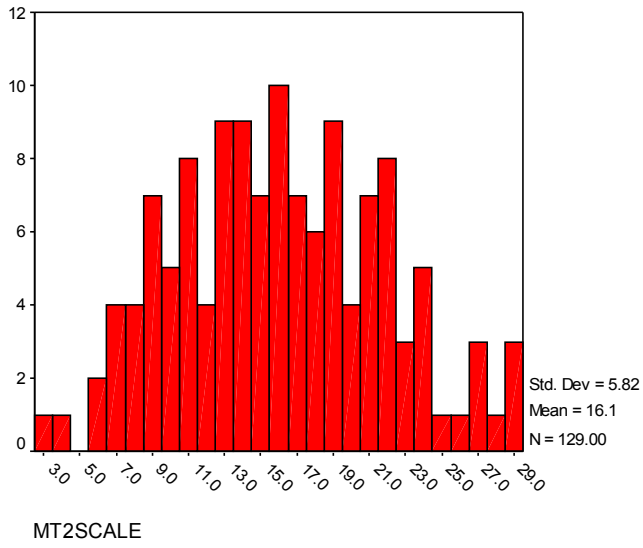
CS 3 Midterm 2 Standards and Solutions

Fall 2006

This exam was probably more difficult that I would have liked, although it seemed to go reasonably well for many of you.. The average score was right around 50%, and people were spread evenly on either side of that average.



Problem 1. One and only one letter different... (4 points)

The predicate `one-ltr-different?` takes two words, and returns `#t` if one and only one of the letters in ~~the first~~ one word is different from the letter at the corresponding position in the ~~second~~ other word. Otherwise, `one-ltr-different?` should return `#f`.

`(one-ltr-different? 'abcde 'abxde)`	➔	`#t`
`(one-ltr-different? 'abcde 'abxxe)`	➔	`#f`
`(one-ltr-different? 'abcde 'abcde)`	➔	`#f`
`(one-ltr-different? 'abcde 'bacde)`	➔	`#f`
`(one-ltr-different? 'abcde 'abxcde)`	➔	`#f`

Below is a buggy version of `one-ltr-different?`. Describe (precisely) what the inputs are that will return an incorrect value for this version.

```
(define (one-ltr-different? wd1 wd2)
  (cond ((and (empty? wd1) (empty? wd2))
         #t)
        ((or (empty? wd1) (empty? wd2))
         #f)
        ((not (equal? (first wd1) (first wd2)))
         (equal? (bf wd1) (bf wd2)))
        (else (one-ltr-different? (bf wd1) (bf wd2)))
        ))
```

The problem with this recursion is in the first base case, where both sentences are checked if empty. The problem has a single recursive case, which is called when the first letters of each word are equal. So, this first base case will return `#t` any time that the words are identical, which is clearly wrong. This base case should return `#f`, or even better should be removed, so that the second base case can check for empty sentences.

While many of you got this correct, there were just as many that were fooled in a specific way: that this procedure would incorrectly count as true a words that had two different letters sandwiching a identical letter. This confusion stemmed from the third base case:

```
(cond...

 ((not (equal? (first wd1) (first wd2)))    ;; base case 3
  (equal? (bf wd1) (bf wd2)))
```

This base case will work correctly: it tests for whether the first letters of each word are different, and then checks whether the rest of each word are equal. This is a base case, solveable without recursion. Many of you seemed to think that it was testing whether the second letter of each word was equal, and ignoring the rest of the sentence.

This question was vauge on what should happen if the words were of unequal length. An example case showed a return value of `#f` (which is what was intended, necessitating the first base case), but the wording describing the function of `one-ltr-different?` wasn't clear. If you mentioned a case of unequal word lengths but failed to mention the incorrect return value when the words were the same, you got partial credit.

Problem 2. From the beginning: a bunch of "easy" HOFs. (9 points).

In the following problems, you need to use HOFs to write procedures that mimic the functionality of `first`, `last`, `butfirst`, `butlast`, and `appearances`. You

- cannot use recursion or helper procedures
- cannot use any of `first`, `last`, `butfirst`, `butlast`, or `appearances`;
- cannot use the procedure `item`; and
- cannot go outside of the framework code given.

If you think a solution is impossible, you very well may be right: some of these are impossible! Write `IMPOSSIBLE` next to the problem to indicate that.

Don't get stuck on this problem: there is plenty more test after this...

> These proved fairly difficult for many of you, although in most cases the solution weren't very tricky. This question was intended to get at the basic operation of accumulate and keep.

```
;; Mimic first with the following procedure, if possible
(define (my-first sent)
  (accumulate
    (lambda (__left____right___)

        _____________left_____________________________________)
    sent))


;; Mimic last with the following procedure, if possible.
(define (my-last sent)
  (accumulate
    (lambda (__left____right___)

        _____________right____________________________________)
    sent))


;; Mimic butfirst with the following procedure, if possible
(define (my-butfirst sent)   ;; mimicking butfirst
  (accumulate
    (lambda (__left____right___)
```

> We accepted answers of `IMPOSSIBLE` here, since that is what we though was right before we saw some of your (perfectly possible) answers! Good job! We gave this solution an extra credit point, to assuage our embarrassment.

```
            (if (equal? (se left right) sent)
                right                ;; last time, lose the 'first' element
                (se left right))     ;; not the last time, keep the sent
        ____________________________________________________________)
    sent))


;; Mimic butlast with the following procedure, if possible.
(define (my-butlast sent)
  (accumulate
    (lambda (__left____right___)

            (if (word? right)
                (se left)            ;; first time, lose 'right'
                (se left right))     ;; not the first time, keep everything
        ____________________________________________________________)
    sent))
```

```
;; Mimic butfirst with the following procedure, if possible
(define (my-butfirst sent)
  (keep
    (lambda (___________________)

        ____________________IMPOSSIBLE_____________________________)
    sent))


;; Mimic butlast with the following procedure, if possible.
(define (my-butlast sent)
  (keep
    (lambda (___________________)

        ____________________IMPOSSIBLE_____________________________)
    sent))
```

`my-appearances` was a standard use of keep:

```
;; Mimic appearances with the following procedure, if possible.
(define (my-appearances item sent)

  (___count____________
     (keep
       (lambda (_____wd________)

           ______(equal item wd)_____________________________________)
       sent) ))
```

Problem 3. Recursion is like decoding words... (8 points)

Write a recursive procedure `decode-word` which takes a sentence of encoded letters and returns the word that they form when decoded. Decoding is done with a sentence `*codes*` which contains 26 codes, one for each alphabetical letter in order. For instance:

```
(define *codes*  '(12 3 4 1 2 14 ...twenty more codes here...)
(decode-word '(4 12 3)) ➔ cab
(decode-word '(14 12 1 2)) ➔ fade
```

The sentence `'(a b c d e f …)` is likely to prove useful in your solution. You can assume that all encoded letters given as input to `decode-word` exist inside `*codes*`. Do not use any higher order functions—rather, use recursion.

```
(define (decode-word coded-sent)
```

This question involved two separate recursions, and it was easiest to separate these into two different procedures. `decode-word` is quite easy to write when you assume that there is a procedure `decode-letter`:

```
(define (decode-word coded-sent)
   (if (empty? coded-sent)
       ""
       (word (decode-letter (first coded-sent))
             (decode-word (bf coded-sent)))))
```

The only real "trick" here was to realize that you were building a word from the sentence.

The `decode-letter` portion of this problem becomes more clear when you have specified a good procedure name, the arguments that it takes, and what it returns. In this case a it takes single coded letter (a number), and returns a letter. This procedure will typically involve recursion because it look this letter up in the `*codes*` sentence. So, the procedure will need to have local copy of the code so that it can pass the butfirst of the codes in the recursive call (i.e., so it can recurse down the sentence). This means making the codes sentence a parameter (and, as you will see, the sentence of the alphabet). We could change the call to `decode-letter` inside `decode-word`, or better use a `decode-letter-helper` procedure:

```
(define (decode-letter coded-ltr)          ;; version 1
    (dl-help coded-ltr
             *codes*
             '(a b c d e f g h i j k l m n o p q r s t u v w x y z)))

(define (dl-help coded-ltr codes alphabet)
   (if (equal? coded-ltr (first codes))
       (first alphabet)
       (dl-help coded-ltr (bf codes) (bf alphabet))))
```

This version recurses down both the codes and alphabet sentences at the same time, and when it has found the `coded-ltr` in the `codes` sentence, the proper return value is right there in the `alphabet` sentence.

There are other ways to write dl-help. For instance, using `position` and `item` (and no recursion):

```
(define (decode-word coded-ltr)          ;; version 2: position and item
    (item (+ 1 (position coded-ltr *codes*))
          '(a b c d e f g h i j k l m n o p q r s t u v w x y z)))
```

You need to add `1` to what `position` returns, if you recall, to use it in `item`. And, there are plenty of other solutions. For instance, some of you essentially wrote `position` recursively, and passed what it returned a call to `item`.

Problem 4. Moving in the second dimension (A: 8 points, B: 6 , C: 9 points)

Consider a higher order procedure `every2d` which takes a procedure and *two* sentences. The procedure must take two arguments, and `every2d` will build a sentence by calling the procedure with the corresponding values of each of the input sentences.

If the two sentences are of unequal length, `every2d` should return the word `UNEQUAL`.

`(every2d + '(1 2 3) '(100 200 300))`	➔	`(101 202 303)`
`(every2d word '(cs i fu) '(3 s n))`	➔	`(cs3 is fun)`
`(every2d word '(a long long sent) '(short))`	➔	`UNEQUAL`

Part A. Below is code for `every2d`, which consists of a single call to `e2d-helper`. Use the provided header to `e2d-helper`, and fill in the body as an accumulating recursion so that `every2d` works correctly.

Do not write any additional helper procedures.

```
(define (every2d proc sent1 sent2)
   (e2d-helper  proc  sent1  sent2 '() )
   )


(define (e2d-helper proc sent1 sent2 answer)
```

A reasonable solution looked like

```
 (define (e2d-helper proc sent1 sent2 answer)
    (cond ((and (empty? sent1) (empty? sent2))
           answer)
          ((or (empty? sent1) (empty? sent2))
           'UNEQUAL)
          (else (e2d-helper proc (bf sent1) (bf sent2)
                            (se (proc (first sent1) (first sent2))
                                answer) )) ))
```

By using an accumulating recursion, which builds up the final results in the `answer` variable, the procedure can return wait until one or both of the sentences are of equal length after running through the full recursion, because either `answer` or the word `UNEQUAL` can simply be returned. An embedded recursion wouldn't be able to do this. Several of you got around this by changing the second base case above to something like:

```
          ((not (equal? (count sent1) (count sent2)))
           'UNEQUAL)
```

This works well enough, but is pretty "gross" because it checks the count of the sentence through each step of the recursion, even though it can only be triggered the first time through. Still, it received full credit if it was used with an accumulating recursion, rather than an embedded one (the embedded one would work here because the base case would succeed the before any combining of solutions in the recursive case had happened).

Part B. Below is a possibly buggy version of `prefix-values-removed`, from the "Roman Numerals" case study, written using `every2d`.

Recall, `prefix-values-removed` takes a `number-sent` with possible prefixes, and returns a sentence of numbers that when summed will be the arabic representation of the roman numeral in question. (The Roman Numerals case study code is in Appendix A).

Be sure to notice that extra zeros are added to the sentences that `every2d` is passed.

```
;; takes a sentence of digit values, possibly containing prefixes
(define (prefix-values-removed number-sent)
   (every2d (lambda (shifted orig)
              (if (< shifted orig)
                  (- orig shifted shifted)
                  orig)
              )
            (se 0 number-sent)
            (se number-sent 0)
            ))
```

Provide three *good* test cases involving prefixes. Include the return value, and comment on whether the return value is correct (i.e., correct in that the Roman Numerals code will work correctly when it uses this version of `prefix-values-removed`).

Your test cases should test as wide a range of different (possibly problematic) conditions as possible.

> This version of `prefix-values-removed` works correctly in all cases, although it returns a different value from the `prefix-values-removed` given in the case study. For instance, the `decimal-value` for the roman numeral 1946 would be
>
> `(1000 100 1000 10 50 5 1)`
>
> The original `prefix-values-removed` in the case study would change the sentence to
>
> `(1000 900 40 5 1)`
>
> This `prefix-values-removed` would return the sentence
>
> `(1000 100 800 10 30 5 1 0)`
>
> and this is at the crux of this problem: being able to see how how the above code turns the first sentence of `digit-values` into this later. The easiest way to see this is to line up the two sentences that are passed to `every2d`, in which the elements are shifted by adding a zero in front of one of the sentences.
>
> ```
> shifted: (   0   1000   100   1000  10   50   5   1 )
>    orig: ( 1000   100  1000    10   50    5   1   0 )
> ```
>
> The procedure passed to every2d checks each element of the sentences, and either returns the element from the "original" sentence or returns the original with two times the "shifted" subtracted from it, based on whether the shifted element is smaller than the original (i.e., whether the shifted element represents a prefix). In this way, the procedure for every2d can compare consecutive elements of the sentence.
>
> Note that the code thinks that the very first element in the original sentence is a number that had a prefix before it, which is impossible. But, because it subtracts zero twice, the value is unchanged.

Cute, huh? Both versions lof `prefix-value-removed` return sentences that will sum to the same number, which is how "correctness" is defined.

Test cases were to involve prefixes. The best conditions to test are a `decimal-value` sentence with a prefix at the front (this was a source of problems in the case study), a prefix at the end, and a prefix only in the middle. This procedure worked correctly on all of these sentences:

Roman numeral	Arabic value	decimal-values (the input)	output
`CMXXII`	`922`	`(100 1000 10 10 1 1)`	`(100 800 10 10 1 1 0)`
`XIV`	`14`	`(10 1 5)`	`(10 1 3 0)`
`MCMXI`	`1911`	`(1000 100 1000 10 1)`	`(1000 100 800 10 1 0)`

Note that many of you tested prefix-value-removed with invalid inputs. That is, inputs that didn't come from valid roman numerals—sentences with multiple prefixes in a row, etc. It is very hard to judge "correctness" when the input isn't valid!

Part C. Write `decode-word` (described in problem 3) without using explicit recursion; rather, use only higher order procedures. *Do* use the HOF `every2d` (you can assume that you have a properly working version).

Again, the sentence `'(a b c d e f g ...)` is likely to prove useful.

This was a hard problem. Since this is the same procedure as in question 3, understanding the concepts used there potentially helped you a lot in this case. Again, there should be two different procedures: `decode-word` and `decode-letter`. These involve processing elements in a sentence without concern for the other elements of the list: perfect for `every` and `every2d`!

```
(define (decode-word coded-sent)                ;; HOF
   (accumulate word
               (every decode-letter coded-sent)))
```

Again, the only real trick here was turning the sentence that `every` returns into a word as specified in the problem statement. Also, if your `decode-letter` procedure needed other arguments, you would have had to use a `lambda` statement.

`decode-letter` seems oh-so-perfect for `every2d`: each element in two sentences (`*codes*` and the alphabet sentence) needs to be looked at in order, checking for equality with a parameter (`coded-ltr`). The only problem is `every2d` is going to return something for each of the 26 elements in the two lists, and we only want one thing returned. A `keep` seems more appropriate, but we don't have a `keep2d` procedure.

Two solutions are most obvious:(1) using a `keep` outside of an `every2d`, to get rid of the extra things `every2d` returned (2) making the `every2d` both transform the element we want to keep and remove the elements we don't. First solution first:

```
(define (decode-letter coded-ltr)          ;; HOF version one
```

```
   (keep (lambda (wd)
           (not (number? wd)))
         (every2d (lambda (code alpha-ltr)
                    (if (equal? coded-ltr code)
                        alpha-ltr
                        1))                    ;; keep will remove this
                  *codes*
                  '(a b c d e f ...)
                  )))
```

In version two, the `every2d` procedure returns the empty list `'()` when it doesn't want to keep an element, and this will be 'disappeared' when `every2d` uses `se` to combine the return values.

```
(define (decode-letter coded-ltr)          ;; HOF version two
   (every2d (lambda (code alpha-ltr)
              (if (equal? coded-ltr code)
                  alpha-ltr
                  '()))                    ;; this will disappear when every2d
                                           ;; combines the return values up
            *codes*
            '(a b c d e f ...)
             ))
```

Many students used `every2d` in another way, namely to combine `*codes*` and the alphabet sentence into one sentence. For instance, `(every2d word *codes* '(a b c d e f ...))` will return a sentence like `'( 12a 3b 4c 1d 2e 14f ...)`. This single sentence can then processed as above just using `every` (where the `lambda` needs to take apart the words that `every2d` created).

```
(define (decode-letter coded-ltr)
   (every (lambda (letter-code)
            (if (equal? coded-ltr (bf letter-code))
                (first letter-code)
                '()))
          (every2d word *code* '(a b c d e f ...))))
```

We saw some solutions which didn't make use of `every2d`, but instead wrote `decode-letter` using `item` and `position`. The intent of the problem was to use higher order functions, namely `every2d`. Some of you, however, still managed to write non-recursive (and impressive) versions of `position` using `accumulate`.

Problem 5. Counting descendants. (A: 2, B: 6 points)

Consider the procedure `num-descendants` that returns the number of descendants a fish will make within a certain number of generations. The procedure takes the number of generations as its argument.

A descendant is a direct child or a descendant's child. The number of descendants in one generation would be the number of direct children made, the number in two generations would be the children and grandchildren, and so on. Do not count the original fish in this number!

This particular species of fish will make three children each year that it is between 2 years old and 6 years old, inclusive. The fish is too young to make children before that, and dies at the beginning of its 7th year. Therefore, `(num-descendants 1)` should return `15`.

The following is a buggy version of `get-descendants`:

```
;; buggy version
(define (num-descendants gen)
  (nd-help gen 0) )

(define (nd-help gen age)
  (cond ((= gen 0) 0)
        ((< age 2)
         (nd-help gen (+ age 1)))
        ((< age 6)
         (+ (nd-help (- gen 1) 0)
            (nd-help (- gen 1) 0)
            (nd-help (- gen 1) 0)
            ))
        ((> age 6) 1)
       ))
```

Part A: What will the result be, using the buggy version above, of `(num-descendants 1)`?

> The answer here is `0`. The `nd-help` procedure recurses twice based on the condition `(< age 2)`, increasing `age` each time. Then, the recursive case conditioned with `(< age 6)` spawns three new recursive calls, each with `gen` set to zero. Each of these calls immediately hits the first base case, and return `0`. Understanding this process was crucial to answering part B correctly.

Part B. Fix the buggy version of `num-descendants` so that it works correctly.

Your solution should follow the general strategy of the buggy version—using tree recursion—rather than using a mathematical approach with the procedures `*`, `expt`, and so forth.

> In this recursion, you are tasked with summing the number of fish that are created during the program execution. The basic strategy to this program is that recursive calls are made each time a fish lives for a year, and the call knows what age and what generation (down from the original fish) the fish is. So, recursive calls are made for two reasons:
>
> - when the fish grows a year older (and doesn't die): generation stays the same, and age is one greater.
> - when the fish creates a new fish (i.e., when a fish is born). The recursive call is for the new fish, with generation one less and age of zero).
>
> In addition to knowing what the recursive calls mean, you also need to determine how to return numbers in order to create the proper summation (and overall return value). In `pascal`, numbers were added in the base cases, which is possible here. Numbers can also be added in some recursive cases (e.g., see the solution to problem 5 (`pascal-calls`) in the second midterm of fall 2004. There two basic approaches to fixing this procedure, and they differ in where they add numbers.

One method is to count fish when they die. So, we will need to add 1 when the age is greater than 6, and add 1 when the generation is equal to 0 (because we don't want the fish to age in the normal progression here, since this is the maximum generation away that we will count). But, with these two additions, we've over counted by 1, because we've counted the original fish. An easy place to correct for that is in num-descendants, rather than in the tree-recursive procedure. (Notice also that we needed to correct the tests for the conditions from what was written in the buggy procedure).

```
;; method of counting fish at death
(define (num-descendants gen)
   (- (nd-help gen 0) 1))   ; subtract one so we don't count the original fish

(define (nd-help gen age)
  (cond ((= gen 0)                  ; this fish is the maximum generation
         1)                         ; away, count it now and no births allowed
        ((< age 2)
         (nd-help gen (+ age 1)))      ; too young to spawn, one year older
        ((<= age 6)
         (+ (nd-help (- gen 1) 0)      ; birth 1
            (nd-help (- gen 1) 0)      ; birth 2
            (nd-help (- gen 1) 0)      ; birth 3
            (nd-help gen (+ age 1))))  ; same fish, one year older
        ((> age 6)
         1)                            ;fish dies
        ))
```

The second method for counting fish is at birth. In this method, we want the base case for death and maximum generation to be zero. To count births, we simply add 3 when the original fish spawns:

```
;; method of counting fish at birth
(define (num-descendants gen)
  (nd-help gen 0) )

(define (nd-help gen age)
  (cond ((= gen 0) 0)
        ((< age 2)
         (nd-help gen (+ age 1)))
        ((<= age 6)
         (+ (nd-help (- gen 1) 0)
            (nd-help (- gen 1) 0)
            (nd-help (- gen 1) 0)
            3                           ;; count the births
            (nd-help gen (+ age 1))))
        ((> age 6) 0) ))
```

With this solution, there is no need to correct the amount in `num-descendants`: since we are counting births, we won't count when the original fish is born, only counting that fish's descendants. Easy!