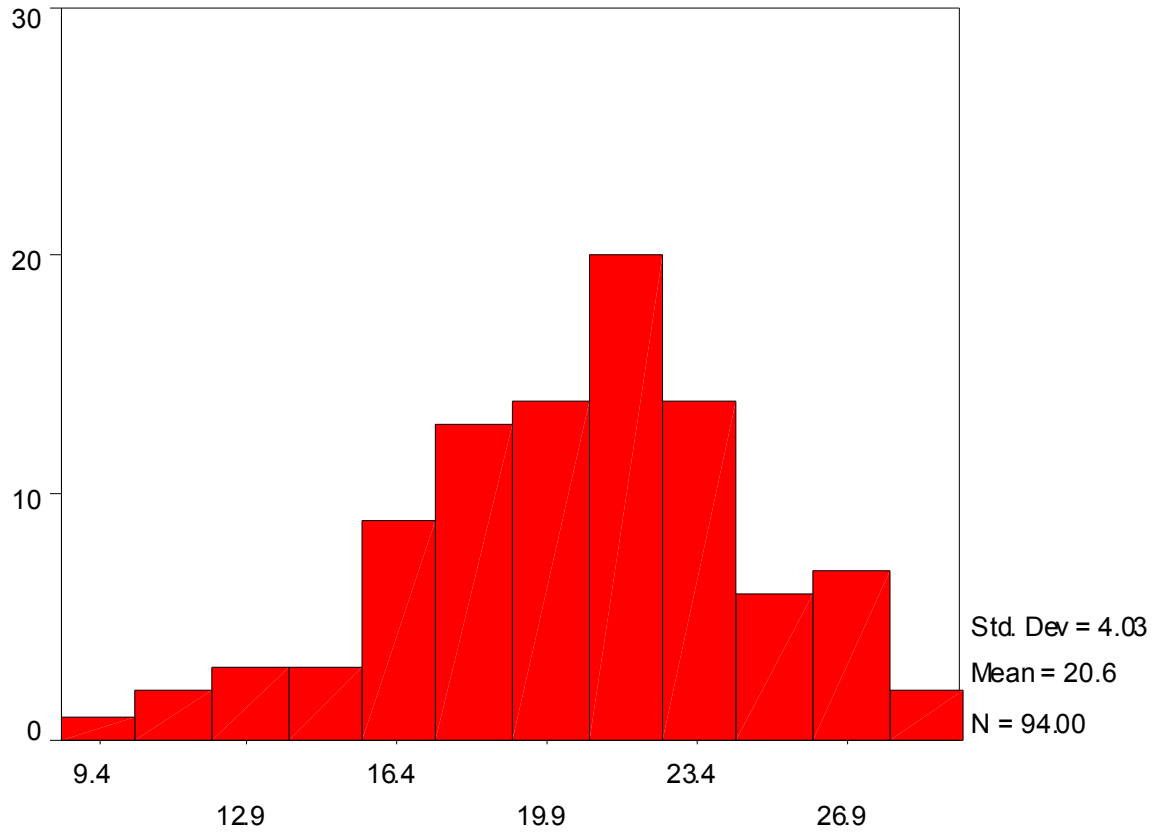
CS3 Midterm 1
Standards and Solutions

Spring 2007

Here is a histogram of the scaled scores (that it leans to the right a bit, and that the minimum score is a 9, means I think you did well on the exam):



m1-scale

Problem 1. And the return value is... (10 points [#9 is worth 2 points])

Write the result of evaluating the Scheme expression that comes before the ➔. If the Scheme expression will result in an error, write *ERROR* in the blank and describe the error.

1	`(se 'I (word 'am (se 'sam)))` ➔ `error - (se 'sam)` isn't a valid argument to `word`
2	`(quote (word sam I am))` ➔ this returns the sentence `(word sam I am)`

3	`(appearances`
	`  (bf 'sam)`
	`  (se 'that 'sam 'I 'am 'sam 'a 'a 'a))`
	
	➔ 1 – appearances is checking for the word `am`.

4	`(cond ('false 'is-true)`
	`  ('lies 'are-truth)`
	`  (else 'im-confused))` ➔ `is-true`

5	`(equal? 3 (or 1 2 3))` ➔ `#f`
6	`(and #t (and #t (and #t (and #t (and #t (and #t #f))))))` ➔ `#f`
7	`(member? 'I (word 'you 'and 'I))` ➔ `#t`
8	`( 3 + 5 )` ➔ `error - 3 is not a function`

9	`(define (mystery sent)`
	`  (or (empty? sent)`
	`  (and (+ 1 (first sent))`
	`  (mystery (bf sent)) ) ) )`
	
	`(mystery '(2 3 4))` ➔ `#t` – the `and` will return true when given `(+ 1 (first '(2 3 4))` as its argument.

Problem 2. Whatever floats your boat (A: 6 points, B: 4 points)

This problem involves a procedure `can-order?`, which takes two ranks in the United States navy and returns `#t` if and only if the first rank is "above" the second and can, therefore, order the other one around. The following table lists the ranks:

rank	Explanation
5	*5 star admiral*
3	*3 star admiral*
1	*1 star admiral*
cpn	*captain*
cmd	*commander*
ltn	*lieutenant*
en	*ensign*

Part A:

Write `can-order?` in the form of the "better solution" in the *Difference Between Dates* case study (the second attempt that successfully wrote `day-span`, after the dead end was reached in the first attempt). You can assume that the ranks passed to `can-order?` are valid.

Chooose good names for your parameters and helper procedures, and add relevant comments above every procedure. Partial credit will be awarded for solutions that don't follow the form of the better solution in *Difference Between Dates*.

The "better solution" in the case study involves turning ranks into numbers, and then comparing the numbers instead of the ranks directly. One solution to this:

```
(define (can-order? rank1 rank2)
  (> (value rank1) (value rank2)))

(define (value rank)
  (cond ((equal? rank 'cpn) 0)
        ((equal? rank 'cmd) -1)
        ((equal? rank 'ltn) -2)
        ((equal? rank 'en) -3)
        (else rank) ) )
```

Most of the answers also changed the admiral ranks, which made for a slightly more complicated `cond`, but was worth full credit. Collectively, you did really, really great on this problem. One TA thought it would be too tricky , but you certainly got the point of the case study and proved us wrong. Some of you lost poinst for forgetting comments, but all of you named your procedures and parameters well. Good job.

Part B:

There are many possible valid calls to `can-order?`. For this problem, you will write test cases for the procedure, but we don't want a large list. Instead, we want you to describe what the general classes of tests cases are. That is, think about how the test cases can be grouped, such that the cases in a group are similar in how they check for errors or otherwise test the program.

There are not many groups. For each group, briefly describe what the similarity is and provide a single test case. Be sure to include the correct result of the test case.

We were looking for three groups here:

1. Test cases where the ranks are the same. For instance, `(can-order? 'cmd 'cmd)` should return `#f`.
2. Test cases where the first rank passed is higher, e.g., `(can-order? 'cmd 'en)`, which should return `#t`.
3. Test cases where the second rank passed is higher, e.g. , `(can-order? 'en 'cmd)`, which should return `#f`.

Problem 3. And and or were walking down the street... (5 points)

A friend tells you that `and` always returns the last argument given to it, unless there is an error. He types:

`(and 'joe 'bob)` and it returns `bob`

`(and 'joe 'bob 'briggs)` and it returns `briggs`

Give an example showing that he is wrong.

__`(and #f 'joe)`_____

Basically, anything with `#f` in a position other than the last works here, because it will cause the `and` to return `#f`.

He then tells you that `or` always returns the first argument it is passed, typing:

`(or 3 4 5)` which returns `3`

Again, can you prove him wrong by providing a counter example?

__`(or #f 3)`_____

> Anything in which `#f` is the first argument and a true value comes after will prove him wrong.

Finally, your friend wrote a procedure `not-in-order?`, which takes three numeric arguments and returns true if and only if they are neither in ascending or decending order. It might not work!

```
(define (not-in-order? a b c)
  (or (not (or (> a b) (< b c)))
      (not (or (< a b) (> b c)))))
```

What will this procedure return for three ascending arguments? ____`#f`_____

What will this procedure return when the middle argument is larger than the other two? ____`#t`_____

Will this procedure always return the same thing in calls where exactly two of the arguments are the same? __yes (always returns `#t`)___

> These problems just required carefully working through the cases.

Problem 4. Recursive `general-day-span` (A: 5 points, B: 8 points)

These questions concern the case study *Difference between dates, recursive version.* The code for this case study is included at the end of this exam (which is Appendix C in your reader).

Part A:

Correctly define the following functions by filling in the blanks. Solutions should fit easily! (Points will be deducted, otherwise).

You can (and should) use procedures defined in the case study!

```
;; takes a month-name and a date-in-month, and returns a valid date.
;;  e.g., (make-date 'january 3) ==> (january 3)
(define (make-date month-name date-in-month)
     ___(se month-name date-in-month)______ )
```

```
;; takes a month-name and a number representing days, and returns true
;; if the month has that many days in it.
;;  e.g., (has-days? 'january 31) ==> #t
(define (has-days? month-name num-days)
   ___(equal? (days-in-month month-name) num-days)___)
```

```
;; Takes a non-december date, and return the name of the following
;; month.
;;  e.g., (next-month-name '(january 3)) ==> february
(define (next-month-name date)
   ____(name-of (next-month-number date))_____________  ) or

   ____(name-of (+ 1 (month-number (month-name date))))____  )


;; Takes two dates, and returns the number of months spanned by those
;; dates (including the months for each of the dates).  Assume the
;; first date is earlier than the second.
;;  e.g., (month-span '(june 16) '(july 4)) ==> 2
(define (month-span earlier-date later-date)

    __(+ 1 (- (month-number (month-name later-date))
              (month-number (month-name earlier-date))))______
```

> OK, so the last one didn't fit so well, but we were lenient on that point. We did take off one point if you passed the wrong argument into a DbD procedure (for instance, a date to `month-number`). We took of ½ a point if you used `first` instead of `month-name`, for violating the data abstraction.

Part B:

In the case study, recursion was used in the `day-sum` procedure, which was called by `general-day-span`. For this problem, you will write a recursive `general-day-span-r` which, in a sense, combines the functionality of `day-sum` and `general-day-span` into one procedure.

You can use helper procedures from part A or from the case study, except for `day-sum`. (Assume the procedures from part A work correctly). You can also write your own helper procedures, but they must be non-recursive.

Your solution must use recursion—that is, your definition of `general-day-span-r` must include a meaningful call to `general-day-span-r`.

```
(define (general-day-span-r earlier-date later-date)
```

> This was the most difficult problem on the exam, because in most solutions there were several details to worry about. However, there were multiple solutions to this problem.
>
> The most important aspect was to see that you couldn't simply add the days remaining of `earlier-date` and `(date-in-month later-date)`, then recursively add up the months in between like in `day-sum`. Anyone who attempted to write the procedure like day sum received little credit. Most people lost points because they either over-counted months or wrote a base case that didn't account for days-remaining properly (in other words, that months have different number of days in them).
>
> There were two clear tasks in this question:
> 1) Writing a proper recursive function (worth 4 points)
> 2) Having the right base case for that function (4 points)

A lot of people mixed what would have been a good base case for one solution with a recursive call for the other, and as a result could only gain as many as 4 points (we gave which ever was higher – the base case or recursive case score). Finding what solution you intended, in order to give partial credit, was often a challenge. Half a point was taken off for using `first` of a date to get the name of the month instead of using `month-name` (a data abstraction violation).

Again, there are many ways to write this function. Here are some:

```
;; Solution 1.  Counting days-in-month and subtracting the
;;  over-counting in the base case
(define (general-day-span-r earlier-date later-date)
  (if (equal? (month-name earlier-date)  (month-name later-date))
     (- (date-in-month later-date)
        (date-in-month earlier-date))
     (+ (days-in-month (month-name earlier-date))
        (general-day-span-r (make-date (next-month-name earlier-date)
                                       (date-in-month earlier-date))
                            later-date) ) ) )
```

In the recursive case of solution 1, the month of the each date is added. This will include the days-in-month of earlier-date, which shouldn't be included. However, the base case then subtracts that number, and corrects for the overcounting. This wasn't a particularly common solution...

In solution 2 below, the days remaining for the `earlier-date` is added in the recursive case, but the `earlier-date` for subsequent recursive calls is changed so that `date-in-month` is 1. In this way, the days-remaining becomes the full month for each subsequent call, and that is how the months are added. A base case of what remains in the later-date make everything work well.

```
;; Solution 2.  Counting days remaining and changing the date in
;;  month of the earlier date
(define (general-day-span-r earlier-date later-date)
  (if (equal? (month-name earlier-date)  (month-name later-date))
      (date-in-month later-date)
      (+ (days-remaining earlier-date)
         (general-day-span-r (make-date (next-month-name earlier-date)
                                        1)
                             later-date) ) ) )
```

Solution 3 has a different test than the first two solutions, stopping when the months are consecutive. At first glance it looks like this solution will be wrong, because it adds the days-in-month for the month of the initial earlier-date, but (in the base case) the days-remaining for the earlier-date of the last recursive call (which will be the month before the month in the initial later-date). But, if you carefully work through the recursion, any extra days that are added on because of different lengths of those two months is corrected for in the subsequent calls. Tricky, but correct!

```
;; Solution 3.  Counting days-in-month and then stopping at
;;  consecutive months (this could also be done backwards from later-date)
(define (general-day-span-r earlier-date later-date)
  (if (consecutive-months? earlier-date later-date)
      (+ (days-remaining earlier-date)
         (date-in-month later-date) )
      (+ (days-in-month (month-name earlier-date))
         (general-day-span-r (make-date (next-month-name earlier-date)
                                        (date-in-month earlier-date) )
                             later-date) ) ) )
```

There were other ways of solving this as well. For instance, there were also solutions that involved changing what the procedure takes and identifying the "dates" (the initial call) as dates or something modified (every subsequent recursive call), and then treating these two recursive cases separately. Another solution many attempted was to count a single day in each recursive case, but nobody created the date for the "next-day" correctly when a new month was reached. That would have been a nice solution, however.

For every solution, partial credit was given when the idea was met and implemented at least somewhat properly. In many cases, students failed to receive partial credit because they mixed up cases and didn't even give the correct number of arguments to `general-day-span-r`.

Problem 5. Scrambled and confused (A: 5 points, B: 2 points, C: 5 points)

Consider the procedure `scramble` defined below:

```
;; takes a word and a sentence of numeric positions, and returns the
;; word formed by the letters (of the original word) pointed to by each
;; position, in the order defined by the sentence of positions.
(define (scramble wd positions)
  (if (empty? positions)
      ""
      (word (item (first positions) wd)
            (scramble wd (bf positions)))))
```

Part A: For the the expression

```
(scramble 'fred '(4 3 1 2))
```

list all of the calls to `scramble` that will result, in order, and what the return value of each call will be. This is the same information that tracing `scramble` will provide.

There were a total of 5 calls to scramble that would occur during the evaluation of the the above expression:

```
(scramble 'fred '(4 3 1 2)) ➔ defr
(scramble 'fred '(3 1 2))   ➔ efr
(scramble 'fred '(1 2))     ➔ fr
(scramble 'fred '(2))       ➔ r
(scramble 'fred '())        ➔ ""
```

Part B: Do a particular scramble on any word using sentence operators. Define the procedure `scramble-532`, using only the procedures `word`, `first`, `last`, `butfirst`, and `butlast`, which takes a five-letter word and scrambles it with the position sentence `'(5 3 2)`. For example,

`(scramble-532 'abcde)` ➔ `ecb`

```
(define (scramble-532 wd)
```

This was designed to be easy :

```
 (define (scramble-532 wd)
   (word (last wd) (first (bf (bf wd))) (first (bf wd))))
```

Part C:

Consider a better version of `scramble` (called `better-scramble`) in which the sentence of positions can be more complicated. For `better-scramble`, each position can be

- a positive number, which references the letter at that position.
- a negative number, which references the letter at the position counting from the end of the word towards the front
- the word `first`, which references the first letter of the word.
- the word `last`, which references the last letter of the word.

For example,

`(better-scramble 'abcde '(-2 first 3 last))`	➔	`dace`
`(better-scramble 'abcde '(-4 first -2))`	➔	`bad`

Fill in the blanks to complete `better-scramble`. Assume that the sentence of positions will always be valid (i.e., won't contain positions out of range, etc.).

```
(define (better-scramble wd positions)
   (if (empty? positions)
       ""
       (word (item (real-position ____wd (first positions)_________ ) wd)
             (better-scramble wd (bf positions)) ) ) )
```

```
(define (real-position _____wd  position________ )
```

```
  (cond ((and (number? position)
              (> position 0) )
         position)
        ((number? position)
         (+ 1 (count wd) position))
        ((equal? position 'first)
         1)
        ((equal? position 'last)
         (count wd) )
        ) )
```