CS61A Summer 2010

George Wang, Jonathan Kotker, Seshadri Mahalingam, Steven Tang, Eric Tzeng

Homework 6

Due: Monday, August 2, 2010, at 7AM

*N*ote: The number of stars (★) besides a question represents the ***estimated*** relative difficulty of the problem: the more the number of stars, the harder the question.

1 How to Start and Submit

First, download `http://inst.eecs.berkeley.edu/~cs61a/su10/hw/hw6.scm` and use it as a template while filling out your procedures. See `http://www-inst.eecs.berkeley.edu/~cs61a/su10/hw-faq.pdf` for submission instructions.

2 Concurrency

1. SICP Exercises
   (a) (★) 3.38
   (b) (★) 3.39
   (c) (★) 3.40
   (d) (★★) 3.41
   (e) (OPTIONAL) (★★)3.42
   (f) (OPTIONAL) (★) 3.44
   (g) (★) 3.46 You can either draw their style of diagram, another diagram, or just describe it in words. Bring this exercise to lab next week!
   (h) (OPTIONAL) (★) 3.48
2. (★★) A *stack* is a data type that allows you to `push` a new item onto the top, and `pop` a new item off the back. Different from a queue, which is first-in, first-out, a stack is last-in, first-out. A classic example is a pez dispenser. A simple OO stack could be implemented like this:

```
(define-class (stack)
  (instance-vars (items '()))
  (method (push item)
    (set! items (cons item items)) )
  (method (pop)
    (let ((top (car items)))
      (set! items (cdr items))
      top)))
```

   This works, but isn't safe for parallel use. If two users `push` or `pop` at the same time, one of the items could be lost.

Rewrite the `stack` class to be safe for concurrent use **using mutexes**. Assume that the only ways to change the state of the stack is to use the push and pop methods. Think about why you can't use serializers!

3 Streams

1. (★★) Streams are used heavily in video and audio transmission (hence, *streaming* a video). By delaying evaluation until when we want the data, Streams give us the illusion of having an infinitely large amount of data in our computer just as how when you stream a radio station, you get the illusion that the entire broadcast is stored in your computer.

   So suppose the raw audio transmission from a radio is stored as a stream of numbers, write a procedure `(smooth-stream stream n)` that takes a stream and return a new stream where the first element of the returned stream is the average of the 1st to n-th element of the input stream, the 2nd element of the returned stream is the average of the 2nd to $n+1$-th element of the input stream, etc. These numbers may represent the amplitude of a signal at that point.

   For example:

   ```
   > (define ints (cons-stream 1 (stream-map 1+ ints)))
   > (define smoothed-ints (smooth-stream ints 4))
   > (ss smoothed-ints)
    (2.5, 3.5, 4.5, 5.5, 6.5, 7.5, 8.5 ...)
   ```

   Because (1+2+3+4)/4 = 2.5, (2+3+4+5)/4 = 3.5, (3+4+5+6)/4 = 4.5, etc. In real world, noise can enter our signal and smoothing the signal dampens the effect of noises.

2. (★) What are the first 15 elements of the following stream:

   ```
   (define mystery
     (cons-stream 0
        (cons-stream 1
           (interleave
               (stream-map (lambda (x) (word 0 x)) mystery)
               (stream-map (lambda (x) (word 1 x)) mystery)))))
   ```

   Try to figure it out without using the Scheme interpreter. Can you describe what stream `mystery` is concisely?

3. (★★) Write a procedure stream-scan. Stream-scan is a cross between map and accumulate. It applies a function to successive elements of a stream and combines the values, like accumulate, but it returns a list of intermediate values, like map. Unlike accumulate, it can handle infinite streams.

   ```
   > (ss (stream-scan + 0 integers))
   (1 3 6 10 15 21 28 36 45 55 ...)
   ```

4. SICP Exercises
   (a) (★★) 3.50
   (b) (★★) 3.51
   (c) (★★) 3.52
   (d) (★) 3.53
   (e) (OPTIONAL) (★) 3.54
   (f) (OPTIONAL) (★★) 3.55
   (g) (★★) 3.56

4 Optional Case Study: Client/Server Battleship

You'll be implementing a simple multi-player battleship game in this case study.

The code for this are the battleship-client.scm and the battleship-server.scm files. The following problems belong in battleship-server.scm.

These files are located at:

`http://inst.eecs.berkeley.edu/~cs61a/su10/hw/battleship-server.scm`

`http://inst.eecs.berkeley.edu/~cs61a/su10/hw/battleship-client.scm`

1. (★) First, make a procedure named `setup-grid`. It should make a 5 by 5 grid out of vectors (a vector of vectors) with all of the starting entries as `#f`, and it should place 3 "battleships" for each player (these battleships should be represented as a word p1 or p2 corresponding to the battleships for player 1 or player 2). In other words, ships are only one single square big, and both players will share the same board.

2. (★★) Define a procedure `hitsearch` that takes in the user's guess. Guesses come in the form of a word of a letter (a-e) and a number (1-5), such as a1 or e5. It should do the following:
   - The procedure should check the given point on the grid to see if it is a hit (note, vectors are numbered 0-4 but the guess will be numbered 1-5).
   - If it is a miss: the procedure should just return the word miss.
   - If it is a hit:
     (a) The procedure should return the word `hit` and that spot on the grid should be set to `#f`.
     (b) Then it should increase the current player's score using the `(increment-current-player-score)` procedure.
     (c) Finally, it should check to see if the current player's score is 3. If it is, then that player has won the game, in which case it should set the variable `game-still-going` to `#f`.

3. (★★★) Make the procedure `battleship-guess` that takes in the user's guess and the name of the player making the guess.
   (a) First, it should check to make sure that the player making the guess is the current player. If it is not, use your `server-broadcast` procedure to remind everyone of the player's name whose turn it is. Note that (current-player-name) provides the current player's name.
   (b) The user's guess should be checked with your hitsearch procedure to determine if it is a hit or miss.
   (c) Then, if no one has won the game yet (ie, if the variable `game-still-going` is still `#t`):
      i. Use your server-broadcast procedure to send out the result to everyone.
      ii. Switch the current player using the `(switch-player)` procedure.
      iii. Broadcast a message with the new `current-player`'s name informing everyone of whose turn it is.h
   (d) If someone has won the game, simply invoke the (win-game) procedure.

4. Extra for Experts: Extend this game as you like it! Some suggestions are:
   (a) The ability to restart the game, keeping track of scores for both players.
   (b) Different Ship Sizes, so ships that are 2, 3, 4, and 5 spaces wide.
   (c) Separate board for each player.

(d) An AI player that plays “smart”.

5 Optional: MapReduce

1. Google uses MapReduce for a variety of search-engine-related tasks involving processing large data sets. An example of such a task is building an inverted word index. For each word in a given set of documents, we want to generate a key-value pair, where the key is the word and the value is all documents in which the word appears.

   Everything you need should already work. Remember to follow the instructions on the lecture notes regarding `ssh`-ing into `icluster` to be able to use the MapReduce commands. This stuff is cool!

   (a) (★) Write a procedure that takes a directory name as argument and returns the inverted index stream.

   (b) (★★) The list above will include words like “a,” “so,” “the,” etc. Suppose now that we only want “important words.” We can use word length as a measure of importance (a real search engine would use a more complex heuristic). Write a procedure that returns an index where all the words are N or more letters long, taking the filename and N as arguments.

2. Google also manages the GMail e-mail service, and they would like to filter out as many spammers as possible. In this problem you will implement a simple spam filtering idea with `mapreduce` so that it can be efficiently applied to the multitudes of e-mail messages in GMail.

   We want to produce a “blacklist” of e-mail addresses that are spamming GMail users. Spam messages are sent to many addresses at once, and so a spam message can be identified by having one of the most commonly-used subject lines. If an address sends many messages with the most common subject lines, it is likely a spammer address. We would like to find the top ten addresses that have sent the most messages with frequently-recurring subject lines.

   The input data is a collection of e-mail records in the file `gmail-messages`. (These are simulated messages, not actual GMail data!) Each e-mail record is a list with the format

   ```
   (from-address to-address subject body)
   ```

   where each element is a double-quoted string. The mapper function will be applied to each e-mail record. An small example set of e-mail records is below.

   ```
   ("cs61a-tb" "cs61a-tc" "mapreduce" "mapreduce is great! lucky students!")
   ("bot1337" "cs61a-ta" "free ipod now!" "buy herbal ipod enhancer!")
   ("bot1338" "cs61c-tf" "free ipod now!" "buy herbal ipod enhancer!")
   ...
   ```

   (a) (★) Our first step is to produce a table with subject lines as keys and counts of occurrences as values. Identify the intermediate key-value pairs and write the mapper and reducer functions.

   (b) (★★) From our tabulation of subject line occurrences in 2(a), we want to find the most common subject lines in the table. We can perform a sort by using the fact that MapReduce sorts by intermediate keys into the reducer groups. Identify the intermediate key-value pairs and write the mapper and reducer functions.

   (c) (★★) Finally, assuming you’ve moved the ten most common subject lines into a list, we want to make a table with from-addresses as keys and counts of e-mails sent with common subject lines as values. You don’t need to sort the table, since the procedures would be identical to those in 2(b). Identify the intermediate key-value pairs and write the mapper and reducer functions.

6 Extra for Experts (Optional)

1. SICP 3.59-3.62
2. Define `hanoi-stream`:

```
> (hanoi-stream 1)
(1)
> (hanoi-stream 2)
(1 2 1)
> (hanoi-stream 3)
(1 2 1 3 1 2 1)
> (hanoi-stream 4)
(1 2 1 3 1 2 1 4 1 2 1 3 1 2 1)
```

7 Feedback

Now that you are done, please leave me some feedback at the following link regarding how the course is going. This is not worth points, but will give us valuable feedback that in turn improves your course experience. Thanks! `http://spreadsheets.google.com/viewform?formkey=dFQxRmx6WEtBUWFvamdvYzlVR1pvRWc6MQ`