## CS 61A Fall 2008 Week 13 Lab
## Monday 11/17 afternoon, Tuesday 11/18, or Wednesday 11/19 morning

For 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`.

**Continued on next page...**

## Week 13 continued:

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 ...)))`.

Exercises:

1. 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.)

**Continued on next page...**

## Week 13 continued:

2(a). One common MapReduce application is a distributed word count. Given a large body of text, such as Project Gutenberg, we want to find out which words are the most common.

Write a `mapreduce` program that returns each word in a body of text paired with the number of times it is used. Remember, the argument to your mapper function is a pair whose key is the document or book name and whose value is a single line of text from a single book. For example,

```
> (define gutenberg-wordcounts (mapreduce ...))
> (ss gutenberg-wordcounts 3)
((the . 300) (was . 249) (thee . 132) ... ) ; These aren't the actual numbers.
```

2(b). Using the output of the program you wrote in part (a), return the most commonly used word.

2(c). Using the output of the program you wrote in part (a), generate a stream of all words used only once.

3(a). Pattern matching is important in many areas of computer science. In this problem, we'll use `mapreduce` to find lines that match a pattern. Assume you're given the function `match?` that takes two arguments, a pattern and a sentence, and returns `#t` if the sentence contains a match for the pattern, or false otherwise. For example:

```
> (match? '(* hard * night *) '(a hard days night))
#t
> (match? '(* hard night *) '(a hard days night))
#f
> (match? '(* night *) '(knight rider))
#f
> (match? '(* walrus) '(I am the walrus))
#t
```

Write a procedure that takes a pattern and a directory name as arguments, and returns a stream of lines (with their keys attached) that match the pattern.

3(b). Try your pattern recognition program on the Project Gutenberg text with patterns to find some of these phrases originally attributed to Shakespeare:

| | |
|---|---|
| pomp and circumstance | one fell swoop |
| foregone conclusion | seen better days |
| full circle | it smells to heaven |
| the makings of | a sorry sight |
| method in the madness | a spotless reputation |
| neither rhyme nor reason | strange bedfellows |