2024-05-13-Deep Dive Into Rewriting

In this note, I will dissect the OhmJS file https://github.com/guitarvydas/das2json/blob/main/defname.ohm and its corresponding action (semantics) file https://github.com/guitarvydas/das2json/blob/main/defname.rwr.

This is only a small piece of a total solution, described in “2024-05-12-OhmJS Example Use-Case”. This is the second, re-parse stage of the pipeline described in that note.

I choose to describe this grammar because it is very short - 8 lines - whereas the main grammar is 54 lines long. The main grammar is the meat of this project, but, is harder to describe in words without being overly boring and TL;DR. The code for the main grammar is in https://github.com/guitarvydas/das2json/blob/main/swib.ohm. I hope that, after you can understand this 8-line example, you can understand the main grammar without further help.

The tiny grammar in question is

```
defname {
  main = text+
  text =
    | defName -- match
    | any -- other
  defName = "⌈" spaces "def" spaces name " (" spaces through<"⌋">
  through<s> = (~s any)+  s
  name = letter alnum* ~alnum
}
```

And the corresponding action rules are:

```
defname {
  main [x+] = ‘«x»’
  text_match [x] = ‘«x»’
  text_other [x] = ‘«x»’
  defName  [lb spaces1 _def spaces2 name lb2 spaces3 misc] = ‘«name»’
  through [misc s] = ‘’
  name [letter alnum*] = ‘«letter»«alnum»’
}
```

Grammar .ohm

Let's begin by looking only at the grammar in question. The purpose of the action rules will be easier to understand once you understand the grammar.

The goal of the re-parser is to grab a chunk of code like the following

```
⟨
def Das2json (_r):(-
    _r.push_new_string ()
    _r.begin_breadcrumb ("Das2json")
    _r.trace ("@0")
XML (_r)
_r.append_returned_string ()
Spaces (_r)
_r.append_returned_string ()
_r.eof ()
_r.end_breadcrumb ("Das2json")
    return Das2json__action__ (_r)-)
⟩
```

and to peel out the function name `Das2json`, discarding the rest. Anything that doesn't match this pattern, delineated by the Unicode brackets " ⟨ ... ⟩ " needs to be just passed through, remaining untouched. The main parser does most of the work, but marks off a chunk of text from which it needs the function name. I could pull out the function name using some other technique, like a REGEX or using Javascript, but, this technique - re-parsing - is dead easy and much easier to write correctly. This technique is "less efficient", but, who cares? The machine does the work. The machine doesn't care. I don't care as long as this re-parse runs "fast enough". If it runs "fast enough", I don't need to waste time making it run faster. I can get on with thinking about more interesting things.

Back to the grammar

The whole grammar is wrapped in ASCII brace brackets and is preceded by a name for the grammar - in this case "defname". OhmJS can use the name of the grammar as one of its initialization options. In this example, the initialization is already handled and we don't need to worry about this feature. You can look at the OhmJS documentation if you want to dig further.

The next line

```
  main = text+
```
Is the main rule. The first rule is taken, by default, to be the default. I used the name “main”, but, I could have used any name.

This rule says that “main” is 1 or more “text” thingies. `+` means one-or-more. `*` mean zero-or-more, and `?` means either zero-of or one-of.

“text” is another rule that can be found in this grammar. Note that all of the rules in this grammar begin with a lower-case letter. The case of the first letter is significant to OhmJS. Lower case means “don’t skip spaces or anything” while upper case means “delete all spaces/newlines/tabs” before trying to match the rule. In this tiny grammar, we want to keep everything intact and can’t afford to skip spaces, so we use lower case rule names. This means that I have to explicitly mark places where spaces need to be matched. In larger grammars, details about spaces becomes noisy and TL;DR, so automated space-skipping can be a useful way to preserve readability of the grammar.

Next, we see the “text” rule:

```
  text =
    | defName -- match
    | any -- other
```

This rule says, “try to match a *defName* thingie, if that fails just match anything (a single character)”. Each branch of the rule must have a unique name, which is specified by the “--” operator. Such names are used in the action code, which we’ll see later. The nuance of branches is fully described by PEG theory - which you don’t need to know in order to just use the grammar.

The next rule is the rule for matching a *defName* thingie. A *defName* is delineated by unicode brackets

```
  defName = "⁅" spaces "def" spaces name " (" spaces through<"⁆">
```

The rule specifies a sequence of 8 things that need to be matched. If OhmJS can’t match all 8, then it gives up and backtracks. In this case, the backtracking leads back to the first branch of the *text* rule. That branch fails, so OhmJS tries

the next rule branch, which happens to be *any,* which matches just about anything, so success on at least one of the branches is guaranteed.

The interesting part of the *defName* rule is the final match that uses an OhmJS parameterized rule

```
through<")">
```

I wrote the rule *through* so that it matches anything and everything until a closing bracket `")"` is encountered and consumed. You can't do this kind of thing using a CFG-based grammar generator. You have to use recursive descent and/or PEG and/or OhmJS to match for this kind of pattern.

The *through* rule is deceptively simple:

```
  through<s> = (~s any)+  s
```

It says, "match anything 1-or-more times, except the parameter *s*". When an *s* is encountered, stop and move on to the next pattern to be matched. The next pattern happens to be an *s*. So, we consume anything except *s*, then we consume an *s.* This rule matches any characters up to (and including) the pattern specified by the parameter *s*.

This is what we want, here. This set of rules matches a *def* line, then skips to the closing bracket `")"`. The effect is that we match a whole Python function definition and break down the *def* line into separate pieces, while matching the rest of the function as one big blob of text. You can't say this in REGEX nor in a CFG-based parser. I could have written this to be more recursive, but, I didn't. If it were possible to have a delineated blob of text nested inside another delineated blob of text, then I would need to rewrite the rule recursively. In this case, I know that doesn't happen, so writing the rule this way is OK.

This kind of skipping over uninteresting detail is the power of PEG, recursive descent, and, OhmJS. The grammar for *defname* is only 8 lines long. Using a CFG based approach, I would have needed to write out - in full - the grammar for all of the stuff that I wanted to skip over. That's a lot of unnecessary work and would have turned this into a mega-project, which would have killed my desire to solve

the problem this way. If I had tried to write this same thing using REGEX, I would have gotten close and then would have to debug and rewrite the pattern many times - REGEX doesn't support rule-calling using a callstack, so I would have needed to think up some tricky work-arounds. Again, this would have become a mega-project and would have killed my desire to go down this path.

The final line of the grammar

```
  name = letter alnum* ~alnum
```

is a mundane specification of what a legal ID is in Python - a letter following by more letters/numbers followed by a non-letter/number. I didn't bother to specify a match for the underscore "_" character, because I wrote the main grammar/rwr stuff and I know that the function that I want to re-parse will never have the underscore character in its name. If we wanted to handle the case of names containing underscores, we'd have to rewrite the *name* rule with a little more nuance. Not rocket science, just much more nuance than I want to explain in this essay.

Rewrite .rwr

Now, let's look at the rewriting specification - the actions

The action rules are:

```
defname {
  main [x+] = '«x»'
  text_match [x] = '«x»'
  text_other [x] = '«x»'
  defName  [lb spaces1 _def spaces2 name lb2 spaces3 misc] = '«name»'
  through [misc s] = ''
  name [letter alnum*] = '«letter»«alnum»'
}
```

Firstly, note that the rewrite spec has the same name and enclosing brace brackets as the grammar has.

Next, we see that there is exactly one rewrite rule for every grammar rule. In the case of multi-branch grammar rules, each branch has a separate rewrite rule.

For example, `text_match` in the rewrite spec corresponds to the grammar rule

```
  text =
    | defName -- match
```

Each rewrite rule has a list of parameters enclosed in square brackets - one parameter for each sub-match specified in the corresponding grammar rule. We are free to choose any name for each parameter, as long as the name is legal in the JavaScript language[1]. Each parameter must have the same suffix (`+`/`*`/`?`) as is found in the corresponding grammar rule[2].

The right-hand side of each rewrite rule is a rewrite string enclosed in Unicode quotes '...'.

---

[1] Note - Javascript, not the Python language.

[2] Ideally, the compiler should check that the suffixes match up, but, for this bootstrap version, I don't do the checking. Furthermore, ideally, the compiler should figure out what the suffixes are, without bothering the programmer with such niggly details. The fact that the compiler doesn't do the checking, yet, means that you can get strange-looking output if you make an error. In many cases, if you see too many commas "`,`", it probably means that you forgot to specify a suffix or got it wrong.

The rewrite string can contain only 2 kinds of things:

1. Raw characters
2. String interpolations

Raw characters are simply copied into the result string.

Interpolations execute some Javascript function and copy the return value - always a string - into the result string.

String interpolations are enclosed in Unicode brackets `«...»`. The "..." can be any valid Javascript expression, but, is usually just the name of a rule parameter. Some time in the future, I might tighten up the definition of what is legal inside a string interpolation, to not rely on Javascript, but, not yet.

For example

```
  text_match [x] = '«x»'
```

says that, if a `text -- match` thingie is matched, the result of the rewrite rule is just a string with the value (a string) of the match stuck in it.

The *defName* rewrite rule is more interesting:

```
  defName  [lb spaces1 _def spaces2 name lb2 spaces3 misc] = '«name»'
```

This says, if a *defName* thingie is matched, break it down into 8 pieces and pass each of the pieces into the rewrite rule as parameters with unique names. The rewrite, itself, says to return a string that contains only the *name* parameter. All of the other 7 parameters are simply ignored and discarded. This is the essence of what we want this parser to do - peel off the function name of the delineated blob of incoming text. Easy peasy - inhale a blob, exhale a trimmed-down blob.

More interesting examples of rewriting can be found in more elaborate .rwr specs, for example in https://github.com/guitarvydas/das2json/blob/main/swib.rwr, where we stick raw Python code into the rewrites. I won't describe that here, under the assumption that the .rwr spec is clear enough to be understood on its own[3].

[3] Contact me if you don't understand some more elaborate rewrite rule.

In the end, the task of compiling / transpiling the `das2json.swib` to Python consists of hacking on the text of the `.swib` file until the output looks like legal Python code. Most of that work is done in the main rewrite - `swib.rwr`. I don't discuss the bulk of that rewrite here, only a supplemental rewrite.

In some (rare) cases, it becomes necessary to use some support code - written in Javascript - to help with the rewrite. For this purpose, the rewrite component - called *Transpile* - has an extra input for support functions written in Javascript. In most cases, though, we simply pour `null.js` into that input, i.e. in most cases we don't need to lean on support code. Again, I won't bother to explain this nuance here, on the assumption that its use will become apparent when studying successively larger examples.

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*