C M S I   3 8 6

# Final Exam Answers

1. It's just

```
Value:         1  2  3  4
Value-Result:  0  2  0  4
Reference:     0  2 -1  4
Name:          0 -1  3  4
```

2. Under static scope, prints 1 - 1 + 1 = 1, Under dynamic scope, prints 1 - 4 + 1 = -2.

3. Under deep binding, prints 17 - 17 = 0, Under shallow binding, prints 30 - 17 = 13.

4. I slipped in some regexes here. You probably might have figured out how to use them while googling during the exam. If not, no worries, there are plenty of other ways to do the substitutions.

```
var mu = function () {
    var data = "MI";
    return {
        value: function () {
            return data;
        },
        rule1: function () {
            if (/I$/).test(data) data += "U";
        },
        rule2: function () {
            data = data + data.slice(1);
        },
        rule3: function () {
            data = data.replace(/III/, "U");
        },
        rule4: function () {
            data = data.replace(/UU/, "");
        }
    };
}();
```

5.

```
class Student:
    def __init__(self, id, name, birthday):
        self.id = id
        self.name = name
        self.birthday = birthday
        self.transcript = {}
    def grade(term, course):
        return self.transcript[term].get(course) if term in transcript else None
    def addTranscriptItem(self, term, course, grade):
        if term not in transcript:
            self.transcript[term] = {}
        self.transcript[term][course] = grade
```

I didn't hide anything here. Because it's Python. Not the way.

6. In this question you were supposed to use two higher-order functions; one for each "loop". Let's start with this one-liner (written across several lines to fit on your screen):

```
function indexOfFirstAllZeroRow (a) {
  return a.map(function (row) {
    return row.every(function (x) {return x === 0})
  }).indexOf(true)
}
```

   This is cool because `indexOf` automatically gives us our -1. However, the solution isn't exactly optimal because it goes through every row; the whole point of this problem that Rubin wanted to get across is the early exit. The function `every` cuts out early, but not `map`. We can use `some` to cut out after the row is found, but the code gets a little ugly.

```
function indexOfFirstAllZeroRow (a) {
function firstAllZeroRow (a) {
  var result = -1
  a.some(function (row, i) {
    return row.every(function (x) {return x === 0}) ? ((result = i), true) : false
  })
  return result
}
```

7. This problem is interesting because we know that in some languages we need closures and in some we do not. In JavaScript (prior to ES6) you must:

```
var a = []
for (var i = 0; i < 10; i++) {
    a[i] = function (i) {return function (x) {return x / (i * i);}} (i)
}
```

   because variables are either global or local to a function, never a block. But in Ruby, Procs work fine:

```
# Ruby
a = Array.new(10) {|i| Proc.new {|x| x / (i * i)}}
```

   because *i* is local to the block and a new block is created for each item of the array. Now in Go, our loop variable will not be local to the block, but we can create a block-level variable ourselves to avoid the closure:

```
package main

import "fmt"

func main() {
    var a [10]func(int)int
    for i := range(a) {
        j := i
        a[i] = func (x int) int {return x / (j * j)}
    }
    fmt.Println(a[1](10), a[2](40), a[5](250), a[9](810))
}
```

   That *j* usage is pretty cool, no? By the way you can try out this code on the Go Playground.

8. So first, here is the naïve non-tail recursive formulation, which is crap, of course:

```
var withoutEvens = function (a) {
    return a.length <= 1 ? [] : [a[1]].concat(withoutEvens(a.slice(2)));
};
```

To make it tail recursive, pass along the list you are "accumulating". Now there are two ways to track your progress through the array as you recurse. One way to pass along the remanining part of the array:

```
var withoutEvens = function (a) {
    var helper = function (a, result) {
        return a.length <= 1 ? result : helper(a.slice(2), result.concat(a[1]));
    };
    return helper(a, []);
};
```

Because in JavaScript it isn't efficient to keep passing along the list of remaining elements to process (because `slice` makes copies), we can pass along an index:

```
var withoutEvens = function (a) {
    var helper = function (i, result) {
        return i >= a.length ? result : helper(i + 2, result.concat(a[i]));
    };
    return helper(1, []); // Cool, eh?
};
```

9.

```
    ⇒
 /   \
#      ∇
|    /   \
#   #      ∇
|   |    /   \
A   B   %      ⊗
       / \    /  \
      ¥   E  •    ∀
     / \    / \   |
    C   D  %   ∀  §
          / \  |  |
         F   G ⊥  I
               |
               H
```

10. Many answers are possible for the table filling in. Here are mine:
   - Str to Num: The number the string looks like, else NaN if it doesn't look like a number
   - Arr to Num: The length of the array
   - Dict to Num: The number of entries in the dict
   - Fun to Num: The number of parameters of the function
   - Bool to Str: "true" or "false"
   - Num to Str: The obvious rendering of the number (in base 10)
   - Arr to Str: The obvious rendering of the array, with square brackets and commas, with some special marker to prevent infinite strings
   - Dict to Str: The obvious rendering of the dict, with braces, colons, and commas, with some special

marker to prevent infinite strings

- Fun to Str: The source code, nicely formatted according to some specific rules
- Anything except null (or another Arr) to Arr: A single element array containing its value. It is possible to have special cases for Strs — make an array of its characters — and Dicts — `{"a":"b", "c":"d"}` ⇒ `["a","b","c","d"]`.
- Anything (except another Dict) to Dict: Probably just {}. I suppose for arrays we can do `["a","b","c","d"]` ⇒ `{"a":"b", "c":"d"}`.
- Anything to Fun: A no-arg function returning that value.

For the extra credit problem: no, JavaScript is not weakly typed; The values `null` and `undefined` do *not* get coerced to objects, and non-functions do not get coerced to functions. I would say it's roughly 90% weakly typed. It's not 100% weakly typed because its designer thought that coercing things to functions and even objects would be going too far, since there is **no consensus on the "obvious" coercion to make**.