# CS 331 – Assignment 5 – Part B

Due: 11:59pm Sunday, April 13

Begin your work on this assignment by downloading the file *assign5b.js* from D2L. In that file you will find five comments of the form `//Problem X`, where $X = 1, 2, 3, 4, 5$. For each of these comments you will find some partially completed code below it. Your task in this assignment is to complete the code below that comment in the correct fashion to meet each of the problem specifications below. To test your solutions, we will do the following from the command prompt on the campus Linux network.

```
$ node
> .load by-name.js              Our file with correct solution to the normal-order evaluator from Part A

> .load tests-partA.js             Get all the boolean and numeric encodings plus the definition of Y

> .load assign5b.js
⋮
                                          We will have additional test cases for your solutions
```

1. You will find the following under the comment `// Problem 1`:

   ```
   // Problem 1
   var SUBREC = ????????
   var Y_SUB = "("+Y+" "+SUBREC+")";
   console.log(eval_and_prettify("(("+Y_SUB+" "+FIVE+") "+THREE+")"));
   ```

   Replace the set of question marks with the appropriate definition of a lambda calculus function that, when Y is applied to it, will result in the lambda calculus subtraction function. Hence the console output should yield `^f.^x.(f (f x))`.

2. You will find the following under the comment `// Problem 2`:

   ```
   // Problem 2
   var LE = ????????????
   var Y_LE = "("+Y+" "+LE+")";
   console.log(eval_and_prettify("(("+Y_LE+" "+FIVE+") "+THREE+")"));
   console.log(eval_and_prettify("(("+Y_LE+" "+THREE+") "+FIVE+")"));
   ```

   Replace the set of question marks with the appropriate definition of a lambda calculus function that, when Y is applied to it, will result in the lambda calculus "less-than-or-equal-to" function, which returns T if its first argument is less than or equal to its second. Hence the console output above should yield `^x.^y.y` and `^x.^y.x`.

3. You will find the following under the comment `// Problem 3`:

   ```
   // Problem 3
   var QUOTIENT = ?????????
   var Y_QUO = "("+Y+" "+QUOTIENT+")";
   console.log(eval_and_prettify("(("+Y_QUO+" "+THREE+") "+TWO+")"));
   ```

   Replace the set of question marks with the appropriate definition of a lambda calculus function that, when Y is applied to it, will result in the lambda calculus integer quotient function. Hence the console output above should yield `^f.^x.x`.

4. In Problem 4, you will be working with the Y-combinator as it can be defined in JavaScript, not the Y-combinator as defined in our lambda-calculus language. As it appears here, called `Y_in_JS`, I am essentially using the version of the combinator that comes from Professor Matt Might at the University of Utah. You may want to read his explanation of the combinator at:
   *http://matt.might.net/articles/implementation-of-recursive-fixed-point-y-combinator-in-javascript-for-memoization/*

   The code you will work from ...

   ```
   // Problem 4

   var Y_in_JS = function (F) {
       return (function (x) {
           return F(function (y) { return (x(x))(y);});
       })
       (function (x) {
           return F(function (y) { return (x(x))(y);});
       }) ;
   } ;
   ```

For example, if we defined a FactGen function as follows:

```
var FactGen = function (fact) {
    return (function(n) {
        return ((n == 0) ? 1 : (n*fact(n-1))) ;
    });
} ;
```

and fed it to this Y-combinator

```
console.log((Y_in_JS(FactGen))(6));
```

we would see 720 as output.

In the code associated with Problem 4, you will see the following:

```
var GCD_gen = ????????

console.log(((Y_in_JS(GCD_gen))(21))(7)) ;
console.log(((Y_in_JS(GCD_gen))(23))(7)) ;
```

Replace the question marks with the appropriate recursive definition of Euclid's GCD algorithm so that, when it is fed to this implementation of JavaScript's Y-combinator, we see 7 and 1 output for the two test cases. Make sure you use the recursive version of Euclid's algorithm. No credit will be given if you use a non-recursive implementation that employs a while loop to control the iteration. After all the whole purpose of this assignment is to have you learn how to implement recursion using the Y combinator.

5. Under the comment `// Problem 5`:

```
// Problem 5

// lex_depth_expr receives an abstract syntax tree from our lambda
// calculus interpreter and return the corresponding expression in
// lexical address form

var lex_depth_expr = function (ast) {
  // Complete the function here
};
```

For example, the following test case:

```
var test5 = lex_depth_expr(parser.parse("(((^x y1 z1.((x y1) z1) ^a.a) ^b.b) ^c.c)"));
require('util').puts(require('util').inspect(test5, false, 20, true));
```

should output:

```
 [ 'ApplyExpr',
 [ 'ApplyExpr',
   [ 'ApplyExpr',
     [ 'LambdaExpr',
       [ 'LambdaExpr',
         [ 'LambdaExpr',
           [ 'ApplyExpr',
             [ 'ApplyExpr',
               [ 'VarWithLexDepth', 2 ],
               [ 'VarWithLexDepth', 1 ] ],
             [ 'VarWithLexDepth', 0 ] ] ] ] ],
     [ 'LambdaExpr',
       [ 'VarWithLexDepth', 0 ] ] ],
   [ 'LambdaExpr',
     [ 'VarWithLexDepth', 0 ] ] ],
 [ 'LambdaExpr',
   [ 'VarWithLexDepth', 0 ] ] ]
```

When you've got all the problems done, upload the completed *assign5b.js* file to the ASSIGNMENT 5 dropbox on D2L. If there are some problems that you do not complete and would cause syntax errors when we load your file, just comment out all the code associated with those problems. If any syntax errors are produced when we load your file, you will receive a zero grade on all five problems.