CONCURRENCY AND MAPREDUCE 14

COMPUTER SCIENCE 61AS

Concurrency

1. What are the possible values of x after the below?

```
(define x 5)
(parallel-execute (lambda () (set! x (* x 2)))
                  (lambda () (if (even? x)
                                 (set! x (+ x 1))
                                 (set! x (+ x 100)))))
```

Solution: 11, 210, 10, 105, 110

2. Consider the procedure squares!, which squares every element of a list in place:

```
(define (squares! lst)
  (if (not (null? lst))
      (begin (set-car! lst (square (car lst)))
             (squares! (cdr lst)))))

> (define lst '(3 4 5))
> (squares! lst)
> lst
> (9 16 36)
```

Use parallel-execute to rewrite squares! so that the squaring is done in parallel. Your new implementation should not impose any restriction on the order in which squares are computed.

Solution:

```
(define (parallel-squares! lst)
  (if (not (null? lst))
      (parallel-execute (lambda () (set-car! lst (square (car lst))))
                        (lambda () (parallel-squares! (cdr lst))))))
```

Mapreduce

General comment:
Remember that unless otherwise specified, you are free to make helper functions for your mapper and reducer.

1. **`count-words`**
   Consider the following MapReduce query:

   ```
   (define (count-words input)
     (list (make-kv-pair (kv-key input) (length (kv-value input))) ))

   (mapreduce count-words + 0 shakespeare)
   ```

   The result is a stream of key-value pairs, where the keys are the names of Shakespeare plays and the values are the number of words in the play.
   `((a-lovers-complaint . 2568) (a-midsummer-nights-dream . 17608) ...)`

   (a) Change either the mapper or the reducer (but not both) to find the length of the longest line in each play. You can either use an existing Scheme primitive or write an entirely new procedure. If you change the reducer you can also change the base case in the mapreduce call. Show your new call to mapreduce.

   Solution: `(mapreduce count-words max 0 shakespeare)`

   (b) Change either the mapper or reducer (but not both) to count the number of times the word ”thou” appears in each play. You can either use an existing Scheme primitive or write an entirely new procedure. If you change the reducer you can also change the base case in the mapreduce call. Show your new call to mapreduce. The key will be the name of the play, the value will be the number of time ”thou” appears.

   Solution:

   ```
   (define (get-thou kv)
     (map (lambda (thou) (make-kv-pair (kv-key kv) 1))
          (filter (lamda (wd) (equal? wd ’thou)) (kv-value kv))))

   (mapreduce get-thou + 0 shakespeare)
   ```

2. **Searching**
   Given a list of seach keywords, we want to find out which documents in our input stream contain each word, using MapReduce. The input key-value pairs have a document name as the key and a list of words from one line as the value. (Note: The same document may have many lines).

   Write a procedure `search` that, given a list of keywords and the input stream, returns a stream sorted by search word in which we can find, for any keyword, the names of the documents containing that word. (Don't worry about cases in which the same keyword appears more than once in a file. Assume that mapreduce returns a sorted key-value pair based on keys).

Solution:

```
(define (search keywords input-stream)
  (define (mapper input-kv)
    (map (lambda (wd) (make-kv-pair wd (list (kv-key input-kv))))
         (filter (lambda (wd) (member wd keywords))
                 (kv-value input-kv))))

  (define (reducer l1 l2)
    (if (and (not (null? l2))
             (memq (car lst2) lst1)) ;prevents duplicates if filename
              is already there
        l1
        (append l1 l2)))

  (mapreduce mapper reducer nil input-stream))
```