# CS3 Midterm2 Review Problems (Spring 2005)

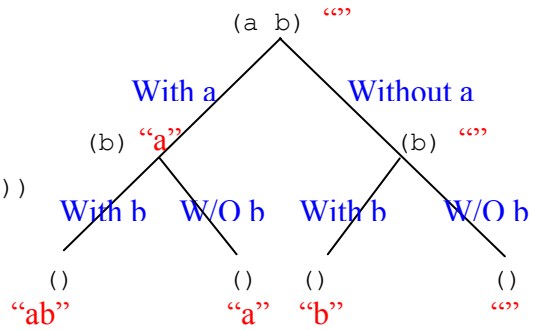
## More complicated recursion

1. **Produce a function to find the length of the longest decreasing substring in a sentence of positive numbers. For example:**
   **(2 1 5 3 2 1 12) → 4**
   **(13 12 11 15 10 5 2 1) → 5**

```
(define (longest length_sofar max_length last sent)
   (cond ((empty? sent) (max max_length length_sofar))
         ((> last (first sent))
          (longest (+ 1 length_sofar) max_length (first sent) (bf sent)))
         (else (longest 1
                        (max max_length length_sofar)
                        (first sent)
                        (bf sent))) ))
```

2. **Write a procedure that returns all words representing subsets of the letters in the input sentence. Assume that the input sentence is not empty and includes distinct letters. Note that the order does not matter in out put subsets so `ab` is considered the same subset as `ba`. Example:**
   **(a b c) → ("", a, b, c, ab, ac, bc, abc)**
   **(w e) → ("", w, e, we)**

```
(define (all_subsets subset_sofar sent)
   (if (empty? sent)
       subset_sofar
       (se (all_subsets
              (word subset_sofar (first sent))
              (bf sent))
           (all_subsets
              subset_sofar
              (bf sent))) ))
```




3. **I have c chips and d drinks, how many ways can I finish all these snack if I ate one at a time? Example: `(snack 1 2)` → 3, which includes (chip, drink, drink), (drink, chip, drink), and (drink, drink, chip).**

```
(define (snack chips drinks)
   (if (or (= chips 0) (= drinks 0))
       1
       (+ (snack (- chips 1) drinks)
          (snack chips (- drinks 1)))))
```

   Trust the recursion! To get to `(snack 3 4)`, you would either have `(snack 2 4)` and then one more bag of chips, or `(snack 3 3)` and then one more drink. So `(snack 3 4)` is just the sum of those two possibilities.

4. **Given a sentence of words, I want to keep only the words that also have their reversed counterparts in the same sentence, and I want these words to be returned in increasing order of their length. If two words have the same length, then return them in the same order as the original sentence. For example, the argument `'(abc gfed mlkj hi ih defg mlkj mlkj jklm)` should**

**return `(hi ih gfed mlkj defg mlkj jklm)`. You may assume the procedure `reverse` is already written for you: `(reverse 'abc)` → `cba`**

```
(define (word-reverse-sent sent)
   (wrs-helper sent sent '()))

(define (wrs-helper sent reduced so-far)
   (cond ((empty? reduced) so-far)
         ((member? (reverse (first reduced)) sent)
          (wrs-helper sent (bf reduced) (sort-length (first reduced) so-
far)))
         (else (wrs-helper sent (bf reduced) so-far))))

(define (sort-length wd sent)
   (cond ((empty? sent) (se wd))
         ((< (count wd) (count (first sent))) (se wd sent))
         (else (se (first sent) (sort-length wd (bf sent))))))
```

For each word we need to check if its reversed counterpart is inside the original sentence. So we use sent to keep track of that, and we keep on shrinking `reduced` to go through each word in the sentence. `so-far` holds the sorted value that we have gathered so far. Here `sort-length` is a helper used to sort the words in a sentence by their length. This works like `insert`: we can always assume that sent is already sorted for this problem.

Rather than keep track of the original sentence sent, we could also just check if the reverse of a word exists in the rest of `reduced`, and then check `so-far` if not.

**Higher-Order-Functions**

5. **Write function that finds the mode (or one of the modes in case of ties) of the numbers in the input sentence. Example:**
   **`(1 2 3 12 1 1)`→ `1`**
   **`(12 3 5 3 12)` → `12` or `3`**

```
; finds the maximum frequency of items
(define (max_freq sent)
   (accumulate max (every (lambda (x) (appearances x sent)) sent)))

;finds the one with the same frequency as max_freq
(define (mode sent)
   (first
      (keep (lambda (x) (equal? (appearances x sent) (max_freq sent)))
            sent)) )
```

6. **Consider a database of student information that consists of a word for each student of with the following structure: `Name;Age;Letter-Grade`. Assuming that student name includes only English letters, write a procedure that returns a sentence of all student ages. For instance:**
   **`(Mary;19;A+ Bill;23;B- Jimmy;44;A Sally;25;C+)` → `(19 23 44 25)`**

```
(define (ages sent)
     (every (lambda (wrd) (keep number? wrd)) sent))
```

7. **Modify your procedure and use it to write a function that returns the name of (one of) the oldest student(s) in class. So for the previous example the procedure should return: `Jimmy`.**
   **Assume that we have a function (name record) that gives us the name part of a record: (name 'Mary;19;A+ ) → Mary**

```
; just finds the maximum age in the return value of the previous function
(define (max_age sent)
   (accumulate max (every (lambda (wrd) (keep number? wrd)) sent)))

;finds the name of the person with the same age as max_age
;compare it to mode function in question 5
(define (oldest sent)
   (name (keep (lambda (record)
                  (equal? (max_age sent) (keep number? wrd)))
               sent)))
```

8. **Write the procedure `last-out` using higher-order-functions. Here's a sample call:**
   **`(last-out 'clint)` → `(clint lint int nt t nt int lint clint)`**

```
(define (last-out wd)
   (accumulate
       (lambda (new so-far)
          (se (word new (first so-far)) so-far (word new (first so-far))))
       wd))
```

   This is very similar to `downup` except that we are spreading out from the last word rather than the first word. `accumulate` is a convenient higher order procedure to use for this because we need to somehow keep track of the last value we created. The new word we add to the sentence `so-far` is always one more letter than the last word processed, which would be on the outer edges of `so-far`, `(first so-far)`.

9. **Before you had written in lab `thoroughly-reversed` using recursion. Lets try it with HOF now.**
   **`(thoroughly-reversed '(scheme is so cool))` → `(looc os si emehcs)`**

```
(define (thorough-reverse sent)
   (accumulate
       (lambda (new-wd so-far-sent)(se so-far-sent new-wd))
       (every (lambda (wd)
                 (accumulate (lambda (new-ltr so-far-wd)
                                (word so-far-wd new-ltr)) wd))  sent)))
```

   This is basically just doing a double reverse. Here we reverse each word of the sentence, then we reverse the sentence itself.

10. **Write the procedure `match-all` so that every person in the first argument is matched to every person in the second argument. However, if the person's name consists only of numbers, then we want to ignore him/her. Example:**
    **`(match-all '(mary 123 helen abby) '(ted 456 bob))`**
    **→ `(mary-ted helen-ted abby-ted mary-bob helen-bob abby-bob)`**
    **Note the order of the return value: we match every person in arg1 to one person in arg2 before matching them to the next person in arg2. Each match is created with a hyphen between the two names.**

```
(define (match-all sent1 sent2)
   (every (lambda (wd2)
            (every (lambda (wd1) (word wd1 '- wd2))
                   (keep (lambda (wd1) (not (number? wd1))) sent1)))
          (keep (lambda (wd2) (not (number? wd2))) sent2)))
```

From the way the problem reads, it seems like we would have to use two `every`s. The order of the return value is based on the people in `sent2` — all the pairs with one person from `sent2` are shown before showing the possible pairs for the next person in `sent2`. Because of this, we can make the second argument to the outer `every` be `sent2`. Also, we do not need to consider names that are actually numbers, so they can be removed. Note that the placeholder name `wd1` in the `lambda` inside `keep` is different and does not interfere with the placeholder `wd1` inside the `lambda` inside `every`. Also, for `keep`, we cannot just do `(keep (not (number?))`. This would actually generate an error saying there are too few arguments to `number?` The inner `lambda` can access the arguments outside of it; this allows us to form pairs accessing `wd2` within `sent1`.

**Lambda**

**11. What does the following evaluate to? If it returns an error, describe what the error is and why it occurred. If it returns a procedure, describe what argument(s) it takes and what it does with it/them. If it returns a value, write out the arithmetic expression of it.**

```
STk> ((lambda (x y)
         (lambda (a b)
            (+ a ((lambda (g h) (* g h)) b 10) x y))) 5 8)
```

This evaluates to a procedure with arguments a and b, which is basically the second lambda.

**12. If the previous expression generated an error, fix it and make a procedure call. If it returned a procedure, how do you call that procedure with arguments `10` and `7` without changing the existing code, and what does it return? (don't add anything inside the left-most parenthesis and the right-most parenthesis.)**

If the above were a procedure, then to call it, we do what we always do. We call it with a left paren and then give it arguments.

```
( ((lambda (x y)
       (lambda (a b)
          (+ a ((lambda (g h) (* g h)) b 10) x y))) 5 8)   10 7)
```

The evaluation that is done: `(+ 10 (* 7 10) 5 8)` → `93`

**13. Assume that I have a function `Even_Apply` that takes two arguments: a one-argument function and a sentence. `Even_Apply` applies the function on all even-position members of the sentence. For example, `(Even_Apply butfirst '(hello world! What a happy day))` → `(hello orld! What "" happy ay)`. Write a function that increments all even-position members of a sentence by one. Example: `(1 2 3 4)` → `(1 3 4 4)`, and `(1 1 1 1 1)` → `(1 2 1 2 1)`**

```
(define (inc_even sent)
      (Even_Apply (lambda (x) (+ 1 x)) sent))
```

**14. Use `Even_Apply` to write a procedure that takes a list of student names and their grades and creates a list of student names and their pass or fail situation. Your procedure should take the minimum passing grade as an input. See the example for more clarification:**
**Passing grade = 60 => `(Ali 75 John 55 Maria 90 Edward 85 George 40)` → `(Ali passed John Failed Maria Passed Edward Passed George Failed)`**

```
(define (pass_fail sent min_pass)
      (Even_Apply (lambda (x) (if (> x min_pass) 'passed 'failed)) sent))
```

**15. Write a function that categorizes all names in a sentence based on their first letter. It should put all names starting with A first; then all names starting with B; and so on… For instance if the input sentence is `(Adam Bobby John Catherine Sara Bill Abraham Jim)`, the function will return `(Adam Abraham Booby Bill Catherine John Jim Sara)`. [this is the first step to implement bucket sort!]**

```
(define (semi_sort sent)
   (every (lambda (letter)
            ;the next part assumes a specific letter and
            ;picks up all names starting with that letter
            (keep (lambda (name) (equal? (first name) letter)) sent))
          '(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)))
```

**Others**

**16. There is a monkey named Sam. It will skip and hop for you depending on how many coins you give it. We have written a procedure `monkey` that will return a sentence describing the order in which Sam skips and hops when given `n` coins. We have also written the procedures `count-skip` and `count-hop` to count the number of times Sam skips and hops, respectively, when given n coins. Example: `(count-skip (monkey 5))` → `7`**

```
(define (monkey coin)
   (cond ((= coin 0) '())
         ((= coin 1) '())
         (else (se 'skip (monkey (- coin 2))
                   'hop (monkey (- coin 1))))))
```

**a) What does `(monkey 4)` return?**

```
(monkey 4) → (skip skip hop hop skip hop skip hop)
     (monkey 2) → (skip () hop ()) → (skip hop)
          (monkey 0) → ()
          (monkey 1) → ()
     (monkey 3) → (skip () hop  skip hop) → (skip hop skip hop)
          (monkey 1) → ()
          (monkey 2) → (skip hop)
```

**b) What does `(count-hop (monkey 3))` return?**

```
(count-hop (monkey 3)) → 2
```

**c) Do `count-hop` and `count-skip` always return the same value when given the same argument? Why?**

They always return the same value, given the same argument. Each `(count-hop (monkey n))` depends on `(count-hop (monkey (- n 2)))` and `(count-hop (monkey (- n 1)))`. So this means that if those two calls to monkey can return the same number of hops as skips, then `(monkey n)` will also return the same number of hops as skips. If you look at (a), you can see that we always add a hop for each skip we do.

**d) Write a formula for estimating `(count-skip (monkey n))`. What is the pattern similar to?**

`(count-skip (monkey n)) = (count-skip (monkey (- n 2))) + (count-skip (monkey (- n 1)))`

This is similar to the Fibonacci sequence of tree recursion. In fact, the number of skips/hops increases in the order of the Fibonacci sequence as `n` increases. The only exceptions are `(monkey 0)` and `(monkey 1)` when we return `()`.

**17. You just finished writing the procedure `condense`, but the neighbor's kid came to play on your computer before you could save it. The correct procedure should do the following: `(condense '(1 2 3 a b 8 9))` → `(6 ab 17)` Here's the buggy code:**

```
(define (condense sent)
  (cond ((empty? sent) sent)               ;; ((<= (count sent) 1) sent)
        ((and (number? (first sent))
              (word? (second sent)))       ;; (not (number? (second sent)))
         (se (first sent) (condense (bf sent))))
        ((and (number? (first sent))
              (number? (second sent)))
         (condense (se (+ (first sent) (second sent)) (bf (bf sent)))))
        ((number? (second sent))
         (se (first sent) (condense (bf sent))))
        (else (condense (se (word (first sent) (second sent))
                            (bf (bf sent)))))))
```

**With the buggy code, what does this return: `(condense '(1 2 3 a b 8 9))`? Explain why. Fix the code so that it returns the correct value. Explain what you had to fix.**

`(condense '(1 2 3 a b 8 9))` generates an error from executing `(second '(17))`. The buggy version wrongly assumes that the given argument is either an empty sentence or a sentence of at least length 2. To fix for this, we can change the base case to `((<= (count sent) 1) sent)`. This is the equivalent of "adding" an additional case `((= (count sent) 1) sent)`, but because sent of length 1 or 0 give the same return value, we can just combine them into one case.

Now if we try `(condense '(1 2 3 a b 8 9))` again, we get `(1 2 3 ab 8 9)`. This bug comes from the second case. It should be checking for the case of a number followed by a letter. However, numbers are also considered words: `(word? 5)` → `#t`. Thus we need to find another way to check whether the number is followed by a word or number:
`(and (number? (first sent)) (not (number? (second sent))))`

Another way to fix the second bug would be to switch the order of case 2 and case3.