CS 61A Summer 2010 Week 8A Lab
Monday 8/9 Afternoon

1. For this lab you'll need three people, each on a separate workstation.

One person should choose to be the server, and should do this:

```
> (load "~cs61a/lib/im-server.scm")
> (im-server-start)
```

Make a note of the IP address and port number that this prints!

The other people will be clients. They should do this:

```
> (load "~cs61a/lib/im-client.scm")
> (im-enroll "123.45.67.89" 6543)    ; use actual numbers from server!
```

but using the server's IP address instead of 123.45.67.89 and the server's port number instead of 6543. (Note that the IP address must be enclosed in quotation marks.)

The clients can then send each other messages:

```
> (im 'cs61a-xy "Hi there, how are you?")
```

The messages can't include more than one line.

A client can leave the IM system by running

```
> (im-exit)
```

The server can quit (which disconnects all the clients) with

```
> (im-server-close)
```

For the rest of this lab, we'll be using `MapReduce`, a programming paradigm developed by Google that uses higher-order functions to allow a programmer to process large amount of data in parallel on many computers. `Hadoop` is an open source implementation of the `mapreduce` design.

Any computation in `mapreduce` consists primarily of two functions: the *mapper* and the *reducer*. (Note: The Google `mapreduce` paper in the course reader says "the `map` function" to mean the function that the user writes, the one that's applied to each datum; this usage is confusing since everyone else uses "`map`" to mean the higher-order function that controls the invocation of the user's function, so we're calling the latter the *mapper*:

```
(map mapper data)
```

Similarly, we'll use `reduce` to refer to the higher-order function, and `reducer` to mean the user's accumulation function.)

The mapper function takes a `(input-key . input-value)` pair as its argument and returns a list of `(finalkey . intermediatevalue)` pairs. (It returns a list of pairs, not a single pair, both to allow

more than one intermediate value per input value (e.g., separating a line into words) and to allow for the possibility of returning an empty stream, meaning no intermediate values at all, so that the mapper can also effectively `filter` the input data.)

The reducer function is applied to each group of `intermediate-value`s with the same `final-key`s, and it returns a `(finalkey . finalvalue)` pair. (The reducer takes two values as arguments: one new value and one partially accumulated value, just like the two-argument function used with `accumulate`.) Since there are many groups, the outputs of the reducer function are appended together to form the final output stream.

We've developed a system that allows you to run `mapreduce` computations on a computer cluster from within STk. Our version of `mapreduce` in Scheme uses slightly simpler semantics than the original `MapReduce`.

The syntax for performing a `mapreduce` computation in STk is as follows:

```
(mapreduce <mapper> <reducer> <base-case> <input>)
```

Mapper is a one-argument procedure that specifies the function that `map` applies.

Reducer is a procedure specifying the reduction function. *Base-case* is the base case for the reduction function. It is similar to the base case argument in Scheme's `accumulate`.

Input specifies the input data to the MapReduce computation. This argument may be a string, the name of a *distributed file directory* stored on all the machines in the parallel cluster, e.g., `"/gutenberg"` for the Project Gutenberg collection of public domain books. Alternatively, it may be a stream (not the name of a stream!) that was produced by an earlier invocation of `mapreduce`.

If you forget to save the output of mapreduce, you can always run `(get-last-mapreduce-output)`, which returns the last stream `mapreduce` returned.

For example, suppose we want to count the number of lines in the collected works of Shakespeare. Our input would be a set of key-value pairs with the name of a play as the key and a line of text as the value. The input is provided in a distributed file directory named `"/gutenberg/shakespeare"`.

We could solve this problem with `mapreduce` as follows:

```
(mapreduce (lambda (input-key-value-pair)
             (list (make-kv-pair 'line 1)))   ; mapper
           +                                  ; reducer
           0                                  ; base case
           "/gutenberg/shakespeare")          ; data
```

Since we're trying to get a total count for all the works of Shakespeare, not a separate count for each play, we give every intermediate pair the same key. We used the word `line`, but anything would have worked. The value is `1` because each line counts as one line!

What we want the reducer to do is add up all the `1`s from all the files. So we don't need a complicated reducer; we just use `+`.

In this case, there's only one instance of `reduce` adding up all the values, but in more general cases there'll be one per key, and so what `mapreduce` returns is not a single reduced value, but a stream of key-value pairs. In this case it'll be a stream of length 1:

`((line .` *number*`))`

To get just the number, we'd say `(kv-value (stream-car (mapreduce ...)))`. To use `mapreduce`, you must first ssh into the Berkeley clusters by using the command
`ssh cs61a-XX@icluster1.eecs.berkeley.edu`
where you replace `XX` with your login.

Exercises:

2. The example above is inefficient because the map phase happens in parallel, but the reduce phase happens on a single machine, since all the keys are the same, and each group of same-key pairs go to a single `reduce` instance. Fix this example so that the plays are line-counted in parallel, but we still get a single total line count at the end. (Hint: Do something to the value that `mapreduce` returns.)