CS61A Discussion Week 13   (mapreduce, analyzing evaluator)
TA: Evan
OH: Monday 10:30-12p, 1-2p @ H50 LAB

*** (Know how to use ACCUMULATE/STREAM-ACCUMULATE)**
*** MAPREDUCE**

```
(mapreduce <mapper> <reducer> <null-value> <input> <??>)
```
where

<**mapper>** is a procedure of one argument with the following domain/range:

```
 key-value pair --> [<mapper> ] -->  LIST of key-value pairs
```

<**reducer>** is a procedure of two arguments, like the combiner of accumulate:

```
  curr-VALUE    --> [<reducer>]
  accum-so-far  --> [         ] -->  next-accum-so-far
```

Note the reducer deals with the VALUES of the key-value pairs returned by the mapper.

<**null-value>** is the null-value used in accumulate

<**input>** is either:
  a special string telling mapreduce to use a given input:

```
   Input Directory             Format of input
"/beatles-songs"          (Beatles album . song title )
"/gutenberg/shakespeare" (Shakespeare play . line from play )
"/gutenberg/dickens"      (Dickens story . line from story  )
"/sample-emails"          (sample-email . (from to subject msg))
```

 OR it may be a result of a previous mapreduce!

<??> is an optional flag, where if your provided reducer is "commutative/associative" (i.e. output of reducer can be fed into BOTH inputs of the reducer and still give correct results), it will use an extra optimization step after the MAP phase.

Also, we use the following data abstraction for kv-pairs:

```
make-kv-pair : cons         BUT, use data abstraction to help
you
kv-key       : car          distinguish MAPREDUCE kv-pairs from
kv-value     : cdr          other pairs (i.e. pairs in values)
```

Phase 1. **Map:**
Take every key-value pair in the input. Apply mapper on it. Result is collection of key-value pairs produced by the mapper.
(The data is split up and multiple processes are applying the mapper at the same time)

Phase 2. **Sort into Buckets**:
For each key, make a bucket, and dump the values into it. Also, sort the keys in some order. **(The same processes above are now throwing the results into the appropriate bucket)**

Phase 3. **(Group) Reduce**:
For each bucket, use the reducer repeatedly to combine all the VALUES into a single value. Attach the result to the key again. **(IMPORTANT! Each bucket is handled by one task)**

The end result is a stream of pairs output by Phase 3 (one pair per bucket)

EXAMPLE

```
(mapreduce (lambda (album-song-pair)
             (list (make-kv-pair 'song (kv-value album-song-pair))))
           (lambda (x y) (+ y 1))
           0
           "/beatles-songs")
```

*** Note the call to LIST, don't ever forget that the mapper must return a LIST of key-value pairs**

```
MAP STAGE:
 Input: (<album> . <song-title>)
 Output: ((song . <song-title>))

SORT STAGE:
 For each (song . <song-title>) seen above, put <song-title>
into the song bucket.
 Looks like (song . (<title> <title> ... <title>))

REDUCE STAGE:
 Given the bucket (song . (<title> <title> ... <title>))
 Reduce (<title> <title> ... <title>) using
   (lambda (x y) (+ y 1)) and base case 0.
 Output: (song . <# of songs>)
```

Draw the picture for above.

Here are a few problems to consider. Ask yourself the following:

1. What VALUES does the task deal with? That is, what should go in the VALUE of kv-pairs generated by the mapper.
2. How many GROUPS do I want? The KEY of kv-pairs generated by the mapper determines which group it will go to.
3. How do I combine the VALUES in each GROUP?

* What is the longest line of Shakespeare?

* Which play contains the line “To be or not to be”? Can we also figure out which line number it appears on?

```
* Which character has the most spoken lines in all of
Shakespeare?
  (Note lines starting with  “CHARACTER:” )
```

```
* How many emails contain “ipod” in the message? Of these, who
sends
  the most of those?
```

ANALYZING EVALUATOR

The intuition is quite easy. Just remember, we "compile" expressions into procedures that take in an environment. This is mainly for speeding up procedure calls (and note, NOT for just recursive procedures).

For instance, in mc-eval, let's suppose I use the square procedure a lot.

Here is a sample call to square:

```
(define (square x) (* x x))
(square 7)
```

```
[1] not self-evaluating, not a symbol ....
       application: eval square, eval 7
       apply square to operands: (7)
[2] Not primitive... It's a compound procedure:
       extend environment, evaluate (* x x)
[3] eval (* x x): not self-evaluating, not a symbol ...
       application: eval *, lookup x, lookup x,
       apply *
```

Now, every time we call square, we have to go through mc-eval's cond clause, checking for what type of expression the body of square is (in step [3]).

What if we could analyze the square procedure once, so that we KNOW what type of expression the body of square is? Save ourselves some trouble! Well, that's what analyzing evaluator does.

So, what we'd like to see, is that when calling square, in step [3], we "know" it's an application, so we jump straight into action:

```
[new step 1]  lookup *, lookup x, lookup x, apply *.
```

How do we do this? First we analyze the expression. After analysis, then we package the information into a procedure:

```
(lambda (env) (apply (lookup * env)
                     (list (lookup x env) (lookup x env) )))
```

Then, every time we call square, we just call the above procedure with the appropriate environment. (i.e. for (square 8), give it an environment where x is 8).
In analyzing-eval, the compiled expression won't look exactly like this, because we have to handle general cases.

So here's the model:
(define (analyzing-eval exp env) ((analyze exp) env))

Analyze the expression, then when it's time for evaluation, plug in the environment.

Analyze-lambda:
```
(lambda <param> <body>)
  <body>  =analyze=>  <analyzed-body>
  ==========================================>
       (lambda (env)
             (make-procedure <param> <analyzed-body> env))
```

Here is where most of the benefits of analysis will come. The difference is that we analyze the body BEFORE we make the procedure, so when it comes to calling this procedure, all we have to do is take the analyzed-body and pass in the appropriate environment (as we mentioned earlier about square)

Analyze-application: (for lambda-created procedures)
```
(<proc> <operands>)
   <proc>     =analyze=>  <analyzed-proc>
   <operands> =analyze=>  <analyzed-operands>
  ============================================>
 (lambda (env)
   (let ((proc (<analyzed-proc> env))
     (operands (map (lambda (a) (a env)) <analyzed-operands>)))
     ((procedure-body proc)
        (extend-environment (procedure-parameters proc)
                            operands
                            (procedure-environment proc)) ))
```

From the above, you can deduce the following rule:

In a given transcript, you will see speedup if some non-primitive procedure is called more than once!

Question about analyzing eval:

1. Which of the following would have speedup in analyzing eval?

   a. (+ 1 2)

   b. (((lambda (x) (lambda (y) (+ x y))) 5) 6)

   c. (map (lambda (x) (* x x)) '(1 2 3 4 5 6 7 8 9 10))

```
d. (define fib
      (lambda (n)
        (if (or (= n 0) (= n 1)) 1
           (+ (fib (- n 1)) (fib (- n 2))))))
   (fib 5)

e. (define fact
     (lambda (x) (if (= x 0) 1 (* x (fact (- x 1))))))

f. (accumulate cons nil '(1 2 3 4 5 6 7 8 9 10))
```