The making of a JavaScript Lisp

Sam Auciello

01 May 2013

I. Background

In September of 2012 I began the process that would eventually lead to the creation of a language I've named Hot Cocoa Lisp. The language is a combination of JavaScript and Lisp. Here is a simple example Hot Cocoa Lisp program that prints out each element in a list.

```
;; create a new variable conaining a list of three words
(var words [ "yes" "no" "maybe" ] )

;; define a new function.
;; '#' here is the beginning of a lambda expression
(def print-list
     (# (word-list)
        (for (word word-list)
             (console.log word))))

(print-list words)
```

The complete documentation for programming in Hot Cocoa Lisp can be found at https://github.com/olleicua/hcl or in the appendix.

My original goal was to learn about programming language design and try designing a small language. Initially I expected the language I designed to be relatively simple and that my Plan of Concentration would be centered more around discussing the decisions that go into language design abstractly. Over time, the idea of a Lisp-like language that could run in the context of JavaScript increasingly appealed to me and I began to focus more on the implementation details than on abstract design ideas. I'm fairly pleased

with this decision for at least three reasons. First, working on a large programming project was a good way to refine my general coding and project management skills. Second, I feel much more confident that I've gained an understanding of the nature of programming language design, having looked at the practical challenges of making a real working language. Third, there is a much deeper sense of satisfaction in having made something that works than in talking about how things work in general.

About a year ago, I started to become quite interested in JavaScript, largely from watching lectures by Douglas Crockford such as "Douglas Crockford: The JavaScript Programming Language".[1] In his lectures and in his book, JavaScript: the Good Parts, Crockford describes a particular set of approaches to JavaScript programming that emphasizes functional programming with closures and prototypal inheritance.[2] It struck me that his approach to programming in JavaScript seemed quite powerful and unfortunately unique to JavaScript. It seemed unfortunate not because other languages ought to support this approach more fully, but because of the ways in which JavaScript's design has been compromised. JavaScript has a complicated history. It was designed in about two weeks and its standard was locked in place before there was a chance to revise it. For this reason, JavaScript contains many mistakes. I would argue one of its most significant problems is that its syntax was intended to mimic that of Java, a language with which it has very little in common. Java's syntax turns out to be quite unnatural for functional programming and, as a result, it can be quite difficult to realize the language's full potential. Crockford has called JavaScript Lisp in C's clothing. I see my project primarily as an attempt to put it back in Lisp's clothing.[3]

Transcompiling

One popular solution to the problems with JavaScript is to program in a language that compiles to JavaScript. This is sometimes referred to as transcompiling because, as opposed to traditional compiling where the output is a lower-level language, these compilers output a language with a similar level of abstraction. CoffeeScript is the most popular such solution.[4] Syntactically, CoffeeScript resembles a combination of Ruby and Python while semantically still being largely derivative of JavaScript. It manages to fix most of the major problems of JavaScript and it has the advantages of being a large, well supported project with an active community. I found that it had two major drawbacks. The first seems difficult to avoid: although the

code gets written in CoffeeScript, because the compiled output is executed as JavaScript, any errors will reference JavaScript line numbers—meaning that debugging requires the developer to still work with JavaScript. This can effectively eliminate much of the benefit of CoffeeScript. The second drawback is specific to the language: because of the way CoffeeScript's syntax uses whitespace for logical blocks and objects, you can run into ambiguities like the following:

```
// Desired JavaScript:
func_that_takes_2_objects({a: 1, b: 2}, {x: 7, y: 8});

// CoffeeScript that you might think will work but won't:
func_that_takes_2_objects
  a: 1
  b: 2,
  x: 7
  y: 8

// The above actually produces:
func_that_takes_2_objects({a: 1, b: 2, x: 7, y: 8});

// CoffeeScript that will work
obj1 =
  a: 1
  b: 2
obj2 =
  x: 7
  y: 8
func_that_takes_2_objects obj1, obj2
```

At best, this limits how functional your programs can be and creates an extra annoyance. At worst, the language becomes counterintuitive and forces the developer to spend time figuring out its nuances, thus violating the simplicity CoffeeScript strove to achieve in the first place. My project differs most significantly from CoffeeScript in how it approaches these two issues. It solves the second one handily by using the simplest, most cleanly unambiguous syntax available: Lisp-style S-expressions. I will talk about my attempt at a solution to the first problem shortly.

Another transcompiled language that is worth noting is LispyScript.[5] It deserves mention not because it is particularly popular; it is fairly obscure,

but it is at least superficially similar to my project. Both are transcompiled languages with the syntax of Lisp. The largest difference is in motivation. LispyScript was designed primarily to add a particular set of powerful functional features from Lisp to JavaScript. My language aims to provide a solution to the same problems CoffeeScript tries to address (i.e. the bad parts of JavaScript) while encouraging a clean, functional style. Additionally, because my first project goal was to learn about language design, I spent a lot of time on implementation, building up a set of general purpose tools for parsing where other projects might prefer a smaller, tighter parser tailored to the specific syntactic needs of the particular language. I also devised an attempted solution to the persistent problem of transcompiled languages needing to be debugged in JavaScript. It occurred to me that adding annotations of the original source code as comments in the compiled JavaScript might make debugging a little simpler. In practice, annotations helped a little, but the problem was compounded by a larger issue with my project. Because my language emphasizes a function coding style, it is not uncommon for code to contain long strings of chained function calls which are often complicated by syntaxes that require odd JavaScript constructs like function wrapping, the purpose of which may not be entirely apparent at first glance. This style can make for long strings of unintelligible code that may not be easy to map to the annotation above them. For example:

```
;; Hot Cocoa Lisp:

(def choose
     (# (m n)
        (if (or (= 0 n) (= m n)) 1
          (+ (choose (--1 m) n)
             (choose (--1 m) (--1 n))))))

// JavaScript:
choose = (function(m, n) {  return ((((0 === n)) || ((m === n))) ? 1 :
  (choose((m - 1), n) + choose((m - 1), (n - 1)))); });
```

Another solution to the debugging issue that is starting to gain traction is called source mapping[6], and it is an attempt to give browsers access to a mapping between the original source and the compiled JavaScript that can then be used to help developers pinpoint the source of errors in their code. I definitely think Hot Cocoa Lisp could benefit from source mapping approach, but it was beyond the scope of my project.

Previous design iterations

My first idea was to design a Lisp-like interpreted language implemented in JavaScript that could run in a browser. I had built up a much more complex type system for this based on prototypal inheritance most of which was eventually scrapped. I was discouraged from writing an interpreter by a variety of factors. One of the perceived advantages was that as long as there were no bugs in my implementation, I could have full control over runtime error messages and they could correspond to the Lisp source. The disadvantage of this is that I would need to fully implement error handling. Getting the idea to write a compiler that added annotations of the original source to the compiled source was the largest factor in abandoning the idea of writing an interpreter.

My next idea was to implement a very bare version of a language that included Lisp-style compile-time macros and then implement most of the language in macros. This idea was inspired somewhat by LispyScript.[5] The basic language would have only four built-in syntaxes:

- `macro` would define a new macro.
- `quote` would tell the compiler to simply print its argument as a literal data structure (probably wrapped in some way).
- `quasiquote` would tell the compiler to find any internal pieces of its argument of the form `(unquote foo)` and replace them with the result of compiling `foo`, then print it as a literal data structure (probably wrapped in some way).
- `js` would take a format string and a bunch of values to interpolate into it and instruct the compiler to print out that string.

As with the popular Java-based Lisp variant Clojure, `quote`, `quasiquote`, and `unquote` had the syntactic shortcuts `'`, `` ` ``, and `~` respectively.[7, clojure.org/reader]

The hope was that all other Lisp syntaxes could be built using macros. Implementing a language this way turned out to be more difficult than I had imagined. The macros had to be able to apply to one another and they needed to work by generating some JavaScript code and calling `eval()` on it in JavaScript which slows things down a lot. The logic of manipulating Lisp abstract syntax trees and raw JavaScript code, and keeping track of which

was which, turned out to be way more involved than I had foreseen.

I eventually decided to go the simpler route and just make a working compiler and consider adding macros later. In the end I abandoned macros and even quoting entirely. Without macros or delayed evaluation, quoting seemed to create unnecessary complications without adding any real power. I eventually changed the internal name of symbol tokens in the parser to identifiers to correspond to their behaving more like identifiers in JavaScript.

II. Hot Cocoa and Hot Cocoa Lisp

Hot Cocoa

Hot Cocoa is a set of JavaScript libraries for implementing languages that I built using the Node.js[8] command line interface. It includes a parsing system for determining the structure of code, a semantic analysis system for applying meaning to that structure, a templating system for generating compiled code, a system for organizing built-in functions, and a testing system for making sure that everything works. The project is hosted at https://github.com/olleicua/hot-cocoa. It can be included in a project using npm[9] with:

```
$ npm install hot-cocoa
```

Parsing

My parsing system was based on the discussion of parsing in Programming Language Pragmatics by Michael L. Scott. [10, ch. 2] The system works in two stages: scanning the original source for tokens and parsing a sequence of tokens into a parse tree. The scanner's purpose is to divide a string of raw code into the smallest possible meaningful pieces called tokens. Each token will be assigned a type by the scanner, such as operator, numeric literal, or identifier. My scanner API takes an array of token type objects and a string of input source code. The array of token types should look like the following:

```
var tokenTypes = [
  { t:'number', re:/^\d+/ },     // one or more digits
  { t:'word', re:/^[a-z]+/ },    // one or more letters
  { t:'whitespace', re:/^\s+/ } // one or more whitespace characters
];
```

In a loop, the scanner tries each of the regular expressions in the token type list until one matches at the beginning of the source code, creates a new token object of the associated type with the matched string as a value, and shifts the matched string off of the beginning of the source code. A token type with the name `whitespace` is special. By default `whitespace` tokens are omitted from the result. The above token type list could transform

```
foo 1 2 bar 3 baz qux 4
```

into

```
[ { type: 'word', value: 'foo' },
  { type: 'number', value: '1' },
  { type: 'number', value: '2' },
  { type: 'word', value: 'bar' },
  { type: 'number', value: '3' },
  { type: 'word', value: 'baz' },
  { type: 'word', value: 'qux' },
  { type: 'number', value: '4' } ]
```

Once a sequence of tokens is generated, they can be parsed into a tree. A parse tree is basically a way of organizing a sequence of tokens into a form that has meaningful structure using a context-free grammar. For example, a scanner could break up the string

```
'(foo bar ) ( baz qux )'
```

into tokens representing words and parentheses but a parser would be needed to determine that `foo` and `bar` are grouped together and that `baz` and `qux` are separately grouped together.

I implemented two different parsing algorithms with the same API. Recursive descent is the faster of the two, but CYK is guaranteed to work for arbitrarily ambiguous grammars in reasonably well bounded time ($O(n^3)$) so long as there is at least one valid parsing of the input token list.[11] My parsing API takes a context-free grammar formatted as a JSON object and a list of tokens and returns a JSON parse tree. Continuing the previous example, consider the following grammar:

```
var parseGrammar = {
  "_program": [
```

```
    ["_function", "_program"],
    []
  ],
  "_function": [
    ["word", "_arguments"]
  ],
  "_arguments": [
    ["number", "_arguments"],
    []
  ]
};
```

With the above sequence of tokens, this grammar would produce the following tree:

```
[ { "type": "_program", "tree": [
  { "type": "_function", "tree": [
    { "type": "word", "value": "foo" },
    { "type": "_arguments", "tree": [
      { "type": "number", "value": "1" },
      { "type": "_arguments", "tree": [
        { "type": "number", "value": "2" },
        { "type": "_arguments", "tree": [ ] }
      ] }
    ] }
  ] },
  { "type": "_program", "tree": [
    { "type": "_function", "tree": [
      { "type": "word", "value": "bar" },
      { "type": "_arguments", "tree": [
        { "type": "number", "value": "3" },
        { "type": "_arguments", "tree": [ ] }
      ] }
    ] },
    { "type": "_program", "tree": [
      { "type": "_function", "tree": [
        { "type": "word", "value": "baz" },
        { "type": "_arguments", "tree": [ ] }
      ] },
      { "type": "_program", "tree": [
```

```
        { "type": "_function", "tree": [
          { "type": "word", "value": "qux" },
          { "type": "_arguments", "tree": [
            { "type": "number", "value": "4" },
            { "type": "_arguments", "tree": [ ] }
          ] }
        ] },
        { "type": "_program", "tree": [ ] }
      ] }
    ] }
  ] }
] } ]
```

Semantic analysis

After the parsing organizes the code into a structure, the next step is to extract meaning from that structure using semantic analysis.[10, ch. 4] My semantic analysis system provides a mapping from the various node types in the tree structure (in this case `_program`, `_function`, `_arguments`, `word`, and `number`) to functions for handling them. For a simple interpreted language, these functions could return the program's output. For a more complex one like Hot Cocoa Lisp, they return a much simpler data structure called an abstract syntax tree that is isomorphic to the structure of Lisp syntax. For parsing a simple Lisp-like language, the abstraction of a parsing and semantic analysis library is not really necessary. A much simpler algorithm could have been used to generate the abstract syntax tree, but I enjoyed the exercise of building up the infrastructure, and I think it helped me to build a richer understanding of language implementation as well as API design.

Templating

When I realized that I was going to make a compiler, it occurred to me that I needed a templating system to format the compiled JavaScript source. My templating system mostly consists of a format function which takes a format string and a values object or array as arguments. Values are interpolated into the format string in place of `~TAGNAME~` where `'TAGNAME'` is a key in the values object. If no key is specified (i.e. '`~~`') then the key is the integer number of empty interpolations preceding this one. For example:

```
format("(~~) (~~) (~~)", [1, 7, 19]); // "(1) (7) (19)"
format(" *~stars~* _~underbars~_ ",
       { stars: "foo", underbars: "bar" }); // " *foo* _bar_ "
```

Function maps

I also made a system for organizing built-in functions that I called function maps. The basic idea was to have a JavaScript object that relates the name of a built-in function to a compilation function that generates JavaScript source for that function. In its most basic form, this compilation function can be defined by a format string. For example, the Lisp `if` function is simply defined by the format string:

```
'(~~ ? ~~ : ~~)'
```

The function map also keeps track of synonyms and provides a mechanism for associating properties with functions.

Testing

I also built a test system with two parts. The first is an API that takes an array of pairs (arrays with two elements). If the first of the pair is a function, then it is called inside of a try block, and its result or error message is used as the first value. The two values are then compared, and the test is considered passed if they are equal. The API then prints to standard out how many tests were passed and what was expected and gotten in any tests that failed. The second part of the system is an executable that recursively scans the current working directory and its children for files that match `**/tests/*.js` or `**/*.test.js`, executes them with Node.js, prints their output, and summarizes the number of tests tried and passed. The executable test script can be installed and run using:

```
$ npm -g install hot-cocoa
$ hot-cocoa-test
```

Hot Cocoa Lisp

I built the Hot Cocoa Lisp language on top of Hot Cocoa. Its project is hosted at https://github.com/olleicua/hcl and it can be installed using npm with:

```
$ npm -g install hot-cocoa-lisp
```

Programming in Hot Cocoa Lisp is a lot like programming in JavaScript. The biggest difference is that the function calling convention is changed to that of Lisp, so

```
my_cool_function(first_argument, second_argument);
```

becomes

```
(my_cool_function first_argument second_argument)
```

Additionally, as with Lisp, all of the syntaxes built into the language follow this structure, so

```
if (some_condition) {
  do_a_thing();
} else {
  some_other_thing();
}
```

becomes

```
(if some_condition (do_a_thing) (some_other_thing))
```

Here is an example program that generates the first twenty lines of Pascal's triangle:

```
;; memoize
(def memoize
     (# (func)
        (let (memo {})
          (# (args...)
             (let (json (JSON.stringify args))
               (or (get memo json)
                   (set (get memo json) (func.apply undefined args))))))))

;; choose w/ memoize
(def choose
     (memoize (# (m n)
                 (if (or (= 0 n) (= m n)) 1
```

```
                    (+ (choose (--1 m) n)
                       (choose (--1 m) (--1 n)))))))

(def max 20)

(times (row max)
     (times (col (+1 row))
      (process.stdout.write (cat (choose row col) " ")))
     (process.stdout.write "\n"))
```

Here is another example program that uses the Node.js library to set up a simple http server:

```
;; Transcriebd from from [[http://howtonode.org/hello-node][http://howtonode.org/hello

;; Load the http module to create an http server.
(def http (require "http"))

;; Configure our HTTP server to respond with Hello World to all requests.
(def server (http.createServer
             (# (request response)
                (response.writeHead 200 {"Content-Type" "text/plain"} )
                (response.end "Hello World\n"))))

;; Listen on port 8000, IP defaults to 127.0.0.1
(server.listen 8000)

;; Put a friendly message on the terminal
(console.log "Server running at [[http://127.0.0.1:8000/"][http://127.0.0.1:8000/"]])
```

There are many other details and the documentation for programming in Hot Cocoa Lisp can be found in the README.md file on GitHub: https://github.com/olleicua/hcl#hot-cocoa-lisp or in the appendix. What follows is the documentation and discussion of my implementation of the language.

Generating abstract syntax trees

The first step in compiling Hot Cocoa Lisp is generating an abstract syntax tree. The abstract syntax tree is a data structure that mimics the structure

of the Lisp syntax. In a language like Scheme this is what would result if the entire program were quoted. The abstract syntax tree is generated entirely by the tools in Hot Cocoa. First, the text is scanned and tokenized using the token type list defined in lib/tokenTypes.js. It is worth noting that tokens like `-1`, which could be interpreted as a number or an identifier, end up being numbers for the simple reason that the number token type is defined earlier in the type list. Next, the tokens are arranged into a parse tree using the grammar in lib/parseGrammer.json to give structure to the code and guarantee that there are no syntax errors. Finally, the parse tree is converted into an abstract syntax tree using the node transformations defined in lib/attributeGrammer.js. This final stage also makes use of lib/types.js to build wrapped abstract syntax nodes that, among other things, know how to print themselves. The result of the whole process is that some text like

```
(console.log "Hello World!")
```

is converted into a JavaScript object somewhat like

```
[
  [
    { type: 'identifier', value: '.' },
    { type: 'identifier', value: 'console' },
    { type: 'identifier', value: 'log }
  ],
  { type: 'string' value: '"Hello World!"' }
]
```

It should be noted that because a program can be made up of multiple top-level Lisp expressions, the first step of compilation actually produces a list of abstract syntax trees as opposed to a single tree.

Generating JavaScript code from abstract syntax trees

After the abstract syntax tree is generated, it can be converted to JavaScript code using a recursive algorithm in lib/compile.js to traverse the tree and do the following at each node:

- If the node is a list expression, check to see whether its first element is in the function map in lib/functions.js.

- If the first element is not in the function map, compile all of the other elements and output a JavaScript function call in the form:

  ```
  element_1(element_2_compiled, element_3_compiled, ...)
  ```

- If the first element is in the function map, check to see whether that function has the lazy property set to true. If it does, pass the remaining elements to the function to determine the JavaScript that should be generated. If it doesn't, first compile each of the remaining elements, then pass them to the function to determine the JavaScript that should be generated.

- If the node is not a list expression, simply output its value. If it is an identifier, run it through the mangling function first to escape any characters not normally allowed in JavaScript identifiers.

Some built-in functions can be accessed as primitives in the language without being called. For example:

```
(map [ 1 2 3 ] *2) ; [ 2 4 6 ]
```

This feature is implemented using the mapping between function names and implementations in lib/builtins.js. Any function names that are mentioned in a position other than the beginning of a list expression are defined at the top of the compiled JavaScript, so the above Hot Cocoa Lisp Program would compile to

```
var _times_2 = function(x){return x*2;};
map([1, 2, 3], _times_2); // [2, 4, 6]
```

Dependency management

The implementation also contains some tools for dependency management. A problem can arise in developing a Node.js project with Hot Cocoa Lisp components in separate files linked using `require` and the Node.js module system. If changes have been made to multiple files the command to compile and re-run the program can get prohibitively long, for example:

```
$ hcl dependency1.hcl && hcl dependency2.hcl && hcl -n main_program.hcl
```

As dependency chains get longer and more complex, the developer may need to either keep track of which dependencies have changed or have an exceedingly long and cumbersome compile command. To alleviate this problem somewhat, I have created a compile function in Hot Cocoa Lisp that takes a path to a .hcl file, compiles that file to a .js file and returns the path to the .js file. For example:

```
(var my-cool-library (require (compile "./path/to/library.hcl")))
```

The implementation is complicated somewhat by the fact that Node's module system allows for cyclic dependencies.[8, nodejs.org/api/modules.html] This is dealt with using the module defined in lib/file2js.js which keeps track of every .hcl file that has been compiled in this compilation and does the compilation only if that file has yet to be compiled.

External JavaScript libraries

The project makes use of a few external JavaScript libraries. The language's REPL (read-evaluate-print loop, sometimes also called an interactive prompt) is powered by Node.js's built-in REPL library, which provides the necessary hooks to override the evaluate part of the loop so that input is interpreted as Hot Cocoa Lisp instead of JavaScript.[8, nodejs.org/api/repl.html] It also provides an output hook that allowed me to remove commas and colons from the returned values so that output arrays and objects match the Hot Cocoa Lisp convention of whitespace delimiters. I used the optimist[12] Node.js library to parse command line options and the Uglify.js[13] library to provide an automatic minification option. I used underscore.js[14] extensively throughout my implementation to facilitate a functional style, and I also provided a mechanism for having underscore 1.4.3 automatically exposed to Hot Cocoa Lisp. When the `-u` flag is supplied, the minified version of underscore 1.4.3 in etc/underscore.js that has been slightly modified to avoid a particular interaction with Node.js's module system is included at the top of the compiled JavaScript and a bit of code is added to expose all of the methods in underscore at the top level so that the functional tools from underscore can be used without referencing underscore. For example, the `-u` flag allows:

```
(filter [ 0 1 2 0 3 ] zero?) ; [ 0 0 ]
```

instead of needing

```
(var _ (require "underscore"))
(_.filter [ 0 1 2 0 3 ] zero?) ; [ 0 0 ]
```

Testing

I used the Hot Cocoa test system to make sure everything was still working while I added features and changed code. My tests fall into three categories:

- The parse tests in tests/text2ast.js and tests/text2astRD.js make sure that my parser is generating abstract syntax trees the way I expect it to using both parsing algorithms.
- The compile tests in tests/compile.js check that a variety of short sample code fragments compile and run properly. This is more like a sanity check to make sure that the language is still doing what it should.
- tests/full.js recompiles and runs all of the .hcl files in the examples directory and compares their output to the associated .out file in the examples directory.

Between tests/compile.js and tests/full.js, all of the basic features of the language are tested.

III. Successes and failings

Meta-programming features

I originally imagined that my project would be a real Lisp with things like quoting, delayed evaluation, and macros but the idea got lost along the way. Most of the popular advantages of Lisp are missing. In a real Lisp, code and data have the same structure with the only difference being whether or not a thing is being evaluated. This has powerful consequences for introspection and meta-programming. My language has none of these things. What my language *does* have is the syntactic structure of Lisp which has the effect of promoting a functional coding style. I consider that to be a qualified success. The goal was to put JavaScript back in Lisp's clothing, and I did just that.

Type coercion

JavaScript has many problems, which have been enumerated far better by Crockford than I can do here.[15] My project provides a reasonable solution

to most of the important ones but one in particular seemed like more effort than it was worth to fix. Many of JavaScript's operators do type coercion when faced with incompatible types which can have some very surprising results. For example:

```
> "$" + 1 + 2
'$12'
> "7" * 5
35
> 0 == ""
true
```

Some humorous examples of this phenomenon can be found in the short presentation "WAT".[16] Short of carefully redefining every operator in the language to correctly check the types of its operands at run time, there wasn't much I could do. Since these issues don't show up very often in well organized code, it would be somewhat excessive to replace every instance of `a + b` with:

```
(typeof(a)==='number'&&typeof(b)==='number'&&!isNaN(a)&&!isNaN(b))?
 a+b:throw new TypeError('addition of two values that are not both numbers')
```

I did fix a few things, though. I made certain that the `=` equality function was implemented using the JavaScript `===` operator, which, unlike `==`, doesn't do type coercion—so `(= 0 '')` properly returns false. I made sure that the functions for determining the type of a value properly give arrays the type `'array'`, the value `null` the type `'null'`, and the value `NaN` the type `'nan'`.

Identifiers

A lot of the interesting design decisions that I found myself making had to do with implementation of identifiers in the language. As I've mentioned, identifiers in Hot Cocoa Lisp are importantly not values that can be bound and evaluated like symbols in Lisp. They only do what JavaScript identifiers do. They do differ from JavaScript identifiers in what symbols are allowed in them and they are represented in the compiled JavaScript using an escape mechanism based on underscores. I discovered a problem that can arrise when making use of an external library written in JavaScript. If a global variable is defined in the external library with underscores in its name, it cannot be access normally. For example:

```
// JavaScript
a_global = { ... };
```

Attempting to access the variable with `a_global` will fail because the mangling system translates that to `a__global`. It is always possible to access the variable via the global object with something like `(get global 'a_global')` in Node.js or `(get window 'a_global')` in a browser, but this solution seemed somewhat inelegant. I added the built-in function `from-js`, which checks that its argument is a valid JavaScript identifier at compile time (throwing an error if it isn't) and prints the identifier's value verbatim. This still seems like a somewhat inelegant solution because it requires the developer to know an obscure detail of the language's implementation in a situation that is likely to come up fairly often. I couldn't think of any other solutions short of implementing variable assignment in some way other than with normal JavaScript variables.

Another issue that cropped up with identifiers was with the built-in function `-1`. In many versions of Lisp, there are built-in functions called `1+` and `1-` that add one and subtract one from their argument respectively. I always found these names confusing because `(1- 10)` looks like it should be -9, not 9. I decided to change the convention to `+1` and `-1` to better fit my intuition. The problem with my idea is fairly obvious in retrospect. `-1` is also a number, and because of the way my scanner is set up, anything that could be a number or an identifier is a number. I didn't notice the issue at first because my compiler wasn't checking whether the first element of a list expression was an identifier before seeing if it was in the function map. I noticed the issue when I tried to make the `-1` function accessible in positions besides the beginning of a list expression which would be useful for things like:

```
(map [ 2 3 4 ] -1) ; [ 1 2 3 ]
```

I decided to compromise and call the subtract one function `--1` instead. I'm not sure this is the most elegant solution, but I didn't want to give up on making the naming convention more intuitive. In retrospect, it may not have been worth it. In any event, it was really nice to get a practical look at why the designers of Lisp set things up the way they did.

Functional compiler design

One of the biggest successes of the project was seeing a functional coding style pay off in building the compiler. I used a recursive method which I found quite powerful for traversing the syntax tree. I was able to keep track of things above the current node of the syntax tree using a context object that I passed around in a monad-like way. Using closures, I was able to set up handlers to instantiate variables at the beginning of each scope so that variable creation would always have access to a handler for instantiating that variable at the top of the innermost scope.

IV. Possible future improvements

Fractions

A pretty straightforward thing to add to the language at this point would be fractions. In Common Lisp, it is possible to say something like `3/4` and have it be interpreted as a fraction. For example:

```
;; Common Lisp
(+ 2 1/2) ; 5/2
```

All that would be needed to add fractions to Hot Cocoa Lisp would be to instruct the scanner to recognize tokens of this form as numbers and add some code to make sure that numbers of this form are always printed with parentheses around them. Unfortunately, adding fractions in this way wouldn't fix the problem of JavaScript numbers always being IEEE doubles, which has the effect of making `0.1 + 0.2` not equal to `0.3`:

```
;; Hot Cocoa Lisp
(+ 0.1 0.2) ; 0.30000000000000004
(+ (/ 1 10) (/ 1 5)) ; 0.30000000000000004
```

Dependency linkage

One feature I strongly considered adding but didn't have time to make work properly was a system for linking dependent source files that would work like `require` in Node.js, but in browsers. The basic idea was to paste the contents of the required file into the requiring file in something like the following way:

```
;; Hot Cocoa Lisp
(var foo (require "foo"))
```

```
// JavaScript
var foo = (function() {
  var exports = {}, module = {};
  module.exports = exports;

// contents of file foo here..

  return module.exports;
}).call(this);
```

I would need to carefully consider the details but this could potentially be a much cleaner way to do dependency linkage in browsers.

Browser support

The compiler currently has a browser mode that can be invoked using the `-b` flag. Currently all that the flag does is make sure that the source is wrapped in a `(function() \{ .. \}).call(this)` to avoid accidentally polluting the global namespace and make sure that the prototypal inheritance built-in function will work in environments where `Object.create` has not been predefined. I did, however, build in a system whereby browser-specific implementation details can be put in a separate function map in lib/browser.js that overrides any functions in it when in browser mode. Currently all of the programs in the examples directory can be compiled with the option and will run perfectly in Firefox 20.0. If Hot Cocoa Lisp were to be used seriously in web development, it would need to be made to support a wider variety of browsers which would involve a lot of browser compatibility testing but the infrastructure to make it work in the compiler is there.

Meta-programming features

It might still be possible to add macros to the language. True Lisp macros like those found in Scheme or Common Lisp allow the developer to define new syntaxes that work like new function definitions but give full control over what is evaluated when. When a function is evaluated all of its arguments are evaluated first. When a macro is evaluated, none of its arguments are evaluated, and the macro definition through `quasiquote` and `unquote` determines which arguments get evaluated, when by building up and return-

ing the code that will be evaluated.

LispyScript has macros that look like Lisp macros, but they behave more like C preprocessor macros in that they instruct the compiler to rewrite some source code before generating the compiled code. In Common Lisp and Scheme, macros have the full power of the language. LispyScript macros, like C macros, basically just give the ability to interpolate arguments into some predefined code. Here is a simple example from the LispyScript documentation:[5, lispyscript.com/docs/macros]

```
(macro array? (obj)
  (= (Object.prototype.toString.call ~obj) "[object Array]"))
```

The tilde here is used to interpolate the argument into the code. I've considered adding a similar macro system to my language but I'm still not entirely convinced that it adds enough useful functionality to the language that isn't already available through functions. Adding the sort of macro that is found in Common Lisp or Scheme would be significantly more complex, as it would necessarily require that quoting and delayed evaluation be a part of the language.

V. Conclusion

I see my project as a success because, as I intended to, I have gained a significant understanding of the tools and pitfalls of programming language design and implementation. I began by building up a practical understanding of parsing and the process of generating a parse tree from a sequence of tokens and a context-free grammar. Working through the implementation details of two different parsing algorithms was quite helpful in creating an intuition for syntax and the structure of language. Writing a compiler gave me a good appreciation for the sorts of practical concerns that go into language design and implementation.

Writing a program that writes programs forced me to think abstractly about programming. Being required to simultaneously consider the code that is being generated and the code that is doing the generating was both intimidating and mind-expanding. This sort of abstract programming really forced me to organize my thoughts and properly separate the various concerns with small testable functions. I'm glad that I chose this project because I feel that improving my understanding of the underlying nature of programming

languages will have benefits in all aspects of programming by giving me a better intuition for how the tools that I use work and how to use them effectively.

Many of the skills that I gained and honed in this project were not specific to the nature of the project. Tools like revision control, test-driven development, and package management are useful in any large project. I've become very comfortable working with git and GitHub and developed very good habits that give me the peace of mind of knowing that any change I make can easily be reverted. I also got some useful experience working with GitHub's markdown rendering system. I learned the ins and outs of npm, the package manager for Node.js. I created and published two packages (one depending on the other) and got some helpful exposure to the world of version numbers and dependencies.

Working on a large project by myself forced me to learn a lot (mostly the hard way) about project management. The principles of iterative design were a big part of the process. Knowing when to scrap an idea and start over is extremely important to any large project, and although it was often painful at the time to let go of ideas that weren't working out, I'm really glad to have gotten hands-on experience forcing myself to do this. I also ran into some trouble with what game designers sometimes call "complexity creep". In game design, there is sometimes a desire to add more moving parts to a game. This can cause problems because even if all of the ideas are good on their own, the combination of too many of them makes the game too complex and thus less enjoyable for the players. Because all of the ideas seemed good on their own, the complexity creeps up on the designer. In language design, there is less reason to limit complexity. More moving parts often just means more options for developers which is usually a good thing. However, in project management, more moving parts means more things that can go wrong. It's important to keep things as simple as possible to avoid complications. I found that I could dream up features significantly faster than I could implement them, and while none of them seemed prohibitively complex on their own, all of them turned out to be too much and I had to simplify.

In some ways, the most important thing I gained from the project is the easiest to express. The sense of accomplishment that comes with finishing a large project is a wonderful thing. I can now honestly say that I've invented my own programming language (even if it is just JavaScript with more parentheses).

References

[1] Douglas Crockford: The JavaScript Programming Language
http://www.youtube.com/watch?v=v2ifWcnQs6M
You Tube,
Douglas Crockford
accessed April 2013.

[2] Douglas Crockford
JavaScript: The Good Parts
O'Reilly, YAHOO! Press,
1st Edition, 2008.

[3] JavaScript: The World's Most Misunderstood Programming Language
http://www.crockford.com/javascript/javascript.html
Douglas Crockford,
accessed April 2013.

[4] CoffeeScript
http://coffeescript.org/
Jeremy Ashkenas,
accessed April 2013.

[5] LispyScript
http://lispyscript.com/
Santosh Rajan,
accessed April 2013.

[6] Introduction to JavaScript Source Maps
http://www.html5rocks.com/en/tutorials/developertools/sourcemaps/
HTML5 Rocks tutorials,
Ryan Seddon,
published March 21, 2012,
accessed April 2013.

[7] Clojure
http://clojure.org/
copyright Rich Hickey,
accessed April 2013.

[8] Node.js
http://nodejs.org/

Joyent Inc,
accessed April 2013.

[9] NPM
https://npmjs.org/
created by Isaac Z. Schlueter,
accessed April 2013.

[10] Michael L. Scott
Programming Language Pragmatics
Morgan Kaufmann Publishers, Elsevier,
3rd Edition, 2009.

[11] To CNF or not to CNF? An Efficient Yet Presentable Version of the CYK Algorithm
http://www.informatica-didactica.de/cmsmadesimple/index.php?page=LangeLeiss2009
Informatica Didactica,
Martin Lange, Institut für Informatik
Hans Leiß, Centrum für Informations- und Sprachverarbeitung
Ludwig-Maximilians-Universität München, Germany
accessed April 2013.

[12] node-optimist
https://github.com/substack/node-optimist
created by James Halliday,
accessed April 2013.

[13] Uglify.js
http://lisperator.net/uglifyjs/
created by Mihai Bazon,
accessed April 2013.

[14] Underscore.js
http://underscorejs.org/
created by Document Cloud,
accessed April 2013.

[15] Douglas Crockford's Javascript
http://javascript.crockford.com/
Douglas Crockford,
accessed April 2013.

[16] WAT
http://www.youtube.com/watch?v=kXEgk1Hdze0
You Tube,
Gary Bernhardt
accessed April 2013.

Links

My code

- The project hosting for Hot Cocoa, my implementation library:
  https://github.com/olleicua/hot-cocoa
- The project hosting for Hot Cocoa Lisp:
  https://github.com/olleicua/hcl
- The documentation for programming in Hot Cocoa Lisp:
  https://github.com/olleicua/hcl#hot-cocoa-lisp

Other links

- A lecture by Douglas Crockford about JavaScript:
  http://www.youtube.com/watch?v=v2ifWcnQs6M
- A collection of Douglas Crockford's discussion of JavaScript:
  http://javascript.crockford.com
- A humorous video about type coercion in JavaScript:
  http://www.youtube.com/watch?v=kXEgk1Hdze0