.

The goal is to produce `.frish` code, so the `.py` code is not, actually, strictly necessary, but it acts as a quick sanity check - the Python compiler compiles and checks the syntax. This simple check points out any deficiencies in the generated Python code (just a cross check, for added comfort).

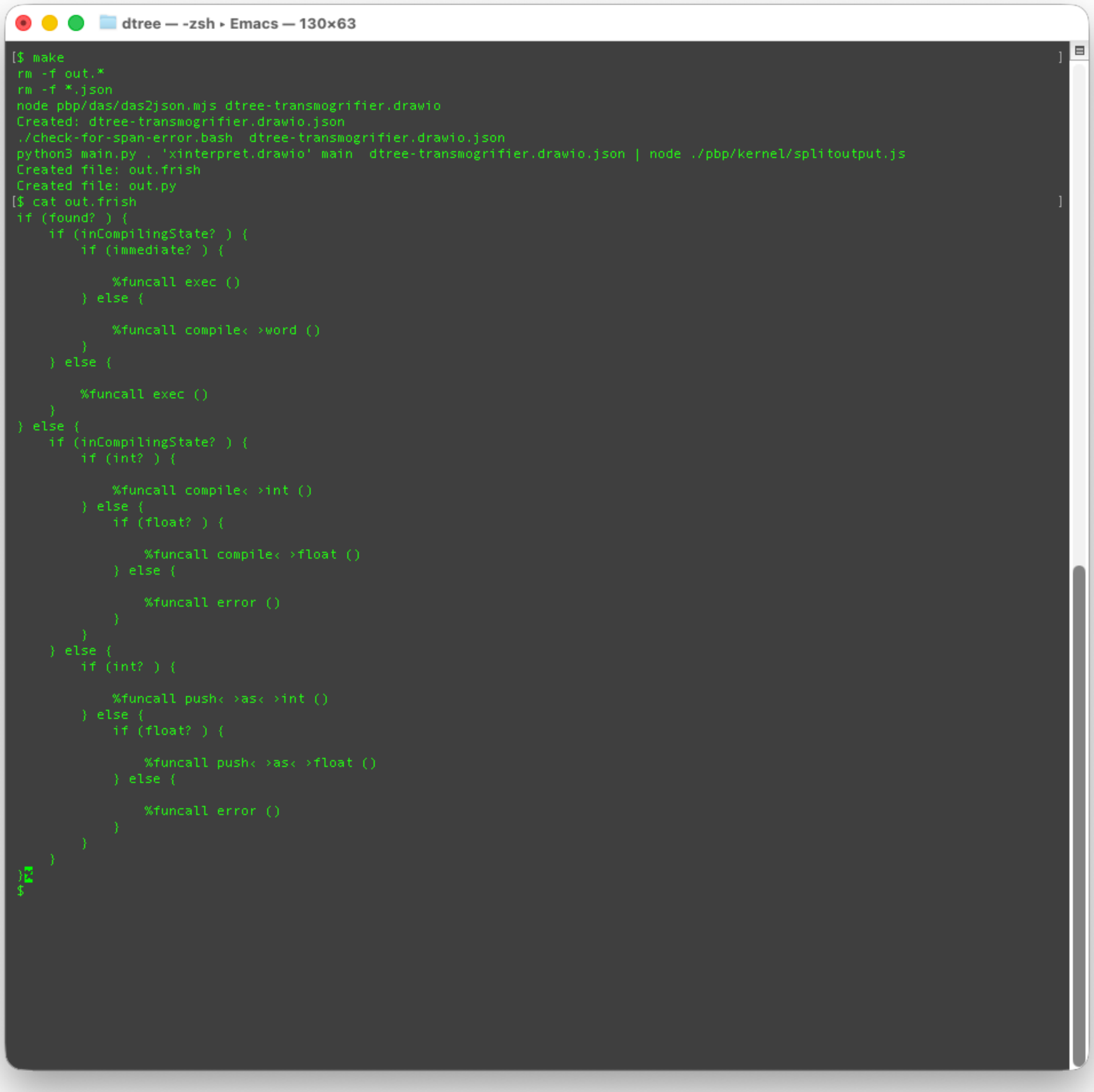
We're done with this sub-project. We have built a decision tree diagram to `.frish` code generator. Now, we can go back to the `.frish` project and we can copy/paste the generated code into the `xinterpret()` routine and test whether this creates a proper forthish interpreter that is portable to 3 languages.

Final

This diagram:



is transmogrified to this code:



```
[$ make
 rm -f out.*
 rm -f *.json
 node pbp/das/das2json.mjs dtree-transmogrifier.drawio
 Created: dtree-transmogrifier.drawio.json
 ./check-for-span-error.bash  dtree-transmogrifier.drawio.json
 python3 main.py . 'xinterpret.drawio' main  dtree-transmogrifier.drawio.json | node ./pbp/kernel/splitoutput.js
 Created file: out.frish
 Created file: out.py
[$ cat out.frish
 if (found? ) {
     if (inCompilingState? ) {
         if (immediate? ) {

             %funcall exec ()
         } else {

             %funcall compile‹ ›word ()
         }
     } else {

         %funcall exec ()
     }
 } else {
     if (inCompilingState? ) {
         if (int? ) {

             %funcall compile‹ ›int ()
         } else {
             if (float? ) {

                 %funcall compile‹ ›float ()
             } else {

                 %funcall error ()
             }
         }
     } else {
         if (int? ) {

             %funcall push‹ ›as‹ ›int ()
         } else {
             if (float? ) {

                 %funcall push‹ ›as‹ ›float ()
             } else {

                 %funcall error ()
             }
         }
     }
 }
 $
```

In the process, I invented a new way to represent spaces within identifiers and function names. Names that contain spaces are emitted with the spaces surrounded by `‹ ›` unicode characters.

```
push‹ ›as‹ ›float
```