Objectives and Background

This MP will introduce you to data abstractions, lambda calculus, and environments. You may wish to read chapter 2 of Essentials of Programming Languages before starting this MP.

Problem 1

Write a big integers data type. You may not use `equal?`, `eqv?`, or arbitrarily large integers.

The interface should be as follows:

```scheme
> (zero)
<representation of zero>
> (one)
<representation of one>
> (plus x y)
<representation of x + y>
> (minus x y)
<representation of x - y>
> (equal x y)
<typeof x = y, #f otherwise>
```

Problem 2

Write a big rationals data type. You may not use `equal?`, `eqv?`, or the scheme representation of rationals or big integers.

The interface should be as follows:

```scheme
> (zero2)
<representation of zero>
> (one2)
<representation of one>
> (plus2 x y)
```

Problem 3

Implement λ-calculus expressions as a datatype.

> (var x)
> (application operator operand)
> (function x body)
> (alpha-r exp x y)
> (beta-r exp x)
> (equal-l x y)

Note: You may have equal-l act as if λx.x ≠ λy.y - i.e., ignore α-equivalence.

For alpha reduction, exp will be a function expression, and x and y will be variables. x will always be the same variable as the argument of exp. You do not have to worry about any of the invalid cases for alpha reduction - however if you use alpha reduction during beta reduction, you will have to check for the invalid cases.

For beta reduction, exp will be a function and x is the argument.

Problem 4

Implement a variable length array. You should not destroy or modify the representation passed into any of the functions - i.e., do not use set! or the like.

> (empty-array)
> (get array i)
<value stored in index i>
> (set array i x)
<representation of new array>
> (get-empty array)
<index of an empty entry>

You may assume that any index which has not been set is empty. You may also assume that the user will not use a particular index until the index is returned by get-empty. 

*get-empty* should return some valid index which is empty.

**Problem 5**

Implement an environment. This should be similar to the environments described in section 2.3 of the book. Again, you should not destroy any environment passed in as an argument.

> (empty-env)
<representation of an empty array>
> (extend-env env xs vals)
<representation of new environment>
> (apply-env env x)
<value of x in env>

**Problem 6**

Implement an environment. This time, we would like to be able to do slightly more complex assignments. You may find it useful to use the previous environment and the variable length array.

> (empty-env2)
<representation of an empty array>
> (extend-env2 env xs vals)
<representation of new environment>
> (apply-env2 env x)
<value of x in env>
> (equals env x y)
<representation of new environment>
> (assign env x val)
<representation of new environment>

*assign* should return an environment in which *x* has the value *val*.

*equals* should return an environment in which *x* and *y* have the same value. Furthermore, if a new value is assigned to *x* via the *assign* function, it will also be assigned to *y*, and vice versa. This linkage will continue until *x* or *y* is extended with a new value, or *x* or *y* is set to be *equal* to some other variable. Note: *equals* sets *y* to be the value of *x*. 
Problem 7

This problem is only required for all the graduate students. Undergraduates may do this problem for extra credit.

Implement the call-by-value version of the Y-combinator, and then use this to implement the power function.

> (y power 2 5)
32

It is alright to have the power function run in $O(n)$ time, instead of $O(\log n)$ time.

Handin

You should hand in a single file named ‘mp2.scm’ with the implementations of the above functions. The names of the functions and number and order of arguments of the functions should be the same as in the problems.

Please see the CS 421 FAQ web page for handin instructions.