Hot Cocoa: A Plan of Concentration

Sam Auciello

01 May 2013

Sponsor: Jim Mahoney
Outside Evaluator: Brandt Kurowski
Additional Reader: Matt Ollis

Work to be Evaluated with Proportional Weights

Field of study: COMPUTER SCIENCE/Programming Languages

A paper and coding project on programming language design and compilation.	50%
Implementing a computational approach to a combinatorial design problem.	20%
Three Computer Science exams in algorithms, programming languages, and internet technology.	30%

Copyright note

Components

The making of a JavaScript Lisp (page 4)

In this essay, I describe my experience learning about programming language design by designing and implementing a Lisp-like language that compiles to JavaScript called Hot Cocoa Lisp. I discuss the successes and failures of the project and the decisions that I made.

Exams (page 29)

In these three exams, show competence in algorithms, in programming languages, and internet technology. The exams were each completed in between a week and ten days between February and April of 2013.

Computational approaches (page 78)

In this paper, I discuss a combinatorial design problem with relevance to experiment design and several computational approaches I have created for solving it.

Appendix (108)

I have included relevant pieces of code from my projects in the appendix.

The making of a JavaScript Lisp

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 returning 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

Exam on Algorithms

I completed the following exam during the week of February 19, 2013. The code it refers to can be found in the files that suppliment this document under exams/algorithms/.

Question 1:

```
Pick any two of the following three problems to analyze:

(i)  sorting a list of numbers

(ii) the partition problem: given a multiset (a set-like construction
that allows multiple copies of the same thing) of integers, is there
a way to partition it into two parts such that the sum of integers in
each part is the same?

(iii) The convex hull problem: given a set of points (x[i], y[i]),
find that list of points that's on the extreme outer "edge", which
form a convex polygon containing all the other points.

For each of the two problems that you choose, pick an algorithm to
examine and answer the following questions. (In order of decreasing
impressiveness, you may choose a good one that you already know and
understand, invent one, or search the literate. In any case, be
explicit about where your algorithm came from, and explain it
clearly.)

(a) Describe what you expect the O() run time to be for the
algorithms, and explain why and and on explictly what type of
problems (e.g. worse case, average case, etc).

(b) Implement and test the algorithms in a language of your choice.
(Do *not* use built-in or library routines for e.g. sorting.)

(c) Run your code on various size data sets that you generate,
recording the number of steps (however you choose to define that) the
algorithms take to run. Show explicitly with a plot of the data from
this "numerical experiment" that the O() behavior is as expected.
```

I chose to look at problems (i) and (ii).

Sorting

I've implemented from memory the following three $O(n \log n)$ sorting algorithms in C:

- Quicksort (in place)
- Mergesort
- Heapsort

Quicksort

Quicksort starts by arbitrarily choosing an element of the list and calling it the pivot. It then swaps the other elements of the list around the pivot until all elements less than the pivot are before the pivot and all elements greater than the pivot are after the pivot. It then recursively calls quicksort on the sublists before the pivot and after the pivot. The base case is lists of length less than 2 which are necessarily already sorted. My implementation operates in place and just for fun uses the XOR swap trick that is probably optimized away by the compiler anyway to make it theoretically extra in-place. Since each number must be visited approximately once in each level of recursion (in different branches) and there are $\log n$ levels of recursion because the size is approximately divided in two at each level, the algorithm should run in $O(n \log n)$ time.

Mergesort

Mergesort starts by dividing the list into two equally sized sublists and recursively calling mergesort on each of them. It then merges the two sorted lists together by taking the lower of the first elements of each list until both lists are empty. As with quicksort the base case is lists of length less than 2. As with quicksort, each number must be visited once in each level of recursion and there are $\log n$ levels of recursion so the algorithm should also run in $O(n \log n)$ time.

Heapsort

unlike mergesort and quicksort, heapsort is not recursive. Heapsort makes use of a datastructure called a heap which is a binary tree in which each node is greater than or equal to it's parent. Adding an element to a heap is a $\log n$ operation because the new element must be added at the bottom of the tree and might need to be compared to (and possibly swapped with) each of its ancestors all the way up the tree to find a valid placement. Removing the smallest element of a heap is similarly a $\log n$ operation because the root element must be swapped to the bottom and then the new root needs to be compared to (and possibly swapped with) it's decendants until if finds a valid placement. Heapsort simply takes the initial list and inserts each element into a heap then produces the sorted list by taking the smallest number from the heap until it is empty. Since we need two $\log n$ operations for each element of the list the algorithm should run in $O(n \log n)$ time. There is an efficient way to store a heap in as an array; the values from the tree are simply read out top to bottom left to right into the array. The parent of a node in the array is at the index $\lfloor \frac{i-1}{2} \rfloor$ where i is the index of the node and the children of a node are at $2i + 1$ and $2i + 2$ respectively. According to wikipedia, it is also possible to run heapsort in place by storing the in this manner in the same place as the initial list. In my implementation I used a separate scratch array to store the heap.

Numerical experiment

My code can be tested with:

```
$ cd sorting
    $ gcc -Wall test.c -o test && ./test
```

I've sumarized the results below:

```
size: 10
n*log_2(n): 30
quicksort: 23
mergesort: 34
heapsort: 15

size: 20
n*log_2(n): 80
quicksort: 59
mergesort: 88
heapsort: 61

size: 30
```

```
n*log_2(n): 120
quicksort: 143
mergesort: 148
heapsort: 110

size: 40
n*log_2(n): 200
quicksort: 245
mergesort: 216
heapsort: 161

size: 50
n*log_2(n): 250
quicksort: 283
mergesort: 286
heapsort: 212
```

These numbers should be regarded as approximate due to my somwhat arbitrary choice of where to increment the counter. All three algorithms clearly grow with proportion to $n \log n$.

The Partition Problem

I've implemented two versions of a brute force solution to this problem, one in C and and one in Ruby.

In C

My solution in C simply enumerates the powerset of the input multiset until it either reaches the end or finds a solution. It does this in a particularly concise manner by taking advantage of the isomorphism between the powerset of a multiset of size n and the binary expansions of the set of integers that satisfy $0 \leq i < 2^n$. My program simply enumerates the integers from 0 to 2^n, checks to see if the partitioning resulting from that integer has equal sums and returns that integer if it does.

My algorithm should run in $O(2^n)$ time since it is literally enumerating the powerset. It has the additinal disadvantage of being limited to input multisets no bigger than the log of the largest integer that can be trivially stored in my architecture. In this case 63 since my implementation requires the integer to be signed to allow negative numbers to signify no partitioning having been found. I've run my code on the prepared lists of orders 5, 10, 20, and 50. I specifically used a mix of lists that can and cannot be partitioned to give a better sense of the algorithms behavior. As one might expect, the algorithm runs significantly faster when it finds a solution:

```
$ cd partitions
    $ gcc -Wall brute.c -o brute && ./brute
    [ 53, 24, 20, 81, 90 ]
    [ 53, 81 ] [ 24, 20, 90 ]
    2^n:32
    steps:10

    [ 59, 54, 88, 86, 64 ]
    no partitioning exists
    2^n:32
    steps:32

    [ 53, 55, 74, 44, 66, 1, 92, 34, 42, 57 ]
    [ 53, 55, 74, 1, 34, 42 ] [ 44, 66, 92, 57 ]
    2^n:1024
    steps:424

    [ 2, 81, 10, 66, 50, 43, 22, 80, 89, 67 ]
    no partitioning exists
    2^n:1024
    steps:1024

    [ 0, 0, 13, 0, 36, 40, 23, 38, 38, 25, 53, 37, 85, 10, 51, 1, 74, 71, 33,
      34 ]
    [ 13, 36, 40, 38, 38, 25, 53, 37, 51 ] [ 0, 0, 0, 23, 85, 10, 1, 74, 71,
      33, 34 ]
    2^n:1048576
    steps:20405

    [ 87, 68, 97, 55, 26, 48, 62, 48, 43, 80, 75, 13, 25, 62, 76, 39, 3, 80,
      37, 7 ]
    no partitioning exists
    2^n:1048576
    steps:1048576
```

```
[ 67, 78, 63, 45, 55, 13, 7, 75, 37, 84, 68, 68, 54, 4, 9, 90, 70, 58, 39,
  1, 19, 13, 33, 59, 88, 14, 98, 83, 69, 17, 87, 61, 81, 21, 64, 19, 8,
  71, 6, 34, 88, 8, 17, 69, 57, 94, 47, 57, 4, 65 ]
[ 67, 78, 63, 45, 55, 13, 7, 75, 37, 84, 68, 68, 54, 9, 90, 70, 58, 39,
  19, 33, 88, 98 ] [ 4, 1, 13, 59, 14, 83, 69, 17, 87, 61, 81, 21, 64, 19,
  8, 71, 6, 34, 88, 8, 17, 69, 57, 94, 47, 57, 4, 65 ]
2^n:1125899906842624
steps:89645056
```

In ruby

My solution in ruby uses a less technically efficient recursive backtracking search and simply stores the partitioned state using two smaller lists. It makes up for these inefficiencies with two improvements. First it sorts the intial list so that larger numbers will be tried first and then it checks the sums of the two smaller lists at each step. If the difference between the two smaller sums is greater than the sum of the remaining unassigned numbers then there is no solution below this point and the search can backtrack. I used a ruby monkey patch to allow arrays of numbers to keep track of their sums internally as numbers are moved between them to avoid the overhead of recalculating the sum each time. Because it is a backtracking search that goes to a maximum depth of n with a branching ratio of 2 it should run in $O(2^n)$ time. Numerical exeriment indicates that my improvement is cutting this down significantly. My recursive search function is being called roughly between $\frac{2^n}{10}$ and $\frac{2^n}{100}$ times even when no solution is found:

```
$ ruby brute.rb
size: 5
2^n: 32
[ 561, 778, 931, 344, 719 ] can not be partitioned
steps: 13
[ 151, 278, 115, 396, 512 ] can not be partitioned
steps: 9
[ 122, 962, 821, 662, 414 ] can not be partitioned
steps: 9
[ 863, 941, 245, 256, 296 ] can not be partitioned
steps: 13
[ 670, 315, 177, 915, 997 ] can not be partitioned
steps: 5
 ---
```

```
    size: 10
    2^n: 1024
    [ 284, 544, 463, 649, 761, 120, 501, 484, 92, 41 ] can not be partitioned
    steps: 113
    [ 461, 695, 579, 443, 284, 624, 871, 319, 517, 874 ] can not be partitioned
    steps: 263
    [ 306, 321, 873, 572, 999, 358, 569, 619, 597, 957 ] can not be partitioned
    steps: 255
    [ 514, 758, 276, 603, 938, 219, 840, 622, 36, 120 ] can not be partitioned
    steps: 105
    [ 469, 871, 605, 731, 691, 68, 610, 966, 544, 193 ] can be partitioned as
[ 966, 691, 605, 544, 68 ] [ 871, 731, 610, 469, 193 ]
    steps: 131
     ---
    size: 20
    2^n: 1048576
    [ 61, 764, 989, 36, 791, 271, 456, 274, 61, 672, 948, 82, 677, 605, 554,
  314, 816, 949, 714, 225 ] can not be partitioned
    steps: 37827
    [ 644, 129, 214, 119, 378, 552, 705, 539, 611, 943, 222, 340, 173, 32,
  312, 911, 689, 526, 330, 568 ] can not be partitioned
    steps: 61741
    [ 834, 16, 154, 318, 901, 645, 506, 751, 496, 865, 733, 935, 779, 293,
  376, 949, 220, 626, 269, 641 ] can not be partitioned
    steps: 64287
    [ 618, 725, 998, 999, 520, 260, 409, 663, 731, 424, 889, 580, 582, 620,
  548, 659, 953, 422, 712, 951 ] can not be partitioned
    steps: 179453
    [ 485, 710, 299, 615, 412, 151, 630, 333, 373, 195, 78, 887, 920, 666,
  783, 256, 318, 465, 791, 766 ] can not be partitioned
    steps: 87827
     ---
    size: 25
    2^n: 33554432
    [ 68, 767, 397, 164, 873, 994, 135, 246, 708, 329, 172, 612, 352, 370,
  820, 841, 31, 36, 722, 477, 961, 18, 85, 66, 285 ] can not be partitioned
    steps: 618285
    [ 482, 566, 270, 370, 714, 643, 337, 340, 939, 112, 943, 367, 199, 789,
  320, 19, 727, 390, 440, 607, 323, 216, 878, 987, 721 ] can not be partitioned
    steps: 1715443
    [ 48, 552, 458, 16, 207, 749, 248, 359, 265, 505, 393, 731, 941, 981, 562,
266, 476, 555, 671, 501, 504, 909, 188, 953, 929 ] can not be partitioned
    steps: 1300743
    [ 395, 165, 443, 767, 513, 590, 613, 518, 514, 70, 328, 887, 679, 135,
  979, 939, 149, 83, 990, 669, 545, 521, 409, 772, 924 ] can not be partitioned
    steps: 1687875
    [ 953, 176, 401, 934, 954, 961, 744, 133, 154, 729, 539, 536, 951, 136,
  489, 493, 925, 208, 699, 955, 555, 255, 671, 736, 399 ] can be partitioned as
[ 961, 955, 954, 953, 951, 934, 925, 401, 176, 133 ] [ 744, 736, 729, 699,
  671, 555, 539, 536, 493, 489, 399, 255, 208, 154, 136 ]
    steps: 260
    ---
```

Question 2:

```
In a language of your choice, illustrate a depth-first and
breadth-first tree search, preferably using a stack and a queue,
for a small "sliding block" puzzle.

A sample search might be to get from

     start        finish

     2 1 3        1 2 3
     5 4 6        4 5 6
     7 8 .        7 8 .

where the "." is the empty square; the two possible first moves
slide either the 6 or the 8 to bottom right corner.

Is one sort of search better than the other for this problem?
```

I've implemented both versions of the search in Hot Cocoa Lisp. Breadth first search is the clear winner finding an efficient solution in under 2 seconds:

```
$ cd sliding_blocks
$ time node breadth_first.js
solution found! [5,2,1,4,3,0,1,2,5,4,3,0,1,4,5,8]
```

```
node breadth_first.js  1.76s user 0.11s system 99% cpu 1.883 total
```

The depth first version took longer than I wanted to wait but I gave it the simpler position

```
. 5 2
1 4 3
7 8 6
```

which can quickly be solved with [3,4,1,2,5,8] and got the result:

```
$ node depth_first.js
solution found! [3,6,7,8,5,4,7,8,5,4,7,8,5,4,3,6,7,8,5,4,7,8,
                 5,4,7,8,5,4,3,6,7,8,5,4,7,8,5,4,7,8,5,4,1,2,5,8]
```

The algorithm is quite simple and I've abstracted it out into *sliding_ blocks/search.hcl.*
A simple test of my sliding block puzzle api can be found in *sliding_ blocks/manual_ solution.hcl*:

```
$ node manual_solution.js
2 1 3
5 4 6
7 8 .

1 2 3
4 5 6
7 8 .

2 1 3
5 4 6
7 8 .
```

Question 3:

```
Explain in your own words what exactly is meant by "P" and "NP" as
complexity classes in computer science, why this is such an important
question, and what is and isn't known about them. Give explicit
examples of problems that are in each of these classes, and explain
why the known algorithms have behaviors consistent with your P and NP
descriptions. You may use external sources if you need to - and if so
be clear which ones you used, of course - but the point here is to
convey to us your understanding, not to just summarize a wikipedia
article.
```

In complexity theory, P stands for polynomial and refers to the class of problems which can be solved within polynomial time. NP stands from non-deterministically polynomial and refers to the class of problems for which a given solution to the problem can be verified in polynomial time. The question of whether or not $P = NP$ is simply that of whether or not all non-deterministically polynomial problems can be solved in polynomial time. It has been proven that that if any of the class of NP problems are in P then they all are and it is typically conjectured that $P \neq NP$. There are many problems of practical significance that are known to be in NP but for which there are no known algorithms for solving them in polynomial time. If it were shown that $P = NP$ it would mean that we have much more efficient solutions than we have so far found to many problems that have been looked at extensively. Two examples of NP problems follow.

The hamiltonian cycle problem
Given a graph determine whether there exists a path that starts a given node and by following edges touches each node in the graph exactly once before returning to the original node. This problem is clearly in NP because given a solution to the problem one need only traverse the given path once (in $O(n)$ time) to determine whether it is in-fact a solution. A simple algorithm for finding such a solution is a backtracking search which follows edges looking for a valid path. Since each node could in principle have as many as $n - 1$ edges this search has a branching ratio of order n and a depth of n thus the algorithm runs in $O(n^n)$ time. An even simpler algorithm would be to enumerate the permutations of the nodes and then chech whether a path exists for that ordering. This would run in $O(n!)$ time.

The boolean satisfiability problem Given a formula involving ands, ors, nots, parentheses, and variables that contain unspecified boolean values, determine whether an possible assignement of true or false to each of the variables exists such that the formula evaluates to true. The problem is in NP because given an efficient boolean logic system a solution can be verified in a single pass. The simplest algorithm for finding a solution is probably to simply enumerate the possible true/false assignements which are isomorphic to the powerset of the set of variables and thus requires $O(2^n)$ time.

I used wikipedia to refresh my memory about these two problems.

Question 4:

```
Explain what a "hash table" is, and what it's O() behavior looks
like. Implement one and use it to make a histogram of a word counts in
a large text file. The details (collision algorithm, hash function,
programming language) are up to you.
```

A hash table is a particular implementation of a key value store data structure (i.e. python's dictionaries, php's associative arrays, or javascript's objects). The hash table makes use of a hash function which is a function that is designed to deterministically map keys to seemingly random indices. The hash function should have the property that any two similar keys map to different indices. The hash table simply stores key/value pairs in a sparse array at the index determined by the hash function. Because the hash function is deterministic retreiving the key value pair is as simple as running the key throught the hash function and jumping to the resulting index. There is no need to look through the entire list for the pair we need. This meens hash tables make read and write operations $O(1)$ with respect to the number of elements in the hash.

I've implemented a hash table in C. It handles hash collisions by keeping key/value pairs in "buckets" which form linked lists at each index. To find a given pair, my program hashes the given key and finds the associated index then follows the linked list it finds there until it finds a "bucket" with the given key of reaches a null bucket (which would mean the pair is not in the hash). Because I'm statically allocating only 256 buckets (for the simple reason that my hash function is particularly straight-forward with one byte indices) read/write operations should be $O(1)$ when there are around 256 keys and with larger numbers of keys the complexity will approach $O(n)$ for a linked list divided by a constant factor of 256. For some reason on my MacBook, I get segfaults when I try to read files over a certain size in C so I'm just using the first chapter of Moby Dick:

```
$ cd hash
$ gcc -Wall words.c -o words && ./words moby_ch1
```

I've also made general test of the hash table api that can be run with:

```
$ gcc -Wall test.c -o test && ./test
```

Question 5:

```
Discuss the the connections between and ideas behind between a
lossless compression algorithm (your choice which) and information
entropy.  Using a large text file (perhaps the same one from the
previous problem), calculate an approximation to the information
entropy.  Find how much that file can be compressed, using a standard
compression tool (or one you've implemented, but that's not required)
and discuss how that is related to the entropy. Repeat for a file of
random text, and explain how the those results compare.
```

Information entropy is basically a measure of how random a set of data appears to be. Lossless compression relies on patterns in a file which can be represented more succinctly for example a series of fifteen xs can be represented by some encoding of the number 15 and the character x. If a file has the maximum amount of entropy for a file of its size then, by definition, it contains no patterns and cannot be compressed. The less entropy the file contains the more it can be compressed. Shannon's definition of entropy uses a number between 0 and 1 which should be equal to the optimal theoretical compression ratio.

I've generated a file of random bytes of the same size as the first chapter of Moby Dick and I've created a short script in ruby to calculate the entropy of both files:

```
$ cd entropy
$ ruby entropy.rb moby_ch1
0.55706724002445

$ ruby entropy.rb random
0.998049624345567
```

I refreshed my memory of the formula for entropy using the discussion at: http://stackoverflow.com/questions/990477/how-to-calculate-the-entropy-of-a-file.

I also used gzip to determine the practical compression ratios of the files:

```
$ ls -l
...
-rw-r--r--  1 olleicua  staff  12243 Feb 25 20:57 moby_ch1
```

```
-rw-r--r--  1 olleicua  staff  12243 Feb 25 21:02 random

$ gzip moby_ch1

$ gzip random

$ ls -l
...
-rw-r--r--  1 olleicua  staff   5893 Feb 25 20:57 moby_ch1.gz
-rw-r--r--  1 olleicua  staff  12278 Feb 25 21:02 random.gz

$ ruby -e "p 5893.0 / 12243.0"
0.481336273789104

$ ruby -e "p 12278.0 / 12243.0"
1.00285877644368
```

The observed compression ratio for the first chapter of Moby Dick is probably lower for the simple reason that gzip looks for patterns spanning multiple characters and my entropy calulation only looke at single characters. A more thorough calculation would have also looked at groupings of consecutive characters of sizes up to the size of the file. The compression ratio for the random text is probably greater than on for the simple reason that gzip was unable to compress it at all but did add headers to specify things like file length, type of encoding used, and checksum.

Exam on Programming Languages

I completed the following the week of March 12, 2013. The code it refers to can be found in the files that suppliment this document under exams/languages/. This exam was turned in just under two days late.

Question 1:

```
As a way to demonstrate your understanding of programming ideas,
discuss the concepts behind following programming buzz words across
at least three languages that you're familiar with that allow different
programming styles, perhaps C, Python, and a Lisp.

(a) First organize the terms into groups of concepts, showing which
    are variations of the same concept or idea across or within
    languages, or closely related concepts, or opposites.

(b) Then for each of these concept groups, discuss the ideas behind
    that group, and give concrete code snips across these languages
    to illustrate them.

Be clear that I *don't* want you to just define each of these words;
instead, I want you to use them as the starting point for a
conversation with examples about some of the fundamential notions of
how programming languages work, and how those notions vary from
language to language.

In alphabetic order, the words are

      API
      argument
      array
      bind
      callback
      class
      closure
      collection
      comment
      concurrent
```

```
compiled
data structure
dynamic
exception
fork
function
functional
global
hash
immutable
imperitive
inheritance
interface
interpreted
iterate
lexical
link
list
lazy
lexical
macro
method
name
namespace
object
overload
parse
scope
package
pattern
pass by reference
pass by value
pointer
recursion
side effect
stack overflow
static
symbol
syntactic sugar
thread
```

```
test
throw
type
variable
vector
```

Control flow

Programming languages need mechanisms for designating which instructions are executed when. The most common forms this takes are conditionals and iteration. Conditionals are used to determine whether a section of code should execute and iteration is used to execute code multiple times. Control flow can also take the form of functions and function calls. Functions can be thought of like mathematical functions that map a set of inputs onto a set of outputs but in the context of control flow it is usually best to think of them as pieces of code that can be broken out and reused. It is often useful to use functions to break up code even when a given function will only be called once in the execution of the program simply because thinking in terms of smaller reusable, general purpose functions makes code easier to read and much easier to modify later. Compare the following two solutions to the same problem in Ruby:

```
print "what is your email address? "
email = gets.strip
if /.+@.+\..+/.match email
  puts "your email is #{email}"
else
  puts "that isn't an email"
end
```

and

```
def getEmail
  print "what is your email address? "
  return gets.strip
end

def validateEmail email
  return /.+@.+\..+/.match email
end
```

```
email = getEmail
if validateEmail email
  puts "your email is #{email}"
else
  puts "that isn’t an email"
end
```

For something this simple the first is probably best but if any of the steps gets significantly more complex (for example we could imagine wanting to check that the email address ends in a real top-level domain) then the second style starts to become much nicer to work with.

Another way of breaking code out into smaller modular pieces is macros. Macro can refer to different things in different contexts but a macro generally is a way of making a sort of meta statement to the language of the form “when I say X what I really mean is Y”. For example C macros can be used to intstruct the compiler to rewrite sections of code before the rest of compilation begins:

```
#define SWAP(a, b) (a)^=(b);(b)^=(a);(a)^=(b);

...

if (swap_needed(x, y)) {
  SWAP(x, y)
}
```

The compiler simply replaces the macro call in the source code with the code to be substituted in. In this case the code is far more legible if it simply says swap then if it spelled out the swapping process. This could of course have been done with a function call but in C a macro can sometimes have performance benefits in these cases because each function call requires additional memory allocation.

In lisp macros work somewhat differently. A lisp macro can be used to define a completely new syntax and unlike C macros which use a completely separate language to define them, lisp macros are written in lisp. Lisp macros also allow you to control when code is evaluated. In a normal function call in Common Lisp like

```
(defun foo (x) (+ 1 x))
(foo (* 2 4))
```

the arguments of the function are evaluated before the funciton call. So in the above example all that the function `foo` sees is the result of the multiplication: 8. Inside of a lisp macro, you can control exactly when things are evaluated for example:

```
(defmacro foo (x)
  `(progn
    ,(when (eq '* (car x))
      `(format t "argument was a multiplication"))
    (+ 1 ,x)))
```

The ` symbol tells the Common Lisp interpreter that what follows shouldn't be evaluated right away except for the inner pieces preceded by `,`. This macro takes an unevaluated expression x and returns the code to add one to the result of the expression being evaluated preceded by a print statement reporting that the expression began with an asterisk if it did. A more practical example of a macro might be a debugging statment:

```
(defmacro debug (x) `(format t "~a: ~a" ',x ,x))
(setq foo 1)
(debug foo) ; FOO: 1
```

Because the macro can control exactly when it's arguments are evaluated it has access the unevaluated form of the argument and can, in this case, print it out as a label. Delaying evaluation in this way is sometimes called lazy evaluation. In some languages, like Haskell, all evaluation is delayed as long as possible.

Another useful form of control flow is exception handling. Exception handling allows programs to handle problem situations gracefully. For example in Python:

```
def func_that_cant_handle_zero(arg):
  if arg == 0:
    raise Exception("Everything is terrible!")
  return "normal results"

try:
  func_that_cant_handle_zero(0)
except Exception:
  print "bad things happened"
```

Even though our function recieved input that it didn't know what to do with, we can account for the problem using a try/except statement rather than simply crash the program.

It many programming environments it is desireable and possible to have multiple lines of code running at the same time. This is called concurrency. One way of doing this is with a fork. A fork statement tells the currently running process to split into separate processes that have no straightforward way of communicating with one another and have access to all of the information that the parent process had prior to that point. One major advantage of this approach is that forked processes can have access to copies of the same data structures which they can then manipulate without worrying about how it effects the other process. Another common way of handling concurrency is called threads. Threads allow sections of code to be run at the same time within the same program. They can be much faster than forks because they don't require everything the process has access to to be duplicated and the have the advantage of sharing access to the same data (not just duplicates). They also have the disadvantage of sharing access to the same data. It becomes important to worry about the precise order inwhich things can happen and being very careful not to make any assumptions about what has happened already in another thread.

Assignment

Programs often keep track of many different sorts of data at once. It is vitally useful to be able to map different pieces of data to helpful names so that the data can be referred to later. Languages have many ways of doing this. The most common is simple variable assignement where some bit of data called a value gets bound to a variable name:

```
# Ruby or Python
name = "value"

// JavaScript
var name = "value"

// C
char* name = "value"

;; Scheme
(define name 'value)
```

The `var` in JavaScript is optional but without it the variable is treated as global which is usually wrong (more on that shortly). The `char*` in C specifies the type of variable. In this case a pointer to a character (the asterisk designates a pointer). In C strings are stored as sequencial characters terminated by the `NULL` character `'\0'` and stored in variables as a pointers to the first character. The `'` in Scheme designates that following symbol shouldn't be evaluated. In this case both `name` and `value` are symbols and second is bound to the first so that the first evaluates to the second.

Variable assignment can also take the form of argument passing. In this case the names are specified when a function is defined and the values are specified when the function is called:

```
// C
void my_function(int x, int y) {
  printf("x is %d and y is %d\n", x, y);
}

void main() {
  my_function(2, 3); // x is 2 and y is 3
}

;; Scheme
(define (my-function x y)
   (format #t "x is ~a and y is ~a" x y))

(my-function 2 3) ; x is 2 and y is 3
```

In both cases the arguments are treated just as bound variables for the purpose of that function call. The `int`s in C are required and specify the type of the argument which, because C is statically typed must be known beforehand. The special form of `define` seen here in Scheme is syntactic sugar for:

```
(define my-function (lambda (x y)
   (format #t "x is ~a and y is ~a" x y)))
```

In this case `my-function` is a symbol that is being bound to this lambda function.

In larger projects it is often necessary to limit which names are accessible in which contexts. These contexts are called scopes or namespaces. Scopes are often nested so that names from the outer scope are accessible from the inner scope but names from the inner scope are hidden from the outer scope. For example in JavaScript:

```
var foo = 1, bar = 2;
(function() { // functions form scopes in JavaScript
  var foo = 3, baz = 4;
  console.log(foo, bar, baz); // 3 2 4
})()
console.log(foo, bar); // 1 2
console.log(baz); // ReferenceError: baz is not defined
```

If a JavaScript variable is set (e.g. `foo = 1`) without being initialized with the `var` keyword then it is put in the outermost global namespace. This is bad because it means that forgetting the word `var` in one place can cause variables to have unexpected values anywhere in your code. Not all languages define scopes this way. In Ruby method scopes don't nest this way (functions in ruby are called methods):

```
foo = 1
def bar
  print foo
end
foo() # NameError: undefined local variable or method ‘foo’

def foo
  bar = 2
end
foo()
bar # NameError: undefined local variable or method ‘bar’
```

Insted of having nested function scopes, ruby has nested class and objects scopes. Ruby makes use of th `@` sigil to denote instance and class variables so:

```
class Foo
  @@bar = 1 # these are class variables
  @@qux = 2
  def baz
    print @@bar
    @@qux = 7
  end
  def snap
    print @@qux
  end
end
Foo.snap() # 2
Foo.baz() # 1
Foo.snap() # 7
```

Ruby also uses the \$ sigil to denote global variables. I find Ruby's approach to scope to be really nice. It assumes that all varaibles are only needed in the local scope unless a sigil specifies otherwise.

Sometimes it is useful to have a lot of names/value associations wrapped up in a specific isolated context that can be passed around as a data structure. This is precisely what a hash is; a set of key/value pairs that is treated as a single value. In Python it is called a Dictionary and in JavaScript it is synonymous with object.

Types

Computer programs handle and use data and data typically requires structure. Types are a way of classifying data so that the program knows how to interpret it. For example, in C, the series of four bytes:

```
00000000 11011110 11011110 1100110
```

could represent the integer 7303014 or the string `'foo'`. Types typically come in two categories. Atomic types like integers and booleans are just simply a data of that type. Structured types like arrays, instances, and hashes can contain other types of data. For example, you could have an array of booleans, a hash mapping strings to numbers, or even an array of arrays of hashes. Low level languages like C allow you to interact directly with the actual machine representations of these types in memory which has the advandage of allowing you to fully control the way that data is stored in memory. This also requires you to keep track of the way that the data is stored in memory. High level languages like Ruby often have complex dynamic structures for storing arbitrary data. For example, anywhere you can put a value in Ruby, you can put a value of any type and the language will figure out how to represent that in memory wihtout you needing to worry about it.

```
x = 10
```

is just as valid as

```
x = [10, 20, ["foo", true, nil]]
```

This makes Ruby a dynamically typed language. Dynamically typed languages have the advantage of flexibility. This flexibility can however cause bugs if for example a function is was designed to take an integer as an argument but instead is passed a null value. Considder the following in Ruby and in Java:

```
// Ruby
def doMath x
  return 10 * (x + 2) - (x / 3)
end

numbers = {
  :one => 1,
  :two => 2,
  :three => 3,
  :four => 4,
  :five => 5
}

doMath numbers[:six]

// Java
public int doMath(int x) {
  return 10 * (x + 2) - (x / 3);
}

public void main() {
  Map<string, int> numbers = new HashMap<string, int>();
  numbers.set("one", 1);
  numbers.set("two", 2);
  numbers.set("three", 3);
  numbers.set("four", 4);
  numbers.set("five", 5);

  doMath(numbers.get("six"));
}
```

In Ruby you would get a relatively unhelpful error message about there being no '+' method for `nil`. In Java you would actually get an error message for the `.get()` call when no entry is found for `'six'` but even if you simply called `domath(null)` you would get a compile time error about the wrong type being passed. This is what is meant when Java is referred to as type safe. The programmer has to specify the type of everything but the result is that the program won't compile unless all of the types are correct. The result is that there are fewer bugs and more reduntant text in the code.

Object-oriented Programming

There are several ways to think about object-oriented programming. One way is to think of classes as user defined types. Many languages embrace this idea. For example in Ruby, the built-in types are, themselves classes which can be modified just as easily as user defined ones:

```
class Integer
  def double; self*2; end
end

puts 10.double # prints "20"
```

This is called a monkey patch or sometimes, "duck punching". Python allows for a similar programming pattern but unlike Ruby, Python won't let you directly modify the built-in types instead requiring that a new class be defined that inherits from the built-in type:

```
class MyInt(int):
  def double(self):
    return self*2

print MyInt(10).double() # prints "20"
```

Inheritance here allows a subclass to take on all of the characteristics of it's superclass (methods, properties etc.) and then redefine or add new ones. An instance of the `MyInt` class here behaves in all ways just like a normal Python integer except that it also has a double method. When a method from the superclass is redefined this way it's sometimes called overloading. In Python all objects inherit from the default object which defines some methods like `__repr__` which is called when the string is printed out. Overloading allows Python classes to have custom representations:

```
class Foo:
  pass

class Bar:
  def __repr__(self):
    return "[BAR INSTANCE]"

print Foo() # <__main__.Foo instance at 0x100c658c0>
print Bar() # [BAR INSTANCE]
```

JavaScript has a fairly unusual approach to objects. Most modern scripting languages have some form of key/value association type. Ruby and Perl call them hashes, PHP calls them associative arrays, and Python calls them dictionaries. Javascript simply calls them objects. The can be created as literals like in Python or Ruby:

```
# Python
x = { "foo": 1, "bar": 2 }

# Ruby
x = { "foo" => 1, "bar" => 2 }

// JavaScript
var x = { foo: 1, bar: 2 }
```

Unlike Python or Ruby, JavaScript uses prototypal ineritance as opposed to classical inheritance. With classical inheritance classes can inherit from other classes and objects can be instances of classes. With prototypal inheritance objects simply inherit from other objects. For example:

```
# Ruby
class Person
  attr_accessor :first_name, :last_name
  def initialize *args
    @first_name, @last_name = args
  end
  def full_name
    "#{@first_name} #{@last_name}"
  end
end
```

```
joe = Person.new "Joe", "Smith"
puts joe.full_name # Joe Smith

// JavaScript
var defaultPerson = {
  full_name: function() {
    return this.first_name + " " + this.last_name;
  }
};

var joe = Object.create(defaultPerson);
joe.first_name = "Joe";
joe.last_name = "Smith;
console.log(joe.full_name());
```

The `Object.create()` call here returns a new object that inherits from the passed object. Because the object simply inheritted from another object the properties of that object can later be changed.

Funcitonal Programming

Functionaly programming is about writing functions that have no sides effects. This means that each function only interacts with the rest of the program via arguments passed in and return values. Such functions can be seen as mathematical functions that map a set of inputs onto a set of outputs.

Where an imperative program is a sequence of instructions to be followed in order, a functional program is a collection of well defined transformations built up from one another with a final outer function that transforms the program's input into the program's output. One of the greatest advantages of this approach is that small well defined functions are much easier to test and debug than large unweildy ones. Python's doctests can be very helpful for these sorts of tests:

```
def addOne(x):
  """
  Example:
  >>> addOne(2)
  3
  """
  return x + 1
```

It is common for recursion to be used in place of iteration in functional languages. For example, given the problem of determining whether a list contains a given element in Python one might do the following:

```
def contains(list, element):
  for e in list:
    if e == element:
      return True
  return False
```

Whereas in Scheme, it would be more common to see:

```
(define (contains l element)
  (cond
   ((null? l) #f)
   ((= element (car l)) #t)
   (#t (contains (cdr l) element))))
```

Callbacks are also a very common pattern in functional programmming. The idea behind a callback is that a function can take another function as one of it's arguments and call that function when it's done. Often passing the results of the first function to the second instead of returning them. Node.js uses this technique to guarantee that input and output operations are non-blocking. For example considder the common use case of querying a database for some data and sending it to the user as JSON. In a traditional web server environment like PHP, the process would be frozen while the database was processing the request:

```
// PHP
$sql = "SELECT stuff FROM tables";
$query_result = db_query($sql); // execution is stopped here
echo json_encode($query_result);

// go back to serving other requests
```

In Node.js this problem is solved using callbacks:

```
// Node.js
var sql = 'SELECT stuff FROM tables';
db_query(sql, function(query_result) {
  serve_request(JSON.stringify(query_result));
});

// go back to serving other requests
```

Because the db_query() function returns immediately, serving other requests can resume immediately. Behind the scenes, Node.js has a pool of threads that it uses to handle the actual database querying. Because JavaScript functions have closures the callback will have access to the scope inwhich it was defined which makes Node.js work particularly well.

APIs

An api, or application interface is an interface to a section of code. The api hides irrelevant implementation details so that the progammer can focus on the parts that matter. For example in a Ruby program, if I need to sort a list of numbers I don't need to know what sorting algoritm is being used. I merely need to know how to interface to the built-in library that does sorting.

```ruby
# implementation
class Array
  def sorted?
    (size - 1).times do |i|
      return false if self[i] > self[i + 1]
    end
    return true
  end
  def sort
    result = shuffle
    return result if result.sorted?
    return sort
  end
end

# api
numbers = [60, 99, 61, 26, 82, 19, 44, 76, 29, 23]
sorted = numbers.sort
```

I don't need to see the implementation to know how to use it. All I need to know how to use it is that Arrays have a .sort method that takes no arguments and returns a sorted copy of the Array. These pieces of information are the api. In this case it may also be worth knowing that the builtin quicksort implementation has been overloaded with the factorial time bogosort algorithm since this will be much slower than expected.

Apis are most useful in general purpose libraries. For example, the node.js package optimist is a useful library for parsing command line arguments. If you needed to know how it worked in order to use it then it would hardly be worth using it at all as you could simply make your own. Instead it provides an api for it's use. It can then be treated like a black box. As long as you know which methods to call with which arguments and what they will do, you can ignore the details of how.

A more complex example of an api is the Google Maps API. Google Maps is a big, fully featured web application for using maps. The inner workings of the application are controled by Google but they expose the API as a system by which web developers can embed maps on web pages and manipulate them in JavaScript. The web developer needed understand all of the implementation details of the map application as long as they understand how to use the mechanisms provided to manipulate the resulting maps.

Question 2:

```
Write six programs implementing solutions to the following two
problems across the three languages with different styles.
(These may be the same three from question 1, but don't need
to be.)

In each case, include docs and tests appropriate to the style of that
language, including explicitly what verision of what language you ran,
in what environment, what steps compiled and/or ran the code, and what
the input and output looked like.

Use these programs to illustrate some different currently popular
programming paradigms, as well as your mastery of the vernacular
within these programming language communities.

As a postscipt, discuss which languages you found well suited
to which problem, and why.

The two problems are
```

```
A) the perfect squares crossword puzzle

   Replace the * below with twentyfive base 10 digits to form a
   crossword-like array of thirteen 3-digit perfect squares, with each
   3-digit number reading across or down.  (121 = 11^2 for example is
   a 3-digit perfect square.)

      *   * * *   * * *   *
      * * *   * * *   * * *
      *   * * *   * * *   *

B) family tree

   Write a program to generate a visual family tree from a .csv (comma
   separated value) file of people.

   Each line in the file should represent a person, and include at
   least (name, father, mother, date born, date died). The data format
   is up to you, but should be (a) well defined, and (b) allow for
   multiple people with the same name.  Generate some (fake) data to
   run your code on, which includes at least 10 people across at least
   3 generations.

   The family tree should be either ascii art or easily displayed
   image (e.g. .png, .pdf, .svg, .html, ...) as you choose. You may
   use an external graphics library appropriate to the language; if
   so as usual quote your sources explicitly.
```

Square Crosswords

I very quickly found an answer to problem by inspection assuming squares can occur multiple times in the solution. My solution relies on palindromic squares to create a highly symetrical solution using 11^2, 12^2, and 22^2:

```
1   1 2 1   1 2 1   1
4 8 4   4 8 4   4 8 4
4   4 8 4   4 8 4   4
```

I decided to try answering the more difficult quesiton of whether this can be solved using each square at most once. I was able to find the following solution using a recursive search in ruby:

```
$ cd crosswords
$ ruby recursive_search.rb
841
484
144
169
441
961
625
256
225
676
576
729
196
```

I transccribed this by hand to the folowing:

```
8   1 6 9   2 2 5   1
4 8 4   6 2 5   7 2 9
1   4 4 1   6 7 6   6
```

My ruby code makes use of Ruby classes to create scopes that fully encapsulate the calculation. This isn't really necessary for an application with so few moving parts but it seemed like the natural thing to do in classical object-oriented language like Ruby. This particular script has two major flaws. Firstly, the logic of when a given square is allowed to fit in a particular space is coded in ad hoc manner which is excessively verbose and somewhat hard to read as well as being inellegant and non-general. Secondly, not having any of the structure of the puzzle built into the program there was no obvious way to translate the solution from the form it is stored in (an array of strings in an arbitrary order) to the crossword format it appears in above. Seeing as by this point it was clearly a flawed first pass, I did the translation by hand and moved on to a better approach in Hot Cocoa Lisp.

My Hot Cocoa Lisp program has a hardcoded list of spaces that will need to contain a digit from two different squares. It then uses this information to automatically constrain the search. This makes it more general and legible than the previous iteration but it still leaves no clear way to translate the output to crossword form.

```
$ hcl recursive_search.hcl
$ node recursive_search.js
[ '841', '484', '144', '169', '441', '961', '625', '256', '225',
  '676', '576', '729', '196' ]
```

I wrote the final version in C and used a somewhat object oriented approach involving structs to organize the puzzle. In this version I kept track of the solution in a 3x11 grid of digits to make sure the output could straightforwardly be made to look like a crossword. I then hardcoded 13 space structs each of which contains the coordinates of the three digits in that space and a bit map denoting which of those digits will have been filled in before this space is assigned a square.

```
$ gcc -Wall recursive_search.c -o recursive_search
$ ./recursive_search
 8   1 6 9   2 2 5   1
 4 8 4   6 2 5   7 2 9
 1   4 4 1   6 7 6   6
```

Family Trees

I wrote a ruby script called *gen_ csv.rb* that generates a random family and stores it in *people.csv*:

```
$ cd family_trees
$ ruby gen_csv.rb
$ cat people.csv
id,first_name,last_name,father,mother,born,died
0,Christy,Jackson,7,8,1928,2010
1,Stacey,Taylor,5,6,1942,-1
2,Claudia,Jackson,1,0,1973,-1
3,Jason,Taylor,1,0,2005,-1
4,April,Jackson,1,0,1986,-1
5,Martha,Taylor,9,10,1903,1971
6,Thelma,Bass,-1,-1,1904,1965
7,Fred,Jackson,-1,-1,1888,1948
8,Joann,Beck,-1,-1,1897,1987
9,Wade,Taylor,17,18,1865,1926
10,Zachary,Currie,-1,-1,1871,1943
11,Patricia,Jackson,7,8,1935,2002
12,Anna,Haynes,-1,-1,1924,2013
13,Gail,Haynes,12,11,1991,-1
```

```
14,Valerie,Currie,9,10,1912,1996
15,Dan,Haynes,11,12,1997,-1
16,Pamela,Bass,6,5,1938,1990
17,Regina,Taylor,21,22,1829,1886
18,Sylvia,Jacobs,-1,-1,1829,1927
19,Barbara,Newton,-1,-1,1926,-1
20,Jeanne,Bass,19,16,1989,-1
21,Lewis,Taylor,-1,-1,1789,1840
22,Mark,Simon,27,28,1796,1858
23,Max,Taylor,17,18,1876,1967
24,Frederick,Taylor,1,0,1985,-1
25,Johnny,Currie,10,9,1911,1974
26,Renee,Bass,19,16,1973,-1
27,Katie,Simon,29,30,1757,1842
28,Joyce,Bond,-1,-1,1760,1844
29,Justin,Simon,-1,-1,1724,1786
30,Victor,Schneider,-1,-1,1718,1785
```

The first tree generating program I wrote was in Ruby. I used a simple scripting approach to generate a graphviz file and complile it to a .png file using `dot`.

```
$ ruby display_tree.rb
$ dot -Tpng tree.graphviz > tree.png
```

This approach seemed too easy so I decided to try making an ascii version in C. Unfortunately intelligently rendering a complex directed graph in two dimensions turns out to be a fairly non-trivial algorithmic problem and it seemed like an poor use of my time to learn and re-write the dot algorithm (or even worse invent my own) so instead I made a console based family tree explorer:

```
$ gcc -Wall display_tree.c -o display_tree
$ ./display_tree

Christy Jackson (1928 - 2010)

Mother: Joann Beck (1897 - 1987)
Father: Fred Jackson (1888 - 1948)
Spouce: Stacey Taylor (1942 - present)
```

```
Children:
Frederick Taylor (1985 - present)
April Jackson (1986 - present)
Jason Taylor (2005 - present)
Claudia Jackson (1973 - present)

warning: this program uses gets(), which is unsafe.
Enter a relative to navigate to (mother, father, spouce, child_n): father

Fred Jackson (1888 - 1948)

Mother: N/A
Father: N/A
Spouce: Joann Beck (1897 - 1987)
Children:
Patricia Jackson (1935 - 2002)
Christy Jackson (1928 - 2010)

Enter a relative to navigate to (mother, father, spouce, child_n): spouce

Joann Beck (1897 - 1987)

Mother: N/A
Father: N/A
Spouce: Fred Jackson (1888 - 1948)
Children:
Patricia Jackson (1935 - 2002)
Christy Jackson (1928 - 2010)

Enter a relative to navigate to (mother, father, spouce, child_n): child_0

Patricia Jackson (1935 - 2002)

Mother: Joann Beck (1897 - 1987)
Father: Fred Jackson (1888 - 1948)
Spouce: Anna Haynes (1924 - 2013)
Children:
Dan Haynes (1997 - present)
Gail Haynes (1991 - present)
```

```
Enter a relative to navigate to (mother, father, spouce, child_n): mother

Joann Beck (1897 - 1987)

Mother: N/A
Father: N/A
Spouce: Fred Jackson (1888 - 1948)
Children:
Patricia Jackson (1935 - 2002)
Christy Jackson (1928 - 2010)

Enter a relative to navigate to (mother, father, spouce, child_n): child_1

Christy Jackson (1928 - 2010)

Mother: Joann Beck (1897 - 1987)
Father: Fred Jackson (1888 - 1948)
Spouce: Stacey Taylor (1942 - present)
Children:
Frederick Taylor (1985 - present)
April Jackson (1986 - present)
Jason Taylor (2005 - present)
Claudia Jackson (1973 - present)

Enter a relative to navigate to (mother, father, spouce, child_n): child_3

Claudia Jackson (1973 - present)

Mother: Christy Jackson (1928 - 2010)
Father: Stacey Taylor (1942 - present)

Enter a relative to navigate to (mother, father, spouce, child_n): quit
```

For my third version I simply re-wrote the tree explorer in Hot Cocoa Lisp with a more functional style.

```
$ hcl display_tree.hcl
$ node display_tree.js

Christy Jackson (1928 - 2010)
```

```
Mother: Joann Beck (1897 - 1987)
Father: Fred Jackson (1888 - 1948)
Spouce: Stacey Taylor (1942 - present)
Children:
Claudia Jackson (1973 - present)
Jason Taylor (2005 - present)
April Jackson (1986 - present)
Frederick Taylor (1985 - present)

Enter a relative to navigate to (mother, father, spouce, child_n): quit
```

Question 3:

```
Discuss the strengths and weaknesses of these programming languages as
you see them. What sorts of problems or situations are good fits to
these languages, and why? Which do you personally like, and why?  Be
specific, giving examples that justify your comparisons and
conclusions. (This may well cover some ground you've already discussed
in the previous two problems.  If so, you don't have to repeat any of
that, just refer back to it and bring up anything that you feel hasn't
yet been brought forward.)
```

I feel that Python works quite well as a teaching language; even if for no other reason than that it forces new programmers to properly indent their code. It also has a better selection of good libraries to do INSERT THING COMPUTERS DO HERE than most modern scripting languages, making it a good choice for a lot of practical applications. In general I get tired of the little problems with Python. The one that irks me the most lately is the limitted nature of lamdas. This email: http://mail.python.org/pipermail/python-dev/2006-February/060621.html from Python's designer Guido van Rossum explains why he has no intension of changing this. He begins by claiming that there is no reasonable way to make the syntax work. This is clearly ridiculous:

```
lambda(arg1, arg2):
  statement one
  statement two
```

It basically seems to boil down to him not wanting Python to be like lisp. I don't know what he thinks makes Python better than lisp but either way I find that the more I program the more I want to be programming functionally and the less I like Python.

I rather like Ruby for object-oriented programming and general scripting. The block passing structure has a way of making simple tasks simpler and more complex tasks surprisingly manageable. It can be very terse which I like because it means less extraneous typing and more expressive power. The way that Ruby treats objects seems very much like the way Java treats objects to me. The two largest differences between the two languages seem to be a) that Java is strongly typed and Ruby is duck typed, and b) that Ruby is much newer and has a lot more helpful features. The biggest flaw I see with Ruby is that it doesn't really have functions. It has methods which are necessarily attached to classes and objects (and if you embrace this then the language can be quite powerful). It has code blocks which can be passed to functions but aren't really functions in that they can't be treated as values. It has procs and lambdas that effectively are functions but they are so far removed from the normal use case that their syntax is obscure and forgettable.

I find Lisp-like languages to have a syntax particularly conducive to functional programming. Since every piece of the language has a consistent syntax it's very easy to think in terms of function calls (in a way everything is). In some ways the entire point of monads is that every deterministic operation (inside a computer or otherwise) is just a transformation from one state of the universe to the other.

Exam on Internet Technology

I completed the following the week of April 16, 2013. The code it refers to can be found in the files that suppliment this document under exams/internet/.

Internet Exam

Question 1:

```
(a) Assume that a computer in the Marlboro computer lab has
just been initialized. It has its network configured, but
has otherwise not been used. Explain what packets and protocols
would be exchanged were someone try to visit (a) cs.marlboro.edu
or (b) mit.edu with a browser. Be explicit, and use this
scenario to explain as many of the fundamental layers and
services as you can.

(b) Illustrate your answer to the first part with actual
packet captures from wireshark or a similar tool in an
analogous situation.
```

cs.marlboro.edu:

1. The lab computer does a DNS (domain name system) query to one of the local campus DNS servers at 10.1.2.2, 10.1.2.17, or 10.2.0.2. This query uses ethernet protocol to communicate with the router in the lab and the IP (internet protocol) layer to direct the packet to whichever DNS server is being used on port 53. It then uses UDP (user datagram protocol) as the transport layer to send request for the IP address associated with cs.marlboro.edu.

1. Noting that the request came from within campus network, the DNS server responds with a CS's local IP address: 10.1.2.19

1. The lab computer next sends an HTTP (hyper-text transfer protocol) request, again using ethernet to communicate with a router and IP to direct the packet to CS (using the recently discoverred IP address) on port 80. This time it uses TCP (transmission control protocol) to establish and maintain connection where both parties keep track of what has been sent and recieved so that any data that gets lost or

arrives out of sequence can be re-requested. At the application layer, an HTTP request like the following is made:

```
GET / HTTP/1.1
Host: cs.marlboro.edu
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_8_3) AppleWebKit/536.28.10 (KH
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding gzip, deflate
DNT: 1
Connection: keep-alive
```

1. CS sends an HTTP response like the following:

```
Connection: Keep-Alive
    Content-Type: text/html; charset=ISO-8859-1
    Date: Mon, 22 Apr 2013 17:04:12 GMT
    Keep-Alive: timeout=15, max=50
    Server: Apache
    Transfer-Encoding: chunked
```

Followed by an HTML (hyper-text markup language) document describing the page. In this case the document contains references to three other resources:

- /home/home$_{\mathrm{style}}$.css
- /images/mc-logo.gif
- /images/cs$_m$$_{\mathrm{edu}}$.png

Each of these resources are then queried for the same way as in steps 3 and 4.

mit.edu

1. The lab computer does a DNS query to one of the local campus DNS servers at 10.1.2.2, 10.1.2.17, or 10.2.0.2. This query uses ethernet protocol to communicate with the router in the lab and the IP layer to direct the packet to whichever DNS server is being used on port 53. It then uses UDP as the transport layer to send request for the IP address associated with mit.edu.

1. The DNS server responds with its cached mit.edu IP address: 18.9.22.69

1. The lab computer next sends an HTTP request, using IP to direct the packet to mit.edu (using the recently discoverred IP address) on port 80. The packet's path is much more complicated this time as it leaves the campus and goes as far as New York before getting to Cambridge. It uses TCP to establish and maintain connection where both parties keep track of what has been sent and recieved so that any data that gets lost or arrives out of sequence can be re-requested.

1. mit.edu sends an HTTP response not unlike that of CS followed by an HTML (hyper-text markup language) document describing the page. In this case the document contains references to 22 other resources. 15 of these resources are also at mit.edu and can be retrieved using the same connection. 3 of them are at web.mit.edu which is an alias for WWW.MIT.EDU.EDGEKEY.NET which is in turn an alias for e7086.b.akamaiedge.NET which has the IP address 23.6.134.151. A new TCP connection is started to retrieve them. The remaining four resources from ajax.googleapis.com (an alias for googleapis.l.google.com who's IP address is 173.194.73.95), google-analytics.com (which has five ip addresses: 74.125.226.209, 74.125.226.210, 74.125.226.211, 74.125.226.212, and 74.125.226.208), ssl.gstatic.com (who's ip address is 74.125.226.239), and dnn506yrbagrg.cloudfront.net (who's ip address is 54.240.190.146). It's worth noting that the the resource from ajax.googleapis.com is jquery and whenever this computer visits any web page in the future that links to the jquery hosted at ajax.googleapis.com, the lab computer will be able to refer to the cached copy it gets from this request and avoid needing to repeat the request.

wireshark:

My packet captures are included in the packets directory. They were made using the filter `dst host 10.1.2.19 or src host`10.1.2.19| for cs.marlboro.edu and `dst host 18.9.22.69 or src`host 18.9.22.69| for mit.edu.

Question 2:

```
Design a SQL database schema for an online go server,
including games, players, obvservers, and whatever else
seems appropriate. Use this situation to explain the notions
of database normalization, many-to-many, many-to-one,
and one-to-one relations.

Construct an explicity schema for your design in a SQL system of your
choice, populate it with some sample data, and illustrate how it could
be accessed with a few representative typical interactions.
```

My Schema for the server I constructed for the following example was fairly simple. It consists of two tables connected by two many-to-one relations. Rails automatically generates a schema file `db/schema.rb`. Mine follows:

```
ActiveRecord::Schema.define(:version => 20130417185645) do

  create_table "games", :force => true do |t|
    t.text     "moves"
    t.datetime "created_at",      :null => false
    t.datetime "updated_at",      :null => false
    t.integer  "white_player_id"
    t.integer  "black_player_id"
  end

  create_table "users", :force => true do |t|
    t.string   "username"
    t.datetime "created_at", :null => false
    t.datetime "updated_at", :null => false
  end

end
```

The `white$_{\mathrm{player}}$$_{\mathrm{id}}$` and `black$_{\mathrm{player}}$$_{\mathrm` fields in the `games` table are foreign keys to the `users` table forming two many-to-one relations which in this case means that a given game has only one white player and one black player while a given user can be the black or white player in many games. This information could have been equivalently kept with a schema like the following:

```
ActiveRecord::Schema.define(:version => 20130417185645) do

  create_table "games", :force => true do |t|
    t.text     "moves"
    t.datetime "created_at",      :null => false
```

```
      t.datetime "updated_at",      :null => false
      t.string   "white_player_username"
      t.datetime "white_player_created_at", :null => false
      t.datetime "white_player_updated_at", :null => false
      t.string   "black_player_username"
      t.datetime "black_player_created_at", :null => false
      t.datetime "black_player_updated_at", :null => false
    end

  end
```

The reason for an additional table is what is called database normalization. Normalization is process of organizing a database schema so that no information is repeated in multiple places. In the above schema information about individual users would be repeated for each game that user played. The advantage of not repeating this information is two-fold. First it makes the data storage more efficient in the long run. Second, if anything should need to change (even if no reason to change it can be foreseen) only needing to change it in one place is a huge boon.

It would also have been possible implement observers in my database by having a many-to-many relation between these two tables using an additional table like the following:

```
  create_table "observations", :force => true do |t|
    t.datetime "created_at",      :null => false
    t.datetime "updated_at",      :null => false
    t.integer  "user_id"
    t.integer  "game_id"
  end
```

This would be a many-to-many relation because a given user could observe many games and a given game could be observed by many users. Many-to-many relations always involve a separate lookup table like this.

One-to-one relations are generally less common. They are implemented the same way as many-to-one relations except that a given record from the table without the foreign key is only meant to be associated with a single record from the table with the foreign key. This can be enforced in most database systems by making the foreign key field unique. This is fairly rare because usually it is simpler (and therefore preferable) to add all of the fields from one of the tables in the one-to-one relation to the other.

The rails interactive console provides a programatic interface to the database. The following creates a pair of players and a pair of games:

```
> sam = User.new :username => "Sam"
=> #<User id: nil, username: "Sam", created_at: nil, updated_at: nil>
> jim = User.new :username => "Jim"
=> #<User id: nil, username: "Jim", created_at: nil, updated_at: nil>
> sam.save
=> true
> jim.save
=> true
> g = Game.new :white_player => sam, :black_player => jim
=> #<Game id: nil, moves: nil, created_at: nil, updated_at: nil,
     white_player_id: 1, black_player_id: 2>
> g.save
=> true
> g = Game.new :white_player => jim, :black_player => sam
=> #<Game id: nil, moves: nil, created_at: nil, updated_at: nil,
     white_player_id: 2, black_player_id: 1>
> g.save
=> true
```

This is equivalent to the sql:

```
INSERT INTO users (username) VALUES ("Sam"), ("Jim");
INSERT INTO games (white_player_id, black_player_id) VALUES (1, 2), (2, 1);
```

The data can later be accessed with something like:

```
> sam = User.find_by_username "Sam"
=> #<User id: 1, username: "Sam", created_at: "2013-04-19 04:43:39",
     updated_at: "2013-04-19 04:43:39">
> g = sam.games
=> [#<Game id: 13, moves: nil, created_at: "2013-04-19 04:44:22",
      updated_at: "2013-04-19 04:44:22", white_player_id: 1,
      black_player_id: 2>,
  #<Game id: 14, moves: nil, created_at: "2013-04-19 04:44:37",
    updated_at: "2013-04-19 04:44:37", white_player_id: 2,
   black_player_id: 1>]
> jim = sam.games[0].black_player
=> #<User id: 2, username: "Jim", created_at: "2013-04-19 04:43:42",
     updated_at: "2013-04-19 04:43:42">
```

This is equivalent to the sql:

```
sqlite> SELECT * FROM users WHERE username = "Sam";
1|Sam|2013-04-19 04:43:39.489586|2013-04-19 04:43:39.489586

sqlite> SELECT * FROM users, games WHERE
        users.username = "Sam" AND
        (users.id = games.white_player_id OR
         users.id = games.black_player_id);
1|Sam|2013-04-19 04:43:39.489586|2013-04-19 04:43:39.489586|13||
  2013-04-19 04:44:22.647385|2013-04-19 04:44:22.647385|1|2
1|Sam|2013-04-19 04:43:39.489586|2013-04-19 04:43:39.489586|14||
  2013-04-19 04:44:37.718101|2013-04-19 04:44:37.718101|2|1

sqlite> SELECT * FROM users WHERE id = 2;
2|Jim|2013-04-19 04:43:42.240019|2013-04-19 04:43:42.240019
```

Question 3:

```
Implement a prototype at least part of the go server from the previous
question with a technology stack of your choice. You don't need to do
the go game and graphics itself (though you can if you want); instead,
consider mainly the fundamental data model, views, controller, and
user interactions of the web pages and the database that you'd expect
for any standard web app.

Explain your choices of technology, and use this example to
demonstrate your familiarity with current options.

Deploy the app in an environment of your choice. (You may use this as
a chance to discuss web servers and their configuration, but that
isn't the main point.)
```

I've implemented a go server using Ruby on Rails. I chose the framework mostly because I will be using it for work a lot soon and it seemed like a good excuse to refamiliarize myself with it. I also find Rails to be an ideal choice for rapid prototyping for two reasons. Firstly, it was designed to fascilitate rapid developement with plenty of helpful command line scripts that automatically generate code and a fully-featured MVC system that makes the most common things as easy as possible. Secondly, the ammount of mementum behind the rails community ensures that things are supported and work most of the time meaning that prototyping is less likely to devolve into debugging or working around an issue with the framework or re-inventing the wheel where features are missing. Rails has a philosophy of convention before configuration. The means that many of the APIs in the framework assume a certain default behavior (like that the bar method of the FooController class will be mapped to the template found at app/views/foo/bar.html.erb). There are generally ways to override these default behaviors but you still need to know the conventions before you can do anything. This can make learning Rails a bit daunting but it really pays off with rapid prototyping. I also considdered using a Node.js framework like Tower.js as that would give me an excuse to use Hot Cocoa Lisp but that would have required me to learn a new framework and seemed outside the scope of the exam. I did get a chance to write a short piece of client side code in Hot Cocoa Lisp.

I've deployed the app using heroku. It can be found at http://go-server.herokuapp.com. I chose Heroku because for a simple demo like this where I don't really need much bandwidth or processing power anyway, their free option is quite reasonable and deployment is as easy as setting up and pushing a git repository to the right place. I ran into a slight issue because I was using sqlite3 in development Heroku doesn't support sqlite and I had to set it up to us pgdb in production.

The app itself is incredible simple. Users "log in" by supplying a name for themself. If the name corresponds to a user in the database then they are authenticated as that user (so pretending to be somebody else is pretty easy). If the user doesn't exist yet then a new one is created. This means that if you forget or misspell your username you might have issues using the app. Anyone can create a game at which point they are assigned a random color. If a game is missing one or more player it will be listed as open and any player will be able to join as the missing color. Players can make moves when it is their turn by clicking on the intersections of the board or clicking 'pass'. There is nothing to stop a player from going on an intersection that has already been played on (the stone will simply be replaced with one of this player's color). There is no logic for removing captured stones and there is no score estimation or even a concept of winning. A game is considdered finished when both players have passed successively.

Question 4:

> `Using the go server from the previous question, discuss the possible security issues in such an installation, including at least user authentication, cookies, sql injection, cross-site scripting, denial of service attacks, and whatever else strikes your fancy. Be as specific as you can.`

The simplest class security issues withmy go server are intentional ommisions. I didn't bother to implement user authentication or the rules of go so anyone could trivially impersonate another player, place a stone on a non-empty square, or violate the Ko rule. Slightly less obvious issues could crop up if, for example I failed to create a server-side check that it was the current user's turn before making a move, or that a game had an opening before somebody tried to join it. Without these checks someone could trivially spoof the relevant HTTP request to make two moves in a row or supplant one of the players in a game they weren't involved in.

Another important class of security issues is called cross-site scripting. This is when a user submits some form of content that intended to be automatically displayed as HTML and includes some JavaScript. Since this JavaScript will be executed by anyone visiting the page, it can easily access their cookies allowing the attacker to impersonate them. Effectively elliminating this sort of attack requires that only a limitted set of html tags and properties be allow in user submitted content to make sure that JavaScript cannot be included. My Installation dodges this but not having user submitted content. A related but importantly different security issue is cross-site request forgery. This is when a user that is authenticated with one website visits a completely different site the second site causes their browser to make a request that requires the users authentication on the first site. For example, if Alice is logged into her bank account looks at a conversation on an open forum, it is possible for Eve to have placed an image tag on the forum page that instead of pointing to an image, points the URL of a request on Alice's bank website. The bank website could protect Alice from this sort of attack using a CSRF token. The basic strategy is to generate a random number with any request for a web form and put that number in a hidden input tag in the form. Then when the form is submitted the server can ignore the request and give an error message if the token submitted doesn't match the one given out in the first place. Ruby on Rails implements this by default for all forms. This means you can feel secure in knowing that visitting websites besides go-server.herokuapp.com won't allow malicious websites to make moves for you in your go games.

Another very important class of security issues is SQL injection. This is when a user submits data via a form which is intended to be stored in a database but the user submits SQL code instead which allows them to gain control of your database. The archetypal example is entering something like `';DROP TABLE users;--` enterred into a for that is being processed on the server by something like:

```
sql = "SELECT * FROM users WHERE name = '" + user_submitted_data + "';"
result = db_query(sql);
```

The sql request becomes `SELECT * FROM users WHERE name =` '';DROP TABLE users;– ';| and all data in the users table is lost. The approriate solution to this is to sanitize user inputs so that things like quotes and semicolons are escaped and the user submitted text is treated as it was intended to be. An even better approach in most practical is to use a well tested framework or library for all database access that does this sanitization for you. This saves from the danger of forgetting to sanitize inputs in some places. ActiveRecord, the ORM (object relational mapping) system that rails uses to interface to databases does just this.

Denial of service attacks are a bit different in that they don't usually do any sort of long term damage. Rather, they make a website unavailable or less available by bombarding the server with requests. A traditional DOS attack where you simply repeatedly send requests to a server from a sigle machine are relatively easy to prevent simply by having systems that detect unusually high volumes of traffic from a single source and reject requests all requests from those sources. A distributed denial of service attack (or DDOS) is such an attack is made in a way that makes the traffic appear to (or actually) come from different sources. There are several complex ways of doing this and basically no consistent ways of preventing it. I suspect (though I have not looked into it) that Heroku provides some basic protection from traditional DOS attacks. I am in some sence protected from DDOS attacks by the mere fact that nobody would go to the trouble setting up a DDOS for a simple Go server rails demo. I might be vulnerable to attack on other sites hosted by Heroku but I'm not particularly worried as DDOSes tend to target high profile sites with some sort of political purpose.

Question 5:

```
Finally, discuss your personal opinions and preferences on old and new
technologies and trends in web development, including perhaps the core
technologies of HTML, XHTML, and HTML5 (whatever you think that
means), CSS and its compilers, Javascript and its compilers and
popular libraries, the backend languages and frameworks like PHP,
Rails etc, popular systems such as Wordpress, as well as options like
Node, Backbone, Bootstrap and so on. The specific list is up to you;
the point here is to show that you have an overview and understanding
of where the field was and is going.
```

Anyone can tell you that internet has changed quite rapidly in recent decades. I think the most significant change has also been the most obvious one: more people use it. When Tim Berners-Lee invented the world wide web a seriously doubt he forsaw FaceBook. I that the rise of social networks has had less to do with the progression of underlying technology then with the change in needs of the web's users. This increase in internet use has (and will likely continue to have) positive and negative effects on the field. Specifically, the increased economic demand for new web applications has fueled many innovative projects. Rails for example has grown into a powerful tool that allows developers to quickly get from an idea to a working prototype. Meanwhile, the demand for higher performance servers has spawned Node.js which powerfully shakes the way people think about concurrency.

The increase in internet use also means that multiple significant demographics that don't have a good understanding of computers now have a stake in the direction of the internet. These include lay-people using social networks, business people who want to profit from the internet's success, and amateur web developers who understand the technologies well enough to use them but never bother to go deeper. Trends like the continued popularity of PHP (and it's use in popular applications like FaceBook, WordPress, and Wikipedia) worry me. People who don't know much about computers can easily look PHP, see how successful it is, and conclude it is a good thing.

Of course these forces aren't totally new either. There have always been forces keeping things from turning out perfectly. The history of the internet is full of stories about a technology not doing everything we wanted it to and people finding inellegant work-arounds that later became industry standards. The document object model and libraries like JQuery is a really good example. The DOM didn't have the features that were needed so we built around it. I see this as making lemonade when life gives you lemons. Languages that compile to JavaScript and CSS are like lemonade.

The internet is a complicated thing with many interested parties and many interlocking moving parts. I think that's why I find it so fascinating. I hope whatever contributions I am able to make to it's future in my lifetime help people make sense of and do cool things with it.

Computational approaches to finding pairs of latin squares with maximal orthogonality, row completeness, and diagonal completeness

I. Introduction

The objective of my project was to apply computational search techniques to the problem described in *Crossover designs in the presence of carry-over effects from two factors* by Lewis and Russell.[1] A summary of this problem follows.

The goal is to find a design for a particular type of experiment that will minimize particular forms of systematic bias. For our purposes, a **design** consists of a pair of latin squares A and B of order n. A **latin square** of order n is an $n \times n$ grid with each cell filled in with one of n symbols such that each row and column of the grid contains each symbol exactly once. Here is an example latin square of order 3:

$$\begin{matrix} 0 & 1 & 2 \\ 1 & 2 & 0 \\ 2 & 0 & 1 \end{matrix}$$

A (somewhat contrived) example of the sort of experiment that could benefit from these designs follows. We have n types of cheese, n types of wine, and n impartial judges. We want to determine which combination of one cheese and one wine the judges like best. With an unlimited supply of each wine and cheese, each judge could simply try each of the n^2 combinations. Unfortunately we might only have n servings of each cheese and wine, or our judges may only have time to try n combinations, or it may for some other reason be prohibitively expensive to have more than n^2 trials. We can set up the experiment with a series of n trials by constructing a pair of order n latin squares A and B. In trial i judge j will try the combination of cheese $A_{i,j}$ and wine $B_{i,j}$.

By constraining both grids in the design to being latin squares, we make sure of two things. Firstly, we can be certain that each judge tries each cheese and each wine once. Secondly, we can be certain that each wine and each cheese will be tried exactly once in each time slot. This will eliminate systematic bias from, for example, always trying cheddar last or only having one judge try Merlot.

In this paper I talk about a property of multisets that I will call **redundancy**. The redundancy of a multiset is the minimum number of elements that must be removed from it to make all of its elements distinct.

A latin square A is considered **row complete** if the multiset of ordered pairs $\{(A_{i-1,j}, A_{i,j}) \mid 1 \leq i < n, 0 \leq j < n\}$ has a redundancy of 0. That is to say that each pair of adjacent symbols in the square occurs exactly once. Here is an example of a row complete latin square of order 4:

$$\begin{array}{cccc} 0 & 1 & 3 & 2 \\ 1 & 2 & 0 & 3 \\ 2 & 3 & 1 & 0 \\ 3 & 0 & 2 & 1 \end{array}$$

Note that I am using the convention of starting indices with 0 here and in the rest of this paper.

An ordered pair of latin squares A and B is considered **orthogonal** if the multiset of ordered pairs $\{(A_{i,j}, B_{i,j}) \mid 0 \leq i < n, 0 \leq j < n\}$ has a redundancy of 0. That is to say that each pair of symbols taken from the same position in each square occurs exactly once in the design. Here is an example of a pair of orthogonal latin squares of order 3:

$$\begin{array}{ccc} 0 & 1 & 2 \\ 1 & 2 & 0 \\ 2 & 0 & 1 \end{array} \qquad \begin{array}{ccc} 0 & 2 & 1 \\ 1 & 0 & 2 \\ 2 & 1 & 0 \end{array}$$

Each design is assigned a series of five metrics M_0, M_1, M_2, M_3, M_4. The five metrics are as follows:

- M_0 is the number of places where an orthogonal pair is

repeated or the redundancy of $\{(A_{i,j}, B_{i,j}) \mid 0 \leq i < n, 0 \leq j < n\}$. We want to minimize this number to maximize the number of wine/cheese combinations tried. - M_1 is the number of mistakes in row completeness of A or the redundancy of $\{(A_{i,j-1}, A_{i,j}) \mid 0 \leq i < n, 1 \leq j < n\}$. We want to minimize this number to eliminate systematic bias that might be introduced by having the same sequence of two cheeses tried multiple times. For example, repeatedly trying cheddar just before havarti could introduce systematic bias into the evaluation of havarti. - M_2 is the number of mistakes in row completeness of B or the redundancy of $\{(B_{i,j-1}, B_{i,j}) \mid 0 \leq i < n, 1 \leq j < n\}$. We want to minimize this number to eliminate systematic bias that might be introduced by having the same sequence of two wines tried multiple times. For example, repeatedly trying Merlot just before Pinot Noir could introduce systematic bias into the evaluation of Pinot Noir.

- M_3 works very much like M_1 and M_2 except that it looks at

the pairs formed by taking one cell from A and the cell immediately to the right of it from B. For example, $(A_{1,2}, B_{1,3})$, or generally $(A_{i,j}, B_{i,j+1})$ such that $0 \leq i < n, 0 \leq j < n-1$. M_3 is the number of repeats in these pairs or the redundancy of $\{(A_{i,j-1}, B_{i,j}) \mid 0 \leq i < n, 1 \leq j < n\}$. We want to minimize this number to eliminate systematic bias from, for example, repeatedly trying cheddar just before Pinot Noir.

- M_4 works just like M_3 except in the opposite direction. It looks at the pairs formed by taking one cell from A and the cell immediately to the left of it from B. For example, $(A_{1,2}, B_{1,1})$, or generally $(A_{i,j}, B_{i,j-1})$ such that $0 \leq i < n, 1 \leq j < n$. M_4 is the number of repeats in these pairs or the redundancy of $\{(A_{i,j}, B_{i,j-1})\ \ 0 \leq i < n, 1 \leq j < n$. We want to minimize this number to eliminate systematic bias from, for example, repeatedly trying Merlot just before havarti.
  Lewis and Russell's work was motivated by experiments in the telecommunications industry. In their experiments, what I have been thinking of as cheese was circuit conditions formed from properties of a pair of telephones and what I have been thinking of as wine was a combination of several variables associated with a transmission such as bandwidth, signal gain or loss, noise level, and coding distortion. Rather than having n judges try a series of n wine/cheese combinations, they had n pairs of subjects conduct a series of n conversations under different circuit conditions.

A common algorithm for exploring a space of possibilities is called **backtracking search**. It can be used any time that solving a problem can be represented by a series of decisions with well defined consequences. The basic structure of the recursive form of the algorithm follows:

```
Function search():
  If a solution has been found:
    Return the solution
  Otherwise:
```

```
Initialize an empty list of solutions
Iterate over the options at this point:
  Try this option
  If no solution can possibly be found past this point:
    Return and empty list
  Otherwise:
    Call search() and append the result to the list of
    solutions
  Undo this option
Return the list of solutions
```

This technique can be used to find combinatorial designs by viewing the process of filling in the symbols in the design as a series of decisions between the various symbols that could occupy each space.

There is a phenomenon in combinatorics called **combinatorial explosion** where the number of designs at a given order grows very rapidly as a function of the order. For this reason, my methods use several techniques to reduce the space that must be searched. One such technique is to find symmetries that allow sections of the space to be eliminated by showing that those sections are equivalent to other squares that I am searching for. For example, permuting the names of the symbols in a square would produce another equivalent square so my search methods can often safely assume that the first row of a square is sorted in ascending order because any square I find for which this isn't true would be equivalent to one for which it is. Another technique is to keep track of the best metrics found so far and consider there to be no possible solutions beyond this point any time our intermediate metrics are worse.

In this paper I will describe several computational methods for minimizing the five metrics at various values of n. In these methods I tried first to minimize the value of $2M_0 + M_1 + M_2$, then, given the space of designs that meet that minimum, find the smallest possible value for both M_3 and M_3+M_4. The idea behind this is to give row completeness and orthogonality equal weight and then to consider diagonal completeness as secondary to that. It is considered that systematic bias from, for example, having cheddar always follow havarti is likely to be more prominent than systematic bias from having cheddar always follow Merlot, and it is easy to imagine a scenario in which having cheddar always follow Merlot is more of a concern than having Pinot Noir always follow havarti (for example, if cheeses have more of a lasting taste than wines). The statistical rationale for these metrics is more thoroughly described in Lewis and Russell.

II. Results

The following table summarizes my findings. For each value of n from 1 to 20, I've listed the lowest value I was able to find for $2M_0 + M_1 + M_2$, and among squares with that value, the lowest value for M_3 that I was able to find as well as the lowest value for $M_3 + M_4$ that I was able to find. Finally, I list the method used to find that result. In some cases, the best result for M_3 came from one method and the best result for $M_3 + M_4$ came from another. In these cases the methods are listed separately below. In some cases the same metrics were found through multiple methods. The squares themselves are listed with the description of that method with their metrics in the form $M = [M_0, M_1, M_2, M_3, M_4]$.

n	$2M_0 + M_1 + M_2$	M_3	method	$M_3 + M_4$	method
1	0	0	A	0	A
2	4	0	A	0	A
3	6	0	A	0	A
4	8	4	A	8	A
5	10	5	C	10	C
6	12	6	B,E	12	B, E
7	14	7	C	14	C
8	16	8	B, C, E	20	B
9	12	25	B	50	B
10	12	25	B	50	B
11	22	0	C	22	C
12	0	12	D, E	60	D
13	26	13	C	26	C
14	28	0	C	56	C
15	30	15	A	45	A
16	0	16	D	64	D
17	34	17	C	34	C
18	36	81	A	162	A
19	38	19	C	38	C
20	0	20	D	140	D

method key:

- A: Lewis and Russell

- B: Row permutations
- C: Directed terraces of cyclic groups
- D: Directed terraces of dihedral groups
- E: Generating arrays

The following table summarizes Lewis and Russell's results for the purpose of comparison. Numbers in **bold** have been improved upon by the methods described in this paper, numbers with no style have been matched, and numbers in *italic* were only found using Lewis and Russell's method:

n	$2M_0 + M_1 + M_2$	M_3	$M_3 + M_4$
1	0	0	0
2	4	0	0
3	6	0	0
4	8	4	8
5	10	5	**15**
6	12	**7**	**14**
7	14	**14**	**35**
8	16	**10**	20
9	**18**	18	36
10	**20**	27	54
11	22	**33**	**77**
12	**24**	**34**	**68**
13	26	**39**	**91**
14	28	**41**	**82**
15	*30*	*15*	*45*
16	**32**	**70**	**140**
17	34	**85**	**204**
18	*36*	*81*	*162*
19	38	**114**	**247**
20	**40**	**92**	**184**

III. Methods

The code for the methods listed below is available at on Github at https://github.com/olleicua/latin-squares. Documentation for the code can be found there.

Lewis and Russell

Lewis and Russell present two non-computational methods for finding reasonably good designs, one that works at even orders and another that works at odd orders.[1, section 3]

At even orders they begin by defining the two sequences $s = (0, 1, n-1, 2, n-2, 3, ...\frac{n+2}{2}, \frac{n}{2})$ and $t = (1, 3, 5, ...t-1, t, 2, 4, ...t-2)$. The pair of latin squares A and B are formed by the following formula:

$$A_{i,j} = s_j + i \bmod n$$

$$B_{i,j} = s_j + t_i \bmod n$$

This is a special case of the cyclic directed terrace method below.

Note that mod in these formulas and in the rest of this paper should be interpreted as the operation in computer science that is equivalent to divide by and take the remainder. So for example, 7 mod 3 = 1. This operation should be thought of as returning an integer as opposed to an infinite set of integers.

At odd orders Lewis and Russell do the following. Let $m = \frac{t+1}{2}$ and $q = \lfloor \frac{m+1}{2} \rfloor$. Construct a sequence s whose even indexed elements are $(0, m+1, m+2, ...n-1, q)$ and whose odd indexed elements are $(m-1, ...q+1, q-1, ...1)$. Construct sequence t such that $t_i = s_i \times 2 \bmod n$. The pair of latin squares A and B are formed by the following formula:

$$A_{i,j} = s_j + i \bmod n$$

$$B_{i,j} = t_j + i \bmod n$$

This is a special case of the row permutations method below.

Their methods are described more thoroughly in their paper, and for the orders where I was unable to find better results, theirs are recorded here.

Results:

Order $n = 15$:

0	8	9	7	10	6	11	5	12	3	13	2	14	1	4
1	9	10	8	11	7	12	6	13	4	14	3	0	2	5
2	10	11	9	12	8	13	7	14	5	0	4	1	3	6
3	11	12	10	13	9	14	8	0	6	1	5	2	4	7
4	12	13	11	14	10	0	9	1	7	2	6	3	5	8
5	13	14	12	0	11	1	10	2	8	3	7	4	6	9
6	14	0	13	1	12	2	11	3	9	4	8	5	7	10
7	0	1	14	2	13	3	12	4	10	5	9	6	8	11
8	1	2	0	3	14	4	13	5	11	6	10	7	9	12
9	2	3	1	4	0	5	14	6	12	7	11	8	10	13
10	3	4	2	5	1	6	0	7	13	8	12	9	11	14
11	4	5	3	6	2	7	1	8	14	9	13	10	12	0
12	5	6	4	7	3	8	2	9	0	10	14	11	13	1
13	6	7	5	8	4	9	3	10	1	11	0	12	14	2
14	7	8	6	9	5	10	4	11	2	12	1	13	0	3

0	1	3	14	5	12	7	10	9	6	11	4	13	2	8
1	2	4	0	6	13	8	11	10	7	12	5	14	3	9
2	3	5	1	7	14	9	12	11	8	13	6	0	4	10
3	4	6	2	8	0	10	13	12	9	14	7	1	5	11
4	5	7	3	9	1	11	14	13	10	0	8	2	6	12
5	6	8	4	10	2	12	0	14	11	1	9	3	7	13
6	7	9	5	11	3	13	1	0	12	2	10	4	8	14
7	8	10	6	12	4	14	2	1	13	3	11	5	9	0
8	9	11	7	13	5	0	3	2	14	4	12	6	10	1
9	10	12	8	14	6	1	4	3	0	5	13	7	11	2
10	11	13	9	0	7	2	5	4	1	6	14	8	12	3
11	12	14	10	1	8	3	6	5	2	7	0	9	13	4
12	13	0	11	2	9	4	7	6	3	8	1	10	14	5
13	14	1	12	3	10	5	8	7	4	9	2	11	0	6
14	0	2	13	4	11	6	9	8	5	10	3	12	1	7

$$M = [0, 15, 15, 15, 30]$$

Order $n = 18$:

0	1	17	2	16	3	15	4	14	5	13	6	12	7	11	8	10	9
1	2	0	3	17	4	16	5	15	6	14	7	13	8	12	9	11	10
2	3	1	4	0	5	17	6	16	7	15	8	14	9	13	10	12	11
3	4	2	5	1	6	0	7	17	8	16	9	15	10	14	11	13	12
4	5	3	6	2	7	1	8	0	9	17	10	16	11	15	12	14	13
5	6	4	7	3	8	2	9	1	10	0	11	17	12	16	13	15	14
6	7	5	8	4	9	3	10	2	11	1	12	0	13	17	14	16	15
7	8	6	9	5	10	4	11	3	12	2	13	1	14	0	15	17	16
8	9	7	10	6	11	5	12	4	13	3	14	2	15	1	16	0	17
9	10	8	11	7	12	6	13	5	14	4	15	3	16	2	17	1	0
10	11	9	12	8	13	7	14	6	15	5	16	4	17	3	0	2	1
11	12	10	13	9	14	8	15	7	16	6	17	5	0	4	1	3	2
12	13	11	14	10	15	9	16	8	17	7	0	6	1	5	2	4	3
13	14	12	15	11	16	10	17	9	0	8	1	7	2	6	3	5	4
14	15	13	16	12	17	11	0	10	1	9	2	8	3	7	4	6	5
15	16	14	17	13	0	12	1	11	2	10	3	9	4	8	5	7	6
16	17	15	0	14	1	13	2	12	3	11	4	10	5	9	6	8	7
17	0	16	1	15	2	14	3	13	4	12	5	11	6	10	7	9	8

0	1	17	2	16	3	15	4	14	5	13	6	12	7	11	8	10	9
2	3	1	4	0	5	17	6	16	7	15	8	14	9	13	10	12	11
4	5	3	6	2	7	1	8	0	9	17	10	16	11	15	12	14	13
6	7	5	8	4	9	3	10	2	11	1	12	0	13	17	14	16	15
8	9	7	10	6	11	5	12	4	13	3	14	2	15	1	16	0	17
10	11	9	12	8	13	7	14	6	15	5	16	4	17	3	0	2	1
12	13	11	14	10	15	9	16	8	17	7	0	6	1	5	2	4	3
14	15	13	16	12	17	11	0	10	1	9	2	8	3	7	4	6	5
16	17	15	0	14	1	13	2	12	3	11	4	10	5	9	6	8	7
17	0	16	1	15	2	14	3	13	4	12	5	11	6	10	7	9	8
1	2	0	3	17	4	16	5	15	6	14	7	13	8	12	9	11	10
3	4	2	5	1	6	0	7	17	8	16	9	15	10	14	11	13	12
5	6	4	7	3	8	2	9	1	10	0	11	17	12	16	13	15	14
7	8	6	9	5	10	4	11	3	12	2	13	1	14	0	15	17	16
9	10	8	11	7	12	6	13	5	14	4	15	3	16	2	17	1	0
11	12	10	13	9	14	8	15	7	16	6	17	5	0	4	1	3	2
13	14	12	15	11	16	10	17	9	0	8	1	7	2	6	3	5	4
15	16	14	17	13	0	12	1	11	2	10	3	9	4	8	5	7	6

$$M = [18, 0, 0, 81, 81]$$

Row permutations

The idea behind this method is to search the space of row permutations of all squares in the space of row complete squares of a given order. To reduce the size of this space somewhat several symmetries are employed. The first step is to find the set of all row complete latin squares of order n with the following properties:

- The first row and first column are in sequential ascending order.

- If any row is swapped with the first one and then the symbols in the entire square are remapped to make the first property true again, the resulting square, if different, will come lexicographically after the original square. In this case the lexicographic ordering of squares can be produced by reading symbols down the columns from the upper left corner to the lower right corner of each square, concatenating these symbols and putting the resulting numbers in ascending order.

- If the square is reflected across the vertical axis and the symbols are remapped to make the first property true again, the resulting square, if different, will come lexicographically after

the original square.
These properties assure that even though not all possible row complete squares are represented, those that are not will be equivalent to one that is. This means that the entire space of row complete latin squares of that order is being searched.

Once these squares are found, each possible pair of squares A and B is considered. For each pair, the space of permutations of the rows in B is searched for a square $B_{permuted}$ that produces the best metrics for the pair $(A, B_{permuted})$. This search process is accelerated by keeping track of the best metrics found so far and allowing the search to backtrack (by saying that no solution can be found past this point) if an intermediate result is worse then the best metrics found so far.

This means that I effectively search the space of all pairs of row complete latin squares at a given order for the pair with metrics that best match my criteria.

The space of row complete latin squares at orders 6, 8, 9, and 10 had already been found by Ian Wanless.[2] I was able to confirm his results at 6, 8, and 9. My program uses his data and completely searches the space of pairs of row complete latin squares of orders 6, 8, and 9. At order 10, the computer I was using ran out of memory before giving any output, but I was able to get some results by limiting my search to pairs of the form $(A, A_{permuted})$.

Results:
The results that I got were found in under five minutes. I was ultimately prevented from searching further by hardware limitations. The squares shown

below for orders 6, 8, and 9 are the result of a complete search of the space of row complete latin squares at those orders and examining every possible pair. The order 10 results below represent about five minutes of searching the much larger space of row complete latin squares at order 10 and only comparing squares to themselves. I would estimate that this search at order ten might be possible on a better computer in a few days. Because there are 492 row complete squares of order 10 after all of the symmetry considerations,the full search that compares every possible pair could be expected to take on the order of $491! \approx 10^{1109}$ times longer. This is an excellent example of combinatorial explosion in action. At order 9 the search takes seconds; at order 10 the search takes longer than we can easily estimate.

At order $n = 6$ the row permutations method was able to surpass Lewis and Russell in M_3 and $M_3 + M_4$.

$$\begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \\ 1 & 3 & 0 & 5 & 2 & 4 \\ 2 & 0 & 4 & 1 & 5 & 3 \\ 3 & 5 & 1 & 4 & 0 & 2 \\ 4 & 2 & 5 & 0 & 3 & 1 \\ 5 & 4 & 3 & 2 & 1 & 0 \end{array} \qquad \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \\ 4 & 2 & 0 & 5 & 3 & 1 \\ 5 & 4 & 3 & 2 & 1 & 0 \\ 1 & 3 & 5 & 0 & 2 & 4 \\ 3 & 0 & 4 & 1 & 5 & 2 \\ 2 & 5 & 1 & 4 & 0 & 3 \end{array}$$

$$M = [6, 0, 0, 6, 6]$$

At order $n = 8$ the row permutations method found a design that surpasses Lewis and Russell in M_3 and another that matches them in $M_3 + M_4$.

$$\begin{array}{cccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 1 & 6 & 0 & 7 & 2 & 4 & 3 & 5 \\ 2 & 0 & 4 & 6 & 5 & 7 & 1 & 3 \\ 3 & 7 & 6 & 4 & 1 & 0 & 5 & 2 \\ 4 & 2 & 5 & 1 & 7 & 3 & 0 & 6 \\ 5 & 4 & 7 & 0 & 3 & 6 & 2 & 1 \\ 6 & 3 & 1 & 5 & 0 & 2 & 7 & 4 \\ 7 & 5 & 3 & 2 & 6 & 1 & 4 & 0 \end{array} \qquad \begin{array}{cccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 2 & 4 & 3 & 5 & 0 & 7 & 1 & 6 \\ 4 & 6 & 0 & 2 & 1 & 3 & 7 & 5 \\ 5 & 2 & 7 & 6 & 3 & 1 & 0 & 4 \\ 1 & 7 & 4 & 0 & 6 & 2 & 5 & 3 \\ 6 & 5 & 1 & 4 & 7 & 0 & 3 & 2 \\ 3 & 0 & 5 & 7 & 2 & 6 & 4 & 1 \\ 7 & 3 & 6 & 1 & 5 & 4 & 2 & 0 \end{array}$$

$$M = [8, 0, 0, 8, 16]$$

$$\begin{array}{cccccccc} 0&1&2&3&4&5&6&7\\ 1&3&0&5&2&7&4&6\\ 2&0&4&1&6&3&7&5\\ 3&5&1&7&0&6&2&4\\ 4&2&6&0&7&1&5&3\\ 5&7&3&6&1&4&0&2\\ 6&4&7&2&5&0&3&1\\ 7&6&5&4&3&2&1&0 \end{array} \qquad \begin{array}{cccccccc} 0&1&2&3&4&5&6&7\\ 1&3&0&5&2&7&4&6\\ 4&2&6&0&7&1&5&3\\ 5&7&3&6&1&4&0&2\\ 7&6&5&4&3&2&1&0\\ 6&4&7&2&5&0&3&1\\ 3&5&1&7&0&6&2&4\\ 2&0&4&1&6&3&7&5 \end{array}$$

$$M = [8, 0, 0, 10, 10]$$

At order $n = 9$ the row permutations method found a design that surpasses Lewis and Russell in $2M_0 + M_1 + M_2$.

$$\begin{array}{ccccccccc} 0&1&2&3&4&5&6&7&8\\ 1&3&6&0&8&4&7&2&5\\ 2&8&5&7&6&1&0&4&3\\ 3&0&7&1&5&8&2&6&4\\ 4&6&3&8&1&7&5&0&2\\ 5&2&1&4&0&6&8&3&7\\ 6&5&4&2&7&3&1&8&0\\ 7&4&8&6&2&0&3&5&1\\ 8&7&0&5&3&2&4&1&6 \end{array} \qquad \begin{array}{ccccccccc} 4&6&3&8&1&7&5&0&2\\ 8&7&0&5&3&2&4&1&6\\ 2&8&5&7&6&1&0&4&3\\ 5&2&1&4&0&6&8&3&7\\ 3&0&7&1&5&8&2&6&4\\ 1&3&6&0&8&4&7&2&5\\ 7&4&8&6&2&0&3&5&1\\ 6&5&4&2&7&3&1&8&0\\ 0&1&2&3&4&5&6&7&8 \end{array}$$

$$M = [6, 0, 0, 25, 25]$$

At order $n = 10$ the row permutations method found a design that surpasses Lewis and Russell in $2M_0 + M_1 + M_2$ without exhausting the space of row complete latin squares.

$$\begin{array}{ccccccccc} 0&1&2&3&4&5&6&7&8\\ 1&3&6&0&8&4&7&2&5\\ 2&8&5&7&6&1&0&4&3\\ 3&0&7&1&5&8&2&6&4\\ 4&6&3&8&1&7&5&0&2\\ 5&2&1&4&0&6&8&3&7\\ 6&5&4&2&7&3&1&8&0\\ 7&4&8&6&2&0&3&5&1\\ 8&7&0&5&3&2&4&1&6 \end{array} \qquad \begin{array}{ccccccccc} 4&6&3&8&1&7&5&0&2\\ 8&7&0&5&3&2&4&1&6\\ 2&8&5&7&6&1&0&4&3\\ 5&2&1&4&0&6&8&3&7\\ 3&0&7&1&5&8&2&6&4\\ 1&3&6&0&8&4&7&2&5\\ 7&4&8&6&2&0&3&5&1\\ 6&5&4&2&7&3&1&8&0\\ 0&1&2&3&4&5&6&7&8 \end{array}$$

$$M = [6, 0, 0, 25, 25]$$

Directed terraces of the cyclic group

This method makes use of **directed terraces**. Directed terraces are defined as follows. Let G be a group of order n. Let a be an arrangement of the elements of G. Define b to be the sequence of $n-1$ elements such that $b_i = a_i^{-1}a_{i+1}$. If b contains every non-identity element of G then a is a directed terrace.

This method works slightly differently at even and odd orders. At even orders, a backtracking search is used to search the space of pairs of directed terraces x and y of the cyclic group of order n that minimize the number of repeats in the multiset $\{x_i - y_i \bmod n \mid 0 \le i < n\}$. For all even orders, this minimum number of repeats has been shown to be 1.[3] My search also tries to minimize the number of repeats in the two multisets $\{x_{i-1} - y_i \bmod n \mid 1 \le i < n\}$ and $\{x_i - y_{i-1} \bmod n \mid 1 \le i < n\}$. It does this by keeping track of the fewest such repeats found so far and backtracking whenever the current intermediate result has more repeats.

At odd orders, the same thing is done except that instead of using a pair of directed terraces, we use a pair of sequences each of which can be constrained to only have one mistake preventing them from being a directed terrace. That is to say that $\{x_i - x_{i-1} \bmod n \mid 1 \le i < n\}$ will contain exactly one repeat for each sequence. Additionally $\{x_i - y_i \bmod n \mid 0 \le i < n\}$ can be constrained to contain zero repeats.

A pair of latin squares A and B can be produced from a pair of directed terraces (or sequences that approximate directed terraces) x and y using the following formulas:

$$A_{i,j} = x_j + i \bmod n$$

$$B_{i,j} = y_j + i \bmod n$$

This has the effect of propagating the repeats in the various differences that were minimized during the search through the square so that the resulting value for M_0 can be shown to be n times the number of repeats in $\{x_i - y_i \bmod n \mid 0 \le i < n\}$ or n at even orders and 0 at odd orders. The values for M_1 and M_2 are similarly n times the number of terrace mistakes in x and y respectively or 0 at even orders and n at odd orders. The values for M_3 and M_4 will be n times the number of repeats in $\{x_{i-1} - y_i \bmod n \mid 1 \le i < n\}$ and $\{x_i - y_{i-1} \bmod n \mid 1 \le i < n\}$ respectively.

This method effectively searches the space of squares that can be generated in this way. Knowing that $2M_0 + M_1 + M_2$ can be constrained to 2 allows me to substantially reduce the space.
I also used a construction based on this method that produces designs with $M = [0, n, n, n, n]$ at odd prime orders to find results at orders 13, 17, and 19. The construction is to produce a pair of arrangements x and y that approximate cyclic directed terraces of order n. First choose number q such that $0 \leq q < n$. The first two elements of x are 0 then q. Each successive element of x is the previous element times q mod n. If this process produces fewer then n elements before producing repeats then choose a new value for q and try again. There will always be a possible value of q that works.[3] y is then defined to be $(0, x_3, x_4, ...x_{n-1}, x_1, x_2)$ and the latin squares are then produced in the same way above.

Results:

The even ordered results shown below represent 63 hours, 13 minutes, and 55 seconds of running my program. I exhaustively searched the space of pairs of directed terraces of the cyclic groups of orders 2 through 14. The odd ordered results represent 35 hours, 30 minutes, and 34 seconds of running my program. I exhaustively searched the space of pairs of sequences that are one mistake away from being directed terraces of cyclic groups and produce orthogonal pairs of latin squares at orders 1 through 11.

At order $n = 5$ the cyclic directed terrace method found a design that surpasses Lewis and Russell in $M_3 + M_4$.

$$\begin{array}{ccccc} 0 & 1 & 2 & 4 & 3 \\ 1 & 2 & 3 & 0 & 4 \\ 2 & 3 & 4 & 1 & 0 \\ 3 & 4 & 0 & 2 & 1 \\ 4 & 0 & 1 & 3 & 2 \end{array} \qquad \begin{array}{ccccc} 0 & 4 & 3 & 1 & 2 \\ 1 & 0 & 4 & 2 & 3 \\ 2 & 1 & 0 & 3 & 4 \\ 3 & 2 & 1 & 4 & 0 \\ 4 & 3 & 2 & 0 & 1 \end{array}$$

$$M = [0, 5, 5, 5, 5]$$

At order $n = 7$ the cyclic directed terrace method found a design that surpasses Lewis and Russell in both M_3 and $M_3 + M_4$.

0	1	3	2	6	4	5
1	2	4	3	0	5	6
2	3	5	4	1	6	0
3	4	6	5	2	0	1
4	5	0	6	3	1	2
5	6	1	0	4	2	3
6	0	2	1	5	3	4

0	2	6	4	5	1	3
1	3	0	5	6	2	4
2	4	1	6	0	3	5
3	5	2	0	1	4	6
4	6	3	1	2	5	0
5	0	4	2	3	6	1
6	1	5	3	4	0	2

$$M = [0, 7, 7, 7, 7]$$

At order $n = 8$ the cyclic directed terrace method found a design that surpasses Lewis and Russell in M_3.

0	1	7	3	6	5	2	4
1	2	0	4	7	6	3	5
2	3	1	5	0	7	4	6
3	4	2	6	1	0	5	7
4	5	3	7	2	1	6	0
5	6	4	0	3	2	7	1
6	7	5	1	4	3	0	2
7	0	6	2	5	4	1	3

0	6	1	2	7	3	5	4
1	7	2	3	0	4	6	5
2	0	3	4	1	5	7	6
3	1	4	5	2	6	0	7
4	2	5	6	3	7	1	0
5	3	6	7	4	0	2	1
6	4	7	0	5	1	3	2
7	5	0	1	6	2	4	3

$$M = [8, 0, 0, 8, 16]$$

At order $n = 11$ the cyclic directed terrace method found a designs that surpass Lewis and Russell at both M_3 and $M_3 + M_4$.

0	1	2	4	3	8	5	9	7	10	6
1	2	3	5	4	9	6	10	8	0	7
2	3	4	6	5	10	7	0	9	1	8
3	4	5	7	6	0	8	1	10	2	9
4	5	6	8	7	1	9	2	0	3	10
5	6	7	9	8	2	10	3	1	4	0
6	7	8	10	9	3	0	4	2	5	1
7	8	9	0	10	4	1	5	3	6	2
8	9	10	1	0	5	2	6	4	7	3
9	10	0	2	1	6	3	7	5	8	4
10	0	1	3	2	7	4	8	6	9	5

0	7	3	6	10	1	2	8	5	4	9
1	8	4	7	0	2	3	9	6	5	10
2	9	5	8	1	3	4	10	7	6	0
3	10	6	9	2	4	5	0	8	7	1
4	0	7	10	3	5	6	1	9	8	2
5	1	8	0	4	6	7	2	10	9	3
6	2	9	1	5	7	8	3	0	10	4
7	3	10	2	6	8	9	4	1	0	5
8	4	0	3	7	9	10	5	2	1	6
9	5	1	4	8	10	0	6	3	2	7
10	6	2	5	9	0	1	7	4	3	8

$$M = [0, 11, 11, 11, 11]$$

0	1	2	5	4	8	10	6	3	9	7
1	2	3	6	5	9	0	7	4	10	8
2	3	4	7	6	10	1	8	5	0	9
3	4	5	8	7	0	2	9	6	1	10
4	5	6	9	8	1	3	10	7	2	0
5	6	7	10	9	2	4	0	8	3	1
6	7	8	0	10	3	5	1	9	4	2
7	8	9	1	0	4	6	2	10	5	3
8	9	10	2	1	5	7	3	0	6	4
9	10	0	3	2	6	8	4	1	7	5
10	0	1	4	3	7	9	5	2	8	6

0	3	9	2	10	6	4	5	7	1	8
1	4	10	3	0	7	5	6	8	2	9
2	5	0	4	1	8	6	7	9	3	10
3	6	1	5	2	9	7	8	10	4	0
4	7	2	6	3	10	8	9	0	5	1
5	8	3	7	4	0	9	10	1	6	2
6	9	4	8	5	1	10	0	2	7	3
7	10	5	9	6	2	0	1	3	8	4
8	0	6	10	7	3	1	2	4	9	5
9	1	7	0	8	4	2	3	5	10	6
10	2	8	1	9	5	3	4	6	0	7

$$M = [0, 11, 11, 0, 33]$$

At order $n = 13$ the cyclic directed terrace method was used to construct a design that surpasses Lewis and Russell at both M_3 and $M_3 + M_4$.

0	2	4	8	3	6	12	11	9	5	10	7	1
1	3	5	9	4	7	0	12	10	6	11	8	2
2	4	6	10	5	8	1	0	11	7	12	9	3
3	5	7	11	6	9	2	1	12	8	0	10	4
4	6	8	12	7	10	3	2	0	9	1	11	5
5	7	9	0	8	11	4	3	1	10	2	12	6
6	8	10	1	9	12	5	4	2	11	3	0	7
7	9	11	2	10	0	6	5	3	12	4	1	8
8	10	12	3	11	1	7	6	4	0	5	2	9
9	11	0	4	12	2	8	7	5	1	6	3	10
10	12	1	5	0	3	9	8	6	2	7	4	11
11	0	2	6	1	4	10	9	7	3	8	5	12
12	1	3	7	2	5	11	10	8	4	9	6	0

0	8	3	6	12	11	9	5	10	7	1	2	4
1	9	4	7	0	12	10	6	11	8	2	3	5
2	10	5	8	1	0	11	7	12	9	3	4	6
3	11	6	9	2	1	12	8	0	10	4	5	7
4	12	7	10	3	2	0	9	1	11	5	6	8
5	0	8	11	4	3	1	10	2	12	6	7	9
6	1	9	12	5	4	2	11	3	0	7	8	10
7	2	10	0	6	5	3	12	4	1	8	9	11
8	3	11	1	7	6	4	0	5	2	9	10	12
9	4	12	2	8	7	5	1	6	3	10	11	0
10	5	0	3	9	8	6	2	7	4	11	12	1
11	6	1	4	10	9	7	3	8	5	12	0	2
12	7	2	5	11	10	8	4	9	6	0	1	3

$$M = [0, 13, 13, 13, 13]$$

At order $n = 14$ the cyclic directed terrace method was able to find a designs that surpass Lewis and Russell at both M_3 and $M_3 + M_4$.

0	1	4	10	3	2	13	8	12	6	11	9	5	7
1	2	5	11	4	3	0	9	13	7	12	10	6	8
2	3	6	12	5	4	1	10	0	8	13	11	7	9
3	4	7	13	6	5	2	11	1	9	0	12	8	10
4	5	8	0	7	6	3	12	2	10	1	13	9	11
5	6	9	1	8	7	4	13	3	11	2	0	10	12
6	7	10	2	9	8	5	0	4	12	3	1	11	13
7	8	11	3	10	9	6	1	5	13	4	2	12	0
8	9	12	4	11	10	7	2	6	0	5	3	13	1
9	10	13	5	12	11	8	3	7	1	6	4	0	2
10	11	0	6	13	12	9	4	8	2	7	5	1	3
11	12	1	7	0	13	10	5	9	3	8	6	2	4
12	13	2	8	1	0	11	6	10	4	9	7	3	5
13	0	3	9	2	1	12	7	11	5	10	8	4	6

0	12	2	1	6	8	9	3	13	10	5	11	4	7
1	13	3	2	7	9	10	4	0	11	6	12	5	8
2	0	4	3	8	10	11	5	1	12	7	13	6	9
3	1	5	4	9	11	12	6	2	13	8	0	7	10
4	2	6	5	10	12	13	7	3	0	9	1	8	11
5	3	7	6	11	13	0	8	4	1	10	2	9	12
6	4	8	7	12	0	1	9	5	2	11	3	10	13
7	5	9	8	13	1	2	10	6	3	12	4	11	0
8	6	10	9	0	2	3	11	7	4	13	5	12	1
9	7	11	10	1	3	4	12	8	5	0	6	13	2
10	8	12	11	2	4	5	13	9	6	1	7	0	3
11	9	13	12	3	5	6	0	10	7	2	8	1	4
12	10	0	13	4	6	7	1	11	8	3	9	2	5
13	11	1	0	5	7	8	2	12	9	4	10	3	6

$M = [14, 0, 0, 28, 28]$

0	1	10	6	3	9	8	13	11	4	12	2	5	7
1	2	11	7	4	10	9	0	12	5	13	3	6	8
2	3	12	8	5	11	10	1	13	6	0	4	7	9
3	4	13	9	6	12	11	2	0	7	1	5	8	10
4	5	0	10	7	13	12	3	1	8	2	6	9	11
5	6	1	11	8	0	13	4	2	9	3	7	10	12
6	7	2	12	9	1	0	5	3	10	4	8	11	13
7	8	3	13	10	2	1	6	4	11	5	9	12	0
8	9	4	0	11	3	2	7	5	12	6	10	13	1
9	10	5	1	12	4	3	8	6	13	7	11	0	2
10	11	6	2	13	5	4	9	7	0	8	12	1	3
11	12	7	3	0	6	5	10	8	1	9	13	2	4
12	13	8	4	1	7	6	11	9	2	10	0	3	5
13	0	9	5	2	8	7	12	10	3	11	1	4	6

0	13	4	11	12	1	5	2	10	6	8	3	9	7
1	0	5	12	13	2	6	3	11	7	9	4	10	8
2	1	6	13	0	3	7	4	12	8	10	5	11	9
3	2	7	0	1	4	8	5	13	9	11	6	12	10
4	3	8	1	2	5	9	6	0	10	12	7	13	11
5	4	9	2	3	6	10	7	1	11	13	8	0	12
6	5	10	3	4	7	11	8	2	12	0	9	1	13
7	6	11	4	5	8	12	9	3	13	1	10	2	0
8	7	12	5	6	9	13	10	4	0	2	11	3	1
9	8	13	6	7	10	0	11	5	1	3	12	4	2
10	9	0	7	8	11	1	12	6	2	4	13	5	3
11	10	1	8	9	12	2	13	7	3	5	0	6	4
12	11	2	9	10	13	3	0	8	4	6	1	7	5
13	12	3	10	11	0	4	1	9	5	7	2	8	6

$$M = [14, 0, 0, 0, 70]$$

At order $n = 17$ the cyclic directed terrace method was used to construct a design that surpasses Lewis and Russell at both M_3 and $M_3 + M_4$.

0	3	9	10	13	5	15	11	16	14	8	7	4	12	2	6	1
1	4	10	11	14	6	16	12	0	15	9	8	5	13	3	7	2
2	5	11	12	15	7	0	13	1	16	10	9	6	14	4	8	3
3	6	12	13	16	8	1	14	2	0	11	10	7	15	5	9	4
4	7	13	14	0	9	2	15	3	1	12	11	8	16	6	10	5
5	8	14	15	1	10	3	16	4	2	13	12	9	0	7	11	6
6	9	15	16	2	11	4	0	5	3	14	13	10	1	8	12	7
7	10	16	0	3	12	5	1	6	4	15	14	11	2	9	13	8
8	11	0	1	4	13	6	2	7	5	16	15	12	3	10	14	9
9	12	1	2	5	14	7	3	8	6	0	16	13	4	11	15	10
10	13	2	3	6	15	8	4	9	7	1	0	14	5	12	16	11
11	14	3	4	7	16	9	5	10	8	2	1	15	6	13	0	12
12	15	4	5	8	0	10	6	11	9	3	2	16	7	14	1	13
13	16	5	6	9	1	11	7	12	10	4	3	0	8	15	2	14
14	0	6	7	10	2	12	8	13	11	5	4	1	9	16	3	15
15	1	7	8	11	3	13	9	14	12	6	5	2	10	0	4	16
16	2	8	9	12	4	14	10	15	13	7	6	3	11	1	5	0

0	10	13	5	15	11	16	14	8	7	4	12	2	6	1	3	9
1	11	14	6	16	12	0	15	9	8	5	13	3	7	2	4	10
2	12	15	7	0	13	1	16	10	9	6	14	4	8	3	5	11
3	13	16	8	1	14	2	0	11	10	7	15	5	9	4	6	12
4	14	0	9	2	15	3	1	12	11	8	16	6	10	5	7	13
5	15	1	10	3	16	4	2	13	12	9	0	7	11	6	8	14
6	16	2	11	4	0	5	3	14	13	10	1	8	12	7	9	15
7	0	3	12	5	1	6	4	15	14	11	2	9	13	8	10	16
8	1	4	13	6	2	7	5	16	15	12	3	10	14	9	11	0
9	2	5	14	7	3	8	6	0	16	13	4	11	15	10	12	1
10	3	6	15	8	4	9	7	1	0	14	5	12	16	11	13	2
11	4	7	16	9	5	10	8	2	1	15	6	13	0	12	14	3
12	5	8	0	10	6	11	9	3	2	16	7	14	1	13	15	4
13	6	9	1	11	7	12	10	4	3	0	8	15	2	14	16	5
14	7	10	2	12	8	13	11	5	4	1	9	16	3	15	0	6
15	8	11	3	13	9	14	12	6	5	2	10	0	4	16	1	7
16	9	12	4	14	10	15	13	7	6	3	11	1	5	0	2	8

$$M = [0, 17, 17, 17, 17]$$

At order $n = 19$ the cyclic directed terrace method was used to construct a design that surpasses Lewis and Russell at both M_3 and $M_3 + M_4$.

0	2	4	8	16	13	7	14	9	18	17	15	11	3	6	12	5	10	1
1	3	5	9	17	14	8	15	10	0	18	16	12	4	7	13	6	11	2
2	4	6	10	18	15	9	16	11	1	0	17	13	5	8	14	7	12	3
3	5	7	11	0	16	10	17	12	2	1	18	14	6	9	15	8	13	4
4	6	8	12	1	17	11	18	13	3	2	0	15	7	10	16	9	14	5
5	7	9	13	2	18	12	0	14	4	3	1	16	8	11	17	10	15	6
6	8	10	14	3	0	13	1	15	5	4	2	17	9	12	18	11	16	7
7	9	11	15	4	1	14	2	16	6	5	3	18	10	13	0	12	17	8
8	10	12	16	5	2	15	3	17	7	6	4	0	11	14	1	13	18	9
9	11	13	17	6	3	16	4	18	8	7	5	1	12	15	2	14	0	10
10	12	14	18	7	4	17	5	0	9	8	6	2	13	16	3	15	1	11
11	13	15	0	8	5	18	6	1	10	9	7	3	14	17	4	16	2	12
12	14	16	1	9	6	0	7	2	11	10	8	4	15	18	5	17	3	13
13	15	17	2	10	7	1	8	3	12	11	9	5	16	0	6	18	4	14
14	16	18	3	11	8	2	9	4	13	12	10	6	17	1	7	0	5	15
15	17	0	4	12	9	3	10	5	14	13	11	7	18	2	8	1	6	16
16	18	1	5	13	10	4	11	6	15	14	12	8	0	3	9	2	7	17
17	0	2	6	14	11	5	12	7	16	15	13	9	1	4	10	3	8	18
18	1	3	7	15	12	6	13	8	17	16	14	10	2	5	11	4	9	0

0	8	16	13	7	14	9	18	17	15	11	3	6	12	5	10	1	2	4
1	9	17	14	8	15	10	0	18	16	12	4	7	13	6	11	2	3	5
2	10	18	15	9	16	11	1	0	17	13	5	8	14	7	12	3	4	6
3	11	0	16	10	17	12	2	1	18	14	6	9	15	8	13	4	5	7
4	12	1	17	11	18	13	3	2	0	15	7	10	16	9	14	5	6	8
5	13	2	18	12	0	14	4	3	1	16	8	11	17	10	15	6	7	9
6	14	3	0	13	1	15	5	4	2	17	9	12	18	11	16	7	8	10
7	15	4	1	14	2	16	6	5	3	18	10	13	0	12	17	8	9	11
8	16	5	2	15	3	17	7	6	4	0	11	14	1	13	18	9	10	12
9	17	6	3	16	4	18	8	7	5	1	12	15	2	14	0	10	11	13
10	18	7	4	17	5	0	9	8	6	2	13	16	3	15	1	11	12	14
11	0	8	5	18	6	1	10	9	7	3	14	17	4	16	2	12	13	15
12	1	9	6	0	7	2	11	10	8	4	15	18	5	17	3	13	14	16
13	2	10	7	1	8	3	12	11	9	5	16	0	6	18	4	14	15	17
14	3	11	8	2	9	4	13	12	10	6	17	1	7	0	5	15	16	18
15	4	12	9	3	10	5	14	13	11	7	18	2	8	1	6	16	17	0
16	5	13	10	4	11	6	15	14	12	8	0	3	9	2	7	17	18	1
17	6	14	11	5	12	7	16	15	13	9	1	4	10	3	8	18	0	2
18	7	15	12	6	13	8	17	16	14	10	2	5	11	4	9	0	1	3

$$M = [0, 19, 19, 19, 19]$$

Directed terraces of the dihedral group

This method works just like the cyclic directed terrace method except that it uses terraces based on dihedral groups instead of cyclic groups. The dihedral group of an even order n is the symmetry group of the two sided regular polyhedron with $\frac{n}{2}$ edges. The latin squares and orthogonal differences are calculated using dihedral multiplication and division rather than modular arithmetic. This means that the relevant formulas for generating a pair of full latin squares from a pair of directed terraces uses dihedral multiplication as follows:

$$A_{i,j} = x_j d_i$$
$$B_{i,j} = y_j d_i$$

where d is any arrangement of the elements of the dihedral group. With this method, M_0, M_1, and M_2 can all be constrained to zero at orders 12, 16, and 20.[4] This constraint on M_0 allowed for an additional constraint on the search space. I ran my search at these three orders.

Results:

The results below for orders 12 and 16 represent 24 hours and 17 minutes of running my program. I exhaustively searched the space of pairs of directed terraces of the dihedral group of order 12. I stopped the order 16 search early to make time for the order 20 search which ran for 61 hours, 10 minutes, and 10 seconds. The order 20 search also didn't have time to finish. I would estimate that exhaustively searching the space of pairs of dihedral terraces of order 16 could have easily taken several weeks. Exhaustively searching order 20 would have taken much longer.

At order $n = 12$ the dihedral directed terrace method found a design that surpasses Lewis and Russell on every metric.

0	2	8	1	6	3	5	9	10	11	7	4
1	11	5	0	7	10	8	4	3	2	6	9
2	4	10	3	8	5	7	11	0	1	9	6
3	1	7	2	9	0	10	6	5	4	8	11
4	6	0	5	10	7	9	1	2	3	11	8
5	3	9	4	11	2	0	8	7	6	10	1
6	8	2	7	0	9	11	3	4	5	1	10
7	5	11	6	1	4	2	10	9	8	0	3
8	10	4	9	2	11	1	5	6	7	3	0
9	7	1	8	3	6	4	0	11	10	2	5
10	0	6	11	4	1	3	7	8	9	5	2
11	9	3	10	5	8	6	2	1	0	4	7

0	5	4	3	7	1	11	2	9	6	10	8
1	8	9	10	6	0	2	11	4	7	3	5
2	7	6	5	9	3	1	4	11	8	0	10
3	10	11	0	8	2	4	1	6	9	5	7
4	9	8	7	11	5	3	6	1	10	2	0
5	0	1	2	10	4	6	3	8	11	7	9
6	11	10	9	1	7	5	8	3	0	4	2
7	2	3	4	0	6	8	5	10	1	9	11
8	1	0	11	3	9	7	10	5	2	6	4
9	4	5	6	2	8	10	7	0	3	11	1
10	3	2	1	5	11	9	0	7	4	8	6
11	6	7	8	4	10	0	9	2	5	1	3

$$M = [0, 0, 0, 12, 48]$$

At order $n = 16$ the dihedral directed terrace method found designs that surpass Lewis and Russell on every metric without searching the entire space.

0	1	2	5	7	13	6	15	4	10	14	3	11	9	12	8
1	0	15	12	10	4	11	2	13	7	3	14	6	8	5	9
2	3	4	7	9	15	8	1	6	12	0	5	13	11	14	10
3	2	1	14	12	6	13	4	15	9	5	0	8	10	7	11
4	5	6	9	11	1	10	3	8	14	2	7	15	13	0	12
5	4	3	0	14	8	15	6	1	11	7	2	10	12	9	13
6	7	8	11	13	3	12	5	10	0	4	9	1	15	2	14
7	6	5	2	0	10	1	8	3	13	9	4	12	14	11	15
8	9	10	13	15	5	14	7	12	2	6	11	3	1	4	0
9	8	7	4	2	12	3	10	5	15	11	6	14	0	13	1
10	11	12	15	1	7	0	9	14	4	8	13	5	3	6	2
11	10	9	6	4	14	5	12	7	1	13	8	0	2	15	3
12	13	14	1	3	9	2	11	0	6	10	15	7	5	8	4
13	12	11	8	6	0	7	14	9	3	15	10	2	4	1	5
14	15	0	3	5	11	4	13	2	8	12	1	9	7	10	6
15	14	13	10	8	2	9	0	11	5	1	12	4	6	3	7

0	11	12	6	14	10	1	8	5	2	3	15	13	7	9	4
1	6	5	11	3	7	0	9	12	15	14	2	4	10	8	13
2	13	14	8	0	12	3	10	7	4	5	1	15	9	11	6
3	8	7	13	5	9	2	11	14	1	0	4	6	12	10	15
4	15	0	10	2	14	5	12	9	6	7	3	1	11	13	8
5	10	9	15	7	11	4	13	0	3	2	6	8	14	12	1
6	1	2	12	4	0	7	14	11	8	9	5	3	13	15	10
7	12	11	1	9	13	6	15	2	5	4	8	10	0	14	3
8	3	4	14	6	2	9	0	13	10	11	7	5	15	1	12
9	14	13	3	11	15	8	1	4	7	6	10	12	2	0	5
10	5	6	0	8	4	11	2	15	12	13	9	7	1	3	14
11	0	15	5	13	1	10	3	6	9	8	12	14	4	2	7
12	7	8	2	10	6	13	4	1	14	15	11	9	3	5	0
13	2	1	7	15	3	12	5	8	11	10	14	0	6	4	9
14	9	10	4	12	8	15	6	3	0	1	13	11	5	7	2
15	4	3	9	1	5	14	7	10	13	12	0	2	8	6	11

$$M = [0, 0, 0, 16, 112]$$

0	1	2	10	6	15	13	8	14	9	12	3	5	11	7	4
1	0	15	7	11	2	4	9	3	8	5	14	12	6	10	13
2	3	4	12	8	1	15	10	0	11	14	5	7	13	9	6
3	2	1	9	13	4	6	11	5	10	7	0	14	8	12	15
4	5	6	14	10	3	1	12	2	13	0	7	9	15	11	8
5	4	3	11	15	6	8	13	7	12	9	2	0	10	14	1
6	7	8	0	12	5	3	14	4	15	2	9	11	1	13	10
7	6	5	13	1	8	10	15	9	14	11	4	2	12	0	3
8	9	10	2	14	7	5	0	6	1	4	11	13	3	15	12
9	8	7	15	3	10	12	1	11	0	13	6	4	14	2	5
10	11	12	4	0	9	7	2	8	3	6	13	15	5	1	14
11	10	9	1	5	12	14	3	13	2	15	8	6	0	4	7
12	13	14	6	2	11	9	4	10	5	8	15	1	7	3	0
13	12	11	3	7	14	0	5	15	4	1	10	8	2	6	9
14	15	0	8	4	13	11	6	12	7	10	1	3	9	5	2
15	14	13	5	9	0	2	7	1	6	3	12	10	4	8	11

0	14	15	12	1	6	3	7	5	11	4	13	9	10	2	8
1	3	2	5	0	11	14	10	12	6	13	4	8	7	15	9
2	0	1	14	3	8	5	9	7	13	6	15	11	12	4	10
3	5	4	7	2	13	0	12	14	8	15	6	10	9	1	11
4	2	3	0	5	10	7	11	9	15	8	1	13	14	6	12
5	7	6	9	4	15	2	14	0	10	1	8	12	11	3	13
6	4	5	2	7	12	9	13	11	1	10	3	15	0	8	14
7	9	8	11	6	1	4	0	2	12	3	10	14	13	5	15
8	6	7	4	9	14	11	15	13	3	12	5	1	2	10	0
9	11	10	13	8	3	6	2	4	14	5	12	0	15	7	1
10	8	9	6	11	0	13	1	15	5	14	7	3	4	12	2
11	13	12	15	10	5	8	4	6	0	7	14	2	1	9	3
12	10	11	8	13	2	15	3	1	7	0	9	5	6	14	4
13	15	14	1	12	7	10	6	8	2	9	0	4	3	11	5
14	12	13	10	15	4	1	5	3	9	2	11	7	8	0	6
15	1	0	3	14	9	12	8	10	4	11	2	6	5	13	7

$$M = [0, 0, 0, 32, 32]$$

At order $n = 20$ the dihedral directed terrace method found designs that surpasses Lewis and Russell on every metric without searching the entire space.

0	1	2	4	7	3	5	12	19	13	17	9	18	10	15	6	16	11	14	8
1	0	19	17	14	18	16	9	2	8	4	12	3	11	6	15	5	10	7	13
2	3	4	6	9	5	7	14	1	15	19	11	0	12	17	8	18	13	16	10
3	2	1	19	16	0	18	11	4	10	6	14	5	13	8	17	7	12	9	15
4	5	6	8	11	7	9	16	3	17	1	13	2	14	19	10	0	15	18	12
5	4	3	1	18	2	0	13	6	12	8	16	7	15	10	19	9	14	11	17
6	7	8	10	13	9	11	18	5	19	3	15	4	16	1	12	2	17	0	14
7	6	5	3	0	4	2	15	8	14	10	18	9	17	12	1	11	16	13	19
8	9	10	12	15	11	13	0	7	1	5	17	6	18	3	14	4	19	2	16
9	8	7	5	2	6	4	17	10	16	12	0	11	19	14	3	13	18	15	1
10	11	12	14	17	13	15	2	9	3	7	19	8	0	5	16	6	1	4	18
11	10	9	7	4	8	6	19	12	18	14	2	13	1	16	5	15	0	17	3
12	13	14	16	19	15	17	4	11	5	9	1	10	2	7	18	8	3	6	0
13	12	11	9	6	10	8	1	14	0	16	4	15	3	18	7	17	2	19	5
14	15	16	18	1	17	19	6	13	7	11	3	12	4	9	0	10	5	8	2
15	14	13	11	8	12	10	3	16	2	18	6	17	5	0	9	19	4	1	7
16	17	18	0	3	19	1	8	15	9	13	5	14	6	11	2	12	7	10	4
17	16	15	13	10	14	12	5	18	4	0	8	19	7	2	11	1	6	3	9
18	19	0	2	5	1	3	10	17	11	15	7	16	8	13	4	14	9	12	6
19	18	17	15	12	16	14	7	0	6	2	10	1	9	4	13	3	8	5	11

0	16	15	10	8	2	9	14	11	18	6	17	7	4	5	3	19	13	1	12
1	5	6	11	13	19	12	7	10	3	15	4	14	17	16	18	2	8	0	9
2	18	17	12	10	4	11	16	13	0	8	19	9	6	7	5	1	15	3	14
3	7	8	13	15	1	14	9	12	5	17	6	16	19	18	0	4	10	2	11
4	0	19	14	12	6	13	18	15	2	10	1	11	8	9	7	3	17	5	16
5	9	10	15	17	3	16	11	14	7	19	8	18	1	0	2	6	12	4	13
6	2	1	16	14	8	15	0	17	4	12	3	13	10	11	9	5	19	7	18
7	11	12	17	19	5	18	13	16	9	1	10	0	3	2	4	8	14	6	15
8	4	3	18	16	10	17	2	19	6	14	5	15	12	13	11	7	1	9	0
9	13	14	19	1	7	0	15	18	11	3	12	2	5	4	6	10	16	8	17
10	6	5	0	18	12	19	4	1	8	16	7	17	14	15	13	9	3	11	2
11	15	16	1	3	9	2	17	0	13	5	14	4	7	6	8	12	18	10	19
12	8	7	2	0	14	1	6	3	10	18	9	19	16	17	15	11	5	13	4
13	17	18	3	5	11	4	19	2	15	7	16	6	9	8	10	14	0	12	1
14	10	9	4	2	16	3	8	5	12	0	11	1	18	19	17	13	7	15	6
15	19	0	5	7	13	6	1	4	17	9	18	8	11	10	12	16	2	14	3
16	12	11	6	4	18	5	10	7	14	2	13	3	0	1	19	15	9	17	8
17	1	2	7	9	15	8	3	6	19	11	0	10	13	12	14	18	4	16	5
18	14	13	8	6	0	7	12	9	16	4	15	5	2	3	1	17	11	19	10
19	3	4	9	11	17	10	5	8	1	13	2	12	15	14	16	0	6	18	7

$$M = [0, 0, 0, 20, 140]$$

0	1	2	4	7	3	5	13	17	10	18	8	19	14	11	16	9	15	6	12
1	0	19	17	14	18	16	8	4	11	3	13	2	7	10	5	12	6	15	9
2	3	4	6	9	5	7	15	19	12	0	10	1	16	13	18	11	17	8	14
3	2	1	19	16	0	18	10	6	13	5	15	4	9	12	7	14	8	17	11
4	5	6	8	11	7	9	17	1	14	2	12	3	18	15	0	13	19	10	16
5	4	3	1	18	2	0	12	8	15	7	17	6	11	14	9	16	10	19	13
6	7	8	10	13	9	11	19	3	16	4	14	5	0	17	2	15	1	12	18
7	6	5	3	0	4	2	14	10	17	9	19	8	13	16	11	18	12	1	15
8	9	10	12	15	11	13	1	5	18	6	16	7	2	19	4	17	3	14	0
9	8	7	5	2	6	4	16	12	19	11	1	10	15	18	13	0	14	3	17
10	11	12	14	17	13	15	3	7	0	8	18	9	4	1	6	19	5	16	2
11	10	9	7	4	8	6	18	14	1	13	3	12	17	0	15	2	16	5	19
12	13	14	16	19	15	17	5	9	2	10	0	11	6	3	8	1	7	18	4
13	12	11	9	6	10	8	0	16	3	15	5	14	19	2	17	4	18	7	1
14	15	16	18	1	17	19	7	11	4	12	2	13	8	5	10	3	9	0	6
15	14	13	11	8	12	10	2	18	5	17	7	16	1	4	19	6	0	9	3
16	17	18	0	3	19	1	9	13	6	14	4	15	10	7	12	5	11	2	8
17	16	15	13	10	14	12	4	0	7	19	9	18	3	6	1	8	2	11	5
18	19	0	2	5	1	3	11	15	8	16	6	17	12	9	14	7	13	4	10
19	18	17	15	12	16	14	6	2	9	1	11	0	5	8	3	10	4	13	7

0	12	16	5	14	1	15	18	19	6	9	13	11	17	7	2	10	8	3	4
1	9	5	16	7	0	6	3	2	15	12	8	10	4	14	19	11	13	18	17
2	14	18	7	16	3	17	0	1	8	11	15	13	19	9	4	12	10	5	6
3	11	7	18	9	2	8	5	4	17	14	10	12	6	16	1	13	15	0	19
4	16	0	9	18	5	19	2	3	10	13	17	15	1	11	6	14	12	7	8
5	13	9	0	11	4	10	7	6	19	16	12	14	8	18	3	15	17	2	1
6	18	2	11	0	7	1	4	5	12	15	19	17	3	13	8	16	14	9	10
7	15	11	2	13	6	12	9	8	1	18	14	16	10	0	5	17	19	4	3
8	0	4	13	2	9	3	6	7	14	17	1	19	5	15	10	18	16	11	12
9	17	13	4	15	8	14	11	10	3	0	16	18	12	2	7	19	1	6	5
10	2	6	15	4	11	5	8	9	16	19	3	1	7	17	12	0	18	13	14
11	19	15	6	17	10	16	13	12	5	2	18	0	14	4	9	1	3	8	7
12	4	8	17	6	13	7	10	11	18	1	5	3	9	19	14	2	0	15	16
13	1	17	8	19	12	18	15	14	7	4	0	2	16	6	11	3	5	10	9
14	6	10	19	8	15	9	12	13	0	3	7	5	11	1	16	4	2	17	18
15	3	19	10	1	14	0	17	16	9	6	2	4	18	8	13	5	7	12	11
16	8	12	1	10	17	11	14	15	2	5	9	7	13	3	18	6	4	19	0
17	5	1	12	3	16	2	19	18	11	8	4	6	0	10	15	7	9	14	13
18	10	14	3	12	19	13	16	17	4	7	11	9	15	5	0	8	6	1	2
19	7	3	14	5	18	4	1	0	13	10	6	8	2	12	17	9	11	16	15

$$M = [0, 0, 0, 60, 80]$$

Generating Arrays

A **generating array** a grid of pairs with particular properties that allow a row complete latin square to be extrapolated from it. Specifically, a generating array R is an $r \times n$ grid of pairs (x, y) where n is divisible by r, $0 \leq x < r$, and $0 \leq y < \frac{r}{n}$. Each horizontally adjacent pair (a, b) of pairs in the array can be assigned a difference $(a_x - b_x \bmod r, a_y, b_y)$. The array should be constrained such that each possible such difference occurs exactly once and each column contains exactly one of each possible value for y. A row complete latin square A can be filled in using the formula:

$$A_{i,j} = R_{i \bmod r, j} \bmod \frac{r}{n}$$

For example consider the following 3×9 generating array:

$$\begin{array}{ccccccccc}
(0,0) & (0,1) & (1,1) & (2,0) & (1,0) & (0,2) & (2,1) & (1,2) & (2,2) \\
(0,1) & (0,2) & (0,0) & (1,1) & (2,2) & (1,0) & (1,2) & (2,1) & (2,0) \\
(0,2) & (1,0) & (2,2) & (1,2) & (1,1) & (0,1) & (2,0) & (0,0) & (2,1)
\end{array}$$

The difference between, for example, $R_{0,2}$ and $R_{0,3}$ would be calculated as $(1, 1) - (2, 0) = (1 - 2 \bmod 3, 1, 0) = (2, 1, 0)$. This generating array could be used to construct the following row complete latin square:

0	1	4	6	3	2	7	5	8
1	2	0	4	8	3	5	7	6
2	3	8	5	4	1	6	0	7
3	4	7	0	6	5	1	8	2
4	5	3	7	2	6	8	1	0
5	6	2	8	7	4	0	3	1
6	7	1	3	0	8	4	2	5
7	8	6	1	5	0	2	4	3
8	0	5	2	1	7	3	6	4

As with the directed terrace methods, repeats in pair differences for adjacent orthogonal and diagonal pairs in a pair of generating arrays are propagated through the resulting square. For example given a pair of 3×9 generating arrays R and S, if the multiset of differences

$$\{(R_{i,j,x} - S_{i,j,x} \bmod 3, R_{i,j,y}, S_{i,j,y}) \mid 0 \le i < 3, 0 \le j < 9\}$$

has a redundancy of 2 then the resulting pair of latin squares will have $M_0 = 6$.

Generating arrays can be used to construct row complete latin squares at all odd composite orders.[3]
I exhaustively searched the space of pairs of $2 \times n$ and $3 \times n$ generating arrays at orders 2 through 11 keeping track of the pair differences relevant to M_0, M_3, and M_4 and backtracking whenever the metrics were worse then a previously found result. I was also able to begin the 2×12 search.

Unfortunately, none of the results I found with this method were better than results found with other methods although it is worth noting that I was able to find some designs at order 12 with $M_0 = 0$. This means that it is possible to construct pairs of orthogonal row complete latin squares from generating arrays. With more computational resources, this could be a useful approach at orders 12, 14, and 15. Order 15 is of particular interest because Lewis and Russell's results at 15 had no row complete squares. One of the generating array pairs I found at order 12 which produces an orthogonal pair follows:

$(0,0)$	$(0,1)$	$(2,0)$	$(1,0)$	$(3,0)$	$(2,1)$	$(5,0)$	$(1,1)$	$(4,1)$	$(3,1)$	$(5,1)$	$(4,0)$
$(0,1)$	$(0,0)$	$(4,1)$	$(5,1)$	$(3,1)$	$(4,0)$	$(1,1)$	$(5,0)$	$(2,0)$	$(3,0)$	$(1,0)$	$(2,1)$

$(0,0)$	$(4,1)$	$(1,1)$	$(4,0)$	$(3,1)$	$(1,0)$	$(2,1)$	$(0,1)$	$(5,1)$	$(5,0)$	$(3,0)$	$(2,0)$
$(0,1)$	$(2,0)$	$(5,0)$	$(2,1)$	$(3,0)$	$(5,1)$	$(4,0)$	$(0,0)$	$(1,0)$	$(1,1)$	$(3,1)$	$(4,1)$

IV. Conclusion

I have looked at and implemented a variety of computational approaches to this problem. Several improvements to Lewis and Russell's results have been found. The two methods that gave the most promising results were the row permutations method and the dihedral directed terrace method. Both of these methods have somewhat limited potential to produce further useful results. The dihedral directed terrace method is limited to orders n such that $n \geq 12$ and $\frac{n}{4} \in \mathbb{Z}$.[4] The row permutations method is limited by the set of row complete latin squares it searches. At low orders it can be possible to effectively search the entire space of row complete squares. It should be noted however, that the method both Ian Wanless and myself used to reduce this space found 2 squares at order 6, 12 each at orders 8 and 9, and 492 at order 10. There are no known row complete latin squares of order 11 and the space is far to large to exhaustively search. It can only be presumed that a lot more exist at higher orders where this method would thus be far less tractable.

One way of using the row permutations method at higher orders might be to use row complete squares found using other methods. This has the disadvantage of constraining the search to squares with a particular type of structure which could severely limit the improvements that might be made. The generating array method shows particular promise here. I was able to match the $M_0 = 0$ result from the dihedral directed terrace method at order 12 which by itself suggests that the generating array method is promising. I also was able to match my results from the full row permutations method using a single square A from a generating array at order 9 (It has, been shown that all row complete squares at order 9 can be constructed from generating arrays[5, Roman-k and Tuscan-k squares p.14]) and searching the space of designs $(A, A_{permutated})$. I also tried this at order 15 but was unable to get any useful results due to the constraints of available time and hardware. There are definitely designs at order 15 from generating arrays but none that I was able to find had a low enough value for M_0 to make the charts. It is also likely that with more time and better hardware (or perhaps exploring ways of parallelizing the search) results at higher orders would be attainable using the cyclic directed terrace method.

It might also be interesting to try some searches where both M_0 and M_1 or M_2 are allowed to be greater than zero to see if this might allow $2M_0 + M_1 + M_2$ to fall below $2n$. Another interesting direction to go might be to try other search methods like simulated annealing. There are many other possibilities that could be explored and my project was only able to scratch the surface but it was a great opportunity to apply computer science to an interesting combinatorics problem with real world applications and experience combinatorial explosion first hand.

References

[1] S. M. Lewis and K.G. Russell
Crossover designs in the presence of carry-over effects from two factors
Appl. Statist. (1998)
47, Part 3, pp.379-391

[2] Data on Complete and Row-Complete Latin squares
http ://users.monash.edu.au/ iwanless/data/RCLS/
Ian Wanless,
accessed April 2013.

[3] Personal communication with Matt Ollis
2012-2013.

[4] Personal correspondence between Matt Ollis and Ian Wanless
communicated by Matt Ollis, 2012-2013.

[5] Ambrose Sterr,
Computerized mathematics and the mathematics of computation
Plan of Concentration
Marlboro College, 2007.

Appendix: Code

Hot Cocoa Lisp

lib/tokenTypes.js

```
module.exports = [
  { t:'boolean', re:/^(true|false|null|undefined|NaN)\b/ },
  { t:'number', re:/^((-?(0|[1-9][0-9]*)(\.[0-9]+)?([eE][-+]?[0-9]+)?)|-?Infinity)/ },
  { t:'string', re:/^("(\\.|[^\\"])*"|'(\\.|[^\\'])*')/ },
  { t:'identifier', re:/^([a-zA-Z_!?$%&@#|~*+\-=\/<>^][a-zA-Z0-9_!?$%&@#|~*+\-=\/<>^]*
  { t:'(', re:/^\(/ },
  { t:')', re:/^\)/ },
  { t:'[', re:/^\[/ },
  { t:']', re:/^\]/ },
  { t:'{', re:/^{/ },
  { t:'}', re:/^}/ },
  { t:'.', re:/^\./ },
  { t:'comment', re:/^(;[^\n]*\n|;.*$)/ },
  { t:'whitespace', re:/^[\s,:]+/ }
];
```

lib/parseGrammar.json

```
{
  "_program": [
    ["_expression", "_program-tail"]
  ],
  "_program-tail": [
    ["_program"],
    []
  ],
  "_expression": [
    ["_dotted-chain"],
    ["_list"],
    ["_dotted-list"],
    ["_literal-list"],
    ["_object"],
    ["_atom"]
  ],
  "_list": [
    ["(", "_list-tail"]
  ],
  "_dotted-list": [
    ["(", ".", "_list-tail"]
  ],
  "_list-tail": [
    ["_expression", "_list-tail"],
    [")"]
  ],
  "_dotted-chain": [
    ["_obj-reference", ".", "identifier", "_dotted-chain-tail"]
  ],
  "_obj-reference": [
    ["_list"],
    ["_literal-list"],
    ["_object"],
    ["identifier"]
  ],
  "_dotted-chain-tail": [
    [".", "identifier", "_dotted-chain-tail"],
    []
  ],
  "_literal-list": [
    ["[", "_literal-list-tail"]
  ],
  "_literal-list-tail": [
    ["_expression", "_literal-list-tail"],
    ["]"]
  ],
  "_object": [
    ["{", "_object-tail"]
  ],
  "_object-tail": [
    ["_atom", "_expression", "_object-tail"],
    ["}"]
  ],
  "_atom": [
    ["identifier"],
    ["string"],
    ["number"],
    ["boolean"]
  ]
}
```

lib/attributeGrammar.js

```
var types = require('./types.js');

module.exports = require('hot-cocoa').analyzer({
  '_program': function(tree) {
    return this.analyze(tree[1], this.analyze(tree[0]));
  },
  '_program-tail': function(tree, beginning) {
    if (tree[0] && tree[0].type === '_program') {
      var result = this.analyze(tree[0]);
    } else {
      var result = [];
    }
    result.unshift(beginning);
    return result;
  },
  '_expression': function(tree) {
    return this.analyze(tree[0]);
  },
  '_list': function(tree) {
    var result = this.analyze(tree[1]);
    result.position = tree[0].position;
    return result;
  },
  '_dotted-list': function(tree) {
    var result = this.analyze(tree[2]);
    result.unshift(types('identifier', '.', tree[0].position));
    return result;
  },
  '_list-tail': function(tree, beginning) {
    if (tree[0].type === '_expression') {
      var result = this.analyze(tree[1], this.analyze(tree[0]));
    } else {
      var result = types('list', [], tree[0].position);
      result.end = tree[0].position.absolute + 1;
    }
    if (beginning) {
      result.unshift(beginning);
    }
    return result;
  },
  '_literal-list': function(tree) {
    var result = types('list', [], tree[0].position);
    result.push(types('identifier', 'list', tree[0].position));
    var body = this.analyze(tree[1]);
    for (var i = 0; i < body.length; i++) {
      result.push(body[i]);
    }
    return result;
  },
  '_literal-list-tail': function(tree, beginning) {
    return this['_list-tail'](tree, beginning);
  },
  '_object': function(tree) {
    var result = types('list', [], tree[0].position);
    result.push(types('identifier', 'object', tree[0].position));
    var body = this.analyze(tree[1]);
    for (var i = 0; i < body.length; i++) {
      result.push(body[i]);
    }
    return result;
  },
  '_object-tail': function(tree, args) {
    if (args) {
      var key = args[0];
      var value = args[1];
    }
    if (tree[0].type === '_atom') {
      var x = this.analyze(tree[1]);
      var result = this.analyze(tree[2], [this.analyze(tree[0]),
                                          this.analyze(tree[1])]);
    } else {
      var result = types('list', [], tree[0].position);
    }
```

```
    if (key && value) {
      result.unshift(value);
      result.unshift(key);
    }
    return result;
  },
  '_dotted-chain': function(tree) {
    var result = types('list', [], tree[0].position);
    result.push(types('identifier', '.', tree[0].position));
    result.push(this.analyze(tree[0].tree[0]));
    result.push(this.analyze(tree[2]));
    var body = this.analyze(tree[3]);
    for (var i = 0; i < body.length; i++) {
      result.push(body[i]);
    }
    return result;
  },
  '_dotted-chain-tail': function(tree) {
    if (tree.length === 0) {
      return [];
    }
    var result = this.analyze(tree[2]);
    result.unshift(this.analyze(tree[1]));
    return result;
  },
  '_atom': function(tree) {
    switch (tree[0].type) { // perhaps without a switch??
    case '.' :
      return types('identifier', '.', tree[0].position);
      break;
    case 'identifier' :
      return types('identifier', tree[0].text, tree[0].position);
      break;
    case 'string' :
      return types('string', tree[0].text, tree[0].position);
      break;
    case 'number' :
      return types('number', tree[0].text, tree[0].position);
      break;
    case 'boolean' :
      return types('boolean', tree[0].text, tree[0].position);
      break;
    }
  },
  'identifier': function(identifier) {
    return this['_atom']([identifier]);
  }
});
```

lib/compile.js

```
var _ = require('underscore');
var format = require('hot-cocoa').format;
var types = require('./types.js');
var builtins = require('./builtins.js');
var args = require('../lib/args.js');
var fs = require('fs');
var UglifyJS = require("uglify-js");

// load built-in functions
var functions = require('./functions.js');
if (args.b) {
  _.extend(functions.map, require('./browser.js').map);
}

// generate the annotations for the ast at the specified index
var annotation = function(source, asts, index) {
  return source.slice((index > 0) ? asts[index - 1].end : 0, asts[index].end)
    .replace(/^\s*\n/mg, '').replace(/\n\s*$/mg, '').replace(/^/mg, '// ');
};

// validate the arguments of a built in function call
var validate_args = function(number, range, function_name, position) {
  if (range === undefined) {
    throw new Error(format(
      'No argument range specified for `~~` at position ~~',
      [function_name, position]
    ));
  }
  if (typeof(range) === 'number') {
    if (number !== range) {
      throw new Error(format(
        'Wrong number of arguments for `~~`: ~~ for ~~ at position ~~',
        [function_name, number, range, position]
      ));
    }
  } else {
    if (range[0] > number || range[1] < number) {
      throw new Error(format(
        'Wrong number of arguments for `~~`: ~~ for ~~-~~ at position ~~',
        [function_name, number, range[0], range[1], position]
      ));
    }
  }
};

// generate a javascript expression that evaluates to the function to be called
var get_function_reference = function(ast, context) {
  if (ast[0].type === 'identifier' && functions.contains(ast[0].value)) {
    return ast[0].json();
  }
  return compile(ast[0], context);
}

// Compile takes an abstract syntax tree and a context object.  It returns
// JavaScript code.
// The context object contains the following properties:
//   path: The directory that the current .hcl source file is in.
//   dummy_iterator: The number of dummy iterators created, initially 0.
//   add_to_scope: A function which takes one or two arguments.  The first is
//     the name of a variable to be initialized in the current scope, the second
//     is a string of javascript code to determine the initial value of the
//     variable.
var compile = function(ast, context) {

  // TODO: more cleaning up of this function

  if (ast.type === 'list') {

    // validation
    if (ast[0] === undefined) {
      return 'null';
    }
    if (ast[0].type !== 'identifier' && ast[0].type !== 'list') {
      throw new Error(
        format('Cannot call object of type `~~`: `~~` at position ~~',
               [ast[0].type, ast[0].json(), ast[0].position]));
    }

    // prepare the function call
    var function_reference = get_function_reference(ast, context);
    var unmangled_function_refernece = ast[0].value;
    var is_lazy = functions.contains(unmangled_function_refernece) &&
      functions.map[unmangled_function_refernece].lazy;
    var args = _.map(ast.slice(1), function(x) {
      return is_lazy ? x : compile(x, context);
    });
    var format_options = {
      compile: function(x, new_context) {
        // allow a new context to be specified or use the outer one through
        // the closure
        if (new_context === undefined) {
          return compile(x, context);
        }
        return compile(x, new_context);
      },
      context: context
    };

    // built-in function
    if (functions.contains(unmangled_function_refernece)) {
      validate_args(args.length, functions.map[unmangled_function_refernece].args,
                    unmangled_function_refernece, ast.position);
      return functions.format(unmangled_function_refernece, types('list', args),
                              format_options);
    }

    // user defined function call
    return format('~~(~~)', [function_reference, args.join(', ')]);
  }
```

```
  if (ast.type === 'identifier') {
    if (builtins[ast.value] !== undefined) {
      context.add_to_outer_scope(ast.json(), builtins[ast.value]);
    }
  }
  return ast.json();
};

// takes a list of asts and converststs them to Javacscript code
module.exports = function(asts, source, path, overwrite_context) {
  source = source || '';
  var result = '';
  var vars_in_scope = [];
  var vars_declarations = [];

  // context
  var context = {
    path: path,
    dummy_iterator: 0,
    add_to_scope: function(name, value) {
      if (vars_in_scope.indexOf(name) == -1) {
        vars_in_scope.push(name);
        if (typeof(value) === 'string') {
          vars_declarations.push(format('~~ = ~~', [name, value]));
        } else {
          vars_declarations.push(name);
        }
      }
    }
  };
  context.add_to_outer_scope = context.add_to_scope;
  _.extend(context, overwrite_context);
  for (var i = 0; i < asts.length; i++) {
    var ast = asts[i];
    result += format('\n\n~annotation~\n\n~code~;', {
      'annotation': (! context.omit_annotations) ?
        annotation(source, asts, i) :
        '',
      'code': compile(ast, context)
    });
  }

  // underscore
  var underscore = '';
  if (args.u && ! context.repl_mode) {
    if (! context.omit_annotations) {
      underscore += '// Underscore.js\n'
    }
    underscore += fs.readFileSync(
      require('path').dirname(process.mainModule.filename) +
        '/../etc/underscore.js'
    ).toString();
    underscore += 'var _a_ = Object.keys(_); for (var _b_ = 0; _b_ < _a_.leng' +
      '_[_a_[_b_]]; } }';
  }

  // add trailing comments
  if ((! context.omit_annotations) &&
      source.slice(asts[asts.length - 1].end).trim() !== '') {
    result += '\n\n' + source.slice(asts[asts.length - 1].end)
      .replace(/^\n/mg, '').replace(/\n$/mg, '').replace(/^/mg, '// ');
  }

  // add header
  var header = (! context.omit_annotations) ?
    '// compiled from Hot Cocoa Lisp' : '';
  if (vars_in_scope.length > 0) {
    var format_string = args.b ?
      '(function() {\n\n~~\n\n~~var ~~;~~\n\n}).call(this);' :
      '~~\n\n~~var ~~;~~';
    result = format(format_string, [
      header,
      underscore,
      vars_declarations.join(', '),
      result
    ]);
  } else {
    result = format('~~~~~~', [header, underscore, result]);
  }
```

```
  if (args.m) {
    result = format('~~\n~~', [
      header,
      UglifyJS.minify(result, {fromString: true}).code
    ]);
  }

  return result;
}
```

lib/functions.js

```
var _ = require('underscore');
var format = require('hot-cocoa').format;
var helpers = require('./helpers.js');
var file2js = require('../lib/file2js.js');
var mangle = require('./mangle.js');

// check whether the specified identifier is a built-in function and throw an
// error if it is
var require_mutable = function(identifier) {
  if (function_map.contains(identifier) ||
      _.map(_.keys(function_map.map), mangle).indexOf(identifier) !== -1) {
    throw new Error(format('The builtin function `~~` can\'t be redefined',
                           [identifier]));
  }
};

var function_map = require('hot-cocoa').template_map({
  'nop': 'undefined',
  '.': function(args, options) {
    return format('~~~~', [
      options.compile(args[0]),
      _.map(args.slice(1), function(key) {
        return '["' + key.value + '"]';
      }).join('')
    ]);
  },
  'get': '~~[~~]',
  'list': _.compose(helpers.brackets('[', ']'), helpers.joiner(', ')),
```

```
'object': function(args, options) {
  var pairs = [];
  for (var i = 1; i < args.length; i += 2) {
    pairs.push(format('~~: ~~', [
      args[i - 1].type === 'identifier' ?
        '"' + args[i - 1].value + '"' :
        args[i - 1].value,
      options.compile(args[i])
    ]));
  }
  return format('{ ~~ }', [pairs.join(', ')]);
},
'inherit': 'Object.create(~~)',
'begin': function(args) {
  return format('(function() { ~~ return ~~; }).call(this)', [
    _.map(args.slice(0, -1), function(x) { return x + ';'; }).join(' '),
    args.slice(-1)[0]
  ]);
},
'if': '(~~ ? ~~ : ~~)',
'when': function(args) {
  return format('(~~ && (function() { ~~ return ~~; }).call(this))', [
    args[0],
    _.map(args.slice(1, -1), function(x) { return x + ';'; }).join(' '),
    args.slice(-1)[0]
  ]);
},
'cond': function(args, options) {
  return format('(~~ : undefined)', [
    _.map(args, function(pair) {
      return format('~~ ? ~~', _.map(pair, options.compile));
    }).join(' : ')
  ]);
},
'while': function(args) {
  return format('(function() {while (~~) { ~~; }}).call(this)', [
    args[0], args.slice(1).join('; ')
  ]);
},
'for': function(args, options) {
  var iterator_init = '';
  var loop_init_args = _.map(args[0], function(x) {
    return options.compile(x);
  });
  if (args[0].length === 3) {
    var loop_init = loop_init_args.join('; ');
  } else if (args[0].length === 2) {
    require_mutable(loop_init_args[0]);
    var dummy_iterator = format('_i~~_', [options.context.dummy_iterator]);
    options.context.dummy_iterator++;
    options.context.add_to_scope(dummy_iterator);
    options.context.add_to_scope(loop_init_args[0]);
    var loop_init = format('~0~ = 0; ~0~ < ~1~.length; ~0~++',
                           [dummy_iterator, loop_init_args[1]]);
    iterator_init = format('var ~0~ = ~1~[~2~];',
                           [loop_init_args[0], loop_init_args[1], dummy_iterator]);
  } else {
    throw new Error('Invalid loop initialization for `for`');
  }
  return format('(function() {for (~~) { ~~ ~~; }}).call(this)', [
    loop_init,
    iterator_init,
    _.map(args.slice(1), function(x) {
      return options.compile(x);
    }).join('; ')
  ]);
},
'times': function(args, options) {
  require_mutable(options.compile(args[0]));
  options.context.add_to_scope(options.compile(args[0][0]));
  return format('(function() {for (~~) { ~~; }}).call(this)', [
    format('~0~ = 0; ~0~ < ~1~; ~0~++', _.map(args[0], function(x) {
      return options.compile(x);
    })),
    _.map(args.slice(1), function(x) {
      return options.compile(x);
    }).join('; ')
  ]);
},
'error': '(function() {throw new Error(~~);}).call(this)',
'attempt': function(args, options) {

  var try_block, catch_block, finally_block;
  if (args[0][0].value !== 'try') {
    throw new Error('`attempt` must begin with a `try` block');
  }
  if (['catch', 'finally'].indexOf(args[1][0].value) === -1) {
    throw new Error('`attempt` must have a `catch` or `finally` block');
  }

  if (args[1][0].value === 'catch') {
    // there is a catch block
    require_mutable(options.compile(args[1][1]));
    catch_block = format('catch (~~) { ~~ }', [
      options.compile(args[1][1]),
      _.map(args[1].slice(2), function(exp) {
        return format('~~;', [options.compile(exp)]);
      }).join(' ')
    ]);
    if (args.length === 3) {
      if (args[2][0].value === 'finally') {
        // there is a finally block after the catch block
        finally_block = format('finally { ~~ }', [
          _.map(args[2].slice(1), function(exp) {
            return format('~~;', [options.compile(exp)]);
          }).join(' ')
        ]);
      } else {
        throw new Error('third block of `attempt` must be `finally` block');
      }
    }

  } else {
    // there is a finally block and no catch block
    finally_block = format('finally { ~~ }', [
      _.map(args[1].slice(1), function(exp) {
        return format('~~;', [options.compile(exp)]);
      }).join(' ')
    ]);
    if (args.length === 3) {
      throw new Error('third block of `attempt` must be `finally` block');
    }
  }
  try_block = format('try { ~~ }', [
    _.map(args[0].slice(1), function(exp) {
      return format('~~;', [options.compile(exp)]);
    }).join(' ')
  ]);
  return format('(function() {~~ ~~ ~~}).call(this)', [try_block, catch_block, final
},
'+': _.compose(helpers.parens, helpers.joiner(' + ')),
'+1': '(~~ + 1)',
'-': function(args) {
  if (args.length === 1) {
    return format('(- ~~)', args);
  }
  return format('(~~)', [args.join(' - ')]);
},
'--1': '(~~ - 1)',
'*': _.compose(helpers.parens, helpers.joiner(' * ')),
'*2': '(~~ * 2)',
'/': _.compose(helpers.parens, helpers.joiner(' / ')),
'/2': '(~~ / 2)',
'^': 'Math.pow(~~, ~~)',
'^2': '(~0~ * ~0~)',
'sqrt': 'Math.sqrt(~~)',
'%': _.compose(helpers.parens, helpers.joiner(' % ')),
'<': _.compose(helpers.parens, helpers.and_chainer('<')),
'>': _.compose(helpers.parens, helpers.and_chainer('>')),
'<=': _.compose(helpers.parens, helpers.and_chainer('<=')),
'>=': _.compose(helpers.parens, helpers.and_chainer('>=')),
'=': _.compose(helpers.parens, helpers.and_chainer('===')),
'!=': _.compose(helpers.parens, helpers.or_chainer('!==')),
'=0': '(~~ === 0)',
'&': '(~~ & ~~)',
'|': '(~~ | ~~)',
'<<': '(~~ << ~~)',
'>>': '(~~ >> ~~)',
'not': '(! ~~)',
'and': _.compose(helpers.parens, helpers.joiner(' && ')),
'or': _.compose(helpers.parens, helpers.joiner(' || ')),
'xor': '((~0~ || ~1~) && (! (~0~ && ~1~)))',
'def': function(args, options) {
  require_mutable(args[0]);
  options.context.add_to_scope(args[0]);
  if (args.length === 2) {
    return format('~~ = ~~', args);
  }
  return 'undefined';
},
'set': helpers.set('', require_mutable),
'set+': helpers.set('+', require_mutable),
'set-': helpers.set('-', require_mutable),
'set*': helpers.set('*', require_mutable),
'set/': helpers.set('/', require_mutable),
'set%': helpers.set('%', require_mutable),
'\st{+': '~~+}',
'--': '~~--',
'let': function(args, options) {
  var keys = [];
  var values = [];
  for (var i = 1; i < args[0].length; i += 2) {
    keys.push(options.compile(args[0][i - 1]));
    require_mutable(options.compile(args[0][i - 1]));
    values.push(format(', ~~', [options.compile(args[0][i])]));
  }
  var expressions = _.map(args.slice(1, -1), function(exp) {
    return format('~~;', [options.compile(exp)]);
  }).join(' ');
  var final_expression = options.compile(args.slice(-1)[0]);
  return format('(function(~~) {~~ return ~~; }).call(this~~)',
              [keys.join(', '), expressions, final_expression,
         values.join('')]);
},
'#': function(args, options) {
  // sub scope
  var vars_in_scope = [];
  var vars_declarations = [];
  var context = _.extend({}, options.context, {
    add_to_scope: function(name, value) {
      if (vars_in_scope.indexOf(name) == -1) {
        vars_in_scope.push(name);
        if (typeof(value) === 'string') {
          vars_declarations.push(format('~~ = ~~', [name, value]));
        } else {
          vars_declarations.push(name);
        }
      }
    }
  });

  // argument list
  var func_args_list = _.map(args[0], function(name) {
    require_mutable(name.json());
    return name.json();
  });

  // splats
  var splat_assignment = '';
  var m;
  if (m = /^(.*)\.{3}$/.exec(func_args_list.slice(-1)[0])) {
    // the argument list ends in a splat
    func_args_list.pop();
    context.add_to_scope(m[1], format('[].slice.call(arguments, ~~)',
                                       [func_args_list.length]));
  }

  var func_args = func_args_list.join(', ');

  var expressions = _.map(args.slice(1, -1), function(exp) {
    return format('~~;', [options.compile(exp, context)]);
  }).join(' ');
  var final_expression = options.compile(args.slice(-1)[0], context);

  // initialize vars
  var vars_init = (vars_in_scope.length > 0 ?
                                    format('var ~~;', [vars_declarations.join(',
                  ''));

  return format('(function(~~) {~~ ~~ return ~~; })',
                  [func_args, vars_init, expressions, final_expression]);
  },
  'nil?': '(~0~ === null || ~0~ === undefined)',
  'boolean?': '(typeof(~~) === "boolean")',
  'number?': '(typeof(~0~) === "number" && (! isNaN(~0~)))',
  'string?': '(typeof(~~) === "string")',
  'list?': '(Object.prototype.toString.call(~~) === "[object Array]")',
  'object?': '(Object.prototype.toString.call(~~) === "[object Object]")',
  're?': '(Object.prototype.toString.call(~~) === "[object RegExp]")',
  'function?': '(typeof(~~) === "function")',
  'empty?': '(~0~ === null || (~0~).length === 0)',
  'integer?': '(typeof(~0~) === "number" && ~0~ % 1 === 0)',
  'even?': '(~~ % 2 === 0)',
  'odd?': '(~~ % 2 === 1)',
  'contains?': '(~~.indexOf(~~) !== -1)',
  // should this be a builtin??
  'type': '(function(_value_, _signature_) { return ((_signature_ === "[object Array]"
  'string': '(~~).toString()',
  'number': 'parseFloat(~~)',
  'integer': 'Math.floor(parseFloat(~~))',
  're': function(args) {
    return format('(new RegExp(~~, ~~))', [
      args[0].replace(/\\/g, "\\\\"),
      args[1] || '""'
    ]);
  },
  'replace': '~~.replace(~~, ~~)',
  'size': '~~.length',
  'compile': function(args, options) {
    file2js(options.context.path + JSON.parse(args[0]));
    return args[0].replace(/\.hcl"$/, '.js"');
  },
  'from-js': function(args) {
    if (/^[_$a-zA-Z][_$a-zA-Z0-9]*$/.exec(args[0].value) === null) {
      throw new Error(format('`~~` is not a valid JavaScript identifier',
                             [args[0].value]));
    }
    return args[0].value;
  }
});
```

```
// Specify minimum and maximum arguments for functions
function_map.set_properties('nop', { args: 0 });
function_map.set_properties(['++', '--', 'not', 'error', 'inherit', 'empty?',
                             'even?', 'odd?', 'nil?', 'boolean?', 'number?',
                             'string?', 'list?', 'object?', 'function?',
                             'integer?', '+1', '--1', '*2', '/2', '^2', 'sqrt',
                             'string', 'number', 'integer', 'size', '=0',
                             'compile', 'type', 'from-js'], { args: 1 });
function_map.set_properties(['^', 'get', 'contains?', '&', '|', '>>', '<<',
                             'xor'], { args: 2 });
function_map.set_properties(['if', 'replace'], { args: 3 });
function_map.set_properties(['re', 'def'], { args: [1, 2] });
function_map.set_properties(['attempt', 'set', 'set+', 'set-', 'set*', 'set/',
                             'set%'], { args: [2, 3] });
function_map.set_properties(['list', 'object'], { args: [0, Infinity] });
function_map.set_properties(['cond', '-', 'begin'], { args: [1, Infinity] });
function_map.set_properties(['.', 'while', '+', '*', '/', '%', '=', '!=', '<',
                             '>', '<=', '>=', 'and', 'or', 'let', '#', 'for',
                             'times', 'when'], { args: [2, Infinity] });

// Specify some functions as not pre-evaluating all of their arguments
// (i.e. special forms)
function_map.set_properties(['.', 'object', '#', 'let', 'cond', 'for', 'times',
                             'attempt', 'from-js'], { lazy: true });

// Specify synonyms
function_map.set_synonyms('get', ['nth']);
function_map.set_synonyms('error', ['throw']);
function_map.set_synonyms('and', ['and?', '&&']);
function_map.set_synonyms('or', ['or?', '||']);
function_map.set_synonyms('not', ['not?', '!']);
function_map.set_synonyms('++', ['inc']);
function_map.set_synonyms('--', ['dec']);
function_map.set_synonyms('+', ['cat']);
function_map.set_synonyms('*2', ['double']);
function_map.set_synonyms('/2', ['half']);
function_map.set_synonyms('^2', ['square']);
function_map.set_synonyms('%', ['mod']);
function_map.set_synonyms('=', ['is', 'is?', 'eq', 'eq?', 'equal', 'equal?',
                                'equals', 'equals?']);
function_map.set_synonyms('!=', ['isnt', 'isnt?', 'neq', 'neq?']);
function_map.set_synonyms('=0', ['zero?']);
function_map.set_synonyms('<', ['lt?']);
function_map.set_synonyms('>', ['gt?']);
function_map.set_synonyms('<=', ['lte?']);
function_map.set_synonyms('>=', ['gte?']);
function_map.set_synonyms('&', ['bit-and']);
function_map.set_synonyms('|', ['bit-or']);
function_map.set_synonyms('<<', ['bit-shift-left']);
function_map.set_synonyms('>>', ['bit-shift-right']);
function_map.set_synonyms('def', ['var']);
function_map.set_synonyms('#', ['lambda', 'function']);
function_map.set_synonyms('function?', ['lambda?', '#?']);
function_map.set_synonyms('list', ['array']);
function_map.set_synonyms('list?', ['array?']);
function_map.set_synonyms('re', ['regex', 'regexp']);
function_map.set_synonyms('re?', ['regex?', 'regexp?']);
function_map.set_synonyms('size', ['length', 'count']);
function_map.set_synonyms('type', ['typeof']);
function_map.set_synonyms('inherit', ['new']);

module.exports = function_map;
```

lib/types.js

```
var _ = require('underscore');
var format = require('hot-cocoa').format;
var mangle = require('./mangle.js');

// wrap the value in an object wrapper that denotes the specified type
module.exports = function(type, value, position) {
    if (type === 'list') {
        var result = [].slice.call(value, 0);
        result.json = function(not_for_compiler) {
            return format('[~~]', [
                _.map(this, function(x) {
                    return x.json(not_for_compiler);
                }).join(', ')
            ]);
        };
    } else {
        var result = { value: value }
        result.json = function(not_for_compiler) {
            if (this.type === 'identifier' && ! not_for_compiler) {
                return mangle(this.value);
            }
            return this.value;
        };
    }
    result.type = type;
    if (position) {
        result.position = position;
    }
    return result;
}
```

Combinatorics

lib/squares.h

```
typedef struct _list {
  int* list;
  int size;
  int max_size;
}* list;

typedef struct _coord {
  int row;
  int col;
}* coord;

typedef struct _latin_grid {
  int** grid;
  int size;
}* latin_grid;

list new_list(int max_size);
list push(list list, int symbol);
latin_grid new_latin_grid(int size);
```

```c
void print_latin_grid(latin_grid grid);
bool is_latin(latin_grid grid);
bool is_row_complete(latin_grid grid);
bool is_orthogonal(latin_grid grid1, latin_grid grid2);
int row_repeats(latin_grid grid);
int column_repeats(latin_grid grid);
int row_completeness_repeats(latin_grid grid);
int orthogonality_repeats(latin_grid grid1, latin_grid grid2);
int repeats(list list, int space);
int generate_pair_id(int size, int symbol1, int symbol2);
```

lib/squares.c

```c
// Defines an API for interacting with latin squares

#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "squares.h"
#define CELL(square, row, col) ((square)->grid[(row)][(col)])

bool logging;
FILE *logfile;

coord new_coord() {
  return malloc(sizeof(struct _coord));
}

list new_list(int max_size) {
  list list = malloc(sizeof(struct _list));
  list->size = 0;
  list->max_size = max_size;
  list->list = malloc(max_size * sizeof(int));
  return list;
}

list push(list list, int symbol) {
  if (list->size < list->max_size) {
    list->list[list->size] = symbol;
    list->size++;
  }
  return list;
}

latin_grid new_latin_grid(int size) {
  int row, col;
  latin_grid latin_grid = malloc(sizeof(struct _latin_grid));
  latin_grid->size = size;
  latin_grid->grid = malloc(size * sizeof(int *));
  for (row = 0; row < size; row++) {
    latin_grid->grid[row] = malloc(size * sizeof(int));
    for (col = 0; col < size; col++) {
      CELL(latin_grid, row, col) = size;
    }
  }
  return latin_grid;
}

latin_grid square_copy(latin_grid square) {
  int row, col;
  latin_grid result = new_latin_grid(square->size);
  for (row = 0; row < square->size; row++) {
    for (col = 0; col < square->size; col++) {
      CELL(result, row, col) = CELL(square, row, col);
    }
  }
  return result;
}

void grid_write(latin_grid square, coord position, int symbol);

latin_grid normalize_grid(latin_grid grid) {
  int i;
  coord pos = new_coord();
  for (i = 0; i < grid->size; i++) {
    pos->row = 0;
    pos->col = i;
    grid_write(grid, pos, i);
    pos->row = i;
    pos->col = 0;
    grid_write(grid, pos, i);
  }
  return grid;
}

void print_latin_grid(latin_grid grid) {
  int row, col;
  for (row = 0; row < grid->size; row++) {
    for (col = 0; col < grid->size; col++) {
      int symbol = CELL(grid, row, col);
      printf("%c", (symbol < 10) ? '0' + symbol : 'W' + symbol);
    }
    printf("\n");
  }
  printf("\n");
}

void log_latin_grid(latin_grid grid) {
  int row, col;
  for (row = 0; row < grid->size; row++) {
    for (col = 0; col < grid->size; col++) {
      int symbol = CELL(grid, row, col);
      fprintf(logfile, "%c", (symbol < 10) ? '0' + symbol : 'W' + symbol);
    }
    fprintf(logfile, "\n");
  }
  fprintf(logfile, "\n");
}

bool is_latin(latin_grid grid) {
  return (row_repeats(grid) == 0) && (column_repeats(grid) == 0);
}

bool is_row_complete(latin_grid grid) {
  return row_completeness_repeats(grid) == 0;
}

bool is_orthogonal(latin_grid grid1, latin_grid grid2) {
  return orthogonality_repeats(grid1, grid2) == 0;
}

int row_repeats(latin_grid grid) {
  int row, col;
  int count = 0;
  list row_list = new_list(grid->size);
  for (row = 0; row < grid->size; row++) {
    row_list->size = 0;
    for (col = 0; col < grid->size; col++) {
      push(row_list, CELL(grid, row, col));
    }
    count += repeats(row_list, grid->size);
  }
  return count;
}

int column_repeats(latin_grid grid) {
  int row, col;
  int count = 0;
  list column_list = new_list(grid->size);
  for (col = 0; col < grid->size; col++) {
    column_list->size = 0;
    for (row = 0; row < grid->size; row++) {
      push(column_list, CELL(grid, row, col));
    }
    count += repeats(column_list, grid->size);
  }
  return count;
}

int row_completeness_repeats(latin_grid grid) {
  int row, col;
  list pair_list = new_list(grid->size * grid->size);
  for (row = 0; row < grid->size; row++) {
    for (col = 0; col < (grid->size - 1); col++) {
      int pair_id = (CELL(grid, row, col) * grid->size) +
        CELL(grid, row, col + 1);
      push(pair_list, pair_id);
    }
  }
  return repeats(pair_list, grid->size * grid->size);
}

int orthogonality_repeats(latin_grid grid1, latin_grid grid2) {
  int row, col;
  list pair_list = new_list(grid1->size * grid1->size);
  for (row = 0; row < grid1->size; row++) {
    for (col = 0; col < grid1->size; col++) {
      int pair_id = generate_pair_id(grid1->size, CELL(grid1, row, col),
                                     CELL(grid2, row, col));
      push(pair_list, pair_id);
    }
  }
  return repeats(pair_list, grid1->size * grid1->size);
}

int diagonal_repeats(latin_grid grid1, latin_grid grid2) {
  // counts repeated pairs of the (grid1[n,m], grid2[n,m+1])
  int row, col;
  list pair_list = new_list(grid1->size * grid1->size);
  for (row = 0; row < grid1->size; row++) {
    for (col = 0; col < (grid1->size - 1); col++) {
      int pair_id = generate_pair_id(grid1->size, CELL(grid1, row, col),
                                     CELL(grid2, row, col + 1));
      push(pair_list, pair_id);
    }
  }
  return repeats(pair_list, grid1->size * grid1->size);
}

int generate_pair_id(int size, int symbol1, int symbol2) {
  if (symbol1 >= size || symbol2 >= size) {
    return size * size;
  }
  return (symbol1 * size) + symbol2;
}

int repeats(list list, int space) {
  int i;
  int count = 0;
  bool usage[space];
  for (i = 0; i < space; i++) {
    usage[i] = false;
  }
  for (i = 0; i < list->size; i++) {
    if (list->list[i] < space) {
      if (usage[list->list[i]]) {
        count++;
      } else {
        usage[list->list[i]] = true;
      }
    }
  }
  return count;
}

void print_bits(long array, int size) {
  int i;
  for (i = size - 1; i >= 0; i--) {
    printf("%s", (array & (1 << i)) ? "1" : "0");
  }
}

void print_bit_array(int* array, int size) {
  int i;
  for (i = 0; i < size; i++) {
    printf("(%i):", i);
    print_bits((long)array[i], size);
    printf(" ");
  }
  printf("\n");
}

bool verbose;

void report1(latin_grid square) {
  if (verbose) {
    print_latin_grid(square);
  }
  printf("b:%d\n\n", row_completeness_repeats(square));
}

void report2(latin_grid square1, latin_grid square2) {
  if (verbose) {
    print_latin_grid(square1);
    print_latin_grid(square2);
  }

  if (logging) {
    log_latin_grid(square1);
    log_latin_grid(square2);
  }

  int a = orthogonality_repeats(square1, square2);
  int b = row_completeness_repeats(square1);
  int c = row_completeness_repeats(square2);
  int d = diagonal_repeats(square1, square2);
  int e = diagonal_repeats(square2, square1);

  printf("a:%d, b:%d, c:%d, d:%d, e:%d\n",
         a, b, c, d, e);

  printf("2a+b+c:%d, a+b+d:%d, a+c+e:%d, a+b+c+d+e:%d\n\n",
         (2 * a) + b + c,
         a + b + d,
         a + c + e,
         a + b + c + d + e);

  if (logging) {
    fprintf(logfile, "a:%d, b:%d, c:%d, d:%d, e:%d\n",
            a, b, c, d, e);

    fprintf(logfile, "2a+b+c:%d, a+b+d:%d, a+c+e:%d, a+b+c+d+e:%d\n\n",
            (2 * a) + b + c,
            a + b + d,
            a + c + e,
            a + b + c + d + e);
    fflush(logfile);
  }
```

```
}
```

lib/backtrack.c

```
// This file defines the shell of a backtracking search.
// It should be included in another file inwhich the function stubs below are
// implemented.

#include "squares.c"

bool is_finished(latin_grid square, coord position);
bool is_terminal(latin_grid square, coord position);
coord next_coord(latin_grid square, coord position);
bool is_allowed(latin_grid square, coord position, int symbol);
void grid_write(latin_grid square, coord position, int symbol);
void print_success(latin_grid square);

void backtrack(latin_grid square, coord position) {

  if (is_finished(square, position)) {
    print_success(square);
    return;
  }

  if (is_terminal(square, position)) {
    return;
  }

  coord next_position = next_coord(square, position);

  int symbol;
  for (symbol = 0; symbol < square->size; symbol++) {
    if (is_allowed(square, position, symbol)) {
      grid_write(square, position, symbol);
      backtrack(square, next_position);
      grid_write(square, position, square->size);
    }
  }
  grid_write(square, position, square->size); // square->size represents blank
  free(next_position);
```

```c
}

void init();
void loop(int size);
void finish();
int main(int argc, char *argv[]) {

  if ((argc > 2) && (strcmp(argv[2], "--quiet") == 0)) {
    verbose = false;
  } else {
    verbose = true;
  }

  if ((argc > 2) && (strcmp(argv[2], "--log") == 0)) {
    logging = true;
    logfile = fopen("log", "a");
  } else {
    logging = false;
  }

  init();
  int size;
  int max = 3;
  if (argc > 1) {
    max = atoi(argv[1]);
  }
  for (size = 1; size <= max; size++) {
    loop(size);
  }
  finish();
  return 0;
}
```

searches/L$_R$.c

```c
#include "../lib/backtrack.c"

bool is_finished(latin_grid square, coord position) { return false; }

bool is_terminal(latin_grid square, coord position) { return false; }
```

```c
coord next_coord(latin_grid square, coord position) { return new_coord(); }

bool is_allowed(latin_grid square, coord position, int symbol) { return false; }

void set_used(int* array, int a_index, int b_index, bool val) { }

void grid_write(latin_grid square, coord position, int symbol) { }

void print_success(latin_grid square) { }

void init() { }

void gen_cyclic(latin_grid square) {
  int i, j;
  for (i = 1; i < square->size; i++) {
    for (j = 0; j < square->size; j++) {
      CELL(square, i, j) = (CELL(square, (i - 1), j) + 1) % square->size;
    }
  }
}

void even_gen_first_row(latin_grid square) {
  int i;
  CELL(square, 0, 0) = 0;
  for (i = 1; i < square->size; i++) {
    if (i % 2 == 1) {
      CELL(square, 0, i) = (i >> 1) + 1;
    } else {
      CELL(square, 0, i) = square->size - (i >> 1);
    }
  }
}

void even_gen_first_square(latin_grid square) {
  gen_cyclic(square);
}

void even_gen_second_square(latin_grid square1, latin_grid square2) {
  int i, j;
  for (i = 0; i < (square1->size / 2); i++) {
    for (j = 0; j < square1->size; j++) {
      CELL(square2, i, j) = CELL(square1, (i * 2), j);
    }
  }
  for (j = 0; j < square1->size; j++) {
    CELL(square2, (square1->size / 2), j) =
      CELL(square1, (square1->size - 1), j);
  }
  for (i = 1; i < (square1->size / 2); i++) {
    for (j = 0; j < square1->size; j++) {
      CELL(square2, ((square1->size / 2) + i), j) =
                CELL(square1, ((i * 2) - 1), j);
    }
  }
}

void odd_gen_square1_row1(latin_grid square) {
  int i;
  int m = (square->size + 1) / 2;
  int q = (m + 1) >> 1;
  CELL(square, 0, 0) = 0;
  for (i = 1; i < (square->size - m); i++) {
    CELL(square, 0, (i * 2)) = m + i;
  }
  if (square->size > 1) {
    CELL(square, 0, (square->size - 1)) = q;
  }
  for (i = 0; i < (m - 1); i++) {
    if ((m - i) > q) {
      CELL(square, 0, ((i * 2) + 1)) = m - i;
    } else {
      CELL(square, 0, ((i * 2) + 1)) = (m - i) - 1;
    }
  }
}

void odd_gen_square2_row1(latin_grid square1, latin_grid square2) {
  int i;
  for (i = 0; i < square1->size; i++) {
    CELL(square2, 0, i) = (CELL(square1, 0, i) * 2) % square1->size;
  }
}

latin_grid square1, square2;
int a, b, c, d, e;

void loop(size) {
  printf("-- %d --\n", size);
  square1 = new_latin_grid(size);
  square2 = new_latin_grid(size);

  if (size % 2 == 0) {
    even_gen_first_row(square1);
    even_gen_first_square(square1);
    even_gen_second_square(square1, square2);
  } else {
    odd_gen_square1_row1(square1);
    odd_gen_square2_row1(square1, square2);
    gen_cyclic(square1);
    gen_cyclic(square2);
  }

  report2(square1, square2);
}

void finish() {}
```

searches/cyclic$_{\text{even}}$.c

```
#include "../lib/backtrack.c"
#include "../lib/cyclic.c"

// current state
latin_grid square_A, square_B;
coord position;

// row_used_A and row_used_B are bit arrays to keep track of symbol availability
// in the top row of each square. The nth bit of this array is a 1 iff symbol n
// is already taken in that square.
```

```
long row_used_A, row_used_B;
// diff_used_A, diff_used_B, and diff_used_orthogonal are arrays of bits to keep
// track of the availability of pair differences in each square.  The nth bit of
// this array is a 1 iff the difference of n is already used in adjacent pairs
// in that square or in an orthogonal pair.
long diff_used_A, diff_used_B;
int *diff_used_orthogonal, *diff_used_diagonal_AB, *diff_used_diagonal_BA;

int least_AB_repeats, least_diagonal_repeats, least_orthogonal_repeats,
  total_found, equivalent_to_1_used;

int repeat_count(int *repeats) {
  int i, result = 0;
  for (i = 0; i < square_A->size; i++) {
    if (repeats[i] < 0) {
      printf("ASSERT FAILS! : %dth element is %d and should not be negative\n",
             i, repeats[i]);
      exit(0);
    }
    if (repeats[i] > 0) {
      result += repeats[i] - 1;
    }
  }
  return result;
}

// use group theory to fill in the rest of the square from the first row
void fill_in_square(latin_grid square) {
  int row, col;
  for (row = 1; row < square->size; row++) {
    for (col = 0; col < square->size; col++) {
      CELL(square, row, col) =
        cyclic_multiply(row, CELL(square, 0, col), square->size);
    }
  }
}

// use group theory to determine the "difference" between the specified symbol
// and the symbol in the space to the left of the specified position
int row_difference(latin_grid square, coord position, int symbol) {
  if (symbol >= square->size) {
    return square->size;
  }
  if (position->col > 0) {
    return cyclic_divide(symbol, CELL(square, position->row, position->col - 1),
                         square->size);
  }
  return square->size;
}

// use group theory to determine the "difference" between the specified symbol
// and the symbol in the specified position in square_A
int orthogonal_difference(coord position, int symbol) {
  if (symbol >= square_A->size) {
    return square_A->size;
  }
  return cyclic_divide(CELL(square_A, position->row, position->col),
                       symbol, square_A->size);
}

// use group theory to determine the "difference" between the specified symbol
// and the symbol to the left of the specified position in square_A
int diagonal_AB_difference(coord position, int symbol) {
  if (symbol >= square_A->size) {
    return square_A->size;
  }
  if (position->col > 0) {
    return cyclic_divide(CELL(square_A, position->row, position->col - 1),
                         symbol, square_A->size);
  }
  return square_A->size;
}

// use group theory to determine the "difference" between the specified symbol
// and the symbol to the right of the specified position in square_A
int diagonal_BA_difference(coord position, int symbol) {
  if (symbol >= square_A->size) {
    return square_A->size;
  }
  if (position->col < square_A->size - 1) {
    return cyclic_divide(CELL(square_A, position->row, position->col + 1),
                         symbol, square_A->size);
  }
  return square_A->size;
}

// just search the first row
bool is_finished(latin_grid square, coord position) {
  return (position->col >= square->size);
}

// backtrack if a better diagonal result exists
bool is_terminal(latin_grid square, coord position) {
  if (square == square_B) {
    int orthogonal_repeats = repeat_count(diff_used_orthogonal);
    int AB_repeats = repeat_count(diff_used_diagonal_AB);
    int total_diagonal_repeats =
      AB_repeats + repeat_count(diff_used_diagonal_BA);
    if (AB_repeats >= least_AB_repeats &&
        total_diagonal_repeats >= least_diagonal_repeats &&
        orthogonal_repeats >= least_orthogonal_repeats) {
      return true;
    }
  }
  return false;
}

coord next_coord(latin_grid square, coord position) {
  coord result = new_coord();
  result->row = position->row;
  result->col = position->col + 1;
  return result;
}

// check that the symbol, the "difference" between it and the previous symbol
// and the "difference" between it and it's orthogonal pair are all unique
bool is_allowed(latin_grid square, coord position, int symbol) {

  if (square == square_A) {
    if (equivalent_to_1_used == 0 &&
        cyclic_equivalent_to_1(symbol, square->size) &&
        symbol != 1) {
      return false;
    }
  }

  long symbol_mask = 1 << symbol;
  long row_diff_mask = 1 << row_difference(square, position, symbol);

  // TODO: DRY this
  if (square == square_A) {
    return ! ((symbol_mask & row_used_A) ||
              ((position->col > 0) ? row_diff_mask & diff_used_A : 0));
  } else {
    return ! ((symbol_mask & row_used_B) ||
              ((position->col > 0) ? row_diff_mask & diff_used_B : 0));
  }
}

void set_used(long* array, int index, bool val) {
  if (val) {
    long mask = 1 << index;
    array[0] |= mask; // set the value to true
  } else {
    long mask = ~(1 << index);
    array[0] &= mask; // set the value to false
  }
}

// add the symbol to the grid and to the usage arrays
void grid_write(latin_grid square, coord position, int symbol) {

  // TODO: clean this up

  int old_symbol = CELL(square, position->row, position->col);

  if (square == square_A) {

    // old value is available
    set_used(&row_used_A, old_symbol, false);
    // new value is not
    set_used(&row_used_A, symbol, true);

    // diffs
    if (position->col > 0) {
      set_used(&diff_used_A, row_difference(square, position, old_symbol), false);
      set_used(&diff_used_A, row_difference(square, position, symbol), true);
    }

    // cyclic equivalences
    if (cyclic_equivalent_to_1(old_symbol, square->size)) {
      equivalent_to_1_used--;
    }
    if (cyclic_equivalent_to_1(symbol, square->size)) {
      equivalent_to_1_used++;
    }
  } else {
    // old value is available
    set_used(&row_used_B, old_symbol, false);

    // new value is not
    set_used(&row_used_B, symbol, true);

    // diffs
    diff_used_orthogonal[orthogonal_difference(position, old_symbol)]--;
    diff_used_orthogonal[orthogonal_difference(position, symbol)]++;
    if (position->col > 0) {
      set_used(&diff_used_B,
               row_difference(square, position, old_symbol), false);
      set_used(&diff_used_B,
               row_difference(square, position, symbol), true);
      diff_used_diagonal_AB[diagonal_AB_difference(position, old_symbol)]--;
      diff_used_diagonal_AB[diagonal_AB_difference(position, symbol)]++;
    }
    if (position->col < square->size - 1) {
      diff_used_diagonal_BA[diagonal_BA_difference(position, old_symbol)]--;
      diff_used_diagonal_BA[diagonal_BA_difference(position, symbol)]++;
    }
  }
  CELL(square, position->row, position->col) = symbol;
}

void print_success(latin_grid square) {
  if (square == square_B) {
    bool new_result = false;
    int orthogonal_repeats = repeat_count(diff_used_orthogonal);
    int AB_repeats = repeat_count(diff_used_diagonal_AB);
    int total_diagonal_repeats =
      AB_repeats + repeat_count(diff_used_diagonal_BA);
    if (orthogonal_repeats < least_orthogonal_repeats) {
      least_orthogonal_repeats = orthogonal_repeats;
      new_result = true;
    }
    if (AB_repeats < least_AB_repeats &&
        least_orthogonal_repeats == orthogonal_repeats) {
      least_AB_repeats = AB_repeats;
      new_result = true;
    }
    if (total_diagonal_repeats < least_diagonal_repeats &&
        least_orthogonal_repeats == orthogonal_repeats) {
      least_diagonal_repeats = total_diagonal_repeats;
      new_result = true;
    }

    if (verbose && new_result) {
      fill_in_square(square_A);
      fill_in_square(square_B);
      report2(square_A, square_B);
    }

    total_found++;
  } else {
    coord position = new_coord();
    position->row = 0;
    position->col = 1;
    backtrack(square_B, position);
  }
}
```

```c
void init() {
  position = new_coord();
}

void loop(size) {

  // ignore odd orders
  if (size & 1) {
    return;
  }

  printf("-- %d --\n", size);
  if (logging) {
    fprintf(logfile, "-- %d --\n", size);
  }
  total_found = 0;
  row_used_A = row_used_B = diff_used_A = diff_used_B =
    equivalent_to_1_used = 0;
  diff_used_orthogonal = calloc(size + 1, sizeof(int));
  diff_used_diagonal_AB = calloc(size + 1, sizeof(int));
  diff_used_diagonal_BA = calloc(size + 1, sizeof(int));
  least_orthogonal_repeats = 2;
  least_AB_repeats = least_diagonal_repeats = size;
  square_A = new_latin_grid(size);
  square_B = new_latin_grid(size);
  position->row = 0;
  position->col = 0;
  grid_write(square_A, position, 0);
  grid_write(square_B, position, 0);
  position->col = 1;
  backtrack(square_A, position);
  printf("\nfound %i of size %i\n", total_found, size);
  if (logging) {
    fprintf(logfile, "\nfound %i of size %i\n", total_found, size);
  }
}

void finish() {}
```

searches/cyclic$_{\text{odd}}$.c

```c
#include "../lib/backtrack.c"
#include "../lib/cyclic.c"

// current state
latin_grid square_A, square_B;
coord position;

// row_used_A and row_used_B are bit arrays to keep track of symbol availability
// in the top row of each square. The nth bit of this array is a 1 iff symbol n
// is already taken in that square.
long row_used_A, row_used_B;
// diff_used_A, diff_used_B, and diff_used_orthogonal are arrays of bits to keep
// track of the availability of pair differences in each square.  The nth bit of
// this array is a 1 iff the difference of n is already used in adjacent pairs
// in that square or in an orthogonal pair.
long diff_used_orthogonal;
int *diff_used_A, *diff_used_B, *diff_used_diagonal_AB, *diff_used_diagonal_BA;

int least_AB_repeats, least_diagonal_repeats, least_row_repeats_A,
  least_row_repeats_B, total_found, equivalent_to_1_used;

int repeat_count(int *repeats) {
  int i, result = 0;
  for (i = 0; i < square_A->size; i++) {
    if (repeats[i] < 0) {
      printf("ASSERT FAILS! : %dth element is %d and should not be negative\n",
             i, repeats[i]);
      exit(0);
    }
    if (repeats[i] > 0) {
      result += repeats[i] - 1;
    }
  }
  return result;
}

// use group theory to fill in the rest of the square from the first row
void fill_in_square(latin_grid square) {
  int row, col;
  for (row = 1; row < square->size; row++) {
    for (col = 0; col < square->size; col++) {
      CELL(square, row, col) =
        cyclic_multiply(row, CELL(square, 0, col), square->size);
    }
  }
}

// use group theory to determine the "difference" between the specified symbol
// and the symbol in the space to the left of the specified position
int row_difference(latin_grid square, coord position, int symbol) {
  if (symbol >= square->size) {
    return square->size;
  }
  if (position->col > 0) {
    return cyclic_divide(symbol, CELL(square, position->row, position->col - 1),
                         square->size);
  }
  return square->size;
}

// use group theory to determine the "difference" between the specified symbol
// and the symbol in the specified position in square_A
int orthogonal_difference(coord position, int symbol) {
  if (symbol >= square_A->size) {
    return square_A->size;
  }
  return cyclic_divide(CELL(square_A, position->row, position->col),
                       symbol, square_A->size);
}

// use group theory to determine the "difference" between the specified symbol
// and the symbol to the left of the specified position in square_A
int diagonal_AB_difference(coord position, int symbol) {
  if (symbol >= square_A->size) {
    return square_A->size;
  }
  if (position->col > 0) {
    return cyclic_divide(CELL(square_A, position->row, position->col - 1),
                             symbol, square_A->size);
  }
  return square_A->size;
}

// use group theory to determine the "difference" between the specified symbol
// and the symbol to the right of the specified position in square_A
int diagonal_BA_difference(coord position, int symbol) {
  if (symbol >= square_A->size) {
    return square_A->size;
  }
  if (position->col < square_A->size - 1) {
    return cyclic_divide(CELL(square_A, position->row, position->col + 1),
                         symbol, square_A->size);
  }
  return square_A->size;
}

// just search the first row
bool is_finished(latin_grid square, coord position) {
  return (position->col >= square->size);
}

// backtrack if a better diagonal result exists
bool is_terminal(latin_grid square, coord position) {
  if (square == square_B) {
    int row_repeats_A = repeat_count(diff_used_A);
    int row_repeats_B = repeat_count(diff_used_B);
    int AB_repeats = repeat_count(diff_used_diagonal_AB);
    int total_diagonal_repeats =
      AB_repeats + repeat_count(diff_used_diagonal_BA);
    if (AB_repeats >= least_AB_repeats &&
        total_diagonal_repeats >= least_diagonal_repeats &&
        row_repeats_A >= least_row_repeats_A &&
        row_repeats_B >= least_row_repeats_B) {
      return true;
    }
  }
  return false;
}
```

```
coord next_coord(latin_grid square, coord position) {
  coord result = new_coord();
  result->row = position->row;
  result->col = position->col + 1;
  return result;
}

// check that the symbol, the "difference" between it and the previous symbol
// and the "difference" between it and it's orthogonal pair are all unique
bool is_allowed(latin_grid square, coord position, int symbol) {

  if (square == square_A) {
    if (equivalent_to_1_used == 0 &&
        cyclic_equivalent_to_1(symbol, square->size) &&
        symbol != 1) {
      return false;
    }
  }

  long symbol_mask = 1 << symbol;

  if (square == square_A) {
    return ! (symbol_mask & row_used_A);
  } else {
    long orthogonal_diff_mask = 1 << orthogonal_difference(position, symbol);
    return ! ((symbol_mask & row_used_B) ||
              (orthogonal_diff_mask & diff_used_orthogonal));
  }
}

void set_used(long* array, int index, bool val) {
  if (val) {
    long mask = 1 << index;
    array[0] |= mask; // set the value to true
  } else {
    long mask = ~(1 << index);
    array[0] &= mask; // set the value to false
  }
}
```

```
// add the symbol to the grid and to the usage arrays
void grid_write(latin_grid square, coord position, int symbol) {

  // TODO: clean this up

  int old_symbol = CELL(square, position->row, position->col);

  if (square == square_A) {

    // old value is available
    set_used(&row_used_A, old_symbol, false);

    // new value is not
    set_used(&row_used_A, symbol, true);

    // diffs
    if (position->col > 0) {
      diff_used_A[row_difference(square, position, old_symbol)]--;
      diff_used_A[row_difference(square, position, symbol)]++;
    }

    // cyclic equivalences
    if (cyclic_equivalent_to_1(old_symbol, square->size)) {
      equivalent_to_1_used--;
    }
    if (cyclic_equivalent_to_1(symbol, square->size)) {
      equivalent_to_1_used++;
    }
  } else {
    // old value is available
    set_used(&row_used_B, old_symbol, false);

    // new value is not
    set_used(&row_used_B, symbol, true);

    // diffs
    set_used(&diff_used_orthogonal, orthogonal_difference(position, old_symbol), false
    set_used(&diff_used_orthogonal, orthogonal_difference(position, symbol), true);
    if (position->col > 0) {
      diff_used_B[row_difference(square, position, old_symbol)]--;
      diff_used_B[row_difference(square, position, symbol)]++;
      diff_used_diagonal_AB[diagonal_AB_difference(position, old_symbol)]--;
      diff_used_diagonal_AB[diagonal_AB_difference(position, symbol)]++;
    }
    if (position->col < square->size - 1) {
      diff_used_diagonal_BA[diagonal_BA_difference(position, old_symbol)]--;
      diff_used_diagonal_BA[diagonal_BA_difference(position, symbol)]++;
    }
  }
  CELL(square, position->row, position->col) = symbol;
}

void print_success(latin_grid square) {
  if (square == square_B) {
    bool new_result = false;
    int row_repeats_A = repeat_count(diff_used_A);
    int row_repeats_B = repeat_count(diff_used_B);
    int AB_repeats = repeat_count(diff_used_diagonal_AB);
    int total_diagonal_repeats =
      AB_repeats + repeat_count(diff_used_diagonal_BA);
    if (row_repeats_A < least_row_repeats_A) {
      least_row_repeats_A = row_repeats_A;
      new_result = true;
    }
    if (row_repeats_B < least_row_repeats_B) {
      least_row_repeats_B = row_repeats_B;
      new_result = true;
    }
    if (AB_repeats < least_AB_repeats &&
        least_row_repeats_A == row_repeats_A &&
        least_row_repeats_B == row_repeats_B) {
      least_AB_repeats = AB_repeats;
      new_result = true;
    }
    if (total_diagonal_repeats < least_diagonal_repeats &&
        least_row_repeats_A == row_repeats_A &&
        least_row_repeats_B == row_repeats_B) {
      least_diagonal_repeats = total_diagonal_repeats;
      new_result = true;
    }

    if (verbose && new_result) {
      fill_in_square(square_A);
      fill_in_square(square_B);
      report2(square_A, square_B);
    }

    total_found++;
  } else {
    coord position = new_coord();
    position->row = 0;
    position->col = 1;
    backtrack(square_B, position);
  }
}

void init() {
  position = new_coord();
}

void loop(size) {

  // ignore odd orders
  if ((size & 1) == 0) {
    return;
  }

  printf("-- %d --\n", size);
  if (logging) {
    fprintf(logfile, "-- %d --\n", size);
  }
  total_found = 0;
  row_used_A = row_used_B = diff_used_orthogonal = equivalent_to_1_used = 0;
  diff_used_A = calloc(size + 1, sizeof(int));
  diff_used_B = calloc(size + 1, sizeof(int));
  diff_used_diagonal_AB = calloc(size + 1, sizeof(int));
  diff_used_diagonal_BA = calloc(size + 1, sizeof(int));
  least_row_repeats_A = least_row_repeats_B = 1;
  least_AB_repeats = least_diagonal_repeats = size;
  square_A = new_latin_grid(size);
  square_B = new_latin_grid(size);
  position->row = 0;
  position->col = 0;
  grid_write(square_A, position, 0);
  grid_write(square_B, position, 0);
  position->col = 1;
  backtrack(square_A, position);
  printf("\nfound %i of size %i\n", total_found, size);
  if (logging) {
    fprintf(logfile, "\nfound %i of size %i\n", total_found, size);
  }
}

void finish() {}
```

searches/dihedral$_{\text{search}}$.c

```
#include "../lib/backtrack.c"
#include "../lib/dihedral.c"

// current state
latin_grid square_A, square_B;
coord position;

// row_used_A and row_used_B are bit arrays to keep track of symbol availability
// in the top row of each square. The nth bit of this array is a 1 iff symbol n
// is already taken in that square.
long row_used_A, row_used_B;
// diff_used_A, diff_used_B, and diff_used_orthogonal are arrays of bits to keep
// track of the availability of pair differences in each square.  The nth bit of
// this array is a 1 iff the difference of n is already used in adjacent pairs
// in that square or in an orthogonal pair.
long diff_used_A, diff_used_B, diff_used_orthogonal;
int *diff_used_diagonal_AB, *diff_used_diagonal_BA;

int least_AB_repeats, least_diagonal_repeats, total_found,
  equivalent_to_v_used, equivalent_to_u_used;

int diagonal_repeat_count(int *repeats) {
  int i, result = 0;
  for (i = 0; i < square_A->size; i++) {
    if (repeats[i] < 0) {
      printf("ASSERT FAILS! : %dth element is %d and should not be negative\n",
             i, repeats[i]);
      exit(0);
    }
    if (repeats[i] > 0) {
      result += repeats[i] - 1;
    }
  }
  return result;
}

// use group theory to fill in the rest of the square from the first row
void fill_in_square(latin_grid square) {
  int row, col;
  for (row = 1; row < square->size; row++) {
    for (col = 0; col < square->size; col++) {
      CELL(square, row, col) =
        dihedral_multiply(row, CELL(square, 0, col), square->size);
    }
  }
}

// use group theory to determine the "difference" between the specified symbol
// and the symbol in the space to the left of the specified position
int row_difference(latin_grid square, coord position, int symbol) {
  if (symbol >= square->size) {
    return square->size;
  }
  if (position->col > 0) {
    return dihedral_divide(symbol, CELL(square, position->row, position->col - 1),
                           square->size);
  }
  return square->size;
}

// use group theory to determine the "difference" between the specified symbol
// and the symbol in the specified position in square_A
```

```
int orthogonal_difference(coord position, int symbol) {
  if (symbol >= square_A->size) {
    return square_A->size;
  }
  return dihedral_divide(CELL(square_A, position->row, position->col),
                         symbol, square_A->size);
}

// use group theory to determine the "difference" between the specified symbol
// and the symbol to the left of the specified position in square_A
int diagonal_AB_difference(coord position, int symbol) {
  if (symbol >= square_A->size) {
    return square_A->size;
  }
  if (position->col > 0) {
    return dihedral_divide(CELL(square_A, position->row, position->col - 1),
                           symbol, square_A->size);
  }
  return square_A->size;
}

// use group theory to determine the "difference" between the specified symbol
// and the symbol to the right of the specified position in square_A
int diagonal_BA_difference(coord position, int symbol) {
  if (symbol >= square_A->size) {
    return square_A->size;
  }
  if (position->col < square_A->size - 1) {
    return dihedral_divide(CELL(square_A, position->row, position->col + 1),
                           symbol, square_A->size);
  }
  return square_A->size;
}

// just search the first row
bool is_finished(latin_grid square, coord position) {
  return (position->col >= square->size);
}

// backtrack if a better diagonal result exists
```

```
bool is_terminal(latin_grid square, coord position) {
  if (square == square_B) {
    int AB_repeats = diagonal_repeat_count(diff_used_diagonal_AB);
    int total_diagonal_repeats =
      AB_repeats + diagonal_repeat_count(diff_used_diagonal_BA);
    if (AB_repeats >= least_AB_repeats &&
        total_diagonal_repeats >= least_diagonal_repeats) {
      return true;
    }
  }
  return false;
}

coord next_coord(latin_grid square, coord position) {
  coord result = new_coord();
  result->row = position->row;
  result->col = position->col + 1;
  return result;
}

// check that the symbol, the "difference" between it and the previous symbol
// and the "difference" between it and it's orthogonal pair are all unique
bool is_allowed(latin_grid square, coord position, int symbol) {

  if (square == square_A) {
    if (equivalent_to_v_used == 0 &&
        dihedral_equivalent_to_v(symbol, square->size) &&
        symbol != 1) {
      return false;
    }
    if (equivalent_to_u_used == 0 &&
        dihedral_equivalent_to_u(symbol, square->size) &&
        symbol != 2) {
      return false;
    }
  }

  long symbol_mask = 1 << symbol;
  long row_diff_mask = 1 << row_difference(square, position, symbol);
  if (square == square_A) {
    return ! ((symbol_mask & row_used_A) ||
              ((position->col > 0) ? row_diff_mask & diff_used_A : 0));
  } else {
    long orthogonal_diff_mask = 1 << orthogonal_difference(position, symbol);
    return ! ((symbol_mask & row_used_B) ||
              ((position->col > 0) ? row_diff_mask & diff_used_B : 0) ||
              (orthogonal_diff_mask & diff_used_orthogonal));
  }
}

void set_used(long* array, int index, bool val) {
  if (val) {
    long mask = 1 << index;
    array[0] |= mask; // set the value to true
  } else {
    long mask = ~(1 << index);
    array[0] &= mask; // set the value to false
  }
}

// add the symbol to the grid and to the usage arrays
void grid_write(latin_grid square, coord position, int symbol) {

  // TODO: clean this up

  int old_symbol = CELL(square, position->row, position->col);

  if (square == square_A) {
    // old value is available
    set_used(&row_used_A, old_symbol, false);

    // new value is not
    set_used(&row_used_A, symbol, true);

    // diffs
    if (position->col > 0) {
      set_used(&diff_used_A, row_difference(square, position, old_symbol), false);
      set_used(&diff_used_A, row_difference(square, position, symbol), true);
    }
    // dihedral equivalences
    if (dihedral_equivalent_to_v(old_symbol, square->size)) {
      equivalent_to_v_used--;
    }
    if (dihedral_equivalent_to_u(old_symbol, square->size)) {
      equivalent_to_u_used--;
    }
    if (dihedral_equivalent_to_v(symbol, square->size)) {
      equivalent_to_v_used++;
    }
    if (dihedral_equivalent_to_u(symbol, square->size)) {
      equivalent_to_u_used++;
    }
  } else {
    // old value is available
    set_used(&row_used_B, old_symbol, false);

    // new value is not
    set_used(&row_used_B, symbol, true);

    // diffs
    set_used(&diff_used_orthogonal,
             orthogonal_difference(position, old_symbol), false);
    set_used(&diff_used_orthogonal,
             orthogonal_difference(position, symbol), true);
    if (position->col > 0) {
      set_used(&diff_used_B,
               row_difference(square, position, old_symbol), false);
      set_used(&diff_used_B,
               row_difference(square, position, symbol), true);
      diff_used_diagonal_AB[diagonal_AB_difference(position, old_symbol)]--;
      diff_used_diagonal_AB[diagonal_AB_difference(position, symbol)]++;
    }
    if (position->col < square->size - 1) {
      diff_used_diagonal_BA[diagonal_BA_difference(position, old_symbol)]--;
      diff_used_diagonal_BA[diagonal_BA_difference(position, symbol)]++;
    }
  }
  CELL(square, position->row, position->col) = symbol;
```

```
}

void print_success(latin_grid square) {
  if (square == square_B) {
    bool new_result = false;
    int AB_repeats = diagonal_repeat_count(diff_used_diagonal_AB);
    int total_diagonal_repeats =
      AB_repeats + diagonal_repeat_count(diff_used_diagonal_BA);
    if (AB_repeats < least_AB_repeats) {
      least_AB_repeats = AB_repeats;
      new_result = true;
    }
    if (total_diagonal_repeats < least_diagonal_repeats) {
      least_diagonal_repeats = total_diagonal_repeats;
      new_result = true;
    }

    if (verbose && new_result) {
      fill_in_square(square_A);
      fill_in_square(square_B);
      report2(square_A, square_B);
    }

    total_found++;
  } else {
    coord position = new_coord();
    position->row = 0;
    position->col = 1;
    backtrack(square_B, position);
  }
}

void init() {
  position = new_coord();
}

void loop(size) {

  // ignore orders besides 12, 16, and 20
  if (size != 12 && size != 16 && size != 20) {
    return;
  }

  printf("-- %d --\n", size);
  total_found = 0;
  row_used_A = row_used_B = diff_used_A = diff_used_B = diff_used_orthogonal =
    equivalent_to_v_used = equivalent_to_u_used = 0;
  diff_used_diagonal_AB = calloc(size + 1, sizeof(int));
  diff_used_diagonal_BA = calloc(size + 1, sizeof(int));
  least_AB_repeats = least_diagonal_repeats = size;
  square_A = new_latin_grid(size);
  square_B = new_latin_grid(size);
  position->row = 0;
  position->col = 0;
  grid_write(square_A, position, 0);
  grid_write(square_B, position, 0);
  position->col = 1;
  backtrack(square_A, position);
  printf("\nfound %i of size %i\n", total_found, size);
}

void finish() {}
```

searches/generating$_{\text{arrays}}$.c

```
#include "../lib/backtrack.c"
#include "../lib/generating_arrays.c"

// current state
latin_grid square_A, square_B;
coord position;
int rows;

// TODO: revise these comments

// row_used_A and row_used_B are bit arrays to keep track of symbol availability
// in the top row of each square. The nth bit of this array is a 1 iff symbol n
// is already taken in that square.
long *row_used_A, *row_used_B;
// diff_used_A, diff_used_B, and diff_used_orthogonal are arrays of bits to keep
// track of the availability of pair differences in each square.  The nth bit of
// this array is a 1 iff the difference of n is already used in adjacent pairs
// in that square or in an orthogonal pair.
long diff_used_A, diff_used_B;
int *diff_used_orthogonal, *diff_used_diagonal_AB, *diff_used_diagonal_BA;

int least_AB_repeats, least_diagonal_repeats, least_orthogonal_repeats,
  total_found, equivalent_to_value1_used, equivalent_to_label1_used;

int repeat_count(int *repeats) {
  int i, result = 0;
  for (i = 0; i < square_A->size * rows; i++) {
    if (repeats[i] < 0) {
      printf("ASSERT FAILS! : %dth element is %d and should not be negative\n",
             i, repeats[i]);
      exit(0);
    }
    if (repeats[i] > 0) {
      result += repeats[i] - 1;
    }
  }
  return result;
}

// fill in the rest of the square from the rows
void fill_in_square(latin_grid square) {
  int row, col;
  for (row = rows; row < square->size; row++) {
    for (col = 0; col < square->size; col++) {
      CELL(square, row, col) = ga_cycle(CELL(square, row % rows, col),
                                        row / rows,
                                        rows,
                                        square->size);
    }
  }
}

// determine the "difference" between the specified symbol and the symbol in the
// space to the left of the specified position
int row_difference(latin_grid square, coord position, int symbol) {
  if (symbol >= square->size) {
    return square->size * rows;
  }
  if (position->col > 0) {
    return ga_difference(CELL(square, position->row, position->col - 1),
                         symbol,
                         rows,
                         square->size);
  }
  return square->size * rows;
}

// determine the "difference" between the specified symbol and the symbol in the
// specified position in square_A
int orthogonal_difference(coord position, int symbol) {
  if (symbol >= square_A->size) {
    return square_A->size * rows;
  }
  return ga_difference(CELL(square_A, position->row, position->col),
                       symbol,
                       rows,
                       square_A->size);
}

// determine the "difference" between the specified symbol and the symbol to the
// left of the specified position in square_A
int diagonal_AB_difference(coord position, int symbol) {
  if (symbol < square_A->size && position->col > 0) {
    return ga_difference(CELL(square_A, position->row, position->col - 1),
                         symbol,
                         rows,
                         square_A->size);
  }
  return square_A->size * rows;
}

// use group theory to determine the "difference" between the specified symbol
// and the symbol to the right of the specified position in square_A
int diagonal_BA_difference(coord position, int symbol) {
  if (symbol < square_A->size && position->col < square_A->size - 1) {
    return ga_difference(CELL(square_A, position->row, position->col + 1),
                         symbol,
                         rows,
                         square_A->size);
  }
  return square_A->size * rows;
}

// just search the first row
bool is_finished(latin_grid square, coord position) {
  return (position->col >= square->size);
}

// backtrack if a better diagonal result exists
bool is_terminal(latin_grid square, coord position) {
  if (square == square_B) {
    int orthogonal_repeats = repeat_count(diff_used_orthogonal);
    int AB_repeats = repeat_count(diff_used_diagonal_AB);
    int total_diagonal_repeats =
      AB_repeats + repeat_count(diff_used_diagonal_BA);
    if (orthogonal_repeats > least_orthogonal_repeats ||
        (AB_repeats >= least_AB_repeats &&
         total_diagonal_repeats >= least_diagonal_repeats &&
         orthogonal_repeats == least_orthogonal_repeats)) {
      return true;
    }
  }
  return false;
}

coord next_coord(latin_grid square, coord position) {
  coord result = new_coord();
  result->row = position->row + 1;
  result->col = position->col;
  if (result->row >= rows) {
    result->row = 0;
    result->col = position->col + 1;
  }
  return result;
}
```

```c
// check that the symbol, the "difference" between it and the previous symbol
// and the "difference" between it and it's orthogonal pair are all unique
bool is_allowed(latin_grid square, coord position, int symbol) {

  if (square == square_A && position->row == 0) {
    if (equivalent_to_value1_used == 0 &&
        ga_value_equivalent_to_1(symbol, rows, square->size) &&
        ga_value(symbol, rows, square->size) != 1) {
      return false;
    }
    if (equivalent_to_label1_used == 0 &&
        ga_label_equivalent_to_1(symbol, rows, square->size) &&
        ga_label(symbol, rows, square->size) != 1) {
      return false;
    }
  }

  // labels in columns must be unique
  int i;
  for (i = 0; i < rows; i++) {
    if (i != position->row &&
        ga_label(CELL(square, i, position->col), rows, square->size) ==
        ga_label(symbol, rows, square->size)) {
      return false;
    }
  }

  long symbol_mask = 1 << symbol;
  long row_diff_mask = 1 << row_difference(square, position, symbol);

  // TODO: DRY this
  if (square == square_A) {
    return ! ((symbol_mask & row_used_A[position->row]) ||
              ((position->col > 0) ? row_diff_mask & diff_used_A : 0));
  } else {
    return ! ((symbol_mask & row_used_B[position->row]) ||
              ((position->col > 0) ? row_diff_mask & diff_used_B : 0));
  }
}
```

```
void set_used(long* array, int index, bool val) {
  if (val) {
    long mask = 1 << index;
    array[0] |= mask; // set the value to true
  } else {
    long mask = ~(1 << index);
    array[0] &= mask; // set the value to false
  }
}

// add the symbol to the grid and to the usage arrays
void grid_write(latin_grid square, coord position, int symbol) {

  // TODO: clean this up

  int old_symbol = CELL(square, position->row, position->col);

  if (square == square_A) {

    // old value is available
    set_used(&row_used_A[position->row], old_symbol, false);

    // new value is not
    set_used(&row_used_A[position->row], symbol, true);

    // diffs
    if (position->col > 0) {
      set_used(&diff_used_A, row_difference(square, position, old_symbol), false);
      set_used(&diff_used_A, row_difference(square, position, symbol), true);
    }

    // equivalences
    if (position->row == 0) {
      if (ga_label_equivalent_to_1(old_symbol, rows, square->size)) {
        equivalent_to_label1_used--;
      }
      if (ga_label_equivalent_to_1(symbol, rows, square->size)) {
        equivalent_to_label1_used++;
      }
      if (ga_value_equivalent_to_1(old_symbol, rows, square->size)) {
        equivalent_to_value1_used--;
      }
      if (ga_value_equivalent_to_1(symbol, rows, square->size)) {
        equivalent_to_value1_used++;
      }
    }
  } else {
    // old value is available
    set_used(&row_used_B[position->row], old_symbol, false);

    // new value is not
    set_used(&row_used_B[position->row], symbol, true);

    // diffs
    diff_used_orthogonal[orthogonal_difference(position, old_symbol)]--;
    diff_used_orthogonal[orthogonal_difference(position, symbol)]++;
    if (position->col > 0) {
      set_used(&diff_used_B,
               row_difference(square, position, old_symbol), false);
      set_used(&diff_used_B,
               row_difference(square, position, symbol), true);
      diff_used_diagonal_AB[diagonal_AB_difference(position, old_symbol)]--;
      diff_used_diagonal_AB[diagonal_AB_difference(position, symbol)]++;
    }
    if (position->col < square->size - 1) {
      diff_used_diagonal_BA[diagonal_BA_difference(position, old_symbol)]--;
      diff_used_diagonal_BA[diagonal_BA_difference(position, symbol)]++;
    }
  }
  CELL(square, position->row, position->col) = symbol;
}

void print_success(latin_grid square) {
  if (square == square_B) {
    bool new_result = false;
    int orthogonal_repeats = repeat_count(diff_used_orthogonal);
    int AB_repeats = repeat_count(diff_used_diagonal_AB);
    int total_diagonal_repeats =
      AB_repeats + repeat_count(diff_used_diagonal_BA);
    if (orthogonal_repeats < least_orthogonal_repeats) {
      least_orthogonal_repeats = orthogonal_repeats;
      least_AB_repeats = least_diagonal_repeats = square->size * rows;
      new_result = true;
    }
    if (AB_repeats < least_AB_repeats &&
        least_orthogonal_repeats == orthogonal_repeats) {
      least_AB_repeats = AB_repeats;
      new_result = true;
    }
    if (total_diagonal_repeats < least_diagonal_repeats &&
        least_orthogonal_repeats == orthogonal_repeats) {
      least_diagonal_repeats = total_diagonal_repeats;
      new_result = true;
    }

    if (verbose && new_result) {
      fill_in_square(square_A);
      fill_in_square(square_B);
      report2(square_A, square_B);
    }

    total_found++;
  } else {
    coord position = new_coord();
    position->row = 0;
    position->col = 1;
    backtrack(square_B, position);
  }
}

void init() {
  position = new_coord();
}

void run_search(size) {
  printf("-- %d x %d --\n", rows, size);
  total_found = 0;
  row_used_A = calloc(3, sizeof(long));
  row_used_B = calloc(3, sizeof(long));
  diff_used_A = diff_used_B = equivalent_to_value1_used =
    equivalent_to_label1_used = 0;
  diff_used_orthogonal = calloc((size * rows) + 1, sizeof(int));
  diff_used_diagonal_AB = calloc((size * rows) + 1, sizeof(int));
  diff_used_diagonal_BA = calloc((size * rows) + 1, sizeof(int));
  least_orthogonal_repeats = least_AB_repeats =
    least_diagonal_repeats = size * rows;
  square_A = new_latin_grid(size);
  square_B = new_latin_grid(size);
  position->row = 0;
  position->col = 0;
  for (; position->row < rows; position->row++) {
    grid_write(square_A, position, position->row);
    grid_write(square_B, position, position->row);
  }
  position->row = 0;
  position->col = 1;
  backtrack(square_A, position);
  printf("\nfound %i of size %i\n", total_found, size);
}

void loop(size) {

  if (size % 2 == 0) {
    rows = 2;
    run_search(size);
  }
  if (size % 3 == 0) {
    rows = 3;
    run_search(size);
  }
}

void finish() {}
```

searches/row$_{\text{permutations}}$.c

```
#include "../lib/squares.c"
#include "../lib/read_row_complete.c"
```

```
#define MIN(X,Y) ((X) < (Y) ? (X) : (Y))

void grid_write(latin_grid square, coord position, int symbol) {}

int best_orthogonal, best_diagonal, best_2AD, best_2ADE, best_DE_for_A;

latin_grid temp_square2;
unsigned int rows_used;
int current_row;

void initialize(int size) {
  temp_square2 = new_latin_grid(size);
  rows_used = 0;
  current_row = 0;
}

bool is_available(int row) {
  // check whether the rowth bit of row_used is set
  return ! ((1 << row) & rows_used);
}

void choose_row(int row, latin_grid square2) {
  // put the rowth row of square2 into the next row in temp_square2
  int i;
  for (i = 0; i < square2->size; i++) {
    CELL(temp_square2, current_row, i) = CELL(square2, row, i);
  }
  current_row++;
  rows_used |= 1 << row;
}

void unchoose_row(int row, latin_grid square2) {
  // blank the current row of temp_square2 and mark the rowth row as being
  // available from square2
  current_row--;
  int i;
  for (i = 0; i < square2->size; i++) {
    CELL(temp_square2, current_row, i) = square2->size;
  }
  rows_used &= ~(1 << row);
```

```
}

bool check_metric(int metric, int* best, bool complete, latin_grid square1) {
  // check whether this metric is the best in this category
  // return true if it is worse in this category
  // print the squares if it is best in this category and complete
  //printf("m:%d b:%d\n", metric, *best);
  if (metric < *best) {
    if (complete) {
      *best = metric;
      report2(square1, temp_square2);
    }
    return false;
  }
  return true;
}

bool check_metrics(latin_grid square1) {
  // check whether this the best in any category.
  // return true if it is the worse than the best so far in every category
  // print it if the the square is complete and it is the best in some category
  bool is_complete = current_row == square1->size;
  int orthogonal = orthogonality_repeats(square1, temp_square2);
  int diagonal_AB = diagonal_repeats(square1, temp_square2) ;
  int diagonal_BA = diagonal_repeats(square1, temp_square2) ;
  int DE = diagonal_AB + diagonal_BA;
  if (orthogonal < best_orthogonal) {
    if (is_complete) {
      best_DE_for_A = DE;
      best_orthogonal = orthogonal;
      report2(square1, temp_square2);
    }
    return false;
  }
  if (orthogonal == best_orthogonal && DE < best_DE_for_A) {
    if (is_complete) {
      best_DE_for_A = DE;
      report2(square1, temp_square2);
    }
    return false;
  }
  return true;
}

void compare_permutations(latin_grid square1, latin_grid square2) {
  // search the space of permutations of the rows of the second square
  // stop if the current pair is worse in every metric than a pair we've found

  int i;
  for (i = 0; i < square1->size; i++) {
    //printf("row %d available? %d (%d)\n", i, is_available(i), rows_used);
    if (is_available(i)) {
      //printf("choosing row %d for row %d\n", i, current_row);
      choose_row(i, square2);
      bool bad_metrics = check_metrics(square1);
      //printf("bm:%d incomp:%d both: %d\n", (! bad_metrics), current_row < square1->s
      if ((! bad_metrics) && current_row < square1->size) {
        //printf("recurse\n");
        compare_permutations(square1, square2);
      }
      unchoose_row(i, square2);
    }
  }
}

void compare_pairs(int size, square_list squares) {
  initialize(size);
  square_list sq_ls1, sq_ls2;
  for (sq_ls1 = squares; sq_ls1 != NULL; sq_ls1 = sq_ls1->next) {
    for (sq_ls2 = sq_ls1; sq_ls2 != NULL; sq_ls2 = sq_ls2->next) {
      // iterate over each pair of squares (order doesn't matter but each square
      // should be compared to itself

      compare_permutations(sq_ls1->square, sq_ls2->square);
    }
  }
}

int main(int argc, char *argv[]) {
  // read squares from file
  if (argc <= 1) {
    printf("No order specified\n");
    return 1;
  }
  square_list squares = file2squares(argv[1]);
  if (squares == NULL) {
    return 1;
  }
  int size = squares->square->size;

  // initialize best metrics
  best_orthogonal = size;
  best_diagonal = size;
  best_2AD = 3 * size;
  best_2ADE = 4 * size;
  best_DE_for_A = size * size * 2;

  // look at all permuted combinations
  verbose = true;
  compare_pairs(size, squares);

  return 0;
}
```