CS 61A Lecture Notes Week 12

Topic: Metacircular evaluator

Reading: Abelson & Sussman, Section 4.1

Midterm 3 is this week.

We're going to investigate SICP's Scheme interpreter written in Scheme. This interpreter implements the environment model of evaluation.

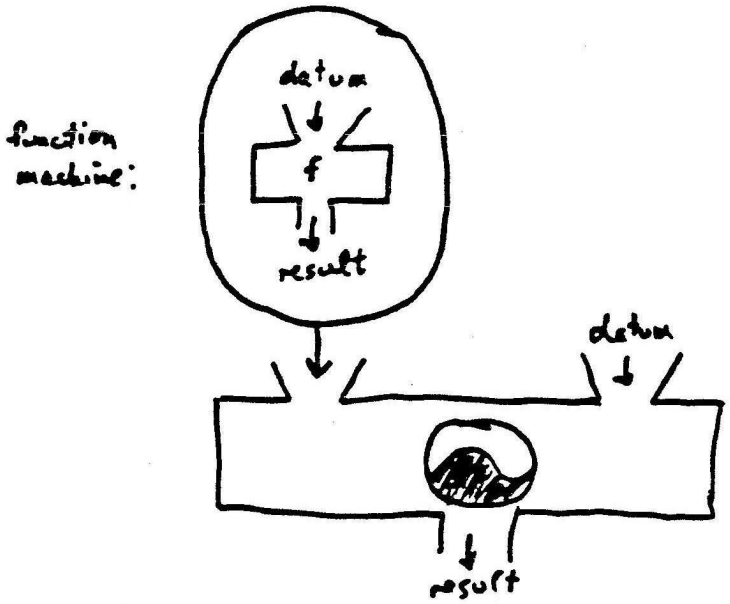
Here's a reminder of the reasons I mentioned in week 6 for studying a Scheme interpreter in Scheme, even though it's obviously not something you'd use in practice:

- It helps you understand the environment model.
- It lets us experiment with modifications to Scheme (new features).
- Even real Scheme interpreters are largely written in Scheme.
- It illustrates a big idea: *universality*.

Universality means we can write *one program* that's equivalent to all other programs. At the hardware level, this is the idea that made general-purpose computers possible. It used to be that they built a separate machine, from scratch, for every new problem. An intermediate stage was a machine that had a *patchboard* so you could rewire it, effectively changing it into a different machine for each problem, without having to re-manufacture it. The final step was a single machine that accepted a program *as data* so that it can do any problem without rewiring.

Instead of a function machine that computes a particular function, taking (say) a number in the input hopper and returning another number out the bottom, we have a *universal* function machine that takes *a function machine* in one input hopper, and a number in a second hopper, and returns whatever number the input machine would have returned. This is the ultimate in data-directed programming.

Our Scheme interpreter leaves out some of the important components of a real one. It gets away with this by taking advantage of the capabilities of the underlying Scheme. Specifically, we don't deal with storage allocation, tail recursion elimination, or implementing any of the Scheme primitives. All we *do* deal with is the evaluation of expressions. That turns out to be quite a lot in itself, and pretty interesting.



Here is a one-screenful version of the metacircular evaluator with most of the details left out. You might want to compare it to the one-screenful substitution-model interpreter you saw in week 6.

```
;;;;;                          In file cs61a/lectures/4.1/micro.scm
(define (scheme)
  (display "> ")
  (print (eval (read) the-global-environment))
  (scheme) )

(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((symbol? exp) (lookup-in-env exp env))
        ((special-form? exp) (do-special-form exp env))
        (else (apply (eval (car exp) env)
                     (map (lambda (e) (eval e env)) (cdr exp)) ))))

(define (apply proc args)
  (if (primitive? proc)
      (do-magic proc args)
      (eval (body proc)
            (extend-environment (formals proc)
                                args
                                (proc-env proc) ))))
```

Although the version in the book is a lot bigger, this really does capture the essential structure, namely, a mutual recursion between `eval` (evaluate an expression relative to an environment) and `apply` (apply a function to arguments). To evaluate a compound expression means to evaluate the subexpressions recursively, then apply the `car` (a function) to the `cdr` (the arguments). To apply a function to arguments means to evaluate the body of the function in a new environment.

What's left out? Primitives, special forms, and a lot of details.

In that other college down the peninsula, they wouldn't consider you ready for an interpreter until junior or senior year. At this point in the introductory course, they'd still be teaching you where the semicolons go. How do we get away with this? We have two big advantages:

- The *source language* (the language that we're interpreting) is simple and uniform. Its entire formal syntax can be described in one page, as we did in week 7. There's hardly anything to implement!

- The *implementation language* (the one in which the interpreter itself is written) is powerful enough to handle a program as data, and to let us construct data structures that are both hierarchical and circular.

The amazing thing is that the simple source language and the powerful implementation language are both Scheme! You might think that a powerful language has to be complicated, but it's not so.

• Introduction to Logo. For the programming project you're turning the metacircular evaluator into an interpreter for a *different* language, Logo. To do that you should know a little about Logo itself.

Logo is a dialect of Lisp, just as Scheme is, but its design has different priorities. The goal was to make it as natural-seeming as possible for kids. That means things like getting rid of all those parentheses, and that has other syntactic implications.

(To demonstrate Logo, run `~cs61a/logo` which is Berkeley Logo.)

Commands and operations: In Scheme, every procedure returns a value, even the ones for which the value is unspecified and/or useless, like `define` and `print`. In Logo, procedures are divided into operations, which return values, and commands, which don't return values but are called for their effect. You have to start each instruction with a command:

```
print sum 2 3
```

Syntax: If parentheses aren't used to delimit function calls, how do you know the difference between a function and an argument? When a symbol is used without punctuation, that means a function call. When you want the value of a variable to use as an argument, you put colon in front of it.

```
make "x 14
print :x
print sum :x :x
```

Words are quoted just as in Scheme, except that the double-quote character is used instead of single-quote. But since expressions aren't represented as lists, the same punctuation that delimits a list also quotes it:

```
print [a b c]
```

(Parentheses *can* be used, as in Scheme, if you want to give extra arguments to something, or indicate infix precedence.)

```
print (sum 2 3 4 5)
print 3*(4+5)
```

No special forms: Except `to`, the thing that defines a new procedure, all Logo primitives evaluate their arguments. How is this possible? We "proved" back in chapter 1 that `if` has to be a special form. But instead we just quote the arguments to `ifelse`:

```
ifelse 2=3 [print "hi] [print "bye]
```

You don't notice the quoting since you get it for free with the list grouping.

Functions not first class: In Logo every function has a name; there's no `lambda`. Also, the namespace for functions is separate from the one for variables; a variable can't have a function as its value. (This is convenient because we can use things like `list` or `sentence` as formal parameters without losing the functions by those names.) That's another reason why you need colons for variables.

So how do you write higher-order functions like `map`? Two answers. First, you can use the *name* of a function as an argument, and you can use that name to construct an expression and eval it with `run`. Second, Logo has first-class *expressions*; you can `run` a list that you get as an argument. (This raises issues about the scope of variables that we'll explore later this week.)

```
print map "first [the rain in spain]
print map [? * ?] [3 4 5 6]
```

• Data abstraction in the evaluator. Here is a quote from the Instructor's Manual, regarding section 4.1.2:

"Point out that this section is boring (as is much of section 4.1.3), and explain why: Writing the selectors, constructors, and predicates that implement a representation is often uninteresting. It is important to say explicitly what you expect to be boring and what you expect to be interesting so that students don't ascribe their boredom to the wrong aspect of the material and reject the interesting ideas. For example, data abstraction isn't boring, although writing selectors is. The details of representing expressions (as given in section 4.1.2) and environments (as given in section 4.1.3) are mostly boring, but the evaluator certainly isn't."

I actually think they go overboard by having a separate ADT for every kind of homogeneous sequence. For example, instead of `first-operand` and `rest-operands` I'd just use `first` and `rest` for all sequences. But things like `operator` and `operands` make sense.

• Dynamic scope. Logo uses dynamic scope, which we discussed in Section 3.2, instead of Scheme's lexical scope. There are advantages and disadvantages to both approaches.

Summary of arguments for lexical scope:

- Allows local state variables (OOP).
- Prevents name "capture" bugs.
- Faster compiled code.

Summary of arguments for dynamic scope:

- Allows first-class expressions (WHILE).
- Easier debugging.
- Allows "semi-global" variables.

Lexical scope is required in order to make possible Scheme's approach to local state variables. That is, a procedure that has a local state variable must be defined within the scope where that variable is created, and must carry that scope around with it. That's exactly what lexical scope accomplishes.

On the other hand, (1) most lexically scoped languages (e.g., Pascal) don't have `lambda`, and so they can't give you local state variables despite their lexical scope. And (2) lexical scope is needed for local state variables only if you want to implement the latter in the particular way that we've used. Object Logo, for example, provides OOP without relying on `lambda` because it includes local state variables as a primitive feature.

Almost all computer scientists these days hate dynamic scope, and the reason they give is the one about name captures. That is, suppose we write procedure P that refers to a global variable V. Example:

```
(define (area rad)
  (* pi rad rad))
```

This is intended as a reference to a global variable `pi` whose value, presumably, is 3.141592654. But suppose we invoke it from within another procedure like this:

```
(define (mess-up pi)
  (area (+ pi 5)))
```

If we say `(mess-up 4)` we intend to find the area of a circle with radius 9. But we won't get the right area if we're using dynamic scope, because the name `pi` in procedure `area` suddenly refers to the local variable in `mess-up`, rather than to the intended global value.

This argument about naming bugs is particularly compelling to people who envision a programming project in which 5000 programmers work on tiny slivers of the project, so that nobody knows what anyone else is doing. In such a situation it's entirely likely that two programmers will happen to use the same name for different purposes. But note that we had to do something pretty foolish—using the name `pi` for something that isn't π at all—in order to get in trouble.

Lexical scope lets you write compilers that produce faster executable programs, because with lexical scope you can figure out during compilation exactly where in memory any particular variable reference will be. With dynamic scope you have to defer the name-location correspondence until the program actually runs. This is the real reason why people prefer lexical scope, despite whatever they say about high principles.

As an argument for dynamic scope, consider this Logo implementation of the `while` control structure:

```
to while :condition :action
if not run :condition [stop]
run :action
while :condition :action
end

to example :x
while [:x > 0] [print :x make "x :x-1]
end

? example 3
3
2
1
```

This wouldn't work with lexical scope, because within the procedure `while` we couldn't evaluate the argument expressions, because the variable `x` is not bound in any environment lexically surrounding `while`. Dynamic scope makes the local variables of `example` available to `while`. That in turn allows first-class expressions. (That's what Logo uses in place of first-class functions.)

There are ways to get around this limitation of lexical scope. If you wanted to write `while` in Scheme, basically, you'd have to make it a special form that turns into something using thunks. That is, you'd have to make

```
(while cond act)
```

turn into

```
(while-helper (lambda () cond) (lambda () act))
```

But the Logo point of view is that it's easier for a beginning programmer to understand first-class expressions than to understand special forms and thunks.

Most Scheme implementations include a debugger that allows you to examine the values of variables after an error. But, because of the complexity of the scope rules, the debugging language isn't Scheme itself. Instead you have to use a special language with commands like "switch to the environment of the procedure that called this one." In Logo, when an error happens you can *pause* your program and type ordinary Logo expressions in an environment in which all the relevant variables are available. For example, here is a Logo program:

```
;;;;;                          In file cs61a/lectures/4.1/bug.logo
to assq :thing :list
if equalp :thing first first :list [op last first :list]
op assq :thing bf :list
end

to spell :card
pr (se assq bl :card :ranks "of assq last :card :suits)
end

to hand :cards
if emptyp :cards [stop]
spell first :cards
hand bf :cards
end

make "ranks [[a ace] [2 two] [3 three] [4 four] [5 five] [6 six] [7 seven]
             [8 eight] [9 nine] [10 ten] [j jack] [q queen] [k king]]
make "suits [[h hearts] [s spades] [d diamonds] [c clubs]]

? hand [10h 2d 3s]
TEN OF HEARTS
TWO OF DIAMONDS
THREE OF SPADES
```

Suppose we introduce an error into `hand` by changing the recursive call to

```
hand first bf :cards
```

The result will be an error message in `assq`—two procedure calls down—complaining about an empty argument to `first`. Although the error is caught in `assq`, the real problem is in `hand`. In Logo we can say `pons`, which stands for "print out names," which means to show the values of *all* variables accessible at the moment of the error. This will include the variable `cards`, so we'll see that the value of that variable is a single card instead of a list of cards.

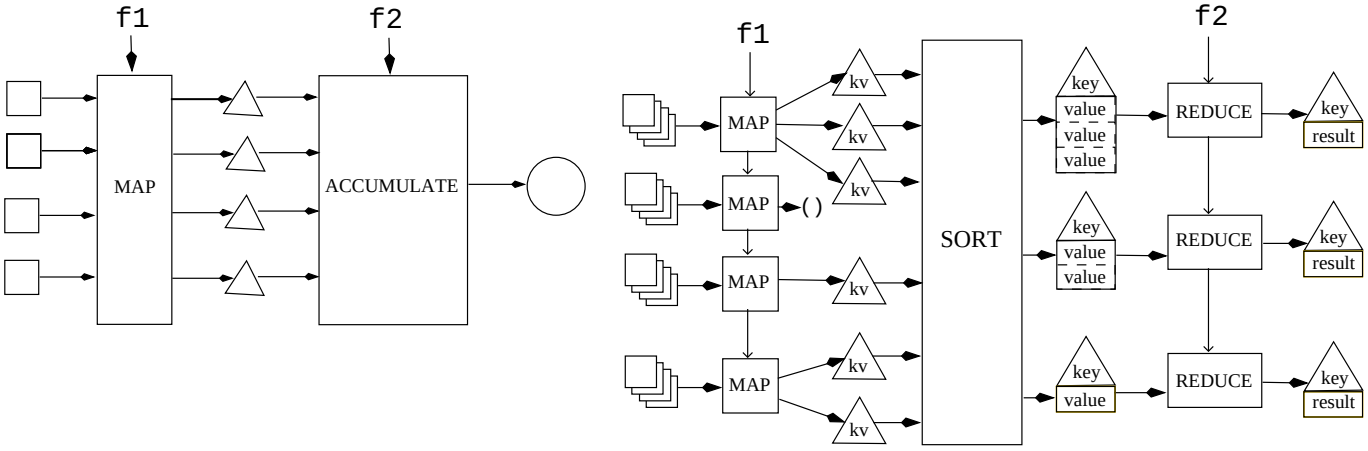
Finally, dynamic scope is useful for allowing "semi-global" variables. Take the metacircular evaluator as an example. Lots of procedures in it require `env` as an argument, but there's nothing special about the value of `env` in any one of those procedures. It's almost always just the current environment, whatever that happens to be. If Scheme had dynamic scope, `env` could be a parameter of `eval`, and it would then automatically be available to any subprocedure called, directly or indirectly, by `eval`. (This is the flip side of the name-capturing problem; in this case we *want* `eval` to capture the name `env`.)

• Environments as circular lists. When we first saw circular lists in chapter 2, they probably seemed to be an utterly useless curiosity, especially since you can't print one. But in the MC evaluator, every environment is a circular list, because the environment contains procedures and each procedure contains a pointer to the environment in which it's defined. So, moral number 1 is that circular lists are useful; moral number 2 is not to try to trace a procedure in the evaluator that has an environment as an argument! The tracing mechanism will take forever to try to print the circular argument list.

• **Mapreduce part 2**

Here's the diagram of mapreduce again:

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



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 element of the file stream represents one line of the file, using a key-value pair whose key is the filename and whose value is a list of words, representing the text of the line. The `cdr` of a file stream is a promise to ask the distributed filesystem for the next line.

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 list 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 merges 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)
                     (stream-map kv-value 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.