2024-05-07 Design Notes for Das2json

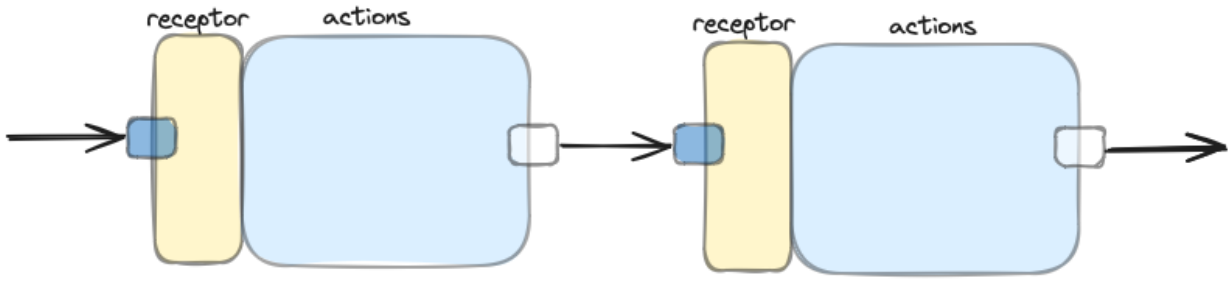
Das2JSON in 0D

Let's rewrite das2json in 0D (using Python) and get rid of the requirement for Odin.

SWIBs and Receptors

We'll use SWIBs - SoftWare Interlocking Blocks.

The general gist of SWIBs is that of forming pipeline, as show in the figure below



A *receptor* is essentially a stream-based *parser.* A receptor accepts incoming Unicode characters one-by-one and executes *actions* based on the path specified by the parse. If the incoming characters don't match any pattern specified in the .swib file, a *syntax error* is displayed and the receptor stops, or, tries to continue parsing the rest of the input. In either case, no further messages are sent on the output pin. This is not shown on the diagram above for sake of simplicity. Each *swib* has at least one[1] extra output pin which corresponds to some kind of error message(s).

---

[1] SWIBs can have many outputs - and, many inputs. You are not constrained to having only one input and two outputs. In practical projects, there might be many *happy paths* through the code and each path may wish to produce outputs and consume inputs on customized ports. Ports can be named, for exactly this kind of usage. The most common ports usually have empty strings as their names (this is like *stdin* and *stdout* in UNIX®). We'll skip over this kind of nuance in this essay, to keep things simple.

Receptors are very similar to *parsers,* except that receptors work on streaming input instead of slurping a whole file in and making a parse tree. Receptors don't create internal parse trees by default, and, simply call *action* code during the parsing operation. This allows *receptors* to handle inputs (and files) of any size, but, does not automatically allow for undoing or reversing the flow of operations, unless care is taken in the design of the .swib file and the actions that it calls.

An analogy for receptor-based *swibs* is that of RNA in biology. RNA walks down the strands of DNA and duplicates the DNA strand as the walk proceeds. A swib receptor does a similar walk, but is not restricted to making a duplicate of its input - it simply invokes *actions* as the walk proceeds.

Back to Das2json...

To implement *das2json* we need to write a simple *swib* pipeline. The pipeline might need only one *swib*. We'll see...

[*I'm building and writing this on-the-fly. The code I present might contain errors. I will fix errors in the next instalment of these essays. I believe in developing ideas in an incremental manner and I don't fear making mistakes - I just back up and fix things as I learn more about the problem space.*]

My preferred approach is to make and test a *swib* for *identity* parsing, then, to hack on it to produce the desired output. In this case, I want to write a receptor that understands the .drawio format and then spits out .json for only the semantically interesting bits contained in the .drawio file, while discarding the semantically uninteresting stuff, like graphics layout details.

A .drawio file is generated by the draw.io[2] diagram editor program. If we can manage to grok the .drawio file and to convert it to Python, then we will be able to grok DPLs (Diagrammatic Programming Languages) drawn in the draw.io editor, i.e. we will be able to compile diagrams to executable code. In this case, we'll use the Python language as the "assembler" for running DPL programs.

We will do all of this in two big steps:

---

[2] https://app.diagrams.net/

1. Draw a DPL program and convert it to .json. This is the job of *das2json* which is the topic of this essay.

2. Slurp the generated .json into some kind of interpreter and run it. This has already been done in the Odin and the Python programming languages. This part will not be discussed in this essay, to keep things simple. You can find the code for this part in the 0D repository[3]. Alternately, we can slurp the .json into a compiler and generate code in some language, then use a compiler(s) for that language to compile the code into executable form. I won't discuss this option in this essay, either, to keep things simple.

So, back to the single task of implementing (1) *das2json...* First, we'll write a *swib* to grok .drawio files and output the same stuff as was input, i.e. the identity transform. This will get us over the big hump in development - designing, implementing and testing the receptor for *das2json.* Later we'll add more interesting actions to the receptor to generate .json for the semantically interesting stuff and to discard / ignore the semantically uninteresting stuff.

So, we begin by figuring out what's in a .drawio file and figuring out how to grok it.

Drawio files

A .drawio file has a header, followed by 1 or more diagrams. Everything in a .drawio file is in XML text format.

Here's the basic *receptor* specification for .drawio files:

```
: Das2json =
  XML Spaces _end

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

: Content =
  <<<
```

[3] https://github.com/guitarvydas/0D

```
    Spaces
    [*
      | "</": _break
      | "<": XML
      | *: Stuff
    ]
  >>>


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

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

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

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

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

This says:

1. To grok a .drawio file, first read it in and grok it as XML, then allow some trailing whitespaces ("spaces") and then expect to see the end-of-file ("_end").

2. The interesting stuff happens in the XML rule and the sub-rules that it invokes. The XML rule says, first allow some leading whitespace, then expect to see a `<` character, then expect to see some more characters as long as they are not the characters[4] `>`, or `<`, or `/>`, or the end-of-file. Next, expect one of `>`, or `/>`, or anything else. If we see `>`, then expect some content, followed by `</`, some more stuff, then `>`. This corresponds to a full-blown XML *element* with content. If, instead, we see `/>`, then we are seeing an *element* with no content and we expect to see nothing else after this point. Finally, at this point, nothing else is legal, so a "syntax error" is raised if we don't see one of the two expected inputs (`>` or `/>`). The syntax error does not need to be explicitly written by the programmer. The underlying swib engine / library takes care of pattern matching and of declaring syntax errors when the pattern match fails.

The rule *Content* says to accept any characters until either `<` or `</` are seen.
1. If `<` is seen, then parse another XML element.
2. If `</` is seen, stop (_break). Stopping will cause the rule to come back up for air, letting the calling rule decide what to do next.
3. Otherwise, just ingest the character and move on, looking for more characters.

Note that the brackets `<<<...>>>` mean to repeat the matching over and over again until `_break` is encountered. This is a "cycle" which only stops cycling when "_break" is encountered.

---

[4] The receptor library allows you to specify more than one expected character at a time. The characters are matched in sequence and the match succeeds only if all characters are matched in the exact sequence specified in the string. For example `/>` is a two-character pattern that matches exactly `/>`. If, say, the input is `/X`, the matcher matches the first character `/`, but, doesn't match the second character `X`, and, thus, fails the match.

The rule *Attributes* is similar, except that it checks for the termination of an attribute list with either `>` or `/>`.

The rule *Stuff* just slurps up characters until one of `>`, `<`, `/>`, or `_end` are encountered. Basically, this rule is intended for accepting any characters until something actionable is encountered. The rule stops when something actionable is encountered. When the *Stuff* rule stops, it effectively punts the work back to the calling rule. The calling rule determines what to expect to see next. In this case, only two possibilities are legal:

1. Expect to see the beginning of an element or
2. Expect to see the ending of an element.

Everything else results in a syntax error.

The rule *Spaces* skips over any space, tab and newline encountered in the input at the current point in the parse. We don't want to deal with any of these whitespace characters explicitly in each of the above rules, so, we let this rule perform the task of skipping spaces.

Note that the syntax of .swib specifications might be further reduced to be even more readable. At present, I want to be very explicit, so I'm using the syntax "as is" without worrying about how to make it more readable.

Implementing the .swib Specification

The goal is to convert the .swib spec into a Python (.py) file and to run the Python.

At first, I simply hand-compiled the .swib spec into Python code and ran it ("make manual-das2son").

After I debugged the hand-written code, I had a pretty good idea of what an automatic compiler should do. Then, I wrote a compiler using 0D, which boils down into calling OhmJS

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*