

Introduction

I perform some experiments with Ohm-js, fill my toolbelt and then move on to more interesting applications...

GitHub

<https://github.com/guitarvydas/OhmSmallSteps>

Source to Source

This is the most basic example I could think of - input a string from a file, dump it out.

See <https://github.com/guitarvydas/OhmSmallSteps>

Ohm file source2source.ohm:

```
Program {  
  code = any*  
}
```

JS (JavaScript) file source2source.js:

```
// usage:  
// npm install ohm-js  
// node source2source.js  
  
const fs = require ('fs')  
const ohm = require ('ohm-js')  
const grammarData = fs.readFileSync('source2source.ohm')  
const grammar = ohm.grammar(grammarData)  
  
const input = fs.readFileSync('s2s.js')  
  
const result = grammar.match(input)
```

```

    if (result.succeeded()) {
        console.log("Matching Succeeded")
    } else {
        console.log("Matching Failed")
    }
}
const semantics = grammar.createSemantics()

// recursive function to get the source of a non-terminal node
// from https://repl.it/talk/learn/Making-your-own-programming-
// language-with-NodeJS/45779
const node_to_source = node => {
    if (node.ctorName == "_terminal") {
        return node.primitiveValue
    } else {
        return node.children.map(n => node_to_source(n)).join('')
    }
}

semantics.addOperation(
    'dump',
    {
        code: (s) => {
            return node_to_source(s._node)
        }
    }
)

console.log(semantics(result).dump())

```

Test file s2s.js:

```

/* a sample comment */
function f(a, b) {
    a = b; // a 2nd sample comment
}

```

usage:

> node source2source.js

1. Get the Grammar to Work

This is the most basic example I could think of - input a string from a file,

dump it out.

See <https://github.com/guitarvydas/OhmSmallSteps>

My workflow is:

1. Get the grammar right.
2. Add semantics.

This is step 1 - see that the grammar is working as expected.

This should print "Matching Succeeded".

See source2source1a.js.

Ohm file source2source.ohm:

```
Program {  
  code = any*  
}
```

JS (JavaScript) file source2source1a.js:

```
// usage:  
// npm install ohm-js  
// node source2source1a.js  
  
const fs = require ('fs')  
const ohm = require ('ohm-js')  
const grammarData = fs.readFileSync('source2source.ohm')  
const grammar = ohm.grammar(grammarData)  
  
const input = fs.readFileSync('s2s.js')  
  
const result = grammar.match(input)  
  
if (result.succeeded()) {  
  console.log("Matching Succeeded")  
}
```

```

    } else {
        console.log("Matching Failed")
    }

```

Test file s2s.js:

```

/* a sample comment */
function f(a, b) {
    a = b; // a 2nd sample comment
}

```

usage:

> node source2source1a.js

2. Add Semantics

This is the most basic example I could think of - input a string from a file, dump it out.

See <https://github.com/guitarvydas/OhmSmallSteps>

Ohm file source2source.ohm:

```

Program {
    code = any*
}

```

JS (JavaScript) file source2source1b.js:

```

// usage:
// npm install ohm-js
// node source2source.js

```

```

const fs = require ('fs')
const ohm = require ('ohm-js')
const grammarData = fs.readFileSync('source2source.ohm')
const grammar = ohm.grammar(grammarData)

```

```

const input = fs.readFileSync('s2s.js')

const result = grammar.match(input)

if (result.succeeded()) {
  console.log("Matching Succeeded")
} else {
  console.log("Matching Failed")
}

const semantics = grammar.createSemantics()

// recursive function to get the source of a non-terminal node
// from https://repl.it/talk/learn/Making-your-own-programming-
// language-with-NodeJS/45779
const node_to_source = node => {
  if (node.ctorName == "_terminal") {
    return node.primitiveValue
  } else {
    return node.children.map(n => node_to_source(n)).join('')
  }
}

semantics.addOperation(
  'dump',
  {
    code: (s) => {
      return node_to_source(s._node)
    }
  }
)

console.log(semantics(result).dump())

```

Test file s2s.js:

```

/* a sample comment */
function f(a, b) {
  a = b; // a 2nd sample comment
}

```

usage:

> node source2source1b.js

Matching Pairs

Ohm can parse matching pairs of delimiters. This can't be done easily with YACC.

Ohm, PEG and recursive-descent parsers in general can include rules that call themselves.

A very simple example is shown below...

```
Program {
  code = matchingBraces+ eol
  matchingBraces =    "{" matchingBraces "}" -- braces1
                    | innards
  innards = "inside"
  eol = "\n"
}
```

The "magic" happens in the "matchingBraces" production. It consumes an opening brace "{", calls itself, then consumes a closing brace "}".

This grammar can parse nested sets of pairs, e.g. "{{inside}}".

Pairtest.js:

```
{{inside}}
```

Pairs.js:

¹ The "- -" annotation is syntactic sugar provided by Ohm, to help with conformance of Arities. There are two rules in "matchingBraces", the first has Arity 3 ("{" matchingBraces "}" has 3 matches) and the second has Arity 1 ("innards" is a single match). The "- -" notation automatically splits its branch off into a sub-rule called matchingBraces_braces. The sub-rule "matchingBraces_braces" has Arity 3, but the alternate in "matchingBraces" has Arity 1 (after the split). After this split, all alternates of "matchingBraces" have Arity 1 and Ohm (JS) is happy.

```

// usage:
// npm install ohm-js
// node pairs.js

const fs = require ('fs')
const ohm = require ('ohm-js')
const grammarData = fs.readFileSync('pairs.ohm')
const grammar = ohm.grammar(grammarData)

const input = fs.readFileSync('pairtest.js')

const result = grammar.match(input)
//console.log(grammar.trace(input).toString())

if (result.succeeded()) {
  console.log("Matching Succeeded")
} else {
  console.log("Matching Failed")
}

const semantics = grammar.createSemantics()

// recursive function to get the source of a non-terminal node
// from https://repl.it/talk/learn/Making-your-own-programming-
// language-with-NodeJS/45779
const node_to_source = node => {
  if (node.ctorName == "_terminal") {
    return node.primitiveValue
  } else {
    return node.children.map(n => node_to_source(n)).join('')
  }
}

semantics.addOperation(
  'dump',
  {
    code: (braced,eol) => {
      return "matching pairs of braces=/" +
node_to_source(braced) + "/"
    }
  }
)

console.log(semantics(result).dump())

```

Flip Chunks of Code

We finally "do something".

The parser figures out "chunkA" and "chunkB" for us.

Then, we use JS to construct a final string with chunkA and chunkB reversed.

This is a cheesy solution, but hey, I believe in single-use DSLs, so it's open season. Anything what will get me from Point a to Point b in a repeatable² fashion is OK with me.

This simple example *could* have been done solely in JS using RegExps and .replace(), but, huge problems, e.g. using a parser to refactor code, are more easily handled with parsers than with RegExp.

flip.ohm:

```
Program {  
  code = chunkA chunkB  
  chunkA = "aaa" eol*  
  chunkB = "bbb" eol*  
  eol = "\n"  
}
```

fliptest.js

```
aaa  
bbb
```

flip.js:

```
// usage:  
// npm install ohm-js  
// node flip.js
```

² and understandable / maintainable


```

const fs = require ('fs')
const ohm = require ('ohm-js')
const grammarData = fs.readFileSync('flip.ohm')
const grammar = ohm.grammar(grammarData)

const input = fs.readFileSync('fliptest.js')

const result = grammar.match(input)

if (result.succeeded()) {
  console.log("Matching Succeeded")
} else {
  console.log("Matching Failed")
}

const semantics = grammar.createSemantics()

// recursive function to get the source of a non-terminal node
// from https://repl.it/talk/learn/Making-your-own-programming-
// language-with-NodeJS/45779
const node_to_source = node => {
  if (node.ctorName == "_terminal") {
    if (node.primitiveValue.toString() == "\n") {
      return ''
    }
  }
  else
    return node.primitiveValue
  } else {
    return node.children.map(n => node_to_source(n)).join('')
  }
}

semantics.addOperation(
  'dump',
  {
    code: (a,b) => {
      return node_to_source(b) + node_to_source(a)
    }
  }
)

console.log(semantics(result).dump())

```

Find and Replace

More JS cheeze. I use the previously built code and JS' `.replace()` with RegExps to edit the final result...

Again, this toy example doesn't show the full power of this kind of parsing/editing.

This is a technique that needs to be stored one's toolbox.

FindAndReplace.js:

```
// usage:
// npm install ohm-js
// node flip.js

const fs = require('fs')
const ohm = require('ohm-js')
const grammarData = fs.readFileSync('flip.ohm')
const grammar = ohm.grammar(grammarData)

const input = fs.readFileSync('fliptest.js')

const result = grammar.match(input)

if (result.succeeded()) {
  console.log("Matching Succeeded")
} else {
  console.log("Matching Failed")
}

const semantics = grammar.createSemantics()

// recursive function to get the source of a non-terminal node
// from https://repl.it/talk/learn/Making-your-own-programming-
// language-with-NodeJS/45779
const node_to_source = node => {
  if (node.ctorName == "_terminal") {
    if (node.primitiveValue.toString() == "\n") {
      return '\n'
    }
  }
  else
    return node.primitiveValue
  } else {
    return node.children.map(n => node_to_source(n)).join('')
  }
}

semantics.addOperation(
```

```

        'dump',
        {
            code: (a,b) => {
                return node_to_source(b) + node_to_source(a).replace(/a/g,
"ABC")
            }
        }
    )

    console.log( semantics(result).dump() )

```

Skip Spaces

To skip spaces, we simply need to change all of the rules so that they begin with uppercase letters.

N.B. The semantic action "code" must also be changed to "Code".

I used the flip code for this test...

skipspacetest.js:

```
ccc ddd
```

skipspaces.ohm:

```

Program {
    Code = ChunkC ChunkD
    ChunkC = "ccc" Eol*
    ChunkD = "ddd" Eol*
    Eol = "\n"
}

```

skipspaces.js:

```

// usage:
// npm install ohm-js
// node skipspaces.js

```

```
const fs = require ('fs')
```

```

const ohm = require ('ohm-js')
const grammarData = fs.readFileSync('skipspaces.ohm')
const grammar = ohm.grammar(grammarData)

const input = fs.readFileSync('skipspacetest.js')

const result = grammar.match(input)

if (result.succeeded()) {
  console.log("Matching Succeeded")
} else {
  console.log("Matching Failed")
}

const semantics = grammar.createSemantics()

// recursive function to get the source of a non-terminal node
// from https://repl.it/talk/learn/Making-your-own-programming-
// language-with-NodeJS/45779
const node_to_source = node => {
  if (node.ctorName == "_terminal") {
    if (node.primitiveValue.toString() == "\n") {
      return ''
    }
  }
  else
    return node.primitiveValue
  } else {
    return node.children.map(n => node_to_source(n)).join('')
  }
}

semantics.addOperation(
  'dump',
  {
    Code: (a,b) => {
      return node_to_source(b) + node_to_source(a)
    }
  }
)

console.log(semantics(result).dump())

```

Skipping Comments

I use the original s2s.js as a test file, add (+) a set of rules that treat comments as whitespace.

skipcomments.ohm:

```
Program {
  Code = any*
  space += slashStarComment
        | slashSlashComment
  slashStarComment = "/*" (~"*/" any)* "*/"
  slashSlashComment = "//" (~"\n" any)* "\n"
}
```

skipcomments.js:

```
// usage:
// npm install ohm-js
// node source2source1b.js

const fs = require('fs')
const ohm = require('ohm-js')
const grammarData = fs.readFileSync('skipcomments.ohm')
const grammar = ohm.grammar(grammarData)

const input = fs.readFileSync('s2s.js')

const result = grammar.match(input)

if (result.succeeded()) {
  console.log("Matching Succeeded")
} else {
  console.log("Matching Failed")
}

const semantics = grammar.createSemantics()

// recursive function to get the source of a non-terminal node
// from https://repl.it/talk/learn/Making-your-own-programming-
// language-with-NodeJS/45779
const node_to_source = node => {
  if (node.ctorName == "_terminal") {
    return node.primitiveValue
  } else {
    return node.children.map(n => node_to_source(n)).join('')
  }
}

semantics.addOperation(
  'dump',
```

```

    {
      Code: (s) => {
        return node_to_source(s._node)
      }
    }
  )

console.log(semantics(result).dump())

```

Grammar Inheritance

Extending a Rule

I've written a grammar where Program2 inherits from Program.

Both grammars are in the same file³.

I use the += operator to extend the rule "Code" in the second grammar Program2.

N.B. we need to call ohm.grammars(...) not ohm.grammar(...) when there is more than one grammar in the file. This call returns a namespace instead of a grammar object.

N.B. The default start rule remains "Code" (inherited from the original grammar). I deal with the start rule later.

inherittest.js:

```

eee
aaa
bbb

```

³ I haven't figured out how to split them across files, yet.

inherit.ohm:

```
Program {
  Code = ChunkA ChunkB -- code
  ChunkA = "aaa" Eol*
  ChunkB = "bbb" Eol*
  Eol = "\n"
}

Program2 <: Program {
  Main = ChunkE
  Code += ChunkE ChunkA ChunkB -- code3
  ChunkE = "eee" Eol*
}
```

inherit.js:

```
// usage:
// npm install ohm-js
// node inherit.js

const fs = require ('fs')
const ohm = require ('ohm-js')
const grammarData = fs.readFileSync('inherit.ohm')
const grammarsNamespace = ohm.grammars(grammarData)

const input = fs.readFileSync('inherittest.js')

const result = grammarsNamespace.Program2.match(input)
// console.log(result)
//
console.log(grammarsNamespace.Program2.trace(input).toString())

if (result.succeeded()) {
  console.log("Matching Succeeded")
} else {
  console.log("Matching Failed")
}
// const semantics = grammar.createSemantics()
```

Overriding a Rule

```
Code := ChunkF ChunkA ChunkB
```

Use the `:=` operator instead of `=` (and `+=`).

overridetest.js:

```
fff  
aaa  
bbb
```

override.ohm:

```
Program {  
  Code = ChunkA ChunkB  
  ChunkA = "aaa" Eol*  
  ChunkB = "bbb" Eol*  
  Eol = "\n"  
}  
  
Program2 <: Program {  
  Code := ChunkF ChunkA ChunkB  
  ChunkF = "fff" Eol*  
}
```

override.js:

```
// usage:  
// npm install ohm-js  
// node inherit.js  
  
const fs = require ('fs')  
const ohm = require ('ohm-js')  
const grammarData = fs.readFileSync('override.ohm')  
const grammarsNamespace = ohm.grammars(grammarData)  
  
const input = fs.readFileSync('overridetest.js')  
  
const result = grammarsNamespace.Program2.match(input)  
// console.log(result)  
//  
console.log(grammarsNamespace.Program2.trace(input).toString())
```



```

if (result.succeeded()) {
    console.log("Matching Succeeded")
} else {
    console.log("Matching Failed")
}

```

Start Rule

To start with a given rule, specify its name (as a String) in match().

```

const result = grammarsNamespace.Program2.match(input,
"StartTest")

```

starttest.js:

```

ggg
aaa
bbb

```

start.ohm:

```

Program {
  Code = ChunkA ChunkB
  ChunkA = "aaa" Eol*
  ChunkB = "bbb" Eol*
  Eol = "\n"
}

Program2 <: Program {
  Code := ChunkF ChunkA ChunkB
  StartTest = ChunkG ChunkA ChunkB
  ChunkF = "fff" Eol*
  ChunkG = "ggg" Eol*
}

```

start.js:

```

// usage:
// npm install ohm-js
// node inherit.js

```

```

const fs = require ('fs')
const ohm = require ('ohm-js')
const grammarData = fs.readFileSync('start.ohm')
const grammarsNamespace = ohm.grammars(grammarData)

const input = fs.readFileSync('starttest.js')

const result = grammarsNamespace.Program2.match(input,
"StartTest")
// console.log(result)
//
console.log(grammarsNamespace.Program2.trace(input).toString())

if (result.succeeded()) {
  console.log("Matching Succeeded")
} else {
  console.log("Matching Failed")
}

```

CST

(Concrete Syntax Tree)

An experiment. I fed a very simple grammar into Ohm, then used the JS debugger (node inspect junk.js) to dissect the results:

dissect.scm:

```
#t
```

dissect.ohm:

```

Scm2JSBasic {
  Program = Form+
  Form = Atom
  Atom = SBoolean
  SBoolean = "#f" | "#t"
}

```

dissect.js:

```

const fs = require ('fs')
const ohm = require ('ohm-js')
const grammarData = fs.readFileSync('dissect.ohm')

```

```

const grammar = ohm.grammar(grammarData)

const input = fs.readFileSync('dissect.scm')

const result = grammar.match(input)

if (result.succeeded()) {
  console.log("Matching Succeeded")
} else {
  console.log("Matching Failed")
}

const semantics = grammar.createSemantics()
const node_to_source = node => {
  if (node.ctorName == "_terminal") {
    return node.primitiveValue
  } else {
    return node.children.map(n => node_to_source(n)).join('')
  }
}

semantics.addOperation(
  'dump',
  {
    Program: (p) => {
      return node_to_source(p)
    }
  }
)

console.log(semantics(result).dump())

function astString(obj) { return obj.toString() }

class Program {
  constructor(p) { this.p = p }
  astString () {
    console.log("<" + this.constructor.name + ">: " +
this.p.constructor.name) ;
    return "P \n" + this.p.map((x) => x.astString()).join('\n')}
}
class Form {
  constructor(s) { this.s = s }
  astString () {
    return "<" + this.constructor.name + ">:" +
this.s.constructor.name + " " + "[" +
this.s.astString().constructor.name + "]" " +

```

```

this.s.astString();}}

class Atom { constructor(a) { this.a = a } astString() { return
this.a.astString() }}
class SBoolean { constructor(b) { this.b = b } astString()
{ return this.b }}

// an operation that uses the above classes
semantics.addOperation(
  'ast',
  {
    Program: (p) => { return new Program(p.ast()) },
    Form: (s) => {
      console.log("Form " + this.s );},
    Atom: (a) => {
      console.log("Atom a=" + a);},
    SBoolean: (b) => {
      console.log("b is " + b._node.ctorName + " and b's ast is
" + b.ast()) ; },
    _terminal: () => { console.log("primitive " +
this.primitiveValue); return this.primitiveValue }
  }
)

console.log()
console.log("result")
console.log(result)
console.log()
console.log("semantics")
var sem = semantics(result)
var tree = sem.ast()
console.log()
console.log("tree")
console.log(tree)

```

Fig. 1 shows this data in diagrammatic form (some parts elided).

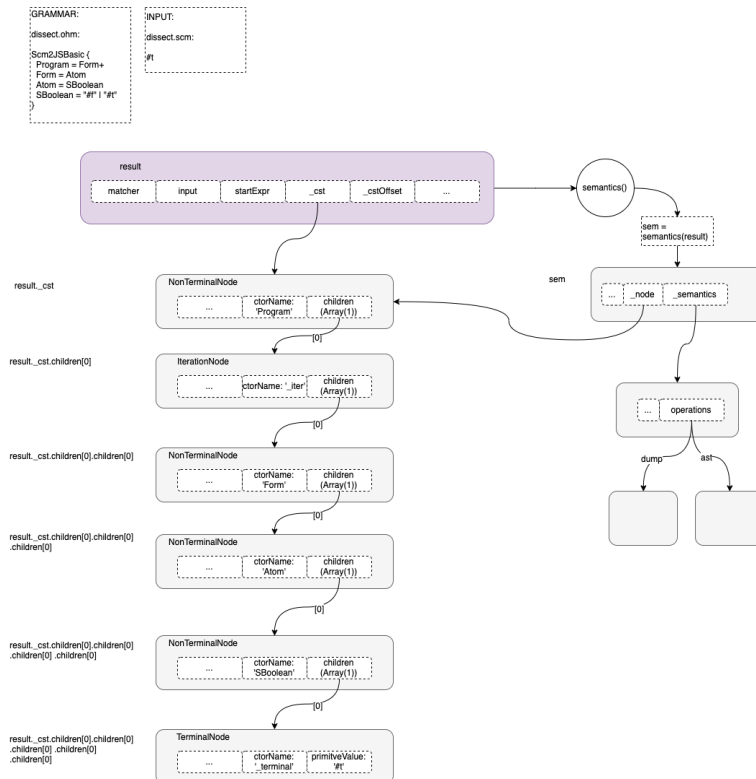


Fig. 1 Overview of Ohm Parse Results

Pipelining Grammars

Ideally, we would want a grammar that inputs a string and produces a token stream which we could then feed into other grammars in a pipeline fashion.

OMeta is able to run grammar pattern matchers on arbitrary input (e.g. strings and tokens).

Ohm-JS cannot do this and works only with input strings (of characters).

We can compromise and produce strings. We'll ignore low-level efficiency for the moment and see that our idea works. Later, we can switch to OMeta if needed. It might turn out that everything works "fast enough" so, we might not need to switch to OMeta and keep our "dumb" implementation.

Lines and Filenames

A token-based representation allows us to stick debugging information into each token.

For example, when there's a syntax error, we might want to know the line & character position of each character and which file it came from.

For now, we can use a trick pioneered by the C compiler - create *pragmas* for file and lines. The pragmas are passed down the pipeline and, if an error occurs, the filename and line number of the erroneous input can be examined and displayed.

In C, the file pragma appears as a single line with the contents '#file⁴ "<...>"'. The line pragma appears as a different single line '#line nn⁵'. The various passes in the pipeline record the pragmas but otherwise ignore them until an error occurs.

Question: if the grammar is embedded as a script in HTML, what is the #file for it?

Answer: I don't know yet. It will depend on the debugger.

Question: if the grammar is embedded as a script in HTML, what is *nn* in the #line for it? Is *nn* a line-offset within the script or is it a line-offset from the beginning of the HTML file?

Answer: I don't know yet.

Question: if the grammar is embedded as a script in HTML, maybe #file and

⁴ Where "<...>" is the filename in quotes.

⁵ Where "nn" is an integer line number.

#line can be replaced by something else, like hidden HTML elements? Is #file even useful in this context?

Answer: I don't know yet.

TODO:

1. We want to create a unity grammar that simply outputs what is input (probably stripping out comments). This might give us ideas on how to proceed - DI via *divide&conquer* and *experimentation*.
2. We want to create a semantics that inserts #line pragmas.
3. We want to create a grammar that expands the ' (QUOTE) shorthand in Scheme to its full form (also Scheme, just harder for humans to read).

I will switch to using an index.html page for these experiments, instead of using node.js...

Math Example in Browser

See <https://github.com/harc/ohm/tree/master/examples/math>.

This example shows how to embed a grammar into a webpage and then process it using different sets of semantics.

The grammar and semantics can be found in <https://github.com/harc/ohm/blob/master/examples/math/index.html>.

The grammar is embedded as a script with type="text/ohm-js". This type distinguishes the ohm grammar from the other scripts and is read into variable "g" by line 63:

```
var g = ohm.grammarFromScriptElement();
```

The actual matching (g.match(...)) of the grammar against some input (string) is done in input.oninput (line 456).

A semantics dictionary is created in line 136:

```
var s = g.createSemantics();
```

Semantic actions are created by the "s.addOperation" and "s.addAttribute" functions.

AddOperation is called to create 3 sets of semantics: 'interpret', 'toTree' and 'toTwoD'.

AddAttribute is called to create a 4th semantics names 'asLisp'.

AddOperation and addAttribute are almost the same, except that the latter (addAttribute) creates properties (not methods) and is memoized (an optimization that reduces the cost of backtracking during the pattern match).

'Interpret' returns a JS value, 'asLisp' returns a JS list (arrays of arrays), 'toTree' and 'toTwoD' return trees of HTML elements.⁶

The function 'input.oninput' (line 456) does calls "show" (../lib.js) for each of the semantics. The 'interpret' semantics returns a JS value which creates a string *div*. The 'asLisp' semantics returns a JS tree which is processed by the stringifyLisp function and creates a string *div*. The semantic functions 'tree' and 'twoD' return trees of HTML elements. The file math.css is creates visualization styles for the elements which result in the final appearance of the elements as a parse tree and as 2D mathematical notation (factions shown as numerators and denominators).

The "show()" function simply appends the elements to the various divs (value,

⁶ The Elements are created by the 'elt' function. The first parameter to 'elt()' is always an element id, which always corresponds to one of the rules in the grammar. The rest of the arguments are sub-trees.

lisp, tree, twoD) and the browser displays the divs. The "show()" function also clears the top-level divs, allowing the display to be iterated. The CSS file (math.css) does most of the real work - by arranging that #tree elements have a visible bottom border, etc.

Ohm and PROLOG

Introduction

PROLOG is just an exhaustive pattern matcher.

PROLOG uses backtracking to exhaustively explore a search space.

One could implement backtracking pattern matching with loops of loops, but the result quickly becomes unmaintainable and defies DI (expression of Design Intent), by overwhelming the code-reader with details.

Once one splits the *expression* of backtracking pattern matching away from the details, one can begin to *think* in terms of relations. PROLOG was an early attempt to make this split⁷.

I don't know of a PROLOG library for JS. Maybe one exists⁸, but that's not my point, here. My point is to show that one can bolt high-falutin' technologies into JS by using parsers.

The easiest-to-understand description of PROLOG (IMO) is Nils Holmgren's⁹

⁷ A more recent technology is miniKanren. Once one learns to think in miniKanren, the possibilities become fantastic and weird very quickly. See Barliman and https://www.youtube.com/watch?v=er_ILvkklsk.

⁸ Or, maybe there is a core.logic miniKanren library for JS?

⁹ See <https://www.t3x.org/bits/prolog6.html>

PROLOG in Scheme.

Can we steal that code? It is written in Scheme and we want JS. Oh wait, Scheme has a super-simple syntax. Maybe we can use an Ohm-based parser to convert the Scheme code into JS, then build an Ohm-based parser that gives us PROLOG syntax? Experiment begins...

Ohm and Scheme

Basic Scheme Parser

The fundamental data type of Scheme is the SEXP (s-expression).

A SEXP is either an atom or a list. A list can contain atoms and other lists.

If we get the basics right, we could use Ohm's grammar inheritance to build bigger grammars.

I don't expect to get that grammar right on the first pass. Divide and Conquer. I try out bits of ideas first, then, modify the bits to suit the bigger picture.

My first attempt at a basic parser for SEXPs:

Scm2jsbasic1.ohm:

```
Scm2JSBasic {  
  Scm = List | Atom | Boolean  
  List = "(" ListInnards ")"  
  ListInnards = (Atom | List)*  
  Atom = Integer | Symbol | Boolean  
  Boolean = "#f" | "#t"  
  Integer = Numchar+  
  Numchar = "0".."9"  
  Symbol = Letchar (Letchar | Numchar | "?" | "_")*  
  Letchar = LC | UC | "+" | "*"
```

```
LC = "a".. "z"  
UC = "A".. "Z"  
}
```

I can already see that I'm going to be replacing Lchar with something more interesting...

Hmm, maybe the whole definition of "symbol" needs to be replaced...

Just to be clear - I am *not* trying to parse all of Scheme, only enough to parse Nils' PROLOG code. This will be a single-use DSL...

Engineering trade-off - I don't need to do everything (the last 5% of details always kill). YAGNI. I hope to transpile Nils' Scheme code to JS, then let the JS compiler do the heavy lifting.

I'm planning to use the Ohm parser as a fancy editor...

Next step - pour all of Nils' code into the test, fix what doesn't parse...

scm2jsbasictest1.scm:

```
(null? xyz)
```

scm2jsbasic1.js:

```
// usage:  
// npm install ohm-js  
// node scm2jsbasic1.js  
  
const fs = require ('fs')  
const ohm = require ('ohm-js')  
const grammarData = fs.readFileSync('scm2jsbasic1.ohm')  
const grammar = ohm.grammar(grammarData)  
  
const input = fs.readFileSync('scm2jstest1.scm')  
  
const result = grammar.match(input)
```

```
if (result.succeeded()) {  
    console.log("Matching Succeeded")  
} else {  
    console.log("Matching Failed")  
}
```

Inherit the Basic Parser

Let's try to inherit the basic scheme parser.

I think that we're going to need to revisit that basic parser many times. We are going to iterate on the basic definition as we build up more interesting parsers.

This is *anti-waterfall* thinking. Beginning with the belief that one doesn't know where this is going to go.

Waterfall design would be the idea that we can get the basic grammar (and all subsequent layers) right without iteration.

Programmers like languages that have REPLs (Read-Eval Print Loop) because it helps them iterate. This is called "design".

Maintainers aren't especially drawn to REPLs for the same reasons, but that's another whole branch of Software Engineering...

Basic Scheme Version #2

Add ! as a valid identifier (first & follow) character.

Add "-" as a valid identifier (follow) character.

Move "-", "_", "!", "?" to first (& follow) character sets.

Add Strings (stunted, no escapes yet).

Add comments as whitespace.

Add () as a valid atom (NullList) and make List use + instead of * (looking ahead to future conversion simplicity).

Add ' recognizer. (Single quote). Single quote is a convenience operator that expands into a QUOTE form. E.G. "'(a b c)" => "(quote (a b c))". In this step, we deal only with the parse. We will re-write QUOTE later...

Scheme macros use `` (back quote) to start a macro. Inside a macro, quoting is "reversed" - everything is quoted unless it is preceded by a comma ",". There are more features, but we are striving only to recognize prolog-6.scm.... YAGNI.

Add ` recognizer. (Back quote, back tick)

Add , recognizer.

Allow for multiple lists in a file.

Lists are made up of cons cells. A cons cell is a 2-tuple {pointer to data, pointer to next cell}. In most "normal" lists, the final "pointer to the next cell" is NIL (i.e. no next cell). As a convenience, the printer prints such lists in the most common notation - openParen, items, closeParen (e.g. "(a b c)"). In some cases, lists end with a datum instead of NIL, and, in those cases the list prints as: openParen, items, dot, last item, closeParen (e.g. "(a b . c)"). All lists are the same - they are made up of cons cells - it is *only* the printer that shows lists in shorthand notation, where the most common lists are null-terminated lists.

sm2jstest2.scm (not a practical piece of code, just a test case):

```
(define (clear_r x)
```

```

(set-car! (cddr x) '(()))
(append (cdr a) `(r! ,l) (cdr g)))

(define (clear_r x)
  (set-car! (cddr x) '(()))
  (append (cdr a) `(r! ,l) (cdr g)))

(a b . c)
((a b . c))

```

scm2jsbasic2.ohm:

```

Scm2JSBasic {
  Program = Scm+
  Scm = List | Atom | Boolean
  QuotedSexp = "'" Scm
  BackQuotedSexp = "`" Scm
  CommaSexp = "," Scm
  List = DottedList | NullTerminatedList
  DottedList = "(" ListItem+ "." ListItem ")"
  NullTerminatedList = "(" ListItem+ ")"
  ListItem = (Atom | List)
  Atom = Integer | Symbol | String | Boolean | NullList |
QuotedSexp | BackQuotedSexp | CommaSexp
  NullList = "(" ")"
  Boolean = "#f" | "#t"
  Integer = Numchar+
  Numchar = "0".."9"
  String = "\"" (~\" any)+ "\""
  Symbol = Letchar (Letchar | Numchar)*
  Letchar = LC | UC | "+" | "*" | "!" | "?" | "_" | "-"
  LC = "a".."z"
  UC = "A".."Z"
  semiColonComment = ";" (~\n any)* "\n"
  space += semiColonComment
}

```

scm2jsbasic2.js:

```

const fs = require ('fs')
const ohm = require ('ohm-js')
const grammarData = fs.readFileSync('scm2jsbasic2.ohm')
const grammar = ohm.grammar(grammarData)

const input = fs.readFileSync('scm2jstest2.scm')

const result = grammar.match(input)
// console.log(grammar.trace(input).toString())

```

```

if (result.succeeded()) {
    console.log("Matching Succeeded")
} else {
    console.log("Matching Failed")
}

```

Basic Scheme vs prolog-6.scm

Try the whole enchilada.

test:

see prolog-6.scm <https://www.t3x.org/bits/prolog6.html>

[ohm grammar:](#)

[see Basic Scheme Version #2](#)

[scm3jsbasic3.js:](#)

```

const fs = require ('fs')
const ohm = require ('ohm-js')
const grammarData = fs.readFileSync('scm2jsbasic2.ohm')
const grammar = ohm.grammar(grammarData)

const input = fs.readFileSync('prolog-6.scm')

const result = grammar.match(input)
//console.log(grammar.trace(input).toString())

if (result.succeeded()) {
    console.log("Matching Succeeded")
} else {
    console.log("Matching Failed")
}

```

Lisp is Assembly Language

Lisp makes a wonderful assembly language - it does just about everything.

Programs can be built in multiple paradigms and, Lisp has no syntax to get in the way.

Scheme is a stripped-down Lisp. CL (Common Lisp) is a production tool.

I think that Lisp should form the basis of most languages and most projectional editors, but, I digress.

Basic Scheme to JS

I will settle on scm2jsbasic2.ohm and call it scm2js.ohm.

Likewise scm2js.js.

test:

prolog-6.scm

scm2js.ohm:

```
Scm2JSBasic {
  Program = Scm+
  Scm = List | Atom | Boolean
  QuotedSexp = "'" Scm
  BackQuotedSexp = "`" Scm
  CommaSexp = "," Scm
  List = DottedList | NullTerminatedList
  DottedList = "(" ListItem+ "." ListItem ")"
  NullTerminatedList = "(" ListItem+ ")"
  ListItem = (Atom | List)
  Atom = Integer | Symbol | String | Boolean | NullList |
  QuotedSexp | BackQuotedSexp | CommaSexp
  NullList = "(" ")"
  Boolean = "#f" | "#t"
  Integer = Numchar+
  Numchar = "0".."9"
  String = "\"" (~\" any)+ "\""
  Symbol = Letchar (Letchar | Numchar)*
  Letchar = LC | UC | "+" | "*" | "!" | "?" | "_" | "-"
```



```

LC = "a".. "z"
UC = "A".. "Z"
semiColonComment = ";" (~"\n" any)* "\n"
space += semiColonComment
}

```

scm2js.js:

```

const fs = require ('fs')
const ohm = require ('ohm-js')
const grammarData = fs.readFileSync('scm2jsbasic.ohm')
const grammar = ohm.grammar(grammarData)

const input = fs.readFileSync('prolog-6.scm')

const result = grammar.match(input)
// console.log(grammar.trace(input).toString())

if (result.succeeded()) {
  console.log("Matching Succeeded")
} else {
  console.log("Matching Failed")
}

```

Unity Parser

The first step towards pipelining grammars is to create a unity parser. The Unity parser contains a grammar plus an unparser.

The grammar we need is the bare minimum to parse prolog-6.scm.

The unparser simply walks the CST (Concrete Syntax Tree) and re-emits the sub-trees as needed.

Spaces and comments are dumped by the parser, so they don't appear in the CST and are not re-emitted.

This exercise showed a deficiency in the original Scm2JSBasic grammar. The grammar, as written, was slurping adjacent symbols and making them into single

symbols. The original grammar ignored spaces, including those that separated adjacent symbols.

The original grammar was written using syntactic rules everywhere.

The fix was to rewrite the grammar using lexical rules for the low-level recognizers.

I, also, switched from using node.js to using HTML for the grammar.

The result is in index.html. (See commit [2e40532e25e63f4d3be1f8f37b18d37df266970d](#)).

index.html: (less prolog-6.scm code):

```
<!doctype html>
<html>
  <head>
    <title>ohm/js in small steps</title>
    <meta charset=utf-8>
    <script src="/Users/tarvydas/projects/ohm/examples/lib.js"></
script>
    <script src="/Users/tarvydas/projects/ohm/dist/ohm.js"></
script>
    <script type="text/ohm-js">

      // An Ohm grammar for Scheme, step 1.
      Scm2JSBasic {
        Program = Form+
        Form = SList | Atom
        QuotedSexp = "'" Form
        BackQuotedSexp = "`" Form
        CommaSexp = "," Form
        SList = DottedList | NullTerminatedList
        DottedList = "(" ListItem+ "." ListItem ")"
        NullTerminatedList = "(" ListItem+ ")"
        ListItem = (Atom | SList)
        Atom = lexical_atom | Syntactic_Atom
        Syntactic_Atom = QuotedSexp | BackQuotedSexp | CommaSexp

        lexical_atom = lexical_integer | lexical_symbol |
lexical_string | lexical_boolean | lexical_nullList
```

```

lexical_nullList = "(" " ")"
lexical_boolean = "#f" | "#t"
lexical_integer = lexical_numchar+
lexical_numchar = "0".."9"
lexical_string = "\"" (~\" any)+ "\""
lexical_symbol = lexical_letchar (lexical_letchar |
lexical_numchar)*
lexical_letchar = lexical_lc | lexical_uc | "+" | "*" | "!" |
"?" | "_" | "-"
lexical_lc = "a".."z"
lexical_uc = "A".."Z"
semiColonComment = ";" (~\" any)* "\"
space += semiColonComment
}

```

```

</script>

```

```

<script>
  var g = ohm.grammarFromScriptElement();
</script>

```

```

<script type="text/test">
  (#t #f)
</script>

```

```

<script type="text/test">
  ( #t #f . #t)
</script>

```

```

<script type="text/test">
  ( . #t)
</script>

```

```

<script type="text/test">
  asymbol
</script>

```

```

<script type="text/test">
  (asymbol)
</script>

```

```

<script type="text/test">
  (asymbol1 asymbol2)
</script>

```

```

<script type="text/test">
  (asymbol1 asymbol2)
  (asymbol3 asymbol4)
</script>

```

```

    <script type="text/test">
... code for prolog-6.scm ...
    </script>

<script>

    var sem = g.createSemantics();

    function toListOfStrings(a) {
return a.join(' ');
    }

    function toPackedString(a) {
return a.join('');
    }

    sem.addOperation(
'unity',
    {
        Program: function(tree) {return
toListOfStrings(tree.unity())},
        Form: function(item) {return item.unity()},
        QuotedSexp: function(_, form) {return "\"" +
form.unity()},
        BackQuotedSexp: function(_, form) {return "`" +
form.unity()},
        CommaSexp: function(_, form) {return "," +
form.unity()},
        SList: function(lis) {return lis.unity()},
        DottedList: function(_lp, items, _dot, lastItem, _rp) {
return "(" + toListOfStrings(items.unity()) + " . " +
lastItem.unity() + ")"},
        NullTerminatedList: function(_lp, items, _rp) {
return "(" + toListOfStrings(items.unity()) + ")"},
        ListItem: function(item) {return item.unity()},
        Atom: function(a) {return a.unity()},
        lexical_integer: function(ns) {return
toPackedString(ns.unity())},
        lexical_symbol: function(c, cs) {return c.unity() +
toPackedString(cs.unity())},
        lexical_string: function(_q1, chars, _q2) {return "\"" +
toPackedString(chars.unity()) + "\""},
        lexical_lchar: function(c) {return c.unity()},
        lexical_numchar: function(c) {return c.unity()},
        lexical_lc: function(c) {return c.unity()},
        lexical_uc: function(c) {return c.unity()},

        lexical_nullList: function(_lp, _rp) {return "()"},
        lexical_boolean: function(b) {return this.sourceString},

```

```

        _terminal: function() { return this.primitiveValue; }
    }
    );
</script>

<script>
    var tests = document.querySelectorAll('script[type="text/
test"]');
</script>

</head>
<body>
    <button onclick="ohmtest()">Click me</button>
    <p id="timestamp"></p>
    <p id="input"></p>
    <p id="output"></p>
    <script>
        var resultsString = '';

        function displayDate () {
            document.getElementById('timestamp').innerHTML = Date();
        }
        function do1Test(testElement) {
            var testString = testElement.innerHTML;
            var r = g.match(testString);
            var tail = "...";
            var charsToDisplay = 40;
            if (testString.length < charsToDisplay) {
                tail = "";
            }
            resultsString = resultsString + "<br>" + r.toString() + " "
+ testString.substring(0,charsToDisplay) + tail +
(r.succeeded() ? (" <--- " + sem(r).unity()) : '');
        }
        function displayTestResults(s) {
            document.getElementById('input').innerHTML = s;
        }
        function ohmtest () {
            resultsString = '';
            console.log(tests);
            console.log(tests[0]);
            tests.forEach(do1Test);
            displayTestResults(resultsString);
            displayDate();
        }
        ohmtest();
    </script>
</body>
</html>

```

```

    </script>
  </body>
</html>

```

Expanding Quotes

In Lisp, "quote" is used so often that it has been given a shortcut, consisting of a single quote.

"a" is internally expanded into "(quote a)" and,

"('(a b))" is internally expanded into "((quote (a b)))".

Unquote is simple to implement as a tweak of the Unity parser:

(See commit f2105736ed1fb19b582986c4e8c2ba588c118eb8) *[this code has a bug - see later]*

```

    sem.addOperation(
      'unquote',
      {
        Program: function(tree) {return
toListOfStrings(tree.unquote())},
        Form: function(item) {return item.unquote()},
        QuotedSexp: function(_, form) {return "(quote " +
form.unquote() + ")"},
        BackQuotedSexp: function(_, form) {return "`" +
form.unquote()}},
        CommaSexp: function(_, form) {return "," +
form.unquote()}},
        SList: function(lis) {return lis.unquote()},
        DottedList: function(_lp, items, _dot, lastItem, _rp) {
return "(" + toListOfStrings(items.unquote()) + " . " +
lastItem.unquote() + ")"},
        NullTerminatedList: function(_lp, items, _rp) {
return "(" + toListOfStrings(items.unquote()) + ")"},
        ListItem: function(item) {return item.unquote()}},
        Atom: function(a) {return a.unquote()}},
        lexical_integer: function(ns) {return
toPackedString(ns.unquote());},
        lexical_symbol: function(c, cs) {return c.unquote() +
toPackedString(cs.unquote());},
        lexical_string: function(_q1, chars, _q2) {return "\"" +
toPackedString(chars.unquote()) + "\""},

```

```

lexical_letchar: function(c) {return c.unquote()},
lexical_numchar: function(c) {return c.unquote()},
lexical_lc: function(c) {return c.unquote()},
lexical_uc: function(c) {return c.unquote()},

lexical_nullList: function(_lp,_rp) {return "()"},
lexical_boolean: function(b) {return this.sourceString},
_terminal: function() { return this.primitiveValue; }
}

```

Expanding Backquotes

In Common Lisp (and Scheme), backquote is used as a shorthand to create lists.

Backquote evolved as a way to create programs that create programs. In Lisp, programs are lists, so programs that create programs are just programs that create lists.

In Lisp, the function "(LIST ...)" creates a list containing all of its arguments. In Lisp, all arguments are evaluated before being passed to a function.

In creating programs that create programs, using Lisp, one of the often-used idioms was a call to the function LIST with arguments that are quoted and some arguments that are evaluated.

This idiom occurred so frequently, that a shorthand was invented - the backquote.

Backquote is a kind of an inverted QUOTE.

Inside a backquoted form, an unquoted sexp is quoted and a form preceded by a comma is not quoted (it is eval'ed in the usual manner).

For example (LIST (QUOTE A) (QUOTE B) (QUOTE C)) is written in shorthand as `(A B C), and, (LIST (QUOTE A) B (QUOTE C)) is written in shorthand as `(A ,B C). Note that, in this latter case, B is evaluated but A and C are quoted. A list of 3 values is created.¹⁰ Clearly, `(A ,B C) is more expressive,

¹⁰ a list of 3 items: (QUOTE A), the value of B, and, (QUOTE C).

and easier to write and read, than (LIST (QUOTE A) B (QUOTE C)), if you know the shorthand.

The following examples form some of the tests I use for the backquote expander:

```
`a -> (quote a)
`,a -> a
`(a) -> (list (quote a))
`(a b c) -> (list (quote a) (quote b) (quote c))
`(a ,b c) -> (list (quote a) b (quote c))
,a --> error (comma not inside backquote)
```

To parse this, I use the grammar already developed. I *could* write some code within the rule "BackQuotedSexp" to rewrite the backquoted code, but, I will use a *trick* - I will switch to another set of semantic rules when I see a backquote. The main set of semantic rules is called 'unbackquote' and the backquote helper is called 'inbackquote'. The semantic rules in 'inbackquote' are written to return reverse-quoted strings, which is what I want when I see a backquote. The BackQuotedSexp rule of the main semantics calls the unparser in 'inbackquote' instead of recursively calling 'unbackquote' (as is done in the straight-forward 'unquote' semantics).

What happens in the case where a backquoted sexp is seen while already processing a backquote? I don't care. This case never happens in the prolog-6.scm code that I want to automatically process. This is, obviously, a place where I leave it as an exercise to the reader: try double-backquoting in a Lisp REPL to test your theory about how this is supposed to work, then mimic this behaviour in the parser.

Numeric, string and boolean atoms are essentially constants, so their values aren't changed by backquote.

(See commit 33eb0eb6f6389ab3423e2561b04e5e9221145ac4).

Pipelining Grammar

I should be able to feed the result from the latest unbackquote expander into the unity transformer.

This will test whether I can pipeline the grammar, i.e. feed one grammar into another. Ohm-JS can only parse character strings. The backquote expander produces strings, so this shouldn't be a problem.

Pipelinetest() runs each test case twice. The first time, the original Scheme is run through unbackquote. Then, the result from unbackquote is run, again, through unbackquote (which should parse, but find no quotes or backquotes to expand). The first run expands all quotes and backquotes. The second run does nothing, but parses the (already expanded) Scheme. The results from the first run should be the same as the results from the second (regression) run, using JS string equality "==".

This means that the results from unbackquote form a valid Scheme program (at least according to this grammar).

(See commit 56569e3b6e33344db9409368f5754454c71b37ce)

This means that we can pipe output from one grammar into another grammar.

The point of pipelining is to remove possible dependencies - Divide and Conquer. Once we have a grammar that expands quotes and backquotes, we know that it works. We never have to alter that grammar again. It does one job.

As we progress further, we will want to tweak the grammar (to give us better handles on parts of the parse). Pipelining ensures that no tweaks can alter the functioning of previously written code (grammars). Quote and backquote

expansion is working and we can forget about it, knowing that it will always work in the same way.

Tweak the Grammar for the 3 Top-Level Scheme Constructs

The minimal Scheme parser that we need to parse and transpile `prolog-6.scm` contains (only) 3 top-level constructs.

1. Global variable definitions
2. Global function definitions
3. Function calls.

A global variable definition looks like `"(define sym <form>)"`.

A global function definition looks like `"(define (sym syms) <form>)"`.

Function calls look like `"(sym <forms>)"`.

To pattern match this (aka parse this), we first need to differentiate syms from other kinds of atoms. I call these Identifiers. Identifiers are symbol names, and cannot be other kinds of atoms (e.g. strings, numbers, etc.).

We can have multiple grammars in an .HTML file, but we need to use `.grammarsFromScriptElements()` which returns a namespace containing multiple grammars, each contained in `<script type="text/ohm-js">...</script>` in the HTML document.

Top Level Forms

In `prolog-6.scm`, there are only 3 kinds of top-level forms:

1. global variable definition
2. global function definition
3. call a function.

I created a new grammar - Scm2JSPass2 - knowing that the grammar does not need to handle quotes and backquotes.

I added new rules to the pass2 grammar, to handle the 3 top level forms. The earlier grammar handled all forms, including top-level ones, with one rule "Form". Now, the pass2 grammar differentiates between the 3 top-level forms.

(See commit 9ad44acc2d62e918f33b652b49e3e8c950bbfcec).

Semantics - Quoting

We've reached the point where we need to consider what every kind of Scheme expression should be transpiled into valid - and meaningful - JS. This is called "semantics".

First, let's look at (quote ...). What does it mean in JS?

'() needs to become an empty list. In JS, we can convert that to an empty array "[]", or we can define a class called List and call "new" on it.

Let's choose the List class solution.

We have constructed a pattern matcher (aka parser) for Scheme. We can simply add a pattern for '()' to the matcher and have it convert "'()'" into "new List()".

Special Forms

In Scheme, almost everything is a function call with evaluated arguments.

There are some special forms, though:

- `cond`
- `let` (and `let*`)
- `if`

These are handled by adding more detailed rules to the grammar.

(See commit [f6c5b87c7a5995ea1a2486c8f05c6d92798c6610](#) and the preceding commits).

Test expressions in `COND` and `IF` are essentially the same, but it helps to give rules different names depending on their context (e.g. test expression in `COND` vs. test expression in `IF`). The transpiler might emit different code for each of these rules, depending on the target language and its syntax.

For example, I created separate parsing rules for the first test+clause in `COND` vs. the rest of the test+clauses of a `COND`. We want to transpile Scheme's `COND` into `if-then-elseif-then...` statements in JS. In this case, the first test+clause needs to transpile to an `"if"` statement, whereas the rest of test+clauses need to transpile to `"} else if (...) {"` statements.

The pattern matcher doesn't care when I split rules, but, this can be helpful in making the transpiler simpler to write.

At this moment, in this commit 've transpiled the special forms into pseudo-code, to watch how certain Scheme phrases are transpiled.

Later, I will convert the pass2 semantics from emitting pseudo-code to emitting JS.

Interlude - Illegal Characters

At this point, we can see that the transpiler, in its present state, allows characters, like "?" and "-" and "*" in identifiers that are illegal in JS identifiers.

I will deal with these later.

The transpiler is a pattern matcher, and, changing single characters to other strings will be easy. For now, it is easier on the eyes (at least for me, since I'm very familiar with Lisp) to leave the illegal characters and debug the operation of the transpiler at the 30,000 foot level.

I proceed using Divide&Conquer and chew on the harder bits first and leaving the "obvious" stuff (least risky) to the end. There is more to learn by attacking the unknowns first. In the worst case, I could use the "SED" tool to change the illegal characters. This problem isn't worrisome at present and can be deferred.

List Constants

Scheme (and Lisp in general) reads code into lists. It interprets the first item in a list as the function and the rest of the items in the list as (evaluated) arguments.

QUOTE is used to create list constants. The Lisp runtime doesn't try to

execute constants, but only creates them. This is very similar to how, say, string constants are created in JS.

For example, the database used in prolog-6.scm is:

```
(define goals '((some (? X))
                 (some (? Y))
                 (neq (? X) (? Y))))
```

which says that the global variable "goals" is created as a list constant containing 3 items.

Pass1 of the transpiler converts this into:

```
(define goals (quote ((some (? X)) (some (? Y)) (neq (? X) (? Y)))))
```

We want this to transpile to something like the following, in pseudo-code:

```
(? X) -> new List("?", "X") (where ? is a valid identifier in
Lisp)
(some (new List("?", "X"))) -> new List("some", new
List("?", "X"))
```

and the whole global constant (global variable) definition becomes:

```
var goals = new List( new List("some",new List("?", "X")),
                     new List("some",new List("?", "Y")),
                     new List("neq", new List("?", "X"), new
List("?", "Y") );
```

Looking at it this way, I see that the prolog-6.scm code uses "(? X)" to create "logic variables" (in this case, a logic variable called X). Lists are used for all sorts of data in Lisp, but in this specific example, (? X) creates a logic variable. Hmm, maybe we want the transpiler to turn this into:

```
var goals = new List( new List("some",new LogicVar("X")),
                     new List("some",new LogicVar("Y")),
```

```
new List("neq", new LogicVar("X"), new  
LogicVar("Y")) );
```

It would be easy to have the transpiler convert patterns like "(? X)" into something more meaningful. Hmm, maybe we should stick to the list version and make such changes later, after the basics are debugged.

Decision made - I'll stick with the List version for now and optimize later...

So, the transpiler should convert

```
(quote (...)) -> new List(...)
```

And, it should do this recursively.

The syntax for JS is wildly different from that of Scheme. In order to continue using (the simpler) Scheme syntax, I transpile the above into valid Scheme. I will unwind this in the final (emitter) pass.

```
(quote (...)) -> (@newList@ (...))
```

(See commit [0c410d333c6370d39b9f648c38dff2f505a5502e](#)).

When Do You Create a New Pass?

The parser consists of two main parts:

- a grammar
- a set of semantics hung off of the grammar

I am using the grammar to help us build the transpiler.

I start a new pass when I want encapsulation. The encapsulation provided by pure OO is not enough. Just creating new sets of semantics is not sufficient to

encapsulate all of the changes I need to make.

In particular, whenever I need to hack on the grammar to make things easier, I need to start a new pass. OO encapsulation won't save me here - I need to start with a simple grammar (e.g. Scm2JSBasic) and freeze my work. Then, I need to create new layers of grammars that perform specific functions for my transpiler.

Ohm-JS has a way to extend grammars in an OO-like manner, but this feature does not actually encapsulate the changes. The final result is a "flat" grammar that may contain hidden dependencies that make debugging harder.

It is much better (safer, less debugging) to use a hierarchy of composed grammars instead of inherited grammars.

I want to freeze what I've got. I don't want new changes to interfere with previously-correct grammars. The only way that I know of to accomplish this is to use *passes* (passes like pipelines in the UNIX® shell. This feature - composition - might be why pipelines became popular).

The List Constant changes could be done in the pass1 grammar, but, then, the layering Architecture would be lost. The grammar would become a ball of lint - collecting every change I've made, instead of laying out the changes in a layered manner.

PEG's backtracking leads to unexpected bugs when new changes are added to an existing grammar.

An example of the problem is seen when comparing the original pass1 grammar with the changes I added in pass2. Pass2 has more details. When trying to understand the grammar, one should begin by looking at the pass1 grammar. The changes for pass2 could have been added by extending the grammar via inheritance and overriding, but overriding can lead to hidden

dependencies and new bugs in previously working code. What I want is a freeze of what already works. Then, I want to compose new changes in layers over what already works. Overriding does not layer my changes - inheritance and overriding changes the final grammar and, possibly, the final result.

Paul Bennett's work on Frame-based software might be a direction to look into. Bennett's Frames are not the same concept as the concept called "frames" in older forms of AI.

Grammar Inheritance

Grammars for Scm2JSListConstants and Scm2JSEmitter are rewritten to inherit from Scm2JSBasic.

Note that this creates new "flat" grammars and does not, by itself, create new isolated passes.

Note that Scm2JSEmitter inherits from Scm2JSBasic and not from its immediate predecessor Scm2JSListConstants (as would be done if we used only OO instead of a pass-based strategy).

OO is essentially a fancy form of cut&paste - it creates new "flat" grammars by piling changes up on predecessor grammars. This applies to all forms of OO, not just to grammars. OO provides only *some* encapsulation, not full encapsulation.

Inheritance produces increasingly complicated software and does not provide a way to "freeze" changes.

I will use the word "isolation" to mean encapsulation that supports freezing.

Emitting JavaScript

I copy&pasted the emitter semantics and created a new semantics (for the same grammar) called 'emitjs'.

I modified the IF-THEN-ELSE code to emit JS `?:` expressions. (See commit `cc41021f1f3efe0b02598ba25020064470f95093`).

Functional JS

Lisp - and Scheme - does not have *statements*. Lisp only has expressions (called s-exprs) and special forms

Every expression returns a value. Always.

To stay true to the semantics of Scheme, I must emit JS code that is expression-based. This is possible in JS using *anonymous functions*. The emitted code is going to look horrible, but it will be a true representation of the original Scheme program. Later, after I see things work and can regression test, I can try to "optimize" some of the sequences, so that they don't as ugly.

For example, whenever the parser sees an atom, we emit a function.

If the atom is a number `N`, then the parser emits
`(function(){return N;})`

If the atom is a string, say `"abc"`, then the parser needs to emit:
`(function(){return "abc";})`

When the parser sees a function call (an s-expr, where the first item is a function and the rest of the items are arguments), it emits a function which evaluates the arguments and returns a function which calls the intended function, say `(fn a b c)` becomes:

`(function(){return fn(a(),b(),c());})11`

Below, is a list of all transformations I think I need:

```
atom x --> (function(){return x;})
SList (a b c) --> (function(){return a(fb(),fc());}) [where fb
and fc are functions]
special (and b c) --> (function(){return _and([b,c]);})
special (or b c) --> (function(){return _or([b,c]);})
special (not b) --> (function(){return (b() == false) ? true :
false;})
special (if x y z) --> (function(){return (x() != false) ? y() :
z();})
special (if x y) --> (function(){return (x() != false) ? y() :
null;})
special (cond ((a b c) (d e f))) --> (function(){return
_cond([[a,b,c],[d,e,f]]);})
special (let ((a b) (c d)) d e f) --> (function(a=b(),c=d())
{return d(),e(),f();})
special (quote symbol) -> (function(){return "symbol";}) [since
JS doesn't have symbols per se]
special (quote N) -> (function(){return N;}) [where N is a
number]
special (quote "abc") -> (function(){return "abc";})
special (quote #f) -> (function(){return false;})
special (quote #t) -> (function(){return true;})
special (quote (...)) -> (function(){return new List(...);})
```

I'm expecting to supply a small runtime library that defines functions `_and()`, `_or()` and `_cond()`. In Lisp, these things work like `&&` and `||` in JS. Each item is evaluated and evaluation *stops* as soon as one item is non-false.¹² It is imperative that unneeded arguments not be evaluated. In the JS version, I will pass an array

¹¹ N.B. for purists: Lisp intentionally does not specify an order for argument evaluation, so it is OK to let the JS compiler pick any order of evaluation. If this were not the case, I would have to add more code to ensure that arguments are evaluated in a specific order. If you don't understand the previous statement, don't worry.

¹² N.B. for purists: Lisp thinks that Booleans are NIL and anything-that-isn't NIL. Scheme reserves `#f` and `#t` for false and true, resp. The Lisp version of Booleans - NIL and not-NIL (a lot like C's NULL=0) leads to trouble, but has become a common idiom and cannot be ignored when dealing with libraries of real code.

of (unevaluated) functions to `_and` and `_or`. This is like how Smalltalk implements control flow - it passes unevaluated blocks (anonymous functions) into branching code. The code decides which branch to take and evaluates (runs) the block.¹³

Most of the mystique of compiler-writing is the act of learning, and coming up with, patterns that make the emitted code suck less.

It used to be that "suck less" meant creating emitted patterns that ran faster than their unoptimized versions.

We have learned, over the years, how to build compilers that are hot.

Thankfully, I am just building a transpiler - I am going to let the JS compiler(s) worry about optimizing my emitted code. I just want it to work, I don't need fast-ness. If I do create some optimizations (i.e suck less), it will probably have to do with making the emitted code more readable to humans.¹⁴ Fast-ness isn't my problem - it's the problem that competent Efficiency Engineers can tackle (once the transpiler is working).

I'll try implementing the above table, then see what breaks. Divide & Conquer...

In the rule `FunctionCall`, the original grammar specifies that the arguments ("actuals") are parsed as `ListItem`. In emitting this code, I want to append `()` to every argument (to effectively evaluate it), but only in function calls. The easiest way that I could think of to do this was to create a new rule `"Arg"` and use it instead of `ListItem` when parsing the arguments to a function. I am using the parser to tell me what the context is and, in the case of arguments to a function call (only) I append `()` to each list item.

¹³ If you didn't notice, I've just explained how CPS (Continuation Passing Style) works. CPS is the ultimate GOTO.

¹⁴ Actually, humans have no business looking at the emitted code. We don't normally look at the assembly code emitted. We used to look at the emitted assembler, but that urge has gone away, as compilers have gotten to be really good. Over time, I expect that no one will care to look at the code emitted by transpilers, such as this.

(See commit 9db14775d9acedfb187a33feaa8bf41b508e4617).

Waterfall

In software, it is never true that one can get it right the first time around.

If you think that you know what you are doing, then you are employing a technique called *the Waterall method*.

Even really smart people don't get it right most of the time - for example, refer to most existing multi-tasking libraries and the agony that these have caused.¹⁵

Workflow

The grammar IDE at <https://ohmlang.github.io/editor/> is useful, although it appears not to accept grammars that inherit from user-defined grammars (not a big loss, since inheritance applied to code serves to de-localize the code, making it more difficult to debug).

My suggested workflow becomes:

- write the grammar (aka the pattern matcher)
- test the grammar against various cases, using the grammar editor <https://ohmlang.github.io/editor/>, starting with simple cases, then escalating in complexity
- once the grammar is fully debugged, build the semantics object(s) associated with the grammar
- copy the output to a file and run node on it, to find further transpilation mistakes.

¹⁵ Currently, multi-tasking is considered "hard", Most of the problem is due to Accidental Complexity - the use of Time-Sharing (a hard problem) and the idea that memory needs to be shared (both, based on old-fashioned criteria).

Mapping Illegals to JS

Scheme allows many more characters as valid identifier characters than does JS.

At this stage in debugging, I want to test the emitted code using the node compiler.

The illegal characters must be mapped to valid JS character strings. This is done using the `.replace()` function and JS REGEXs.

See the function `doReplacements()` in `index.html` for a list of the actual replacements.

Refining Scheme to JS Emission

The JS emitter, as currently written, shows some weaknesses and bugs. Notably, the emitted code is extremely hard to read.

This chapter of the essay describes a second attempt at code emission in JS...

The semantics (meaning) of Scheme is that everything is an expression and, thus, everything returns a value. In Scheme, there are a few basic types. Just about everything else looks like a function (with the open-parentheses moved leftward).

The emitted JS code must evaluate to the same semantics as Scheme. Everything returns a value. This is not a problem for basic types (numbers, strings, booleans), but, JS also specifies several statements (like "if") that are evaluated for side-effects. In mapping Scheme to JS, we must emit JS code that returns values for such statements.

Thankfully, JS allows me to create anonymous functions. I will wrap anonymous functions around the statement types and invoke *return* inside of the functions. I will cause the functions to be evaluated by suffixing them with `()`.

I list the mappings below, as I understand them now...

```
atom --> atom
(fn ...) --> fn(...)
(if e th el) --> (function(){if e {return th;} else {return
el;}})();
(let ((a x) (b y)) d e f) --> (function(a=x, b=y)){ d; e; return
f; })()
(and a b c)
(or a b c)
(not x) --> !x
(cond (a b c) (d e f) (else g h i)) -->
  (function () {
    if a {
      b;
      return c;
    } else if d {
      e;
      return f;
    } else {
      g;
      h;
      return i;
    }
  }) ()
(cond (a b c) (d e f)) -->
  (function () {
    if a {
      b;
      return c;
    } else if d {
      e;
      return f;
    } else {
      return null;
    }
  }) ()
(a b c) -->
  a(b,c)
;; top level
(define g v) -->
  var g = v;
(define (f a b) c d) -->
  function f(a b) { c ; return d; };
(a b c) -->
```

Scheme syntax	Emitted JS
atom	atom
(fn a b c)	fn(a,b,c)

(if e thn els)	<pre> (function(){ if e { return thn; } else { return els; } })(); </pre>
(if e thn)	<pre> (function(){ if e { return thn; } else { return null; } })(); </pre>
(let ((a x) (b y)) d e f)	<pre> (function(a=x, b=y)){ d; e; return f; }()) </pre>
(and a b c)	<pre> (function() { if (a && b && c) { return true; } else { return false; } }) () </pre>
(or a b c)	<pre> (function() { if (a b c) { return true; } else { return false; } }) () </pre>
(not x)	!x
(cond (a b c) (d e f) (else g h i))	<pre> (function () { if a { b; return c; } else if d { e; return f; } else { g; h; return i; } }) () </pre>

(cond (a b c) (d e f))	<pre> (function () { if a { b; return c; } else if d { e; return f; } else { return null; } }) () </pre>
(a b c)	a(b,c)
;; top level	
(define g v)	var g = v;
(define (f a b) c d)	function f(a b) { c ; return d; };
(a b c)	a(b,c)

Again, I won't be doing any checking - I will let the JS compiler check the types for me. I continue to treat the parser as a glorified REGEX, not as a full-blown language builder. Future generations of the tool environment might allow my transpiler to insert file location / debug information to allow easier debugging by humans. Future IDEs might allow the insertion of BREAK statements into the generated code, much like what is currently done with IDEs+compilers+assembler today. LISP has some advanced features for this kind of insertion. I don't know enough about JS at present, to say whether JS does or does not have such features.

Writability vs. Readability

Ohm-JS documentation emphasizes the fact that grammars are "clean" and semantics have been moved out of the grammar¹⁶.

This makes sense for *readability*. The grammar and semantics have been created and checked by the compiler. A Maintenance Engineer can *read* the

¹⁶ S/SL has this property, too.

grammar and then refer to the semantics object.

This separation of semantics from grammar, though, makes *writing* more difficult, since one needs to refer to the grammar when writing semantic rules. For example, I have broken `CondExpression` into many parts. I have to ping-pong between the grammar and the semantics object to ensure that I have correctly named each semantic rule and to make sure that the arities of semantic rules match the grammar. I just changed the grammar to split `Cond` into more pieces - now, I find it difficult to ensure that the semantics object corresponds to the tweaked grammar.

Ideally, I should be able to choose whether I view grammar rules and the semantics rules in one eye-full, or, whether the grammar should be clean, with semantics rules moved elsewhere.

At least, the ohm-js compiler performs arity checks and raises error conditions if there is a mismatch between grammar and semantic rules.

Can this¹⁷ be significantly improved?

Greedy Matching

PEG, and ohm-js, uses *greedy* matching. It matches the longest possible phrase and doesn't turn back.

This means that we can write a rule, like:

```
CondExpressionWithElse = BEGIN lexical_COND FirstCondClause  
MoreCondClause CondElseClause END
```

but not a rule like:

¹⁷ the readability vs. writability conundrum

`StatementBlock = Statement* LastStatement`

The first pattern matches one CondClause then matches more CondClauses.

The second pattern matches a bunch of statements, greedily matching all statements, then I want it to back up and leave one Statement to match LastStatement. This back up doesn't happen and the pattern match fails.

The corrected version of the rule is

`StatementBlock = Statement+`

We can handle the "last" statement in the semantics, since ohm-js gives me an Iter node containing statements.

I would *prefer* to have the pattern matcher tell me which is the last statement, without having to write code to walk the Iter node chain.

Can we rewrite this? Maybe, using ohm-js' lookahead operation.

Maybe we can say something like:

```
StatementBlock = MidStatement* LastStatement
MidStatement = Statement &Statement
LastStatement = Statement ~Statement
```

?

I'll try it out in the editor using "(a) (b) (c)" as input and selecting (from the pull-down) the start rule as StatementBlock.

Note that the editor shows the input string at the top of the tree output window. For every &, it shows the input an extra time. For example when the above rules are use to parse "(a) (b) (c)", the editor shows two "(c)" even though the input contains only one "(c)".

Quoting Revisited

As I get further and try more detailed examples, I see some bugs in the quote expander code.

Firstly,

```
(define db '(5))
```

Comes out as

```
(define db (@newList@ "5"))
```

whereas it should be

```
(define db (@newList@ 5)).
```

This is a variant of the problem we solved in the Scm2JS grammar, and should be easy to repair. Thankfully, the quote expansion is isolated, so I know that changes to it will not affect what we've done so far.

See commit [8946e4f755f5b3083fcf0203c0d3246d0b8cd399](#) which fixes quoted lists of single atoms and a quoted null list. The result still has problems...

Next, we see (1) that the Scm2JS grammar treats the insides of quoted lists as function calls and (2) something looks odd for input `"(define db '(((edge a b)))`).

Actually, (1) is misdiagnosed. The emitter has left-over code that places `"()`" after every item when emitting for `SList_atnewlistat`. I need to remove the spurious `"()`". See commit [0ce488124f292152754348e030b78be3b449fb13](#) (now back on the 'master' branch, once again).

Now, (2) looks more reasonable, but still has spurious "function"s emitted. Again, this is just legacy code - easy to clean up and fix. See commit [6a5a268cccdc04c593c79983bd5d02ccf80160ed](#).

The emitted code is

```
var db = new List(new List(new List("edge","a","b")));
```

but the node compiler balks with the error that List is undefined.

Manually stubbing in List...

```
function List(...args) {  
  };  
  
var db = new List(new List(new List("edge","a","b")));
```

and the code compiles cleanly under node (put the code into "junk.js" and run node: "node junk.js").

Dotted Lists

There is but one instance of a dotted list in prolog-6.scm

```
(define db  
  '(((edge a b))  
    ((edge a f))  
    ((edge a g))  
    ((edge b c))  
    ((edge b d))  
    ((edge c d))  
    ((edge c e))  
    ((edge g h))  
    ((edge d h))  
    ((edge h e))  
    ((edge h f))  
  
    ((path (? A) (? B) ((? A) (? B)))  
      (edge (? A) (? B)))  
  
    ((path (? A) (? B) ((? A) . (? CB)))  
      (edge (? A) (? C))  
      (path (? C) (? B) (? CB)))))
```

I could handle the dotted list in the grammar, but, it is easier to change the code manually to remove the dotted list completely (and rewrite it to call a library function that I'll call `_cons`). I will write the replacement in Scheme and let the transpiler do the rest of the job...

```
(define db
  '(((edge a b))
    ((edge a f))
    ((edge a g))
    ((edge b c))
    ((edge b d))
    ((edge c d))
    ((edge c e))
    ((edge g h))
    ((edge d h))
    ((edge h e))
    ((edge h f))

    ((path (? A) (? B) ((? A) (? B)))
      (edge (? A) (? B)))

    ((path (? A) (? B) (_cons (? A) (? CB)))
      (edge (? A) (? C))
      (path (? C) (? B) (? CB))))))
```

Dissecting the Parse

I found that the parsing results from the ohm editor were misleading. I implemented my own tree viewer, using Lisp-like syntax. See commit [950c308f28e5a3360afc14fb6b9f171c4a4acb32](#).

Larger Tests

The new grammars now handle all of `prolog-6.scm`.

I view the JavaScript output using my favourite pretty-printer (emacs, JavaScript-mode, in my case).

I am manually eyeing the output backwards - starting with the final call to `prove6()`.

I see something suspicious in the transpilation of the function "resolve".

I have saved the output in `prolog-6.js`. Running "node¹⁸" on it shows an error at function "back5".

That gives me at least two errors to look at.

I will start debugging by looking at `back5`. I start with cutting `back5` out of the full test and making a test harness for it.

`Back5` transpiled and formatted, is:

```
function back5(l,g,r,e,n) {
  return (function(){
    if (
      (function(){
        ifpair_Q_(g) && pair_Q_(r) {
          return true;
        } else {
          return false;
        }
      })
    ) {
      return prove5(l,g,cdr(r),e,n);
    } else {
      return prove5(L_l(l),L_g(l),cdr(L_r(l)),L_e(l),L_n(l));
    }
  })
  ();
};
```

The first obvious problem is the `ifpair_Q_`. The branching AND (`&&`) needs to be wrapped in parentheses and, for readability, the "if" could use a space before the expression. Both of these problems can be addressed by adjusting the code

¹⁸ > node prolog6.js

for && (and ||).

Next, there is a problem with CondClauses - MoreCondClause is defined as CondClause*. Instead it should be defined as a CondClause and the * repetition is moved up into CondExpressionWithElse and CondExpressionWithoutElse. Now, each MoreCondClause will be prefixed with "} else if " instead of only the first MoreCondClause having this prefix. (See commit 49ee1506144edb8bbca68e796ad3b4a94cdcb318).

Next, another named let, that I overlooked, is in "lookup". I rewrote this manually.

It seems that the grammar does not handle LETs with no bindings. This appears in my manual rewrite of "print_frame_loop". Putting in dummy bindings (_xx and _yy) solves the immediate problem. Since this problem appears only in my rewrite, I choose not to repair the grammar (at this time).

I created references to cd*r().

Sweeping Changes

In debugging support.js, I hit a variable scoping problem.

The problem appears to go away if I use "let" instead of "var" in the transpiled code.

This is one of the advantages of using the DSL approach. This is a late change in strategy, but the fix is easy and all of the code is regenerated (transpiled).

Now, node complains that "db" is defined more than one. This is true - the original code in prolog-6.scm uses "db" as a global and relies on Lisp's ability to

redefine variables as the code is loaded. Possible solutions: (1) alter the .scm code to create multiple variables with different names, (2) allow multiple (define x ...) statements and transpile them variable assignments if the variable has already been defined (this would require making a table of top-level variables (globals), and checking it before emitting a top-level "let" definition, or ?.

First, it seems prudent to remove the global variable and turn it into a function parameter. This means going back to Scheme, checking that prolog-6.scm still runs, making the change to original Scheme code and seeing that it still gives the same answers...

Running Scheme (mit-scheme) on my macbook produces the output:

```
> scheme
```

```
1 ]=> (load "prolog-6")

;Loading "prolog-6.scm"...
p = (a f)

p = (a b c d h f)

p = (a b d h f)

p = (a g h f)

p = (a f)

p = (a b c d h f)

p = (a b d h f)

p = (a g h f)

y = bar
x = foo

y = baz
x = foo

y = foo
```

```

x = bar

y = baz
x = bar

y = foo
x = baz

y = bar
x = baz
;... done
;Value: #t

```

Next, I make a copy and remove the definitions for prove3 and prove5. I remove the tests for prove3 and prove5.

```

1 ]=> (load "prolog-6a")

;Loading "prolog-6a.scm"...
y = bar
x = foo

y = baz
x = foo

y = foo
x = bar

y = baz
x = bar

y = foo
x = baz

y = bar
x = baz
;... done
;Value: #t

```

That gets rid of the extra references to db. Db is still a global and is used once in the "else" clause of prove6. This use comes about because the pattern-matching "loop" needs to start from the beginning of the database. The mutual recursion between loop6 and back6 is actually a nested set of loops and, deep in the nested loops, the algorithm needs to start matching rules against the whole database. In this case, the recursion is a trick that covers up what is really going

on - loops nested in loops nested in loops - recursion isn't actually needed, except to appease Scheme and allergy sufferers - those who think that "loop" is a four-letter word. In mathematics(?) notation the algorithm would clearly show a nested set of for-all's (\forall). I'll rewrite the algorithm more clearly, later. For now, I'll pass "db" in as a parameter ("whole-db") while ensuring that it gets copied without change in prove6 and back6. See commit 42e75c5499e8e0df65d49ba0b52eb521fc15f1dd.

Quote List Expansion

I've revamped the emitjs semantics to output "cons(...)" instead of "new List(...)".

This is not correct, a quoted list should be converted to "list(...)", where list() inserts a null at the end of the list.

See commit 9ba41fe9151d5bde775852f573f19de3fcc9e251.

Note that was anti-waterfall development, I made progress without needing to get everything right at first. A placeholder was enough. I fixed the error, then iterated. Fixing the error was easy - a minor change to the .semantics() of the emitter.

Let*

The let* (LetSequential) is incorrect. It needs to be transformed into sequential Lets. There are at least 2 ways forward - (1) fix LetSequential, or (2) manually rewrite prolog-6a.scm to remove all let*s.

For now, let's take choice 2 (there is only one let* in prolog-6a.scm).

Function Copy()

The Copy() function is not working correctly.

The first clause returns "x" as an atom. There is no "return" statement emitted.

I will copy/paste it into my test-harness element and zero in on it.

I pare the test case down to:

```
(define (copy x n)
  (cond
    ((not (pair? x)) x)))
```

and look at "pass1 & listConstants & emitter tree".

The parse tree is:

```
(:Program
 (:GlobalFunctionDefinition
  (:Identifier
   (:symbol [[copy]] ))
  (:Identifier
   (:symbol [[x]] )),
  (:Identifier
   (:symbol [[n]] ))
  (:StatementBlock
   (:SequentialStatement
    (:LastStatement
     (:Statement
      (:Form
       (:SList
        (:SpecialForm
         (:CondExpression
          (:CondExpressionWithoutElse [toCL operation] ))))))))))))
```

I see 2 potential problems: (1) there is a comma between the args "x" and "n" and (2) the :CondExpressionWithoutElse is showing "[toCL operation]" instead of a value.

It looks like (1) was intentional, so I will focus on (2).

CondClauseWithoutElse was missing () after firstfirstCondClause.toCL. This is in the toCL operation, so the actual problem has not been repaired, but may be more visible.

Now, the parse tree is appears as:

```
(:Program
  (:GlobalFunctionDefinition
    (:Identifier
      (:symbol [[copy]] ))
    (:Identifier
      (:symbol [[x]] )),
    (:Identifier
      (:symbol [[n]] ))
    (:StatementBlock
      (:SequentialStatement
        (:LastStatement
          (:Statement
            (:Form
              (:SList
                (:SpecialForm
                  (:CondExpression
                    (:CondExpressionWithoutElse
                      (:FirstCondClause
                        (:CondClause
                          (:CondTest
                            (:Form
                              (:SList
                                (:SpecialForm
                                  (:NotExpression
                                    (:Bool
                                      (:Form
                                        (:SList
                                          (:FunctionCall
                                            (:Identifier
                                              (:symbol [[pair_Q_]] ))
                                            (:Arg
                                              (:ListItem
                                                (:Atom
                                                  (:atom
                                                    (:symbol [[x]] ))))))))))))))))
                                                  (
```

```
(:StatementBlock
  (:Atom
    (:atom
      (:symbol [[x]] ))))))) )))))))
```

[^ I put the output into emacs .lisp mode and pretty-printed it.]

The problem appears to be that grammar looks for "StatementBlock = Atom | SequentialStatement". The grammar is over-specified. I simply remove Atom and leave "StatementBlock = SequentialStatement". This now leaves Atom to be parsed by Statement and takes its rightful place as a MidStatement or a LastStatement.

Missing Support Routines

Append is missing. I will write one in Scheme, called Append_Inefficient, and replace all calls to Append by this - in the Scheme code.

N.B. I am now using the Scheme to JS transpiler. I will write AppendInefficient in Scheme and let the transpiler convert it to JS. There are some interesting trade-offs to be had when the transpiler begins to work (even hobbling). Should I write Append() in JS or in Scheme? Or, should I rewrite it in PROLOG? I'm not worried about efficiency (yet), and I am very comfortable writing Scheme. Writing it in Scheme will allow me to debug the implementation in a working compiler - MIT-Scheme.

Instead of dealing with varargs, I wrote AppendInefficient3 which is defined using AppendInefficient.

In a sense, I'm using Scheme as a meta-language...

Working

Now, the full test is working (compared to the Scheme output for `prolog-6a.scm`).

See commit `544c1d52c9278b1cb90724f6452408a8154fc750`.

Recap

We now have the JS code for a PROLOG interpreter. It consists of the `support.js` routines plus the output from our transpiler.

"Prove6" is the entry point.

We need to create a database of facts plus and rules and, a database of goals. We pass these to `prove6()` and it prints the results.

We need several simple things:

- `Prove6()` needs to return a list of answers instead of printing them out.
- A PROLOG-like syntax skin over JS, that converts PROLOG-like statements into databases and calls to `prove6()`. This is probably going to be another grammar (Ohm-JS).
- A way to combine chunks of PROLOG-like statements into JS. Each chunk will be fed into the above PROLOG-to-JS transpiler and the result will be pasted back into the JS program. When done, the final JS program will be fed to the JS compiler/interpreter. In its simplest form this will involve calls to `String.search()` and `String.split()` (etc.). Q: What do we do with the resulting JS code? Can we overwrite `<script>` segments in the DOM? Do we generate another .html page with new `<script>...</script>` segments? Do we generate a .JS file and feed it to `node.js`? The ideal would be a page, an .html file, that contains the mixed PROLOG and JS script, which invisibly transpiles the code and runs and produces a result. [This is the "hard part" for me. I don't yet have enough JS and HTML experience. Grammars are the "easy part" for me:-].

Deeper

Chapter 2 contains a good overview of Ohm:

<https://escholarship.mcgill.ca/concern/theses/j67316286>

OMeta: an Object-Oriented Language for Pattern Matching

<http://www.tinlizzie.org/~awarth/papers/dls07.pdf>

(N.B. Ohm is not OMeta, but Ohm is a descendant of OMeta

N.B. Ohm only understands how to parse characters

N.B. Ohm breaks semantics and grammar apart, whereas OMeta expects semantics to be a part of the grammar description)

OMeta: PhD Thesis

http://www.vpri.org/pdf/tr2008003_experimenting.pdf

OMeta website

<http://www.tinlizzie.org/ometa/>

Ohm examples

<https://cs.lmu.edu/~ray/notes/ohmexamples/>

Ohm github:

<https://github.com/harc/ohm>

article: Making your own programming language with NodeJS

<https://repl.it/talk/learn/Making-your-own-programming-language-with-NodeJS/45779>

Repl.it:

<https://repl.it/>

Ohm paper (and how it differs from OMeta):

<https://ohmlang.github.io/pubs/dls2016/modular-semantic-actions.pdf>

PubNub blogs about Ohm

1. <https://www.pubnub.com/blog/javascript-parser-ohm-makes-creating-a-programming-language-easy/>

2. (includes toAST): <https://www.pubnub.com/blog/build-your-own-symbol-calculator-with-ohm/>

3. (Meow language): <https://www.pubnub.com/blog/add-code-blocks-conditionals-to-ohm/>

[writing a compiler \(incl. toAST\):
https://cs.lmu.edu/~ray/notes/writingacompiler/](https://cs.lmu.edu/~ray/notes/writingacompiler/)

[A Compiler for Ael
\(good introduction - a bit of everything including code generators for JS, C and ASM\):
https://cs.lmu.edu/~ray/notes/aelcompiler/](https://cs.lmu.edu/~ray/notes/aelcompiler/)

Ideas

IDEAS

Ohm-JS does not parse anything but single characters.

In an ideal situation, we would want to parse "tokens".

Tokens would provide:

- separation between Syntactic entities - when we ignore spaces, we would still have (delimited) tokens, for example, in the scanner pass (the very first pass), we would recognize identifiers. If we had tokens, the scanner would emit 3 tokens for: `abc<space>def` - `token(identifier, "abc")`, `token(spaces, " ")`, `token(identifier, "def")`. In later passes, we could ignore spaces and still see distinct identifiers, e.g. `token(identifier, "abc")`, `token(identifier, "def")`. In the current version of Ohm-JS, the string `abc<space>def` is recognized as a single token, when spaces are ignored, e.g. `"abcdef"`. Maybe this problem just needs more thought...
- extra information, such as line#, character position and filename, e.g. `token(identifier, "abc", line=10, startPosition="3", filename="sample.txt")`.

The above could be accomplished by using distinguished strings, for example `"!!!<token identifier abc ... !!!>"`. The "problem" with this solution is that strings are not terminals and, hence, there is no automatic backtracking.

Maybe we could use Unicode characters instead of distinguished strings? For example `"† identifier abc ... ®"`. JS purportedly supports unicode characters, so, in theory, Ohm-JS should be able to parse them, with backtracking. Is this worth

it? Maybe the whole issue is moot?