CS 61A   Lecture Notes   Week 4

Topic: Data abstraction, sequences

Reading: Abelson & Sussman, Sections 2.1 and 2.2.1 (pages 79–106)
[...]

• MapReduce*

In the past, functional programming, and higher-order functions in particular, have been considered esoteric and unimportant by most programmers. But the advent of highly parallel computation is changing that, because functional programming has the very useful property that the different pieces of a program don't interfere with each other, so it doesn't matter in what order they are invoked. Later this semester, when we have more sophisticated functional mechanisms to work with, we'll be examining one famous example of functional programming at work: the `MapReduce` 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.

Much of the computing done at Google consists of relatively simple algorithms applied to massive amounts of data, such as the entire World Wide Web. It's routine for them to use clusters consisting of many thousands of processors, all running the same program, with a distributed filesystem that gives each processor local access to part of the data.

In 2003 some very clever people at Google noticed that the majority of these computations could be viewed as a `map` of some function over the data followed by an `accumulate` (they use the name `reduce`, which is a synonym for this function) to collect the results. Although each program was conceptually simple, a lot of programmer effort was required to manage the parallelism; every programmer had to worry about things like how to recover from a processor failure (virtually certain to happen when a large computation uses thousands of machines) during the computation. They wrote a library procedure named `MapReduce` that basically takes two functions as arguments, a one-argument function for the `map` part and a two-argument function for the `accumulate` part. (The actual implementation is more complicated, but this is the essence of it.) Thus, only the implementors of `MapReduce` itself had to worry about the parallelism, and application programmers just have to write the two function arguments.

`MapReduce` is a little more complicated than just

```
(define (mapreduce mapper reducer base-case data)    ; Nope.
  (accumulate reducer base-case (map mapper data)))
```

because of the parallelism. The input data comes in pieces; several computers run the `map` part in parallel, and each of them produces some output. These intermediate results are rearranged into groups, and each computer does a `reduce` of part of the data. The final result isn't one big list, but separate output files for each reducing process.

To make this a little more specific, today we'll see a toy version of the algorithm that just handles small data lists in one processor.

Data pass through the program in the form of *key-value pairs:*

```
(define make-kv-pair cons)
(define kv-key car)
(define kv-value cdr)
```

A list of key-value pairs is called an *association list* or *a-list* for short. We'll see a-lists in many contexts other than `MapReduce`. Conceptually, the input to `MapReduce` is an a-list, although in practice there are several a-lists, each on a different processor.

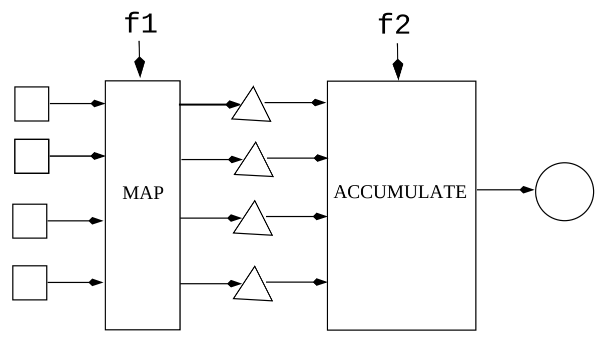
* Except as otherwise noted, the content of this document is licensed under the Creative Commons Attribution 2.5 License.

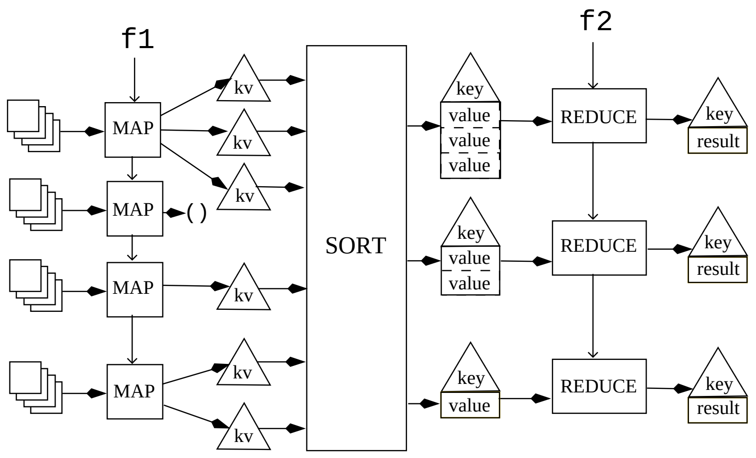
Any computation in `MapReduce` involves two function arguments: the *mapper* and the *reducer*. (Note: The Google `MapReduce` paper in the course reader says "the `map` function" to mean the function that the user writes, the one that's applied to each datum; this usage is confusing since everyone else 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` for the higher-order function, and `reducer` for the user's accumulation function.)

`(accumulate f2 base (map f1 data))` `(mapreduce f1 f2 base dataname)`





The argument to the mapper is always one kv-pair. Keys are typically used to keep track of where the data came from. For example, if the input consists of a bunch of Web pages, the keys might be their URLs. Another example we'll be using is Project Gutenberg, an online collection of public-domain books; there the key would be the name of a book (more precisely, the filename of the file containing that book). In most uses of a-lists, there will only be one kv-pair with a given key, but that's not true here; for example, each line of text in a book or Web page might be a datum, and every line in the input will have the same key.

The value returned by the mapper must be *a list of* kv-pairs. The reason it's a list instead of a single kv-pair, as you might expect, is twofold. First, a single input may be split into smaller pieces; for example, a line of text might be mapped into a separate kv-pair for each word in the line. Second, the mapper might return an empty list, if this particular kv-pair shouldn't contribute to the result at all; thus, the mapper might also be viewed as a filterer. The mapper is not required to use the same key in its output kv-pairs that it gets in its input kv-pair.

Since `map` handles each datum independently of all the others, the fact that many `map`s are running in parallel doesn't affect the result; we can model the entire process with a single `map` invocation. That's not the case with the `reduce` part, because the data are being combined, so it matters which data end up on which machine. This is where the keys are most important. Between the mapping and the reduction is an intermediate step in which the kv-pairs are sorted based on the keys, and all pairs with the same key are reduced together. Therefore, the reducer doesn't need to look at keys at all; its two arguments are a value and the result of the partial accumulation of values already done. In many cases, just as in the accumulations we've seen earlier, the reducer will be a simple associative and commutative operation such as `+`.

The overall result is an a-list, in which each key occurs only once, and the value paired with that key is the result of the `reduce` invocation that handled that key. The keys are guaranteed to be in order. (This is the result of the 61A version of `MapReduce`; the real Google software has a more complicated interface because each computer in the cluster collects its own `reduce` results, and there are many options for how the reduction tasks are distributed among the processors. You'll learn more details in 61B.) So in today's single-processor simulation, instead of talking about `reduce` we'll use a higher order function called `groupreduce` that takes a *list of a-lists* as argument, with each sublist having kv-pairs with the same key, does a separate reduction for each sublist, and returns an a-list of the results. So a complete MapReduce operation works roughly like this:

```
(define (mapreduce mapper reducer base-case data) ; handwavy approximation
  (groupreduce reducer base-case
               (sort-into-buckets (map mapper data))))

(define (groupreduce reducer base-case buckets)
  (map (lambda (subset) (make-kv-pair
                         (kv-key (car subset))
                         (reduce reducer base-case (map kv-value subset))))
       buckets))
```

As a first example, we'll take some grades from various exams and add up the grades for each student. This example doesn't require map. Here's the raw data:

```
(define mt1 '((cs61a-xc . 27) (cs61a-ya . 40) (cs61a-xw . 35)
              (cs61a-xd . 38) (cs61a-yb . 29) (cs61a-xf . 32)))
(define mt2 '((cs61a-yc . 32) (cs61a-xc . 25) (cs61a-xb . 40)
              (cs61a-xw . 27) (cs61a-yb . 30) (cs61a-ya . 40)))
(define mt3 '((cs61a-xb . 32) (cs61a-xk . 34) (cs61a-yb . 30)
              (cs61a-ya . 40) (cs61a-xc . 28) (cs61a-xf . 33)))
```

Each midterm in this toy problem corresponds to the output of a parallel map operation in a real problem.

First we combine these into one list, and use that as input to the sortintobuckets procedure:

```
> (sort-into-buckets (append mt1 mt2 mt3))
(((cs61a-xb . 40) (cs61a-xb . 32))
 ((cs61a-xc . 27) (cs61a-xc . 25) (cs61a-xc . 28))
 ((cs61a-xd . 38))
 ((cs61a-xf . 32) (cs61a-xf . 33))
 ((cs61a-xk . 34))
 ((cs61a-xw . 35) (cs61a-xw . 27))
 ((cs61a-ya . 40) (cs61a-ya . 40) (cs61a-ya . 40))
 ((cs61a-yb . 29) (cs61a-yb . 30) (cs61a-yb . 30))
 ((cs61a-yc . 32)))
```

In the real parallel context, instead of the append, each map process would sort its own results into the right buckets, so that too would happen in parallel.

Now we can use groupreduce to add up the scores in each bucket separately:

```
> (groupreduce + 0 (sort-into-buckets (append mt1 mt2 mt3)))
((cs61a-xb . 72) (cs61a-xc . 80) (cs61a-xd . 38) (cs61a-xf . 65)
 (cs61a-xk . 34) (cs61a-xw . 62) (cs61a-ya . 120) (cs61a-yb . 89)
 (cs61a-yc . 32))
```

Note that the returned list has the keys in sorted order. This is a consequence of the sorting done by sort-into-buckets, and also, in the real parallel mapreduce, a consequence of the order in which keys are assigned to processors (the "partitioning function" discussed in the MapReduce paper).

Similarly, we could ask *how many* midterms each student took:

```
> (groupreduce (lambda (new old) (+ 1 old)) 0 (sort-into-buckets (append mt1 mt2 mt3)))
((cs61a-xb . 2) (cs61a-xc . 3) (cs61a-xd . 1) (cs61a-xf . 2) (cs61a-xk . 1)
 (cs61a-xw . 2) (cs61a-ya . 3) (cs61a-yb . 3) (cs61a-yc . 1))
```

We could combine these in the obvious way to get the average score per student, for exams actually taken.

Word frequency counting. A common problem is to look for commonly used words in a document. For starters, we'll count word frequencies in a single sentence. The first step is to turn the sentence into key-value pairs in which the key is the word and the value is always 1:

```
> (map (lambda (wd) (list (make-kv-pair wd 1))) '(cry baby cry))
((cry . 1) (baby . 1) (cry . 1))
```

If we group these by key and add the values, we'll get the number of times each word appears.

```
(define (wordcounts1 sent)
  (groupreduce + 0 (sort-into-buckets (map (lambda (wd) (make-kv-pair wd 1))
                                          sent))))

> (wordcounts1 '(cry baby cry))
((baby . 1) (cry . 2))
```

Now to try the same task with (simulated) files. When we use the real `mapreduce`, it'll give us file data in the form of a key-value pair whose key is the name of the file and whose value is a line from the file, in the form of a sentence. For now, we're going to simulate a file as a list whose car is the "filename" and whose cdr is a list of sentences, representing the lines of the file. In other words, a file is a list whose first element is the filename and whose remaining elements are the lines.

```
(define filename car)
(define lines cdr)
```

Here's some data for us to play with:

```
(define file1 '((please please me) (i saw her standing there) (misery)
                (anna go to him) (chains) (boys) (ask me why)
                (please please me) (love me do) (ps i love you)
                (baby its you) (do you want to know a secret)))
(define file2 '((with the beatles) (it wont be long) (all ive got to do)
                (all my loving) (dont bother me) (little child)
                (till there was you) (roll over beethoven) (hold me tight)
                (you really got a hold on me) (i wanna be your man)
                (not a second time)))
(define file3 '((a hard days night) (a hard days night)
                (i should have known better) (if i fell)
                (im happy just to dance with you) (and i love her)
                (tell me why) (cant buy me love) (any time at all)
                (ill cry instead) (things we said today) (when i get home)
                (you cant do that) (ill be back)))
```

We start with a little procedure to turn a "file" into an a-list in the form `mapreduce` will give us:

```
(define (file->linelist file)
  (map (lambda (line) (make-kv-pair (filename file) line))
       (lines file)))

> (file->linelist file1)
(((please please me) i saw her standing there)
 ((please please me) misery)
 ((please please me) anna go to him)
 ((please please me) chains)
 ((please please me) boys)
 ((please please me) ask me why)
 ((please please me) please please me)
 ((please please me) love me do)
 ((please please me) ps i love you)
 ((please please me) baby its you)
 ((please please me) do you want to know a secret))
```

Note that `((please please me) misery)` is how Scheme prints the kv-pair `((please please me) . (misery))`.

Now we modify our `wordcounts1` procedure to accept such kv-pairs:

```
(define (wordcounts files)
  (groupreduce + 0 (sort-into-buckets
                    (flatmap (lambda (kv-pair)
                               (map (lambda (wd) (make-kv-pair wd 1))
                                    (kv-value kv-pair)))
                             files))))
```

```
> (wordcounts (append (file->linelist file1)
                      (file->linelist file2)
                      (file->linelist file3)))
((a . 4) (all . 3) (and . 1) (anna . 1) (any . 1) (ask . 1) (at . 1)
 (baby . 1) (back . 1) (be . 3) (beethoven . 1) (better . 1) (bother . 1)
 (boys . 1) (buy . 1) (cant . 2) (chains . 1) (child . 1) (cry . 1)
 (dance . 1) (days . 1) (do . 4) (dont . 1) (fell . 1) (get . 1) (go . 1)
 (got . 2) (happy . 1) (hard . 1) (have . 1) (her . 2) (him . 1) (hold . 2)
 (home . 1) (i . 7) (if . 1) (ill . 2) (im . 1) (instead . 1) (it . 1)
 (its . 1) (ive . 1) (just . 1) (know . 1) (known . 1) (little . 1)
 (long . 1) (love . 4) (loving . 1) (man . 1) (me . 8) (misery . 1) (my . 1)
 (night . 1) (not . 1) (on . 1) (over . 1) (please . 2) (ps . 1) (really . 1)
 (roll . 1) (said . 1) (saw . 1) (second . 1) (secret . 1) (should . 1)
 (standing . 1) (tell . 1) (that . 1) (there . 2) (things . 1) (tight . 1)
 (till . 1) (time . 2) (to . 4) (today . 1) (wanna . 1) (want . 1) (was . 1)
 (we . 1) (when . 1) (why . 2) (with . 1) (wont . 1) (you . 7) (your . 1))
```

(If you count yourself to check, remember that words in the album titles don't count! They're keys, not values.)

Note the call to `flatmap` above. In a real `mapreduce`, each file would be mapped on a different processor, and the results would be distributed to `reduce` processes in parallel. Here, the `map` over files gives us a list of a-lists, one for each file, and we have to append them to form a single a-list. `Flatmap` flattens (appends) the results from calling `map`.

We can postprocess the groupreduce output to get an overall reduction to a single value:

```
(define (mostfreq files)
  (accumulate (lambda (new old)
                (cond ((> (kv-value new) (kv-value (car old)))
                       (list new))
                      ((= (kv-value new) (kv-value (car old)))
                       (cons new old))        ; In case of tie, remember both.
                      (else old)))
              (list (make-kv-pair 'foo 0))        ; Starting value.
              (groupreduce + 0 (sort-into-buckets
                                (flatmap (lambda (kv-pair)
                                           (map (lambda (wd)
                                                  (make-kv-pair wd 1))
                                                (kv-value kv-pair)))
                                         files)))))
```

```
> (mostfreq (append (file->linelist file1)
                    (file->linelist file2)
                    (file->linelist file3)))
((me . 8))
```

(Second place is ”you” and ”I” with 7 appearances each, which would have made a two-element a-list as the result.) If we had a truly enormous word list, we'd put it into a distributed file and use another `mapreduce` to find the most frequent words of subsets of the list, and then find the most frequent word of those most frequent words.

Searching for a pattern. Another task is to search through files for lines matching a pattern. A *pattern* is a sentence in which the word `*` matches any set of zero or more words:

```
> (match? '(* i * her *) '(i saw her standing there))
#t
> (match? '(* i * her *) '(and i love her))
#t
> (match? '(* i * her *) '(ps i love you))
#f
```

Here's how we look for lines in files that match a pattern:

```
(define (grep pattern files)
  (groupreduce cons '()
               (sort-into-buckets
                (flatmap (lambda (kv-pair)
                           (if (match? pattern (kv-value kv-pair))
                               (list kv-pair)
                               '()))
                         files))))

> (grep '(* i * her *) (append (file->linelist file1)
                               (file->linelist file2)
                               (file->linelist file3)))
(((a hard days night) (and i love her))
 ((please please me) (i saw her standing there)))
```

Summary. The general pattern here is

```
(groupreduce reducer base-case
             (sort-into-buckets
              (map-or-flatmap mapper data)))
```

This corresponds to

```
(mapreduce mapper reducer base-case data)
```

in the truly parallel `mapreduce` exploration we'll be doing later.

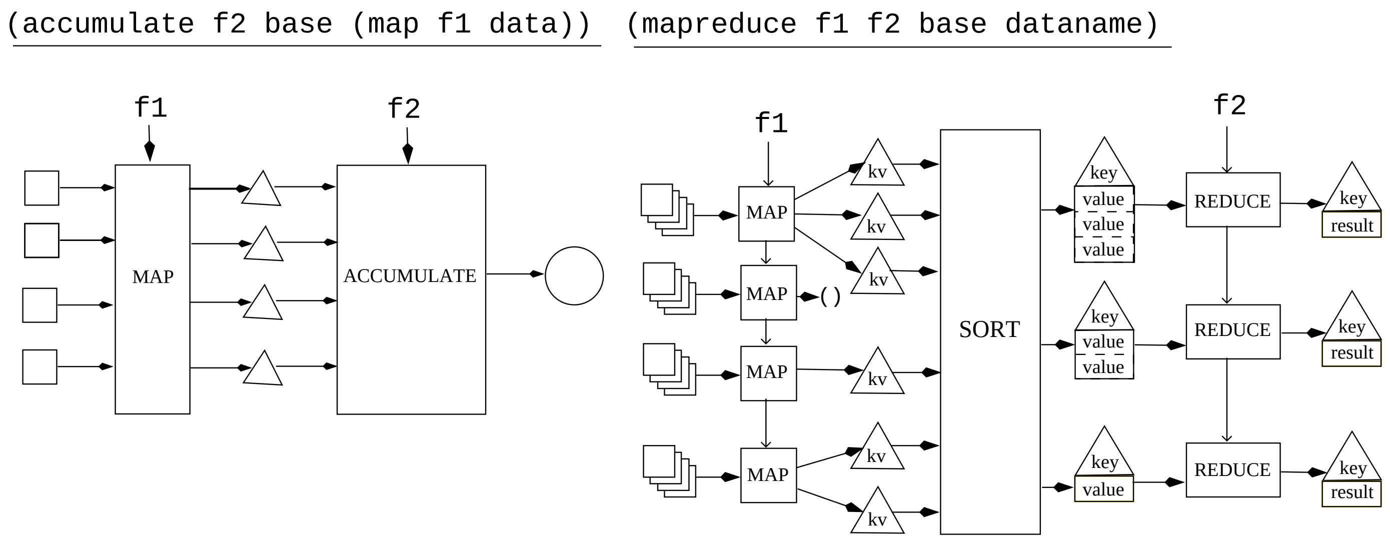
Topic: Metacircular evaluator

Reading: Abelson & Sussman, Section 4.1

[...]

• **Mapreduce part 2**

Here's the diagram of mapreduce again:



The seemingly unpoetic names `f1` and `f2` serve to remind you of two things: `f1` (the mapper) is used before `f2` (the reducer), and `f1` takes one argument while `f2` takes two arguments (just like the functions used with ordinary `map` and `accumulate` respectively).
mapper: `kv-pair` → `list-of-kv-pairs`
reducer: `value`, `partial-result` → `result`

All data are in the form of key-value pairs. Ordinary `map` doesn't care what the elements of the data list argument are, but `mapreduce` works only with data each of which is a key-value pair. In the Scheme interface to the distributed filesystem, a file is a stream. Every line of the file is a key-value pair whose key is the filename and whose value is a list of words, representing the text of the line.

(Actually, "a file is a stream" is a slight handwave. The STk running on the parallel cluster has special versions of `stream-car`, etc., that accept both ordinary streams and special file streams that are really pointers to data in the distributed file system, but that simulate the behavior of streams. So if you print one of these special streams directly, rather than using `show-stream`, you'll see something that doesn't look familiar.)

Each processor runs a separate `stream-map`. The overlapping squares at the left of the mapreduce picture represent an entire stream. (How is a large distributed file divided among map processes? It doesn't really matter, as far as the `mapreduce` user is concerned; `mapreduce` tries to do it as efficiently as possible given the number of processes and the location of the data in the filesystem.) The entire stream is the input to a `map` process; each element of the stream (a kv-pair) is the input to your mapper function `f1`.

For each key-value pair in the input stream, the mapper returns a <u>list</u> of key-value pairs. In the simplest case, each of these lists will have one element; the code will look something like

```
(define (my-mapper input-kv-pair)
  (list (make-kv-pair ... ...)))
```

The interface requires that you return a list to allow for the non-simplest cases: (1) Each input key-value pair may give rise to more than one output key-value pair. For example, you may want an output key-value pair for each *word* of the input file, whereas the input key-value pair represents an entire line:

```
(define (my-mapper input-kv-pair)
  (map (lambda (wd) (make-kv-pair ... ...))
       (kv-value input-kv-pair)))
```

(2) There are *three* commonly used higher order functions for sequential data, `map`, `accumulate`/`reduce`, and `filter`. The way `mapreduce` handles the sort of problem for which `filter` would ordinarily be used is to allow a mapper to return an empty list if this particular key-value pair shouldn't contribute to the result:

```
(define (my-mapper input-kv-pair)
  (if ...
      (list input-kv-pair)
      '()))
```

Of course it's possible to write mapper functions that combine these three patterns for more complicated tasks.

The keys in the kv-pairs returned by the mapper need not be the same as the key in the input kv-pair.

Instead of one big accumulation, there's a separate accumulation of values for each key. The non-parallel computation in the left half of the picture has two steps, a `map` and an `accumulate`. But the `mapreduce` computation has *three* steps; the middle step sorts all the key-value pairs produced by all the mapper processes by their keys, and combines all the kv-pairs with the same key into a single aggregate structure, which is then used as the input to a `reduce` process.

This is why the use of key-value pairs is important! If the data had no such structure imposed on them, there would be no way for us to tell `mapreduce` which data should be combined in each reduction.

Although it's shown as one big box, the sort is also done in parallel; it's a "bucket sort," in which each `map` process is responsible for sending each of its output kv-pairs to the proper `reduce` process. (Don't be confused; your mapper function doesn't have to do that. The `mapreduce` program takes care of it.)

Since all the data seen by a single `reduce` process have the same key, the reducer doesn't deal with keys at all. This is important because it allows us to use simple reducer functions such as `+`, `*`, `max`, etc. The Scheme interface to `mapreduce` recognizes the special cases of `cons` and `cons-stream` as reducers and does what you intend, even though it wouldn't actually work without this special handling, both because `cons-stream` is a special form and because the iterative implementation of `mapreduce` would do the combining in the wrong order.

In the underlying `mapreduce` software, each `reduce` process leaves its results in a separate file, stored on the particular processor that ran the process. But the Scheme interface to `mapreduce` returns a single value, a stream that effectively `stream-append`s the results from all the `reduce` processes.

Running mapreduce: The `mapreduce` function is not available on the standard lab machines. You must connect to the machine that controls the parallel cluster. To do this, from the Unix shell you say this:

```
ssh icluster1.eecs.berkeley.edu
```

If you're at home, rather than in the lab, you'll have to provide your class login to the `ssh` command:

```
ssh cs61a-XY@icluster.eecs.berkeley.edu
```

replacing `XY` above with your login account. `Ssh` will ask for your password, which is the same on the parallel cluster as for your regular class account. Once you are logged into `icluster1`, you can run `stk` as usual, but `mapreduce` will be available:

```
(mapreduce mapper reducer reducer-base-case filename-or-special-stream)
```

The first three arguments are the mapper function for the `map` phase, and the reducer function and starting value for the `reduce` phase. The last argument is the data input to the `map`, but it is restricted to be either a distributed filesystem folder, which must be one of these:

`"/beatles-songs"`	This one is small and has all Beatles song names
`"/gutenberg/shakespeare"`	The collected works of William Shakespeare
`"/gutenberg/dickens"`	The collected works of Charles Dickens
`"/sample-emails"`	Some sample email data for the homework
`"/large-emails"`	A much larger sample email dataset. Use this only if you're willing to wait a while.

(the quotation marks above are required), or the stream returned by an earlier call to `mapreduce`. (Streams you make yourself with `cons-stream`, etc., can't be used.) Some problems are solved with two `mapreduce` passes, like this:

```
(define intermediate-result (mapreduce ...))
(mapreduce ... intermediate-result)
```

(Yes, you could just use one `mapreduce` call directly as the argument to the second `mapreduce` call, but in practice you'll want to use `show-stream` to examine the intermediate result first, to make sure the first call did what you expect.)

Here's a sample. We provide a file of key-value pairs in which the key is the name of a Beatles album and the value is the name of a song on that album. Suppose we want to know how many times each word appears in the name of a song:

```
(define (wordcount-mapper document-line-kv-pair)
  (map (lambda (wd-in-line) (make-kv-pair wd-in-line 1))
       (kv-value document-line-kv-pair)))

(define wordcounts (mapreduce wordcount-mapper + 0 "/beatles-songs"))

> (ss wordcounts)
```

The argument to `wordcount-mapper` will be a key-value pair whose key is an album name, and whose value is a song name. (In other examples, the key will be a filename, such as the name of a play by Shakespeare, and the value will be a line from the play.) We're interested only in the song names, so there's no call to `kv-key` in the procedure. For each song name, we generate a list of key-value pairs in which the key is a word in the name and the value is 1. This may seem silly, having the same value in every pair, but it means that in the `reduce` stage we can just use `+` as the reducer, and it'll add up all the occurrences of each word.

You'll find the running time disappointing in this example; since the number of Beatles songs is pretty small, the same computation could be done faster on a single machine. This is because there is a significant setup time both for `mapreduce` itself and for the `stk` interface. Since your mapper and reducer functions have to work when run on parallel machines, your Scheme environment must be shipped over to each of those machines before the computation begins, so that bindings are available for any free references in your procedures. It's only for large amounts of data (or long computations that aren't data-driven, such as calculating a trillion digits of π, but `mapreduce` isn't really appropriate for those examples) that parallelism pays off.

By the way, if you want to examine the input file, you can't just say

```
(ss "/beatles-songs")        ; NO
```

because a distributed filename isn't a stream, even though the file itself is (when viewed by the `stk` interface to `mapreduce` a stream. These filenames only work as arguments to `mapreduce` itself. But we can use `mapreduce` to examine the file by applying null transformations in the map and reduce stages:

```
(ss (mapreduce list cons-stream the-empty-stream "/beatles-songs"))
```

The mapper function is `list` because the mapper must always return a list of key-value pairs; in this case, `map` will call `list` with one argument and so it'll return a list of length one.

Now we'd like to find the most commonly used word in Beatle song titles. There are few enough words so that we could really do this on one processor, but as an exercise in parallelism we'll do it partly in parallel. The trick is to have each reduce process find the most common word starting with a particular letter. Then we'll have 26 candidates from which to choose the absolutely most common word on one processor.

```
(define (find-max-mapper kv-pair)
  (list (make-kv-pair (first (kv-key kv-pair))
                      kv-pair)))

(define (find-max-reducer current so-far)
  (if (> (kv-value current) (kv-value so-far))
      current
      so-far))

(define frequent (mapreduce find-max-mapper find-max-reducer
                            (make-kv-pair 'foo 0) wordcounts))

> (ss frequent)

> (stream-accumulate find-max-reducer (make-kv-pair 'foo 0) frequent)
```

This is a little tricky. In the `wordcounts` stream, each key-value pair has a word as the key, and the count for that word as the value: `(back . 3)`. The mapper transforms this into a key-value pair in which the key is the first letter of the word, and the value is *the entire input key-value pair*: `(b . (back . 3))`. Each `reduce` process gets all the pairs with a particular key, i.e., all the ones with the same first letter of the word. The reducer sees only the values from those pairs, but each value is itself a key-value pair! That's why the reducer has to compare the `kv-value` of its two arguments.

As another example, here's a way to count the total number of lines in all of Shakespeare's plays:

```
(define will (mapreduce (lambda (kv-pair) (list (make-kv-pair 'line 1)))
                        + 0 "/gutenberg/shakespeare"))
```

For each line in Shakespeare, we make exactly the same pair `(line . 1)`. Then, in the `reduce` stage, all the ones in all those pairs are added. But this is actually a bad example! Since all the keys are the same (the word `line`), only one `reduce` process is run, so the counting isn't done in parallel. A better way would be to count each play separately, then add those results if desired. You'll do that in lab.