2024-05-22 Working Paper - Designing a Syntax for SWIB

Goal

The ultimate goal is to figure out some sort of syntax for receptors and to design an IDE that makes it easy to create Python programs using diagrams.

This work stems from the desire to convert diagrams drawn using draw.io into JSON[1] containing only semantically interesting information, while discarding all of the graphical layout information.

Why?

I believe that we need to break free of the function-based programming[2] mindset. Using functions to express computations is OK, whereas using the same notation, makes expression of some concepts more difficult.

Concurrency and state machines are concepts that become confusing when expressed in function-based form.

I draw inspiration from an older parsing language called S/SL[3], from a newer parsing language called OhmJS[4] and from Harel's StateCharts[5].

---

[1] A hand-built version of such a translator is in 0d/das2json/. The goal here is to re-build this converter app using software components ("0D").

[2] Function-based programming is the idea of using a written, function-based notation to express the operation of CPU-based electronic machines. This idea was calcified as early as 1954 in the FORTRAN programming language (and 1956 in Lisp). CPUs are step-wise sequencers, whereas mathematics is expressed in the form of continuous functions. Some of the capabilities of CPUs are lost when function-based notation is mapped onto the operation of CPU-based hardware. This is especially evident when dealing with concurrency. Functions can express the innards of single nodes in a network, but, functional notation is strained beyond its sweet spot when the notation is used to express the composition of networks containing multiple nodes.

[3] see https://guitarvydas.github.io/2024/01/06/References.html

[4] ohmjs.org

[5] see https://guitarvydas.github.io/2023/11/27/Statecharts-Papers-We-Love-Video.html

OhmJS *almost* does everything I need, except that it builds a full parse tree before calling any semantics code. At some point the input generates a parse tree that is too large to fit in memory and the process fails. The DSL proposed here does a *parse* and invokes code along the way, instead of building a parse tree automatically. This approach has the benefit that it can parse large input source in a streaming manner - on-the-fly - but, has the disadvantage that the process is irreversible by default, i.e. *undo* and *time-reversing* don't fall neatly out of this approach.

I believe in writing code that writes code. I want to have a *parser generator* not a *language specification*. In my opinion, PEG-based technologies are more powerful for building *parser generators* than CFG[6]-based technologies.

[6] Context Free Grammar - YACC and its descendants. Regular Expressions are a constrained form of CFGs (REGEXs lack a stack, no PDFA) and are, thus, even less suitable for this endeavour.

Composing Software using SoftWare Interlocking Blocks (SWIBs)

It is straight-forward to compose software if you use totally-isolated *black boxes.*

To achieve total isolation, data must be isolated *and* control flow must be isolated. Most existing programming languages, e.g. Python, Rust, etc., do not isolate both dimensions.

UNIX® command pipelines, and processes in operating systems, do isolate control flow, but, are generally thought to be very heavy weight.

When control flow is fully isolated, the system must use a *scheduler* to choose - randomly - which software unit to execute. Units cannot be written to rely on an expected order of execution. Units are not executed in a sequential order.

Programmers currently possess the technology to build systems this way, i.e. using *closures* and using the ability to create *queue* data structures*,* but, programmers tend to shy away from this form of composition due to their function-based, sequential mindset and the belief that processes are inherently heavy-weight.

In fact, *processes* in operating systems are just *closures* written in assembler and C[7].

[7] see Greenspun's 10 Rule.

Step 1 - A Parser for XML in Raw Python

As a first step in this process, I created a tentative syntax for the parsing DSL. It looks a lot like S/SL, with the added feature that it can inhale strings containing many characters.

I snipped the design parameters for the parser, such that it is meant only to work with strings - not general data structures or lists. In my experience, this simplification covers most use-cases for *t2t* transpilation. I've been using this simplification - strings only - for several years and haven't yet found it to be lacking in power.

Each *rule* in the DSL creates a string and returns it. Each *rule* starts out with an empty string. Characters are appended to the string as the parse progresses. There are 2 types of *rules* - (1) those which simply return the generated string "^=", and, (2) those which call a rewriting function on the generated string, "@=", returning the result of the rewrite. At present, only 2 rewrite operations are allowed - (1) return the string, (2) discard the string. In the future, I imagine that the rewrite rules with use the more general string construction and string interpolation found in RWR[8].

The parser uses a stack of strings, creating fresh (empty) strings every time a rule is dynamically called.

I manually built a parser for XML in Python[9].

Then, I wrote an OhmJS application that automatically converts the DSL syntax into Python code. I call this process *t2t,* for *text-to-text transpilation*.

The resulting app inhales a .drawio XML file and exhales the same without any changes, i.e. an *identity transform*. The output, itself, is uninteresting, but, the fact that the parser survives and works is interesting.

This work can be found in the repo https://github.com/guitarvydas/das2json on branch `main`. Running `make` from the command line does a *t2t* translation of

---

[8] It's likely that I haven't documented the RWR operations. Email me if you have difficulty understanding RWR rewrite rules and want to know.

[9] I would have used Common Lisp, but, I think that more programmers are frightened by CL.

`das2json.swib` converting it into a Python program called `das2json.py`. The t2t converter code is written in DPL[10] form in the file `das2json.drawio`. The output of that step is a Python program. `Make` then runs this Python program with input from the file `test.drawio`. The makefile rule `compileswib` creates the Python program and the rule `run` runs the python program.

The tentative syntax for the parsing DSL is in the file `das2json.swib`, where it is used to express the operation of an XML parser.

At this moment, the XML parser is specified as...

```
: Das2json @=
  _trace "@0"
  XML Spaces _end

: XML ^=
  Spaces "<" Name Attributes
  [
    | ">": Content "</" Stuff ">"
    | "/>":
  ]

: Content ^=
  <<<
    Spaces
    [*
      | "</": _break
      | "<mxGeometry ":  mxGeometry
      | "<": XML
      | *: Stuff
    ]
  >>>

: mxGeometry @= XML

: Attributes ^=
  <<<
    [*
      | "style=": Style
      | ">": _break
      | "/>": _break
      | _end: _break
      | *: .
    ]
  >>>
```

[10] Diagramming Programming Language

```
: Style @= "style=" String

: Name ^=
  <<<
    [*
      | " ":   _break
      | "\t": _break
      | "\n": _break
      | ">": _break
      | "<": _break
      | "/>": _break
      | _end: _break
      | *: .
    ]
  >>>

: Stuff ^=
  <<<
    [*
      | ">": _break
      | "<": _break
      | "/>": _break
      | _end: _break
      | *: .
    ]
  >>>

: Spaces ^=
  <<<
    [*
      | " ": .
      | "\t": .
      | "\n": .
      | *: _break
    ]
  >>>

: String ^=
  "\"" NotDquotes "\""

: NotDquotes ^=
  <<<
    [*
      | "\"": _break
      | *: .
    ]
  >>>

: EndMxCell @= "</mxCell>" Spaces
```

```
@ Das2json = _return_value
@ mxGeometry = _ignore_value
@ Style = _ignore_value
```

“_” is used as a prefix to builtin functions and DSL keywords.

The last 3 lines specify rewrite lines that are associated with “@=“ rules. At present, only 2 rewrite operations are supported - return the generated string, and ignore the generated string.

Step 2. Iterate, Design a New SWIB Receptor Syntax

```
⊢ Sampler ⊣

: Sampler ^=
  Stuff _end

: Stuff ^=
  <<<
    [*
      | "Hello World": Hello
      | _end: _break
      | *: .
    ]
  >>>

: Hello ^=
  "Hello World”
```

The first line is intended to specify a pipeline of parsers. This example pipeline contains only one parser (receptor) - Sampler.

`<<< ... >>>` specifies a *cycle* which can be exit with an `_break` operation.

[* | ... | ... |*:] specifies a look ahead choice (peek), each branch beginning with “|” and a string to be matched, followed by “:” and a sequence of matching operations. The “else” branch contains the symbol “*” instead of a lookahead string.

The first branch which matches is fired. If no branch is matched, an error is signalled.

A matching operation can be:

- A string to be matched exactly. Once matched, it is appended to the generated string of the enclosing rule.
- A rule name, to be called. The returned string is appended to the generated string of the calling rule.
- “.” To accept any character and to append it to the generated string.
- “_end” which succeeds only if the end of the input stream has been reached.

Step 3. Iterate the Design Some More

I'm trying to understand what primitive operations are needed and where blocking might occur, and, whether it would be easier to implement the pattern-matching engine using features in existing languages (e.g. exceptions in Python), or to bite the bullet and write a VM (Virtual Machine). The necessary features exist in assembler, but, most modern languages restrict access to these features.

One thing is certain: the parser engine is a state machine that blocks, waiting for I/O, in very specific places. Because the blocking happens in predictable places in the code, it should be possible to make the engine be "more efficient" than modern languages that use operating systems and threads and suffer ad-hoc blocking. Regardless, this kind of blocking happens in modern languages, but, is papered-over by operating systems[11].

I didn't keep every iteration, but, here's one:

```
"Sampler", [
    push_new_string,
    enter, "Sampler",
    call, "Stuff"
    append_return,
    exit, "Sampler",
    return_pop
]

"Stuff", [
    push_new_string,
    enter, "Stuff",
    begin_cycle,
    mark,
    peek, "Hello World",
    ?, [
        call, "Hello",
        append_return,
        continue,
        mark,
        peek_eof,
        ?, [
            break
            end_cycle,
```

[11] Note that operating system documentation tends to describe the lifetime of process in terms of state machines (idle, running, blocked, etc.). What is being discussed here is nothing new, but, only a different way of looking at the same problems.

```
            exit, "Stuff",
            return_pop
        ]]]

"Hello", [
    push_new_string,
    enter, "Hello",
    expect_and_append, "Hello World",
    ?, [
        exit, "Hello",
        return_pop
        ]]
```

“?” marks the spots where blocking might occur. The bytecodes following the “?” are enclosed in [...], which is a way of specifying *continuations* of code that run after a block sequence is resumed.

My question, here, is whether blocking should be reflected in the DSL syntax, or if we say that every operation with the prefix “peek” can block. Or ...?

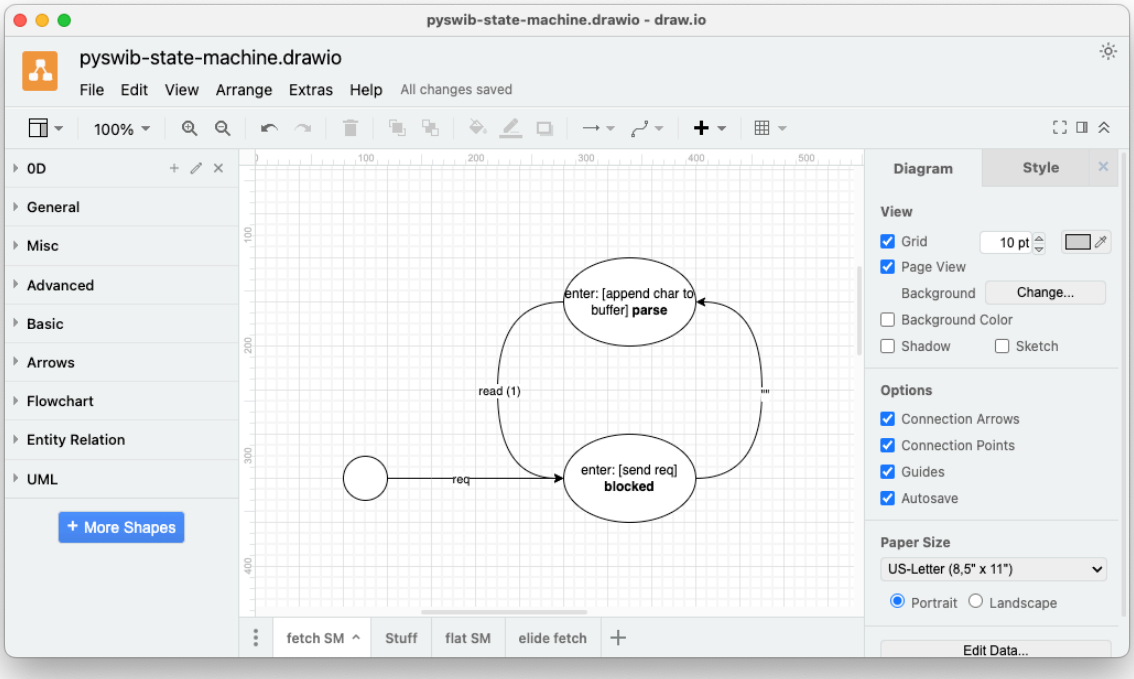
I’m not sure that I like either of the above two choices - explicit operator “?” or implicit operations with a special prefix.

Drawing diagrams of the above might help understand the Design issues... On to step 4.

Step 4. Iterate Again, This Time Drawing Diagrams of the State Machines

I want to further ponder this design space. I start by making drawings.

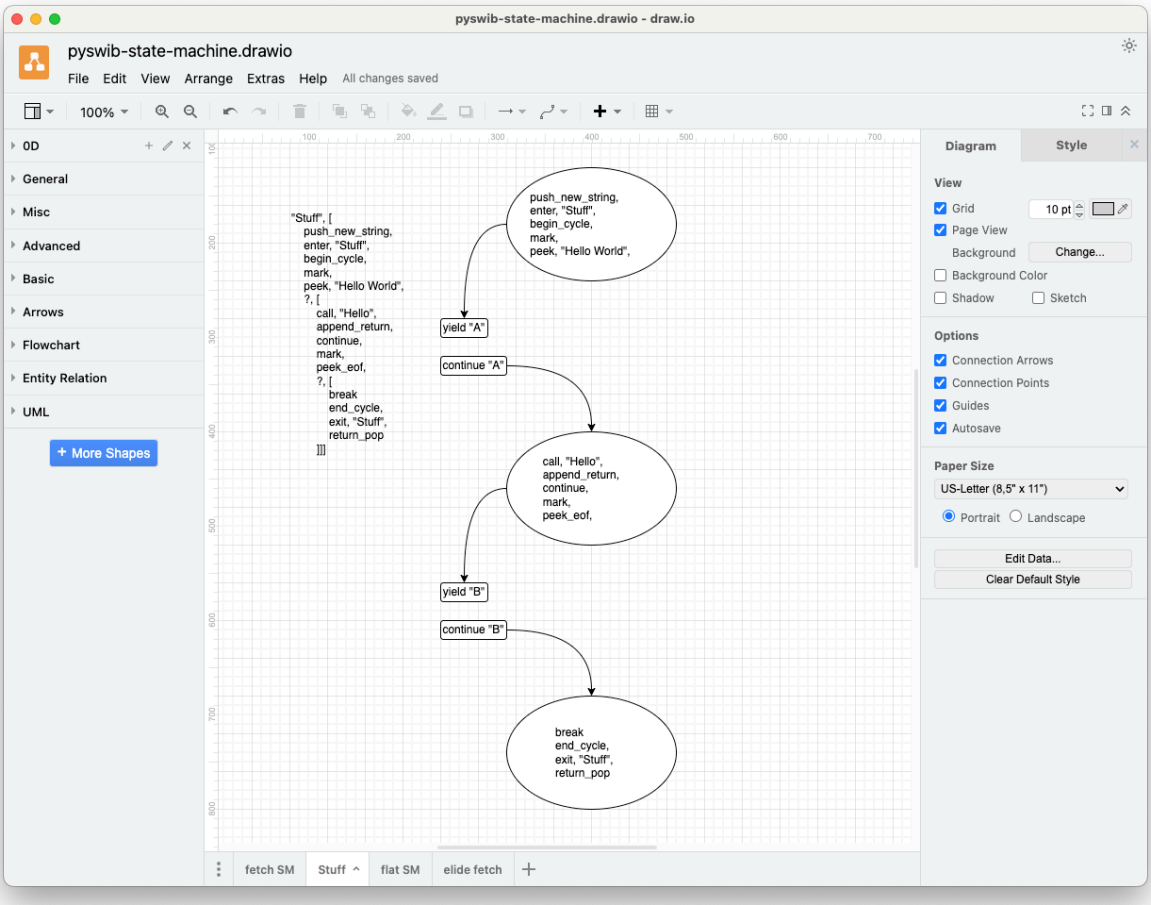
Here is the most simple state machine representing blocking I/O in this system:



Question: what “state” do I need to save in order to resume the parser engine after I/O has been received?

The problem, here, is that this blocking happens deep in the bowels of the code and requires different resumption continuations at each point.

Next, I try labelling each “yield” with some unique identifier to help figure out which continuation needs to be resumed:

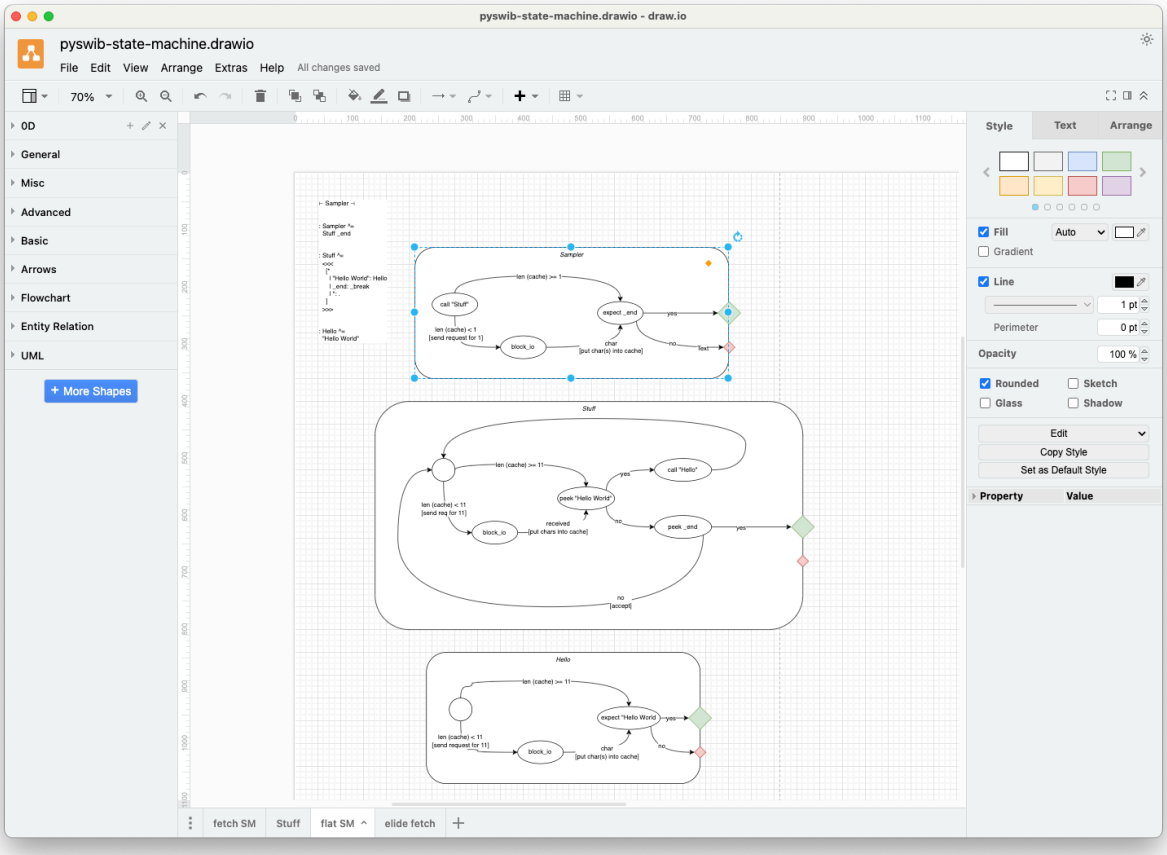


Here, I make the revelation that “yield” is only called if I/O blocking is required. If the necessary characters are already in the cache, then we can continue without blocking.

I had drawn explicit transitions out of each state directly to the next state, to show non-blocking transitions. I erased those transitions when I realized that the non-blocking condition could be faked by inserting a “continue” message at the front of the queue, and calling the message handler for the engine.

I’m not sure that this is such a good idea. In general being explicit usually wins out.

Next, I draw out the state machine in full glory, as a flat machine:

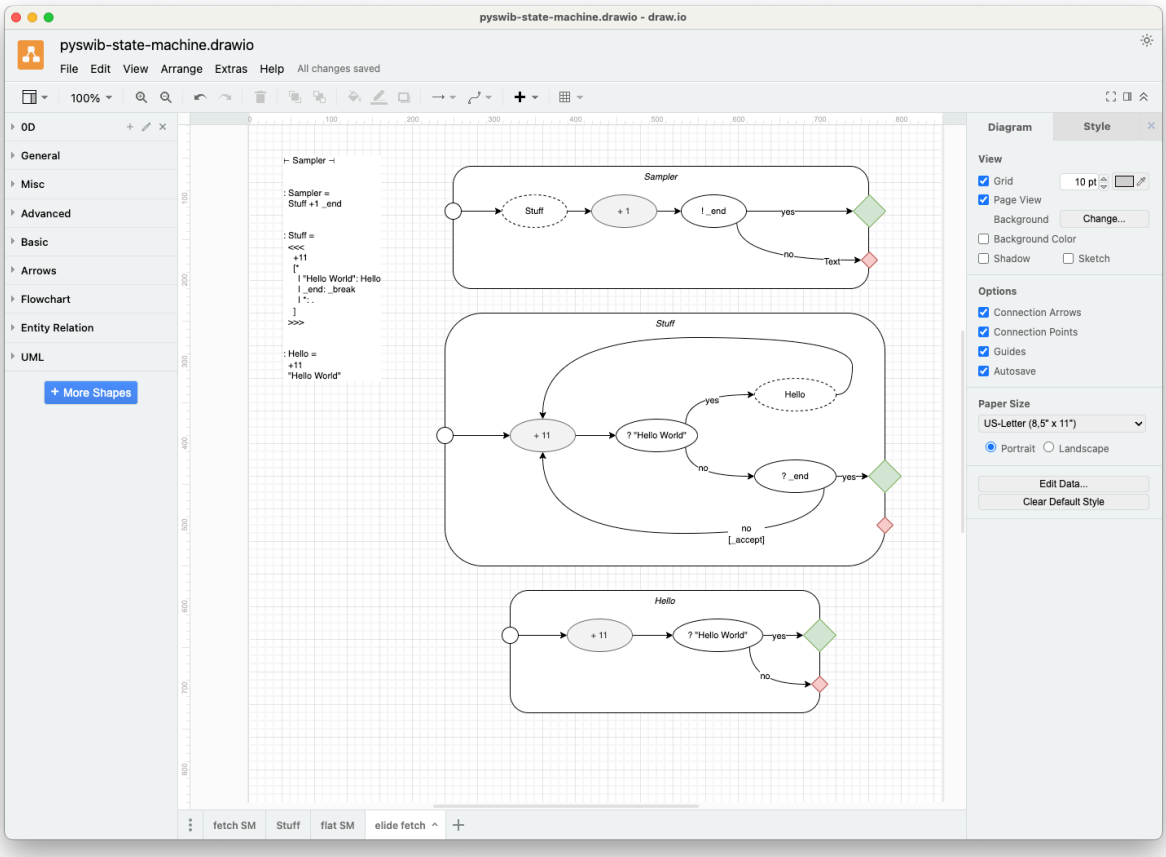


This diagram looks wildly more complicated and less succinct than the DSL code written out as text, but, it does show some interesting patterns.

In all three boxes, the blocking operation and the state transitions due to blocking take up the most real-estate, but, the pattern is essentially repeated in each of the 3 cases.

Can we visually elide the blocking state and transitions?

This next diagram attempts to elide the blocking operations:



This suggests and incorporates a couple of twists:

- The gray bubble represents elided blocking. The number in the bubble represents how many characters will be needed (fewer if end-of-input is reached).

- Only the 1st and 3rd boxes can error out. The 2nd box will never error out. This isn't obvious - to me - from the textual description at the left.

- The diagrams look a lot less busy. I'm beginning to like them more than the textual representation. The diagrams show a lot of details that aren't explicitly shown in the textual version. I think that that is a good thing.

- An "optimization" is apparent. We can ask for multiple characters at once and avoid asking for them one-by-one.

- The boxes look like they are pointing towards the kind of information we would need to store - on a stack - in order to be able to resume the engine after blocking.
- The textual representation of cycle `<<<...>>>` disappears and gets handled by state transitions. Less syntax. Is this good, or will it become clunky when used on larger examples? [Further testing of the ideas using larger and larger inputs will help determine the answer].
- The top-most diagram looks plug-able. We will see when we get to a use-case that needs pipelines.
- Even if I decide on a text-only syntax for this stuff, diagramming the problem space has helped me see aspects that weren't apparent.

--- I'm going to stop here and try to document my progress up to this point. More pondering is still required, but, I like the direction this is heading in. ---

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*