Programming Simplicity

Teasers

Paul Tarvydas

Programming Simplicity Teasers

Paul Tarvydas

This book is for sale at
http://leanpub.com/programmingsimplicityteasers

This version was published on 2023-04-28

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2023 Paul Tarvydas

Table of Contents

Simple Example of a Transpiler Using Ohm-JS

Goal:

To transpile `a = b + c` into JavaScript.

Then, to hack the transpiler to emit Python.

Future: hack the transpiler to emit WASM, Rust, etc., etc.

The point of this note is to demonstrate the use of Ohm-JS for writing transpilers (string -> string transformers) using a *very* simple example.

Code in Ohm-JS

Here's some code, followed by a more-detailed reading...

```
var ohm = require ('ohm-js');

const g = ohm.grammar (`
verysimple {
  Top = letter "=" letter "+" letter
}
`);
const semantics = g.createSemantics ().addOperation ("js"\
, {
    Top (target, eq, v1, plus, v2) {
    return `
${target.js()} = ${v1.js()} + ${v2.js()};
```

```
13  console.log (${target.js()});
14  `},
15      _terminal () { return this.sourceString; }
16  });
17
18  var cst = g.match ("a = b + c");
19  var transpiled = semantics (cst).js ();
20  console.log (transpiled);
```

We call `ohm.grammar` with a string that represents the simple grammar, following Ohm-JS' rules for forming grammars. See the Ohm-JS documentation if you want to dig deeper into what can be done.

The call to `ohm.grammar` starts up the Ohm pattern-matching engine. If the pattern-match succeeds, the engine calls functions listed below in the `semantics` structure.

These functions perform actions on the bits that matched. This is kinda like the way that REGEX operates. In REGEX, you write a pattern in REGEX syntax, and you write an action in REGEX syntax. In Ohm-JS, you write a pattern in Ohm's *grammar* syntax, and you write actions in JavaScript. Of course, you can escape having to write JavaScript code by writing a small DSL that accepts sub-matches and performs actions on them. How do you write such a DSL? Using Ohm-JS, of course.

The JavaScript actions are grouped into grab-bags. You can write more than one set of actions for any given grammar. In Ohm-JS, the grab-bags are given names and follow the rules of JavaScript namespace objects.

There must be one action for each rule in the corresponding grammar (there are exceptions to this rule - see the Ohm-JS documentation). Each action must have *exactly* the same name as its corresponding grammar rule, and, it must have *exactly* one parameter for each sub-match used in the corresponding rule.

For example, in the above code, the main grammar rule is called

"Top" which contains five (5) sub-matches

1. The first sub-match is *letter* (built into Ohm-JS),
2. The character "="
3. Another *letter*
4. The character "+"
5. And, another *letter*

(Yes, you can specify much more interesting patterns. This is a *very* simple example).

The line of code `var cst = g.match ("a = b + c");` starts up the pattern matcher and gives it the source string `"a = b + c"` for matching.

If the pattern match succeeds, the Ohm engine calls the JavaScipt function `Top(...)` and associates the five parameters with match information:

1. *target* is associated with the match `"a"`
2. "=" is given the name *eq*
3. "b" is given the name *v1*
4. "+" is given the name *plus*
5. "c" is given the name *v2*.

More stuff happens under the covers. The Ohm-JS engine sends more information into each parameter, but, we'll ignore those details for now.

The line of code `const semantics = g.createSemantics ().addOperation ("js", {...` sets up a correspondence between the Ohm grammar rule "Top" and various JavaScript functions. In this example, I chose to use the name "js" as a tag for the set of functions that correspond to the grammar. Ohm calls this an *operation*.

The action code applies the *operation* ("js" in this example) to the sub-matches. For example `v1.js()` invokes the "js" operation on the sub-match "v1".

To understand what the operations do, you have to read the JavaScript code.

In this *very* simple example, I chose to play only with JavaScript strings using JavaScript's *template string* syntax. A *template string* is a string delimited by back-quotes. A *template string* can contain JavaScript commands when they are surrounded by the syntax `${...}`. For example, the snippet `${v1.js()}` means to apply the "js" operation on the `v1` sub-match. The way that I've written the JavaScript code, this operation simply pulls out the substring that matched - in this case "b".

In this example, I create strings that look like JavaScript code by sticking in the appropriate syntax required by JavaScript (e.g. semi-colons and a call to the builtin function console.log(...). The final result is:

```
1  a = b + c;
2
3  console.log (a);
```

You should be able to run this Ohm-JS program in the console using `node`, like:

```
1  $ node simple.js
2
3  a = b + c;
4  console.log (a);
5
6  $
```

_terminal()

The line `_terminal () { return this.sourceString; }` does some required housekeeping in Ohm-JS. It specifies that when the matching engine hits bottom, the action is to simply return the matched string.

These housekeeping functions are documented in the Ohm-JS documentation.

_iter()

Some submatches match multiple items. The grammar operations `*`, `+`, and, `?` cause multiple matches. The function `_iter(...)` is a housekeeping function needed to specify actions for such multiple matches.

As usual, see the Ohm-JS documentation for further information on how to correctly deal with `_iter(...)`.

The final three lines of code

```
...
var cst = g.match ("a = b + c");
var transpiled = semantics (cst).js ();
console.log (transpiled);
```

cause the Ohm-JS engine to do the pattern matching, then apply the “js” operation to the results and to print out the return string from the “js” operation (again, this is a *very* simple example, and is written to return strings).

Hacking the Example to Transpile to Python Instead of JavaScript

We can use the same grammar.

We simply need to supply a different set of operations to generate code that looks like Python instead of JavaScript.

All we need to do is to extend the `.operation` stuff:

```
var ohm = require ('ohm-js');

const g = ohm.grammar (`
verysimple {
  Top = letter "=" letter "+" letter
}
`);
const semantics = g.createSemantics ().addOperation ("js"\
, {
    Top (target, eq, v1, plus, v2) {
    return `
${target.js()} = ${v1.js()} + ${v2.js()};
console.log (${target.js()});
`},
    _terminal () { return this.sourceString; }
});
semantics.addOperation ("py", {
    Top (target, eq, v1, plus, v2) {
    return `
${target.py()} = ${v1.py()} + ${v2.py()}
print (${target.py()})
`},
    _terminal () { return this.sourceString; },
    // not used in this example, but will be needed in bi\
gger examples
```

```
26      _iter (...children) { return children.map(c => c.py (\
27  )); }
28  });
29
30  var cst = g.match ("a = b + c");
31
32  var transpiled = semantics (cst).js ();
33  console.log ('*** javascript ***');
34  console.log (transpiled);
35
36  var pytranspiled = semantics (cst).py ();
37  console.log ('*** python ***');
38  console.log (pytranspiled);
```

We pattern-match once (`g.match`) and run two different operations on the results (`.js()` and `.py()`).

You should be able to run the example at the command line

```
1   $ node simple2
2   node simple2
3   *** javascript ***
4
5   a = b + c;
6   console.log (a);
7
8   *** python ***
9
10  a = b + c
11  print (a)
12
13  $
```

Obviously, the differences between JavaScript and Python are very small - in this *very* simple example. Feel free to go to town hacking this example to do more interesting kinds of transpilation.

Hacking the Example to Transpile to WASM Instead of JavaScript and Python

Exercise left to the reader.

Creating a DSL

For starters, the goal should be to emit code that looks like

```
const semantics = g.createSemantics ().addOperation ("js"\
, {
    Top (target, eq, v1, plus, v2) {
    return `
${target.js()} = ${v1.js()} + ${v2.js()};
console.log (${target.js()});
`},
    _terminal () { return this.sourceString; }
});
```

This would let us avoid writing JavaScript. We could run the DSL to generate the above code, then COPY/PASTE the code into a JavaScript program.

What should the syntax for the DSL be? Here's an untested suggestion:

```
SimpleFAB {
  Top [target eq v1 plus v2] = @'
const semantics = g.createSemantics ().addOperation ("js"\
, {
    Top (target, eq, v1, plus, v2) {
    return `
${<<target>>.js()} = ${<<v1>>.js()} + ${<<v2>>.js()};
console.log (${target.js()});
`},
    _terminal () { return this.sourceString; }
});
```

```
12  '@
13  }
```

Hints:

- ‘FAB’ is a short-form for ‘fabrication’.
- Use non-ASCII quotes in pairs and non-ASCII brackets in pairs for easy matching by PEG-based parsers like Ohm-JS.
- COPY/PASTE the above DSL into an example section of the Ohm-Editor, then see if you can create a grammar that groks this DSL.
- A fabrication string is any set of characters between the pairs of quotes `@'...'@`. You don’t need to parse the stuff inside of the string, as long as you write a pattern that begins with `@'` and ends with `'@` and you recognize parameter names bracketed by `‹‹...››`.
- If literal `$` characters appear in the fabrication string, you will need to escape them, to keep JavaScript from complaining.

Why Would Anyone Use Only Strings For This?

[advanced] So, why would I only use strings? Transpilers convert one string into another string. Compilers are transpilers. Compilers convert files of high-level language strings into files of assembler strings. ASM converts a file sof assembler strings into binary bits.

The “good part” of Functional Programming is the idea of converting text to other text, i.e. string to string. This is called “referential transparency”, and, is the cause for all of the arcane restrictions in FP, like no side-effects, etc.

JS (and Python) make string manipulation easy. To do this without getting confused, it helps to do actions in a certain sequence (“order”), but, that’s easy to do just by calling Ohm more than once in the same program (kinda like UNIX pipelines but without the added bloat).

I like to Design first to get something running correctly, then, Optimize later.

Alan Kay is on the record saying that new languages should use existing languages as “assembler”. Sounds like “transpilation” to me.

Git Repository

https://github.com/guitarvydas/simpleohm

Ohm-JS

https://ohmjs.org

Statecharts

TBD for the time being, see https://guitarvydas.github.io/2020/12/09/StateCharts.htn

TBD for now, see https://www.youtube.com/watch?v=r8xoLH0YaE4&t=8s

Hamburger Workbench

TBD

Parsing Idioms

TBD

Essays and Blogs and YouTube, etc.

still evolving…

most recent essays https://publish.obsidian.md/programmingsimplicity/

blogs: https://guitarvydas.github.io most recent Table of Contents https://guitarvydas.github.io/2021/12/10/Table-of-Contents-Dec-01-2021.html

YouTube: https://www.youtube.com/channel/UC-97kq8Phfahz4UWCCav_-zQ https://www.youtube.com/channel/UC9EJr0nKHwadbHUtc5zHdmQ/videos

Twitter: @paul_tarvydas