CS 3 Midterm 2
Spring 06
Standards and Solutions

Problem (8 points) Tail recursion.

Write a tail-recursive definition for `n-matches`, which returns up to the first `n` matches between two sentences. A "match" occurs when the same word is at the same position in both sentences:

`(n-matches '(a b c d) '(a b c d) 3)`	➔	`(a b c)`
`(n-matches '(a x c) '(a b c d) 6)`	➔	`(a c)`
`(n-matches '(c b a) '(a b c) 3)`	➔	`(b)`
`(n-matches '(a b c) '(x y z) 2)`	➔	`()`

The numeric argument to `n-matches` will be 0 or greater. (Note that you will get partial credit if you write a non-tail-recursive solution to the problem.)

The problem posed here was relatively standard, although a few of you failed to understand the problem statement. A large number of you, however, had trouble knowing (and acting on) the difference between a tail and embedded recursion. Remember, in a tail recursion, there are no pending procedures to combine the return values from the recursive calls. Looking at the return values from the example calls above, this problem will be using `se` to combine partial answers to the problem. A tail solution, however, cannot use `se` to combine answers from recursive calls, because that would be a pending procedure! Rather, the combiner should be used to buildup a sentence that is passed as an argument to the recursive call. This extra argument is often present in tail recursive solutions; we call these accumulating recursions.

Another way to understand tail recursions, which might be easier: in tail recursive solutions, the base case(s) must return the full and complete answer to the initial call. (This is a direct consequence of the "no combining recursive return values" constraint). 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.

One complication in this problem is that the recursion already had an argument —the one that kept track of the number of matches—that functioned like those in accumulating recursions. This argument was not sufficient to return the complete answer and make the solution a tail one, however.

Here is a tail solution:

```
(define (n-matches s1 s2 n)
   (n-matches-helper s1 s2 n '()))

(define (n-matches-helper s1 s2 n current)
   (cond ((or (empty? s1)    ;; base cases
              (empty? s2)
              (<= n 0))
          current)           ;; return the accumulated answer
         ((equal? (first s1) (first s2))
         ;; there is a match, so accumulate into current
```

```
              (n-matches (bf s1)
                         (bf s2)
                         (- n 1)
                         (se cur (first s1)) ))
             (else
              ;; no match, keep current the same
              (n-matches (bf s1) (bf s2) n cur))))
```

If you provided an embedded solution to this problem, you could earn up to 6 points (depending on any errors you made in your code). Here is an embedded solution:

```
(define (n-matches s1 s2 n)
   (cond ((or (empty? s1)
              (empty? s2)
              (<= n 0))
          '())
         ((equal? (first s1) (first s2))
          (se (first s1)
              (n-matches (bf s1) (bf s2) (- n 1))))
         (else
          (n-matches (bf s1) (bf s2) n))))
```

Problem (9 points) Higher-order election research

Write a higher-order procedure named `electoral-votes` which takes a predicate as its single argument. The procedure will sum up the 2008 electoral votes for states that satisfy the predicate.

`(electoral-votes california?)`	→	55
`(electoral-votes blue-state?)`	→	212
`(electoral-votes voted-for-bush-in-2004?)`	→	286

The database of states and their electoral votes is in a global variable `*states*`, with the same format as in your third miniproject:

```
(ca55 me04 nj15 …)
```

The predicate takes the state's two-letter abbreviated name as its argument. You do not have to write these predicates; rather, you only need to write `electoral-votes` such that it works properly with any proper predicate.

Do not use any explicit recursion in your solution. Make sure you define and use accessor procedures where appropriate; you will lose some points if you don't.

This question proved trickier than we thought it would. The third miniproject required several procedures of very similar form to this solution. One possible confusion was the way that your solution used an unknown predicate, although this is a concept that is most definitely part of this course. The procedure `electoral-votes` that you were to write is a higher order function is the same way that `keep`, `every`, and `accumulate` are: it takes a procedure as one of its arguments.

The use of a global variable, rather than an extra argument, was an extra wrinkle that seemed to trip up only few of you. This had very little impact on the definition of your solution.

For proper data abstraction, accessors to pull apart the name and number of electoral votes for a given state are necessary. (By using these, our solution code wouldn't be dependent on the format of the state database. And, now that you know about general lists, you can understand how other formats might be better).

```
(define (state-ev state) (bf (bf state))
(define (state-name state) (bl (bl state))
```

Here is a solution using the above accessors:

```
(define (electoral-votes pred)
   (accumulate +
       (every state-ev
             (keep (lambda (state)
                         (pred (state-name state))
                   *states*))))
```

Note that a lambda is necessary for the keep—several of missed this because you didn't realize that the predicate required the state's name, rather than a combination of the name and number of votes.

The specific point breakdown included:
- 3 pt for all the accessors (this included using good names!)
- 1 pt for accumulate usages
- 1 pt for every usage
- 1 pt for keep usage
- 3 pts for constructing the lambda with pred
- 1 pt for using ***states*** variable correctly

Problem (8 points) A unique bug

Write the predicate `all-unique?`, which takes a sentence of any length and returns
- `#f`, if two or more words in the sentence are identical
- `#t`, otherwise

Do not use any explicit recursion. (That is, use higher order functions in your solution). Make sure to think about the full range of inputs.

Also, do not use the predefined procedures `dupls-removed` or `appearances`.

This question was designed to require an `accumulate`, which most of you attempted. (Although, see further below). Remember, when elements in the sentence need to be compared with a HOF, `accumulate` is the best, if not only, way to go. The trick here was that `accumulate` couldn't directly return the answer, because accumulate can't return a Boolean (which isn't a word or a sentence). But, through something like a comparison with the original sentence, `accumulate` made it easy to determine the answer.

```
(define (all-unique? sent)
  (or (empty? sent)
      (empty? (bf sent))
      (equal? sent
          (accumulate (lambda (cur sofar)
                        (if (member? cur (if (word? sofar)
                                             (se sofar)
                                             sofar))
                           sofar
                           (se cur sofar)))
                      sent))))
```

Here are some other non-`accumulate` solutions; although some of you tried to write solutions along these lines, none of you got it exactly right. These are tricky! Here is a solution using the logic of remove duplicates, using the semi-predicate member:

```
(define (all-unique? sent)
   (equal?
      0
      (- (count sent)
         (count (keep (lambda (wd)
                         (not (member? wd (bf (member wd
sent)))))
                      sent)))))
```

Here is an even more tricky solution using the logic of appearances. Note the nested `keep`'s!

```
(define (all-unique? sent)
  (empty? (keep (lambda (wd)
                  (not (equal? (count (keep (lambda (wd1)
                                                 (equal? wd
wd1))
                                              sent))
                               1)))
                sent)))
```

Problem (A: 7, B: 8 points). A troubling occurrence…

Recall the `occurs-in?` homework assignment from earlier in the semester. This predicate took two words as arguments, and returned `#t` if the first argument was contained somewhere in the second.

This question concerns a *semi-predicate* version of `occurs-in?`, named `occurs-in` (that is, without the "`?`"). A semi-predicate returns either `#f` or some scheme value that evaluates to true. In this case, when true `occurs-in` returns the portion of the second argument after and including the first argument:

`(occurs-in 'ab 'xabcab)`	➔	`abcab`
`(occurs-in 'ab 'nothere)`	➔	`#f`
`(occurs-in "" 'anything)`	➔	`anything`
`(occurs-in 'bb 'abbbc)`	➔	`bbbc`

Part A. Use the `occurs-in` semi-predicate to write a definition for `num-occurrences`, which takes the same arguments as `occurs-in` and returns the number of times that the first argument occurs within the second argument. (You can assume that the first argument will not be empty).

`(num-occurrences 'ab 'xabcab)`	➔	`2`
`(num-occurrences 'ab 'nothere)`	➔	`0`
`(occurs-in "" 'anything)`	➔	*not defined*
`(num-occurrences 'bb 'abbbc)`	➔	`2`

Make sure that you make <u>as few calls to `occurs-in` as possible</u> within your code. (As rationale, you could assume that `occurs-in` is a very slow procedure, and that if you call it too many times it will slow down your implementation of `num-occurrences` unacceptably. Do not rewrite portions of `occurs-in` to avoid using it, though!)

For this problem, you needed to understand what a semi-predicate was and be able to use it correctly. Most of you did this reasonably well. You also needed to know how to bind variables using `let` to get full credit on this problem, given the constraint that you should make "as few calls to occurs-in as possible".

```
(define (num-occurrences sub full)
  (let ((occur (occurs-in sub full)))
     (if occur
        (+ 1 (num-occurrences sub (bf occur)))
        0)))
```

The specific point breakdown included:

- -2 if more than 1 call to occurs-in was made (should have used let)
- -2 if the argument to num-occurrences in the recursive case was not correct. Many students made a helper procedure to remove the first N letters from the second argument, where N is the length of the first argument, but this leads to incorrect results for `(num-occurrences 'bb 'abbc)`.
- -1 each for bad argument names and incorrect `let` bindings (mismatched parenthesis)
- -1 for incorrect base case

Part B: Your sometimes brilliant, sometimes completely-lost partner has written the following definition of the semi-predicate `occurs-in` by using the full predicate `occurs-in?` :

```
(define (occurs-in sub full)
   (if (occurs-in? sub full)
      (occurs-in-help sub (bf full) full)
      #f))

(define (occurs-in-help sub full prev-full)
   (if (occurs-in? sub full)
      (occurs-in-help sub (bf full) full)
      prev-full))
```

Briefly explain under which sets of inputs this procedure will:

1) return the correct value for `occurs-in`
2) return an incorrect value for `occurs-in`
3) crash or recurse infinitely (and, therefore, return no value).

Not all of these results may be possible.

A succinct way to describe the following sets is:
1) This will be correct only if there are 0 or 1 occurrences of `sub` in `full`
2) This will be incorrect if there are 2 or more occurrences of `sub` in `full`.
3) This will crash only if `sub` is empty.

By far the most common mistake was to forget the third case, although there was an example of such in Part A.

Problem (A: 6, B: 8 points) Its for the kids

Kindergarteners spend a lot of time learning words of the form *consonant-vowel-consonant*, such as "cat" and "dog". This question will involve the procedure `make-kindergarten-words` (abbreviated `make-kw`), which takes a sentence of consonants and a sentence of vowels, and returns a sentence of **all** the possible kindergarten words that can be made using those letters. The ordering of the result sentence doesn't matter.

`(make-kw '(s t) '(a o))`	➔	`(sas sat sos sot tas tat tos tot)`
`(make-kw '() '(a o))`	➔	`()`
`(make-kw '(l n k t s) '(a e i o u))`	➔	`... A really long sentence (225 words)!`

Part A. Fill in the box below to complete the solution for `make-kw`. Do not use any additional helper procedures.

```
(define (make-kw consonants vowels)
   (every (lambda (c)
             (every (lambda (v)
                        (every (lambda (c2)
                                     (word c v c2))
                            consonants)

                    )
```

```
               vowels))
   consonants))
```

This proved to be a tricky problem for most of you, which wasn't a surprise to us. We graded it quite leniently because, while it is something that you should have been able to figure out given what you know (and, given enough time), it isn't something that you had seen explicitly before.

When an `every` (with, say a sentence `s1` as its argument) is embedded within another `every` (with an argument of `s2`), the inner procedure will be called on every possible pair of elements of `s1` and `s2`. Take some time to understand this; stepping through each call (in your head) of the outer `every` is a good way to start.

Part B. Complete the following recursive solution for `make-kw`, filling in the bodies for recursive cases 1-3 Don't use any higher order procedures.

```
(define (make-kw consonants vowels)
   (make-kw-help1 consonants vowels consonants))


(define (make-kw-help1 front vowel end)
   (cond ((or (empty? front)                          ;; base cases
              (empty? vowel)
              (empty? end))
          '())


         ((and (word? front) (word? vowel) (word? end))  ;; base case 2
                                                      ;;  (error
                                                      ;;   corrected)
          (word front vowel end))
          )


         ((and (word? front) (word? vowel))           ;; recursive
                                                      ;; case 2
          (se (make-kw-help1 front vowel (first end))
              (make-kw-help1 front vowel (bf end))))
           )


         ((word? front)                               ;; recursive
                                                      ;; case 3
          (se (make-kw-help1 front (first vowel) end)
              (make-kw-help1 front (bf vowel) end)))
          )


         (else                                        ;; recursive
          (se (make-kw-help1 (first front) vowel end)    ;; case 4
              (make-kw-help1 (bf front) vowel end))   ;;  (error
          )                                           ;;   corrected)
))
```

We hoped that this part of problem 5 would be easier with the thinking that you had done in *Part A*, and visa versa, and we might even have been right. You have seen recursions of this type—where arguments are sometimes words and sometimes sentences—in procedures like `thoroughly-reversed`. This is a form of tree recursion.

Many of you commented that the last case, labeled "recursive case 4", helped quite a bit; reading that carefully made the general structure of the solution clear. (And, the typo here was quite unfortunate; our apologies).

- 2 pts first blank
- 3 pts. second blank
- 3 pts. third blank