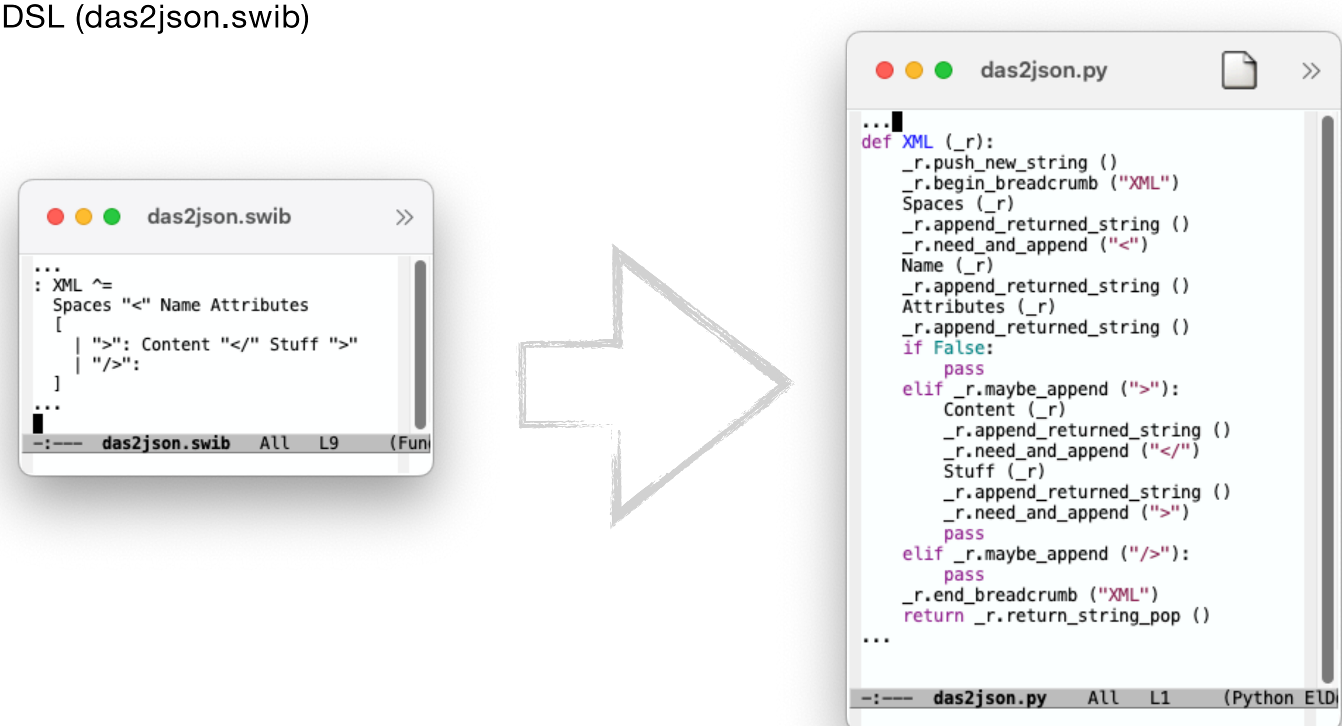
2024-05-12-OhmJS Example Use-Case

Introduction

This example contains a snippet of code written in a “new” language (an SCN[1]) taken out of context from the file https://github.com/guitarvydas/das2json/blob/main/das2json.swib.

DSL (das2json.swib)



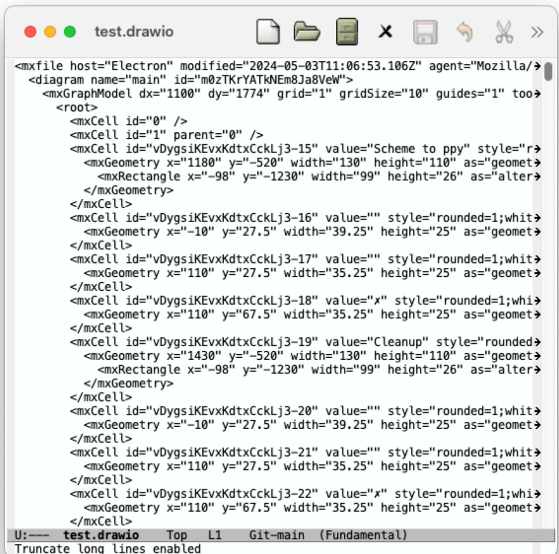
This .swib snippet is the 2nd rule in that file.

The file is used to generate stand-alone Python code (just Python code, not Python intertwined with 0D).

[1] Solution Centric Notation - like a DSL sans steroids.

Running “make” compiles all of `das2json.swib` into Python code, then runs the Python code with input from `test.drawio`. The full code for all of the above is available in the repository and runs.

The input is a stream of XML (the file `test.drawio` contains a special purpose variant of XML called *graphML*). The Python code parses the input XML and outputs the same XML with some parts removed[2].



```
test.drawio
<mxfile host="Electron" modified="2024-05-03T11:06:53.106Z" agent="Mozilla/➔
  <diagram name="main" id="m0zTKrYATkNEm8Ja8VeW">
    <mxGraphModel dx="1100" dy="1774" grid="1" gridSize="10" guides="1" too➔
      <root>
        <mxCell id="0" />
        <mxCell id="1" parent="0" />
        <mxCell id="vDygsiKEvxKdtxCckLj3-15" value="Scheme to ppy" style="r➔
          <mxGeometry x="1180" y="-520" width="130" height="110" as="geomet➔
            <mxRectangle x="-98" y="-1230" width="99" height="26" as="alter➔
          </mxGeometry>
        </mxCell>
        <mxCell id="vDygsiKEvxKdtxCckLj3-16" value="" style="rounded=1;whit➔
          <mxGeometry x="-10" y="27.5" width="39.25" height="25" as="geomet➔
        </mxCell>
        <mxCell id="vDygsiKEvxKdtxCckLj3-17" value="" style="rounded=1;whit➔
          <mxGeometry x="110" y="27.5" width="35.25" height="25" as="geomet➔
        </mxCell>
        <mxCell id="vDygsiKEvxKdtxCckLj3-18" value="✗" style="rounded=1;whi➔
          <mxGeometry x="110" y="67.5" width="35.25" height="25" as="geomet➔
        </mxCell>
        <mxCell id="vDygsiKEvxKdtxCckLj3-19" value="Cleanup" style="rounded➔
          <mxGeometry x="1430" y="-520" width="130" height="110" as="geomet➔
            <mxRectangle x="-98" y="-1230" width="99" height="26" as="alter➔
          </mxGeometry>
        </mxCell>
        <mxCell id="vDygsiKEvxKdtxCckLj3-20" value="" style="rounded=1;whit➔
          <mxGeometry x="-10" y="27.5" width="39.25" height="25" as="geomet➔
        </mxCell>
        <mxCell id="vDygsiKEvxKdtxCckLj3-21" value="" style="rounded=1;whit➔
          <mxGeometry x="110" y="27.5" width="35.25" height="25" as="geomet➔
        </mxCell>
        <mxCell id="vDygsiKEvxKdtxCckLj3-22" value="✗" style="rounded=1;whi➔
          <mxGeometry x="110" y="67.5" width="35.25" height="25" as="geomet➔
        </mxCell>
U:---  test.drawio   Top   L1    Git-main  (Fundamental)
Truncate long lines enabled
```


This parser works in a streaming manner. It does *not* create a parse tree, and, therefore, can handle input files of any size. On the other hand, it cannot - by default - walk the parse up and down and back again the way a tree walker can do. In this kind of parser, time is not reversible and any actions taken during the parse cannot be - easily - undone.

I will discuss only this snippet, hoping that the rest of the code will be understandable after this explanation.

[2] The final goal is to rewrite the graphML heavily, leaving in only the semantically interesting bits (boxes, ports, connections, names).

Overview

This is a rough attempt to show how a simple DSL is mapped into stand-alone, working Python code. I call this kind of simplistic DSL a *notation,* an “SCN” for short. “SCN” stands for *Solution Centric Notation.*

In this case, a “domain expert” wants to think about writing a parser. The parser is expressed using an SCN (called

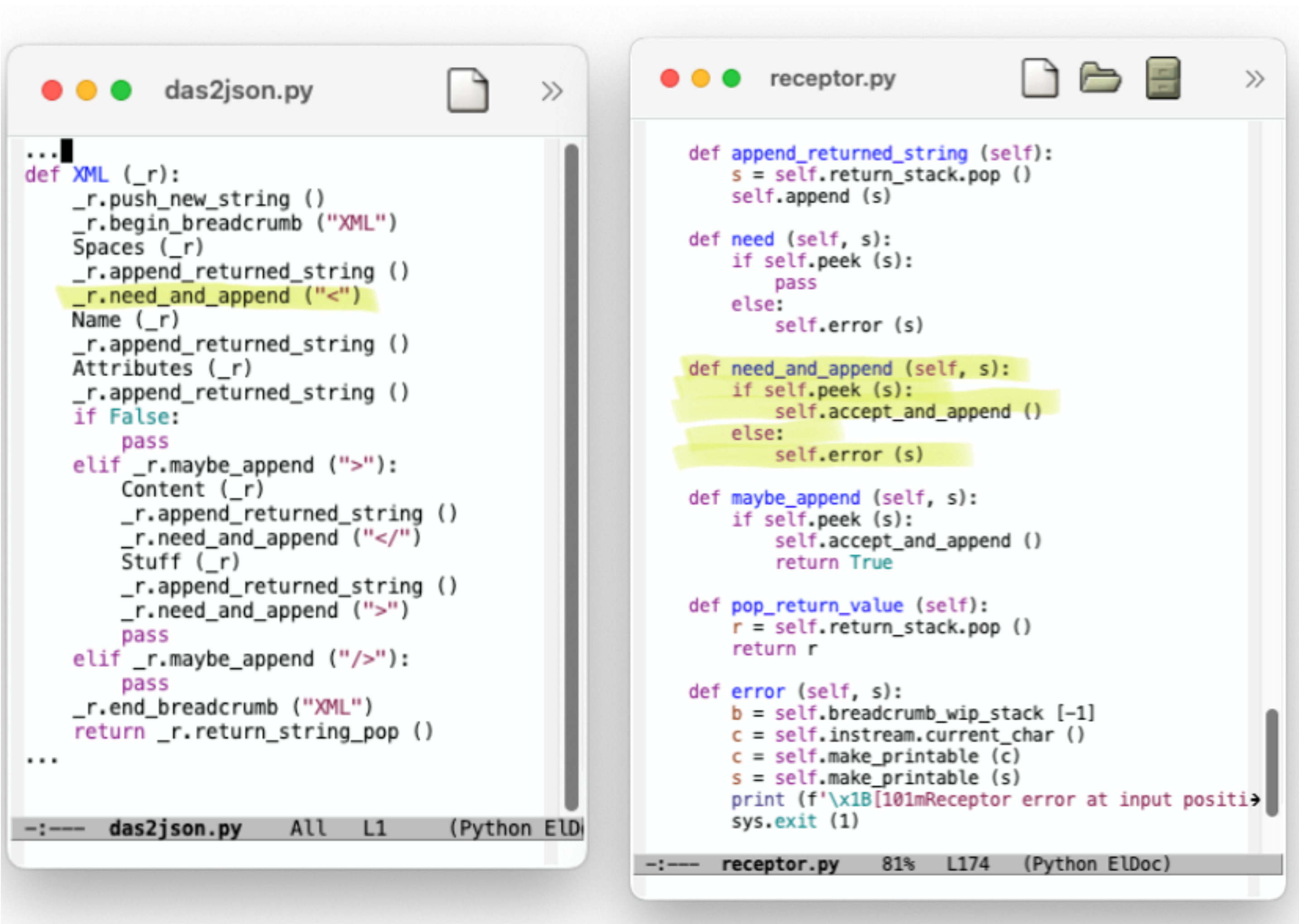
das2json.swib). The SCN is targeted *only* at the idea of what text needs to be matched. The niggly details required to do this kind of matching in a general purpose language - Python in this case - are ignored.

A *t2t* transpiler (essentially a simplistic compiler) is used to map the DSL into full-blown Python including all of the niggly details that were ignored by the “domain expert” when writing the specification in the SCN.

This is the basis of FDD - *Failure Driven Development.* The idea is that a programmer writes code that writes code. Keeping the SCN small makes it possible to regenerate all of the code at the push of a button. The effect is that the programmer can iterate and refine the design, knowing that changing code is easy. The programmer tweaks the code in the SCN, while the system (the development IDE) regenerates all of the code needed to

Generated Python code uses a library

Manually written Python library



das2json.py

```
...
def XML (_r):
    _r.push_new_string ()
    _r.begin_breadcrumb ("XML")
    Spaces (_r)
    _r.append_returned_string ()
    _r.need_and_append ("<")
    Name (_r)
    _r.append_returned_string ()
    Attributes (_r)
    _r.append_returned_string ()
    if False:
        pass
    elif _r.maybe_append (">"):
        Content (_r)
        _r.append_returned_string ()
        _r.need_and_append ("</")
        Stuff (_r)
        _r.append_returned_string ()
        _r.need_and_append (">")
        pass
    elif _r.maybe_append ("/>"):
        pass
    _r.end_breadcrumb ("XML")
    return _r.return_string_pop ()
...
-:---  das2json.py    All   L1     (Python ElD
```

receptor.py

```
    def append_returned_string (self):
        s = self.return_stack.pop ()
        self.append (s)

    def need (self, s):
        if self.peek (s):
            pass
        else:
            self.error (s)

    def need_and_append (self, s):
        if self.peek (s):
            self.accept_and_append ()
        else:
            self.error (s)

    def maybe_append (self, s):
        if self.peek (s):
            self.accept_and_append ()
            return True

    def pop_return_value (self):
        r = self.return_stack.pop ()
        return r

    def error (self, s):
        b = self.breadcrumb_wip_stack [-1]
        c = self.instream.current_char ()
        c = self.make_printable (c)
        s = self.make_printable (s)
        print (f'\x1B[101mReceptor error at input positi→
        sys.exit (1)
-:---  receptor.py    81%   L174    (Python ElDoc)
```

realize the full-blown application. This is the way that compilers work - programmers write programs in HLL syntax and the compiler converts the HLL into runnable assembler code. At the core, compilation is just *t2t* (text to text transpilation).

The rest of this essay contains a more thorough, wordy, sequential breakdown of this particular SCN and includes a reference to the github repository.

FDD is described in more detail in https://guitarvydas.github.io/2021/04/23/Failure-Driven-Design.html.

Description

The goal of `das2json.swib` is to specify a text parser using fewer lines of code than would be required when writing a parser in pure Python. In this newly-invented SCN, we can only talk about *parsing.* The SCN does not support all of the other general operations available in Python. I claim that it is easier to focus on a single problem at a time - in this case, *parsing* (aka "pattern matching") - and, avoid having to think about all of the other general features that are available in most modern General Purpose programming Languages - GPLs.

Most compiler-compilers use CFG-based technology that is language-oriented by default. This *swib receptor* parser, though, is parser-oriented and is just a DSL for specifying a *parser* (instead of specifying a *language*). The grammar, though, looks very similar to CFG-based approaches. This parser uses a recursive-descent strategy that allows it to use rules that can mutually invoke each other. This parser can parse matching pairs of brackets - like PEG-based parsers. CFG-based approaches can't do this. I claim that this kind of parsing is more powerful than CFG-based approaches, but, the differences are subtle.

I've chosen a snippet which contains one rule - the rule called "XML". The intent is to pattern-match basic XML. This rule is not stand-alone and invokes other rules. I don't bother to include or show the other rules. You can see them in the repository, if you want to look more deeply.

In this SCN, a "rule" generates a single string and returns it. Each rule starts out with a fresh empty string and fills the string in when it successfully matches patterns.

At the Python level, this is easy to implement. We make a *stack* of strings and push a new empty string onto the stack every time we dynamically enter a rule. When the rule finishes its work, it pops the string off of the *string* stack and pushes it onto a *return* stack. This SCN uses 2 stacks, not just one. Operation at the Python level is pretty basic. Any competent Python programmer should know how to do this kind of thing. There's no magic here. This SCN limits what you can do in Python and allows you to focus on *only* the task at hand.

```
: XML ^=
  Spaces "<" Name Attributes
  [
```

```
    | ">": Content "</" Stuff ">"
    | "/>":
  ]
```

To read this snippet, we start with `:`. This is like `def` but uses 2 fewer characters. The name of the rule comes next - in this case the rule is called `XML`. Next comes the definition operator `^=` which says that we will allocate a fresh, empty string for the rule, then fill it in, then return it. There is at least one other definition operator - `^@` - but, I'll skip discussing it here, for simplicity.

The next line specifies that 4 matches must succeed before proceeding with the 3rd line. First, we must invoke the sub-rule `Spaces`, then match the string `“<“`, then invoke another sub-rule `Name`, then invoke the sub-rule `Attributes`. Again, any competent Python programmer already knows how to do this, but this notation says it in fewer characters than the corresponding Python code. In Python, sub-rules are invoked by simply calling methods (`Space()`, Name`()`, and, `Attributes()`). String matching is done with `==` or a library routine. In our case, the library routines are found in the file https://github.com/guitarvydas/das2json/blob/main/receptor.py. The code in the library routines is plain-jane Python code, with the only nuance being that we make it possible to un-get characters by using a cache.

After this come a *choice* operation, lines 3-6. A *choice* is like a *case* statement that pattern matches strings against the input stream. The syntax of *choice* is a set of square brackets surrounding several choice legs. A choice leg - basically an `if...elif...` - operation begins with a “|” character followed by the string to be matched followed by a `:` and a sequence of matches if the choice-leg match succeeds.

If we see `“>”`, then we must invoke the sub-rule `Content`, followed by a match of `“</“`, followed by invocation of the sub-rule `Stuff`, followed by a match of `”>”`.

If, instead, we see `“/>”`, then we don't match for anything else.

This pattern should match inputs like

```
<mxCell id=123 style=“...”> <p> Hello </p> </mxCell>
```

And

```
<mxCell is=456 style="...”/>
```

Note that we allow matching multi-character strings, like `“/>”` instead of only matching for single-character strings like `“>”`. The idea of matching multi-character strings makes it necessary to un-get characters and necessitates using a cache. For example if we see `“/X”`, we match the `“/“` and then fail because `“X”` is not `“>”`. In this case, we need to un-get the `“/“` and back up to continue matching. This is just a low-level implementation detail - you don’t really need to know how multi-character strings are matched unless you want to understand the low-level Python code in the library.

As we execute that match, by default, all of the matching characters are appended to the allocated, fresh string associated with this rule invocation (one unique, fresh string is created *each* time a rule is invoked. For example, each time we call `XML()`, a fresh string is allocated on the string stack.

At the end of the rule, the modified string is “returned” to the calling rule. The string is popped off of the string stack and pushed onto the return stack[3].

That’s it. That specifies a parser for XML (along with the sub-rules that I didn’t bother to copy into this note, but, can be found in the repo).

What Python code is generated for the above snippet?

```
def XML (_r):
    _r.push_new_string ()
    _r.begin_breadcrumb ("XML")
    Spaces (_r)
    _r.append_returned_string ()
    _r.need_and_append ("<")
    Name (_r)
    _r.append_returned_string ()
    Attributes (_r)
    _r.append_returned_string ()
    if False:
        pass
    elif _r.maybe_append (">"):
        Content (_r)
```

[3] Could you do this with only 1 stack? Sure. But, the code would be more complicated. Make it correct first, then optimize it later.

```
        _r.append_returned_string ()
        _r.need_and_append ("</")
        Stuff (_r)
        _r.append_returned_string ()
        _r.need_and_append (">")
        pass
    elif _r.maybe_append ("/>"):
        pass
    _r.end_breadcrumb ("XML")
    return _r.return_string_pop ()
```

6 lines of SCN generate 24 lines of Python. The SCN allows you to focus on the problem - pattern-matching the input. The Python code conflates pattern-matching with niggly details and makes it harder to focus on the main task.

Note that the "breadcrumb" lines are tags that are automatically inserted by the SCN compiler to help with debugging.

The parsing operations, like `.need_and_append (">")`, call library routines found in `receptor.py`. These library routines don't do anything magical, but, they are used so frequently that I decided to put them into a library.

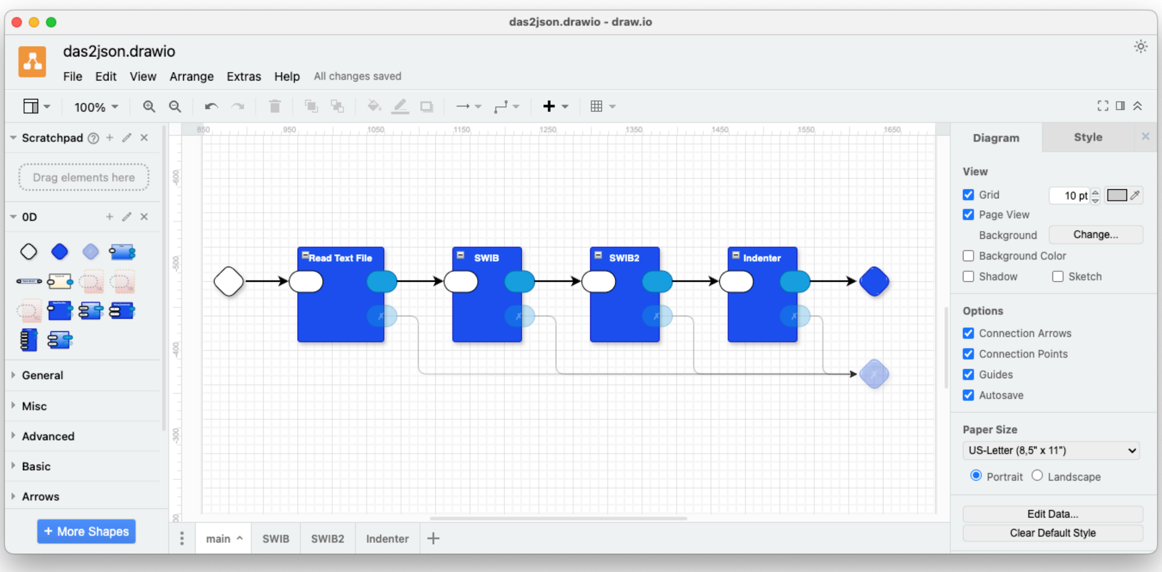
The code generator is written in OhmJS (grammar) and RWR (rewrite rules associated with OhmJS). The grammar is in file https://github.com/guitarvydas/das2json/blob/main/swib.ohm and the rewrite rules are in https://github.com/guitarvydas/das2json/blob/main/swib.rwr.

Note that, in this case, a trick is used to finish the rewrite[4]. I pass the rewritten code through yet another OhmJS+rwr combination to pull out the name of the main rule. The trick is in files https://github.com/guitarvydas/das2json/blob/main/defname.ohm and https://github.com/guitarvydas/das2json/blob/main/defname.rwr. I insert the Unicode brackets " ⎛ ... ⎠ " in the first phase - swib.rwr - to delineate the chunk of text that I want to re-parse. The rest of the text, which is not delineated, remains untouched. The rule "text" in `defname.ohm` handles incoming characters one at a time until a delineated chunk of text is found. The grammar breaks up the delineated chunk into several pieces and the .rwr rewriter uses only one of the pieces while throwing the rest away. I could have done this trick differently, but, I already had access to a parser and it was simpler to simply use the available parser black box than to code up some scheme that saved the

[4] see https://guitarvydas.github.io/2024/05/13/Deep-Dive-Into-Rewriting.html

main function name in some special location. The goal of parsing twice is to help the developer and the only efficiency consideration is whether the trick works “fast enough” for the developer on a development machine (which tends to be more powerful than end-user machines). If we wanted to ship this code as an end-user product, we might want to spend extra time to optimize the code, but, there is no reason to optimize this code if it is to be used only by developers and works “fast enough” for developers.

The pipeline for reading the das2json.swib file and generating Python code from it is pictured below



To allow the use of existing structured-text based tools (“{...}”), another trick is used. I generate code as if it is structure-based, using the brackets “(- ... -)”. At the very end, I run `indenter.js` to convert the structure-based code into legal indented Python code. I decided to use ASCII brackets “(- ... -)” instead of

Unicode brackets simply because my editor (emacs) knows how to indent parenthesis-based code, but, doesn’t deal with Unicode brackets (out of the box). In the early stages of exploring this technology, it made sense to eye-ball the code and this was made easier when it was indented by emacs.

Appendix - See Also

See Also

References *https://guitarvydas.github.io/2024/01/06/References.html*
Blog *https://guitarvydas.github.io/*
Blog *https://publish.obsidian.md/programmingsimplicity*
Videos *https://www.youtube.com/@programmingsimplicity2980*
[see playlist "programming simplicity"]
Discord *https://discord.gg/Jjx62ypR (Everyone welcome to join)*
X *(Twitter) @paul_tarvydas*
More writing *(WIP): https://leanpub.com/u/paul-tarvydas*