CS 135 Fall 2017
Midterm 2 Review
Reminder: Midterm 2 (November 3)

- The midterm will be held on Friday, November 3rd at 6:30PM.
- Check your seating for the midterm on Odyssey.
- The midterm will cover up to and including Module 08 Slide 37.
- There will be NO assignment due Tuesday, November 7.
Clicker Question - box-and-pointer

Which of the following nested box representations match this box-and-pointer representation?

A

\[
\begin{array}{ccc}
1 & 2 & 3 \\
\end{array}
\]

B

\[
\begin{array}{ccc}
1 & 2 & 3 \\
\end{array}
\]

C

\[
\begin{array}{ccc}
1 & 2 & 3 \\
\end{array}
\]

D

\[
\begin{array}{ccc}
1 & 2 & 3 \\
\end{array}
\]

E

\[
\begin{array}{ccc}
1 & 2 & 3 \\
\end{array}
\]
Clicker Question - box-and-pointer

Which of the following nested box representations match this box-and-pointer representation?

A

1 2
2 3

B

1 2
2 3

C

1 2 3

D

1 2 3

E

1 2 3
Clicker Question - List Translation

Given this list:

(list (list) 'cons (list (list 2 'green) 3))

Which of the following is equivalent to the given list?

A  '(empty cons (list 2 'green) 3)
B  '(empty cons (2 'green) 3)
C  '(empty cons (list (list 2 'green) 3))
D  '(() cons ((2 green) 3))
E  '(() 'cons ((2 'green) 3))
Clicker Question - List Translation

Given this list:

\[(\text{list} \ (\text{list}) \ '\text{cons} \ (\text{list} \ (\text{list} \ 2 \ '\text{green}) \ 3))\]

Which of the following is equivalent to the given list?

A  \'(empty \ cons \ (list \ 2 \ 'green) \ 3)
B  \'(empty \ cons \ (2 \ 'green) \ 3)
C  \'(empty \ cons \ (list \ (list \ 2 \ 'green) \ 3))
D  \'(() \ cons \ ((2 \ green) \ 3))
E  \'(() \ 'cons \ ((2 \ 'green) \ 3))
Stepping through Lists

Give the first and second substitution steps as well as the final value for the following expression:

\[
\text{(length \ (rest \ (rest \ (second \ '(((hello \ red) \ (0 \ 1 \ 1 \ 2 \ 3 \ 5) \ ())))))\n}\]
Stepping through Lists

(length (rest (rest (second '((hello red) (0 1 1 2 3 5) ()))))))
⇒ (length (rest (rest '(0 1 1 2 3 5))))
Stepping through Lists

\[
\text{(length (rest (rest (second '(((hello red) (0 1 1 2 3 5) ()))))))}
\]
\[
\Rightarrow \text{(length (rest (rest '(0 1 1 2 3 5))))}
\]
\[
\Rightarrow \text{(length (rest '(1 1 2 3 5)))}
\]
Stepping through Lists

\[
(length \ (rest \ (rest \ (second \ '(((\text{hello red}) \ (0 \ 1 \ 1 \ 2 \ 3 \ 5) \ ()))))))
\]

⇒ \[(length \ (rest \ (rest \ '(0 \ 1 \ 1 \ 2 \ 3 \ 5))))\]

⇒ \[(length \ (rest \ '(1 \ 1 \ 2 \ 3 \ 5)))\]

...

⇒ 4
Template functions - Shapes

Consider the following definitions:

```
(define-struct rectangle (length width colour))
;; A Rectangle is a (make-rectangle Num Num Sym)
;; requires: length, width > 0
```

```
(define-struct triangle (base height colour))
;; A Triangle is a (make-triangle Num Num Sym)
;; requires: base, height > 0
```

```
;; A Shape is (anyof Rectangle Triangle)
```

Write a template function for a Shape and another template function for a list of Shapes.
Template functions - Shapes

;; my-shape-fn: Shape → Any
(define (my-shape-fn shape)
  (cond
    [(rectangle? shape) (... (rectangle-length shape) ... 
      (rectangle-width shape) ...
      (rectangle-colour shape) ...)]
    [(triangle? shape) (... (triangle-base shape) ... 
      (triangle-height shape) ...
      (triangle-colour shape) ...)])
Template functions - Shapes

;; my-list-shapes-fn: (listof Shape) → Any
(define (my-list-of-shapes-fn loshapes)
  (cond
    [(empty? loshapes) . . . ]
    [else (. . . (my-shape-fn (first loshapes)). . . (my-list-of-shapes-fn (rest loshapes)). . . )]))
Insertion Sort - sort-shapes

Using your template functions as a guide, write a function called sort-shapes that sorts a list of Shapes in increasing order of area. If two shapes have the same area, they should appear in the same order as in the original list.
Insertion Sort - sort-shapes

(define (sort-shapes loshapes)
  (cond
    [(empty? loshapes) empty]
    [else (insert-shape (first loshapes)
                          (sort-shapes (rest loshapes))))]))
Insertion Sort - sort-shapes

(define (insert-shape shape sloshapes)
  (cond
   [(empty? sloshapes) (cons shape empty)]
   [(<= (area-shape shape) (area-shape (first sloshapes))) (cons shape sloshapes)]
   [else (cons (first sloshapes) (insert-shape shape (rest sloshapes)))]))

(define (area-shape shape)
  (cond
   [(rectangle? shape) (* (rectangle-length shape)
                            (rectangle-width shape))]
   [(triangle? shape) (* 1/2 (triangle-base shape)
                        (triangle-height shape))])))
Recursing on a list and 2 Nats - sublist

Write a function called sublist which consumes a list, lst, and 2 natural numbers, start and end. sublist should produce the elements in lst indexed from start up to but not including end. If the list doesn’t have sufficient elements at any point then any contents within the range so far are returned. Note that the first element of a list is indexed at 0.

(sublist 2 5 '(a b c d e f)) ⇒ '(c d e)
(sublist 4 8 '(a b c d e f)) ⇒ '(e f)
Recursing on a list and 2 Nats - sublist

;; (sublist lst start end) produces elements from index start, to index end of lst.
;; sublist: (listof Any) Nat Nat → (listof Any)
;; requires: start ≤ end

;; Examples:
(check-expect (sublist empty 1 2) empty)
(check-expect (sublist '(a b c) 0 3) '(a b c))
(check-expect (sublist '(a b c d) 0 0) empty)
(check-expect (sublist '(a b c d) 1 3) '(b c))

(define (sublist lst start end)
  (cond
    [(empty? lst) empty]
    [(= end 0) empty]
    [(= start 0) (cons (first lst) (sublist (rest lst) start (sub1 end)))]
    [else (sublist (rest lst) (sub1 start) (sub1 end))])))
2-dimensional lists - get-cols-range

Consider the following data definition:

;;; A Table is a (listof (listof Any))
;;; requires: all the sublists have the same length

Using sublist, write a function called get-cols-range which consumes a Table and 2 natural numbers, start and end. get-cols-range should produce the table with only columns indexed from start up to but not including end. If the table doesn’t have sufficient columns then any columns within the range are returned. Note that the first column is indexed at 0.

(get-cols-range ’((1 2 3 4 5) (a b c d e) (3 6 9 12 15) (f g h i j)) 1 3)
⇒ ’((2 3) (b c) (6 9) (g h))
2-dimensional lists - get-cols-range

;;; get-cols-range: Table Nat Nat → Table
;;; requires: start ≤ end

(define (get-cols-range table start end)
  (cond
    [(empty? table) empty]
    [else (cons (sublist (first table) start end)
                (get-cols-range (rest table) start end))])))
Recursing on a Nat - add

In this problem, we will be implementing the addition of 2 Nats without using the built-in Racket function `+`.

Write a function called `add` that adds together 2 natural numbers. The only built-in arithmetic function you may use is `add1`. You may not use any helper functions.
Recursing on a Nat - add

;; (add m n) produces the sum of m and n

;; add: Nat Nat → Nat

;; Examples:

(check-expect (add 3 0) 3)
(check-expect (add 7 9) 16)

(define (add m n)
  (cond
    [(= n 0) m]
    [else (add1 (add m (sub1 n)))]))
Recursing in lockstep - hangman

In the game of hangman, one player decides on a secret word and the other player tries to guess the word one letter at a time. In this problem, we will write a function that simulates one such guess.

Write a function called `hangman` that consumes a string called `secret-word` and another string called `guessed`, as well as a single character `guess`. `guessed` is the same string as `secret-word` except all the letters that have not been guessed yet are replaced by "_". `hangman` should produce a new string such that if `guess` is in `secret-word`, all the corresponding blanks in `guessed` are replaced by `guess`. Otherwise, `guessed` is produced.

(hangman "cat" "c_t" #\a) ⇒ "cat"
(hangman "boo" "___" #\o) ⇒ "_oo"
(hangman "onion" "_n_n_" #\p) ⇒ "_n_n_n"
Recursing in lockstep - hangman

(define (hangman/chars secret-word current-state guess)
  (cond
    [(empty? secret-word) empty]
    [(char=? (first secret-word) guess)
      (cons guess
        (hangman/chars (rest secret-word) (rest current-state) guess))]
    [else (cons (first current-state)
      (hangman/chars (rest secret-word) (rest current-state) guess))]]
)

(define (hangman secret-word current-state guess)
  (list->string (hangman/chars (string->list secret-word)
    (string->list current-state)
    guess)))
Recursing at different rates - symmetric-difference

Recall:

;; A NumSet is a (listof Num)
;; requires: the numbers in the list are strictly increasing

Rewrite the function symmetric-difference from Assignment 05 without helper functions.

The function should consume 2 NumSets, set1 and set2, and produce a NumSet with all the elements that are in set1 or set2 but not both.

(symmetric-difference '(1 2 3 4) '(2 4 6 8)) ⇒ '(1 3 6 8)
Recursing at different rates - symmetric-difference

(define (symmetric-difference set1 set2)
  (cond
    [(empty? set1) set2]
    [(empty? set2) set1]
    [(< (first set1) (first set2))
      (cons (first set1) (symmetric-difference (rest set1) set2))]
    [(= (first set1) (first set2))
      (symmetric-difference (rest set1) (rest set2))]
    [else (cons (first set2) (symmetric-difference set1 (rest set2)))]))
;; remove-elem: Any (listof Any) → (listof Any)
;; Examples:
(check-expect (remove-elem 3 empty) empty)
(check-expect (remove-elem 2 '(1 2 2 3)) '(1 2 3))
(check-expect (remove-elem 4 '(dog cat 1)) '(dog cat 1))

(define (remove-elem elem lst)
  (cond
   [(empty? lst) empty]
   [(equal? elem (first lst)) (rest lst)]
   [else (cons (first lst)
                (remove-elem elem (rest lst)))]))

What kind of recursion is this?
A  Pure structural
B  Generative
C  Accumulative
D  I don’t know
(define (remove-elem elem lst)
  (cond
    [(empty? lst) empty]
    [(equal? elem (first lst)) (rest lst)]
    [else (cons (first lst)
                 (remove-elem elem (rest lst)))]))

(check-expect (remove-elem 3 empty) empty)
(check-expect (remove-elem 2 '(1 2 2 3)) '(1 2 3))
(check-expect (remove-elem 4 '(dog cat 1)) '(dog cat 1))

What kind of recursion is this?
A  Pure structural
B  Generative
C  Accumulative
D  I don’t know
;; same-elems?: (listof Any) (listof Any) → Bool
;; Examples:
(check-expect (same-elems? empty empty) true)
(check-expect (same-elems? empty '(cat kitty cat)) false)
(check-expect (same-elems? '(cat 1 dog) '(dog cat 1)) true)
(check-expect (same-elems? '(cat 1 dog) '(dog cat 2)) false)

(define (same-elems? l1 l2)
  (cond
   [(and (empty? l1) (empty? l2)) true]
   [(or (empty? l1) (empty? l2)) false]
   [else (same-elems? (remove-elem (first l1) l1)
                      (remove-elem (first l1) l2))]))

What kind of recursion is this?
A  Pure structural
B  Generative
C  Accumulative
D  I don’t know
;; same-elems?: (listof Any) (listof Any) → Bool
;; Examples:
(check-expect (same-elems? empty empty) true)
(check-expect (same-elems? empty '(cat kitty cat)) false)
(check-expect (same-elems? '(cat 1 dog) '(dog cat 1)) true)
(check-expect (same-elems? '(cat 1 dog) '(dog cat 2)) false)

(define (same-elems? l1 l2)
  (cond
    [(and (empty? l1) (empty? l2)) true]
    [(or (empty? l1) (empty? l2)) false]
    [else (same-elems? (remove-elem (first l1) l1)
                           (remove-elem (first l1) l2))]))

What kind of recursion is this?
A Pure structural
B Generative
C Accumulative
D I don’t know
Association lists - HeightsAL

Write a data definition for an association list used to store peoples’ names and heights. The names will be stored as strings and are unique and the heights will be numbers.
Association lists - HeightsAL

;; A HeightsAL is a (listof (list Str Num))

;; requires: the name are unique

;; the heights are positive
Accumulative recursion - shortest

Write a function called `shortest` that consumes a non-empty HeightsAL and produces the name of the shortest person in the list. Use **accumulative recursion**.

```
(shortest '(()"ron" 180) ("hermione" 164) ("mrs figg" 133)) ⇒ "mrs figg"
```
Accumulative recursion - shortest

;; shortest: HeightsAL → Str
;; requires: heights-al is non-empty

(define (shortest heights-al)
  (shortest/acc (rest heights-al) (first heights-al)))

;; shortest-acc: HeightsAL (list Str Num) → Str

(define (shortest/acc heights-al shortest-so-far)
  (cond
    [(empty? heights-al) (first shortest-so-far)]
    [(< (second (first heights-al)) (second shortest-so-far))
      (shortest/acc (rest heights-al) (first heights-al))]
    [else (shortest/acc (rest heights-al) shortest-so-far)]))
Binary trees - count-internal-nodes

An internal node is a node that has at least one child. Write a function count-internal-nodes that counts the number of internal nodes in a binary tree.

(define-struct node (key val left right))

;; A Node is a (make-node Num Str BT BT)

;; A binary tree (BT) is one of:

;; * empty

;; * Node
Binary trees - count-internal-nodes

(define (count-internal-nodes tree)
  (cond
    [(empty? tree) 0]
    [(and (empty? (node-left tree))
          (empty? (node-right tree))) 0]
    [else (+ 1 (count-internal-nodes (node-left tree))
               (count-internal-nodes (node-right tree)))]))