

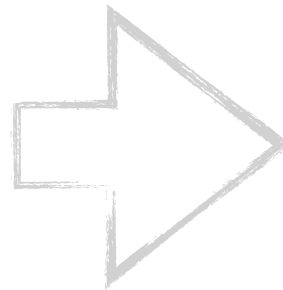
# 2024-05-26-Mapping a DSL onto Python in Sketches

Goal

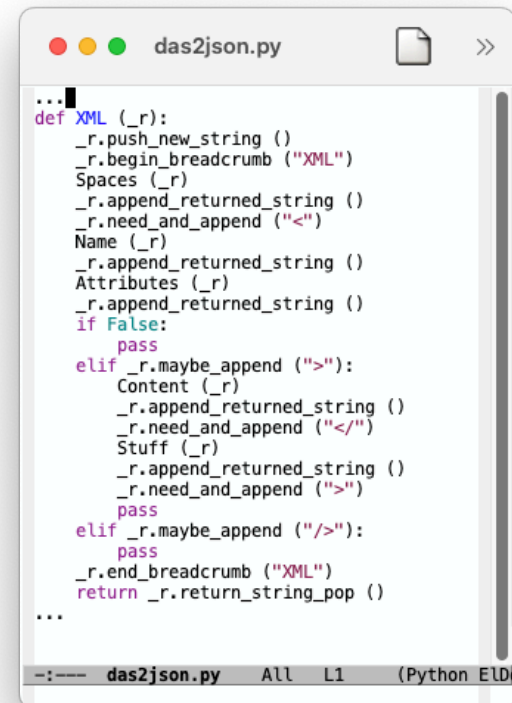
DSL (das2json.swib)



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



Python code (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 ()
...
```

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 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).

A more thorough, wordy, sequential breakdown of this particular SCN can be found in <https://guitarvydas.github.io/2024/05/12/OhmJS-Example-Use-Case.html>. The essay contains references to the github repository for the WIP code for this project.

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

```
...
: XML ^=
  Spaces "<" Name Attributes
  [
  | ">": Content "</" Stuff ">"
  | "/>":
  ]
...
-:--- das2json.swib All L9 (Fun
```

```
...
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
```

Generated Python code uses a library

```
...
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)

Manually written Python library

```
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 position {self.instream.current_pos}
    sys.exit (1)
```

—:— receptor.py 81% L174 (Python ELDoc)

## 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>