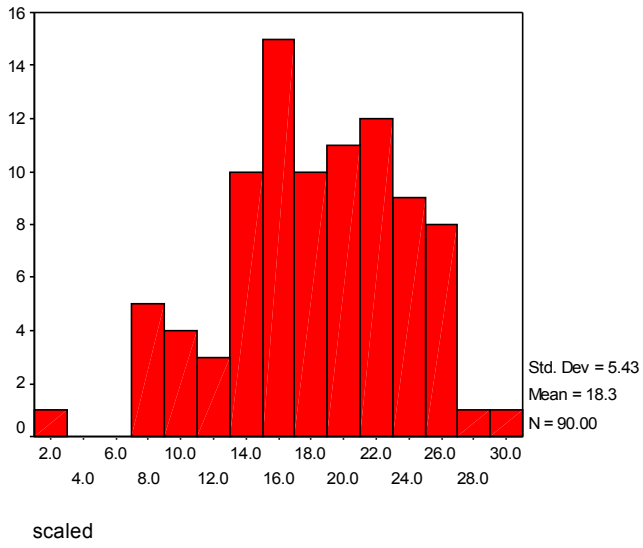
# CS3 Midterm 2 Standards and Solutions
# Spring 2007




**Problem 1. Roman Holiday (7 points)**

A "tail recursion" is one in which return values from recursive calls are not combined in any way. Generally, this is achieved by an accumulating recursion.

Rewrite `roman-sum`, from the Roman Numerals case study, using a tail-recursive solution. The code from the case study is included in an appendix at the end of this exam.

> This problem confused many of you because of the terms "tail recursion" and "accumulating recursion". These were covered in lab and lecture, and knowing either term should have led you to the right solution.
>
> Remember, in a tail recursion, there are no pending procedures to combine the return values from the recursive calls. Saying this another way, in tail recursions the base cases must return the full and complete answer to the initial call. The way this is achieved, in most cases, is through an extra argument in a helper procedure that "accumulates" the answer through successive recursive calls – that is, an accumulating recursion.
>
> Needless to say, an accumulating recursion is very different than the higher order function `accumulate`!
>
> In this problem, the solution given in the roman numerals case study (and, the appendix in the exam) is a non-tail recursive solution (called "embedded"). To change it to an accumulating solution, you need to make a helper procedure in which the recursion took place that had a "so-far", or accumulating, argument. For a few of you, you accumulated a sentence of numbers, and then added them up. Most of you simply kept track of the answer, like in the solution below:
>
> Since the code was supplied at the back of the exam, it was impossible to gain points by doing an embedded recursion (as this was simply copying from the back of the test).

```
(define (roman-sum number-sent)
  (roman-sum-helper number-sent 0))

(define (roman-sum-helper number-sent so-far)
  (cond ((empty? number-sent) so-far)
((empty? (bf number-sent))
 (+ (first number-sent) so-far))
((starts-with-prefix? number-sent)
 (roman-sum-helper (bf (bf number-sent))
           (+ so-far (second number-sent)
              (- (first number-sent)))))
(else
 (roman-sum-helper (bf number-sent)
           (+ so-far (first number-sent))))
) )
```

**Problem 2. At the right price (A: 6 points; B: 6 points) (graded by Alex)**

"The Price is Right" was a game show which started each round in the same way: several guests would guess at the price of an item, trying to get as close as possible to the real price without going over. The person who was closest without going over would win.

In this question, you will work with a procedure `PIR-winner` which takes a sentence of guesses and a real price, and returns the guess that is closest without being greater than the real price:

| | | |
|---|---|---|
| `(PIR-winner '(134 245 199 300 160) '200)` | ➔ | `199` |
| `(PIR-winner '(19 35 56 44 22) '35)` | ➔ | `35` |

(You can assume that there is always at least one guest whose guess is below or equal to the real price, and that no two guesses will be the same).

*Part A.* Write `PIR-winner without` using recursion (i.e., use higher-order procedures).

The easiest solution used a `keep` to remove all the overbids, and then an `accumulate` to find the maximum of the remaining bids:

```
(define (PIR-winner guesses price)
  (accumulate max
      (keep (lambda (n)
              (<= n price))
            guesses)))
```

If your solution looked like that, but you forgot the `accumulate` (trying to apply `max` to a whole sentence), then you lost one point.

Others, however, used only accumulate to process the sentence of guesses, which was more difficult. Care had to be taken to avoid problems when the rightmost guess was an overbid, or the first guess was the winning guess. A solution might look like:

```
;; only accumulate version
(define (pir-winner guesses real-price)
  (accumulate
   (lambda (guess sofar)
     (cond ((> sofar real-price)
            guess)                        ;haven't seen an underbid yet
           ((and (> guess sofar)          ;guess bigger than underbid sofar
                 (<= guess real-price)) ; and still an underbid
            guess)                        ; so return it
           (else sofar)))
   guesses))
```

If your solution didn't somehow filter out over bids, you lost 3 points.

*Part B.* Consider the following recursive version of `PIR-winner`:

```
(define (PIR-winner guesses real-price)
  (pir-win-help guesses real-price real-price))

(define (pir-win-help guesses real-price diff)
  (cond ((or (empty? guesses)
             (empty? (bf guesses)))
         (- real-price diff))
        ((and (<= (first guesses) real-price)
              (>  (first guesses) (- real-price diff)))
         (pir-win-help guesses real-price (- real-price (first guesses))) )
        (else
         (pir-win-help (bf guesses) real-price diff)) ) )
```

What will the return value for the call below be?

`(PIR-winner '(15 30 20 25) '22)` ➔ **20**

The solution above was worth 1 point. A common incorrect solution was that it would loop forever, presumably because of the incorrect first recursive call (that it didn't use `(bf guesses)`). However, because of the way `diff` was passed in call and testing for in that case, that recursive call was bypassed the next time around, and the recursion would happily continue onto the `butfirst` of `guesses` (through the second recursive call).

Briefly describe the set of values for `guesses` and `real-price` that will result in an incorrect return value?

The incorrect value comes about because of the first conditional case (the base case), and that it is incorrectly checking whether the sentence of guesses has one element. As a result, the recursion is going to stop too early. This problem manifests, then, when the last guess is the winning guess. Some of you only wrote that this would happen when there was only one guess, which got 2 out of 3 points. Answers with negative guesses or guesses in which none were less than or equal to `real-price` weren't worth any points, because that was outside the domain of the procedure).

Briefly describe the set of values for `guesses` and `real-price` that will result in an error or an infinite loop? (Assume there is at least one guess less than or equal to the real price).

> There were no situations where the procedure would infinitely loop, and no parameters within the stated domain for which is would generate an error. See the explanation above for the reason it won't loop (this was tricky). If you left this blank, you got no points. If you said something about non-numbers, or other arguments outside of the domain of `pir-winner`, you got full points.

**Problem 3. Last letter standing (A: 7 points; B: 6 points)**

Consider a procedure last-letter that takes a sentence and returns the last letter of the last word:

| | | |
|---|---|---|
| `(last-letter '(the quick brown fox))` | ➔ | `x` |
| `(last-letter '(the quick brown ""))` | ➔ | `error_empty_word` |
| `(last-letter '())` | ➔ | `error_empty_sent` |
| `(last-letter '(a))` | ➔ | `a` |

Note that when there is no last letter, either because the last word is empty or because there are no words, `last-letter` returns a word signaling an error. (And, this word is different depending on the error).

*Part A.* Write last-letter without using higher-order functions (i.e., using recursion)

- Do not use helper procedures.
- Do not use the procedures `last` or `butlast`.
- Use good names for the parameters to your procedures (we will be picky here).

```
;; recursive version
(define (last-letter
```

> You needed to use a two-stage recursion here – that is, a recursion which first recursed down the sentence (word by word), and then second recursed down the word (letter by letter). This is a similar process to `thoroughly-reversed` in lab. The constraint to not use helpers meant that you needed the same recursive procedure to do both stages of the recursion. For instance,
>
> ```
> (define (last-letter w-or-sent)
>   (if (sentence? w-or-sent)
>       (cond ((empty? w-or-sent) "error - empty sent")
>     ((empty? (bf w-or-sent))
>      (last-letter (first w-or-sent)))
>     (else (last-letter (bf w-or-sent))))
>       (cond ((empty? w-or-sent) "error-empty word")
>     ((empty? (bf w-or-sent))
>      w-or-sent)
>     (else (last-letter (bf w-or-sent))))))
> ```
>
> We were picky with name for the parameter: it needed to indicate that it could be either a word or a sentence.
>
> If your solution happened to use helpers or higher-order-procedures, you lost 3 points.

*Part B.* Fill in the framework below for a non-recursive solution to last-letter. (You can assume that, for this procedure, you are only given sentences that have at least two words, each with at least two letters).

- Do not use the procedures `last` or `butlast`.
- Use good names for the parameters to your procedures (we will be picky here).

```
;; HOF version
(define (last-letter sent)
  (accumulate
         (lambda (  left-ltr   right-ltr   )
               right-ltr )
         (accumulate
                 (lambda (  left-wd  right-wd   )
                      right-wd )
                 sent) ) )
```

> The inner-most accumulate will find the last word in the sentence and return it. Then, the outer most accumulate will find the last letter in that word and return that.
>
> Most people got the body of the `lambdas` correct, but many people lost points for not naming their placeholders with descriptive names – you needed to use different names for the inner and outer `lambdas`, and you needed to somehow indicate that the outer was for letters, while the inner was for words. That is the whole point of this "two-stage HOF".
>
> The placeholders for each were each worth one point (yes, we were picky). However, using `left` and `right` for the arguments of both `lambdas` lost only one point total, because, we reasoned, one of the names was semi-reasonable. This was graded differently during the exam (it lost 2 points there), so check with your TA if you need the grade changed.
>
> As an aside, `left` and `right` are not bad names *per se* for the arguments to an `accumulate lambda`, but it does depend on the context. For instance, in situations where the right argument "accumulates" the eventual return value—this is a *very* common situation—it is better to use `sofar` as the second parameter. When writing `last` using `accumulate`, `left` and `right` are fine paramter names, if the `accumulate` is standalone. When they are nested, as above, it makes a *lot* more sense to distinguish between the arguments of the `lambdas`.

**Problem 4. More snacking options (6 points)**

Consider the procedure `snack`, which calculates the number of different ways you might finish some number of chips and some (possibly different) number of drinks, eating one at a time.

```
(define (snack chips drinks)
  (if (or (= chips 0)
          (= drinks 0))
      1                                ;; base case
      (+ (snack (- chips 1) drinks)    ;; tree-recursive case
         (snack chips (- drinks 1)) ) ) )
```

| | | |
|---|---|---|
| `(snack 1 2)` | ➔ | 3 [ (chip, drink, drink), (drink, chip, drink), (drink, drink, chip) ] |

The logic of the tree recursion is, at any stage, there are two ways you can procede: either take a drink, or eat chips. Each way branches recusively, reducing the number available for the appropriate type of food. The base case is 1 because when you run out of either food type, there is only one way to eat what is left.

Fill in the framework below to correctly define `snack3`, which returns the number of ways you might finish chips, drinks, and gum (i.e., three types of food).

| | | |
|---|---|---|
| `(snack3 1 1 1)` | ➔ | 6 [ (chip, drink, gum), (chip, gum, drink), (drink, chip, gum),<br>(drink, gum, chip), (gum, chip, drink), (gum, drink, chip) ] |

Do not use the `snack` procedure in your solution.

The base and recursive cases for `snack3` were similar to those in `snack`, but each contained a main difference. First, here is a solution:

```
(define (snack3 c d g)
  (if
      (= (appearances 0 (se c d g)) 2)

      1

      (+ (if (> c 0)
             (snack3 (- c 1) d g)
             0)
         (if (> d 0)
             (snack3 c (- d 1) g)
             0)
         (if (> g 0)
             (snack3 c d (- g 1))
             0) )))
```

For the base case, you needed to stop (and therefore return 1) when two out of the three arguments were `0` (there are many ways to test for this; using `appearances` wasn't particularly common). This should make sense if you think about it: one you have used up one of the three kinds of food, you still have to figure out the number of ways to use up the other two types recursively – certainly, there is more than one way). Many of you got this right; it was worth 3 points.

The simplest extension of `snack`, conversely, tested whether one of the three was zero (i.e., by simply adding the additional case `(= gum 0)` to the `or` expression in `snack`. This was worth only 1 point.

The second portion to the problem involved the recursive case. Many of you simply extended the recursive case in `snack`: i.e., by making three calls to `snack3` within a + expression, each of which diminished one of the arguments by `1`). However, this clearly wouldn't work when one of the arguments was `0`: a call to `snack3` would then be passed a negative number! The solution above used if expressions to only recurse when the relevant argument was non-zero. There are several different ways to write this logic, but few of you got it correct. This section was worth 3 points; simply extending `snack` into `snack3` was worth 1 point.