Warm-up

10 prisoners are lined up from tallest to shortest and all wear black or white hats. Police Officer is about to ask starting from the tallest to shortest: Is your hat white? Prisoners can make at most one mistake in total answering Yes or No. What strategy they should use to make this happen?
Reminders

• The final exam is scheduled for April 13, 9:00 - 11:30 AM in STC 1012

• Please look up your assigned seating ahead of time

• The exam will cover modules 1 - 13 with an emphasis on the last half of the course.
Abstract List Functions
What is Abstraction?
Abstraction is the process of finding similarities or common aspects, and forgetting unimportant differences.

In this context: writing a function, specifically an “Abstract List Function”

We will consider a few examples.
Abstract list functions overview

In lecture you covered a number of the abstract list functions that Racket has available. Below are the abstract list functions we will be talking about today.

\[
\begin{align*}
;\text{build-list: } & \text{Nat (Nat } \to \text{ Y) } \to \text{ (listof Y)} \\
;\text{map: } & (X \to Y) \text{ (listof X) } \to \text{ (listof Y)} \\
;\text{filter: } & (X \to \text{ Bool}) \text{ (listof X) } \to \text{ (listof X)} \\
;\text{foldr: } & (X Y \to Y) Y \text{ (listof X) } \to Y \\
;\text{foldl: } & (X Y \to Y) Y \text{ (listof X) } \to Y \\
;\text{sort: } & \text{ (listof X) (X X } \to \text{ Bool) } \to \text{ (listof X)}
\end{align*}
\]
ALF: build-list

;;build-list Nat (Nat → Y) → (listof Y)

build-list is an abstract list function which consumes a Nat and a function which consumes a Nat producing Any. build-list creates a list of the natural numbers from 0 to one less than the natural number consumed and applies the function consumed to each of those. Examples:

(build-list 3 identity)

;;⇒ (list 0 1 2)

(build-list 5 sqr)

;;⇒ (list 0 1 4 9 16)

(build-list 4 (lambda (x) (build-list x add1)))

;;⇒ (list empty (list 1) (list 1 2) (list 1 2 3))
ALF: map

;;map (X → Y) (listof X) → (listof Y)

map is an abstract list function which consumes a function and a list and applies that function to every element of that list. Examples:

(map add1 (list 1 2 3))
;;⇒ (list 2 3 4)

(map sqr (build-list 4 identity))
;;⇒ (list 0 1 4 9)

(map string-length (list "shake" "it" "off"))
;;⇒ (list 5 2 3)
ALF: map

Multiply every element in a list by two:

;;(listof Num) → (listof Num)

Calculate a list of distances from the origin for a list of Posns:

;;(listof Posn) → (listof Num)
**ALF: map**

Multiply every element in a list by two:

```
(map (lambda (x) (* x 2)) lo-Num)
```

Calculate a list of distances from the origin for a list of Posns

```
(define dist (lambda (x) (sqrt (+ (sqr (posn-x x)) (sqr (posn-y x))))))
(map dist lo-Posn)
```
ALF: filter

;;filter (X → Bool) (listof X) → (listof X)
filter is an abstract list function which consumes a predicate and a list. If after consume an element of the list, the predicate produces true, the element will be produced in the new list otherwise it is filtered out.
Examples:

(filter positive? (list −2 1 0 −3 4 −7 −4 150))
;;⇒ (list 1 4 150)
(filter not (list true false false true))
;;⇒ (list false false)

(define taylor-lyrics ’("Blank Space" "Shake it Off" "Red" "Fearless" "Love Story"))
(define taylor? (lambda (x) (member? x taylor-lyrics))
(filter taylor? (list "Call me Maybe" "Red" "Party in the USA" "Love Story")))
;;⇒ (list "Red" "Love Story")
ALF: foldr

;;foldr (X Y → Y) Y (listof X) → Y
foldr is an abstract list function which consumes a function, a base case, and a list. foldr applies the given function to every element and the recursive rest of the list. At the final element it applies the function to that element and the base case.

(define lst (list item1 item2 item3))
(foldl function basecase lst)
(function item1 (function item2 (function item3 basecase)))

Examples:

(foldr + 0 (list 1 2 3 4 5))
;;⇒ 15
(foldr string-append " " '("It's " "like " "I " "got " "this " "music " "in " "my " "mind"))
;;⇒ "It's like I got this music in my mind"
**ALF: foldl**

```
(foldl (X Y → Y) Y (listof X) → Y)
```

foldl is an abstract list function which consumes a function, an accumulator, and a list. foldl applies the given function to every element and the accumulator. At the first element it applies the function to that element and the initial accumulator.

```
(define lst (list item1 item2 item3))
(foldl function basecase lst)
(function item3 (function item2 (function item1 basecase)))
```

Examples:

```
(foldl + 0 (list 1 2 3 4 5)) ;;⇒ 15
(foldl cons empty (list 1 2 3)) ;;⇒ (list 3 2 1)
(foldl − −3 (list 1 −2 3)) ;;⇒ 9
```
ALF: sort

`;sort (listof X) (X X → Bool) → (listof X)`

sort is an abstract list function which consumes a function and a list. Sort uses the given function to compare elements in the list. If consumed function produces true if the first of two elements is less than the second. The produced list is re-ordered accordingly.

Examples:

```scheme
(sort (list 5 1 4 2 3) <) ➞ (list 1 2 3 4 5)
(sort (list "Josh" "Adam" "Andy" "Andrew" "Pearl" "Nick") string<?) ➞ (list "Adam" "Andrew" "Andy" "Josh" "Nick" "Pearl")
(sort (list #\c #\a #\b) char<?) ➞ (list #\a #\b #\c)
```
Lambdas - apply-to-all

Write a function `apply-to-all` which consumes a function, `to-apply` as a parameter. `apply-to-all` will produce a function that applies `to-apply` to every item in a list. Include a contract.

```
(check-expect ((apply-to-all add1) '(1 2 3 4 5)) '(2 3 4 5 6))
```
Lambdas - apply-to-all

This function is most easily done using lambdas, but you could also write it using local definitions.

;; apply-to-all: (X → Y) → ((listof X) → (listof Y))
(define (apply-to-all to-apply)
  (lambda (lst) (map to-apply lst)))

(define (apply-to-all-local to-apply)
  (local
    [(define (apply-func lst)
         (map to-apply lst))]
    apply-func))
Abstract List Functions - Choosing ALFs

Write each of the transformations described below using only abstract list functions.

;; Multiply the even numbers by 3
;; Leave the odd numbers alone
(check-expect (mult-even-3 '(1 2 3 4 5 6)) '(1 6 3 12 5 18))

;; Produce true if there are strictly more even numbers than odd numbers
(check-expect (more-evens? '(2 4 6 8 10 1 3 5)) true)
(check-expect (more-evens? '(2 4 6 1 3 5 7)) false)

;; Reverse a list
(check-expect (my-reverse '(1 2 3 4 5 6)) '(6 5 4 3 2 1))
Abstract List Functions - Choosing ALFs

;; Multiply the even numbers by 3
;; Leave the odd numbers alone
(define (mult-even-3 lon)
  (map (lambda (x)
          (cond [(even? x) (* 3 x)]
                [else x]]
               lon))

;; Produce true if there are strictly more even numbers than odd numbers
(define (more-evens? lon)
  (> (length (filter even? lon))
      (length (filter odd? lon))))

;; Reverse a list
(define (my-reverse lst) (foldl cons empty lst))
Abstract List Functions - Choosing ALFs

Write each of the transformations described below using only abstract list functions.

;; Create a list of the first n even Nats, starting from 0
(check-expect (first-n-evens 4) (list 0 2 4 6))

;; Cumulative sum: Produce a list whose i-th element is the
;; sum of the first i elements of input list
(check-expect (cumu-sum empty) empty)
(check-expect (cumu-sum '(1)) '(1))
(check-expect (cumu-sum '(1 2 3 4 5 6)) '(1 3 6 10 15 21))
Abstract List Functions - Choosing ALFs

;; Create a list of the first n even Nats, starting from 0
(define (first-n-evens n)
  (build-list n (lambda (x) (* x 2))))

;; Cumulative sum: Produce a list whose i-th element is the
;; sum of the first i elements of input list
(define (cumu-sum lon)
  (rest (foldr (lambda (x y) (cons (- (first y) x) y))
               (list (foldr + 0 lon))
               lon)))
count-squares

Write a function count-squares that consumes a (listof Int) and produces the number of perfect squares in the given list. A perfect square is an integer that is the square of an integer. Without explicit recursion.

(define (count-squares loi)
  (foldr (lambda (x y) (cond [(integer? (sqrt x)) (add1 y)] [else y]))
         0 loi))
ascending?

Write a function ascending? that consumes a (listof Int) and produces true if the entries of the list appear in a strictly increasing order, and false otherwise. Note that a list with 0 or 1 entries is ascending. This is harder than the last two problems.

(check-expect (ascending? (cons 1 (cons 2 (cons 3 (cons 4 empty))))) true)

define (ascending? loi)

  (or (empty? loi)

    (number? (foldl (lambda (x y) (cond [(false? y) y]
                                                 [(> x y) x]
                                                 [else false]]))
                  (first loi) (rest loi)))))
Binary Search Trees
BST - Review

Remember that a binary search tree (BST) is a tree structure with (key, value) pairs at the nodes. Each node is structured such that every key in right is greater than key, and every key in left is less than key.

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

;; a Node is a (make-node Num Str BST BST)

;; a BST is one of:
;; * empty
;; * Node
To write a template for a BST function we need to consider the two cases outlined by our data definition of a BST.

- empty
- Node

In the case where a BST is a node we have to use all the selector functions. Since left and right are nodes themselves, we recurse the template function on the left and right.
The template for a function that consumes a BST would then look as such.

;; my-bst-fn: BST → Any
(define (my-bst-fn bst)
  (cond
    [(empty? bst) ...]
    [else (... (node-key bst) ...
               (node-value bst) ...
               (my-bst-fn (node-left bst)) ...
               (my-bst-fn (node-right bst)) ...)])
BST Traversal

Accessing every node in a given tree, often called traversing the tree, is a useful process in some cases. This section focuses on traversing a BST.

A traversal is done by recursing on the left subtree, acting on the current node, and recursing on the right subtree while at each step merging the results of these three evaluations in some way.
In-Order Traversal

An in-order traversal only adds one extra restriction. The nodes are visited in the order of left subtree, current node, right subtree. When producing a list using an in-order traversal of the BST you will notice that the produced list is sorted by key.
In-Order Traversal

Given a BST we want to produce a sorted association list of (key, value) pairs within the given bst. This requires that we use an in order traversal of the BST.

;; bst->al: BST → (listof (list Num Str))
(define (bst->al bst)
   (cond
     [(empty? bst) empty]
     [else (append (bst->al (node-left bst))
                   (cons (list (node-key bst) (node-pair bst))
                         (bst->al (node-right bst))))]))
Drivers

We can modify our node structure slightly so that the key is now a name and the value is the age of that person. Nodes are then ordered lexiographically with no repeated names in a single bst.

(define-struct person (name age left right))
;; a Person is a (make-person Str Nat BST BST)

;; a BST is one of:
;; * empty
;; * Person
Drivers
A basic BST using the person structure would look like this.

(define driver-tree (make-person "Adam" 19
    (make-person "Aaron" 14 empty empty)
    (make-person "David" 25 (make-person "Allisa" 16 empty empty))))
Drivers

Using the person structure we can traverse the BST to produce a sorted list of the names of people eligible for a driver’s license. In order to be a licensed driver in Ontario, a person must be at least 16 years old.

;; (drivers bst) produces an ordered list of names corresponding to people who can drive that are contained within bst
;; drivers: BST → (listof Str)
;; Examples:
(check-expect (drivers empty) empty)
(check-expect (drivers driver-tree) (list "Adam" "David"))
;; drivers: BST → (listof Str)

(define (drivers bst)
  (cond
    [(empty? bst) empty]
    [(>= (person-age bst) 16)
     (append (drivers (person-left bst))
            (cons (person-name bst)
                 (drivers (person-right bst)))]
    [else (append (drivers (person-left bst))
               (drivers (person-right bst)))]))
Mutual Recursion Templates
Mutual Recursion Templates

In an unbounded tree, any number of children is allowed for a node. Here is a definition for an unbounded tree:

(define-struct node (key children))
;;; A Node is a (make-node Nat (listof Node))

;;; A Tree is (anyof Node empty)

We have three non-trivial types associated with a tree: Tree, Node, and (listof Node). We can write mutually recursive templates to process these types.
Mutual Recursion Templates

First, we need to handle a single node. A template to process a structure will call all of its selectors:

```scheme
;; my-node-fn: Node → Any
(define (my-node-fn my-node)
  (... (node-key my-node)
       ... (node-children my-node)))
```

But we can improve this. `(node-children my-node)` is a `(listof Node)`, so we can apply a `(listof Node)` template.
Mutual Recursion Templates

;; my-node-fn: Node → Any
(define (my-node-fn my-node)
  (... (node-key my-node)
       ... (my-listof-node-fn (node-children my-node)))))
We can now write a template for \((\text{listof Node})\).

\[
\begin{align*}
\text{;; my-listof-node-fn: (listof Node) } & \rightarrow \text{ Any} \\
\text{(define (my-listof-node-fn lo-node)} & \\
\text{ \hspace{1cm} (cond [(empty? lo-node) \ldots ])} \\\n\text{ \hspace{2cm} [else \ldots (\ldots (first lo-node))]} \\\n\text{ \hspace{3cm} (my-listof-node-fn (rest lo-node))))))))
\end{align*}
\]

Similarly to before, we can call our Node template on the first of the list.
Mutual Recursion Templates

;; my-listof-node-fn: (listof Node) → Any
(define (my-listof-node-fn lo-node)
  (cond [(empty? lo-node) . . . ]
        [else (. . . (my-node-fn (first lo-node))
                  (my-listof-node-fn (rest lo-node)))]))
Mutual Recursion Templates

For some applications, we can even simplify this further using \texttt{foldr}:

\begin{verbatim}
;; my-listof-node-fn: (listof Node) \rightarrow Any
(define (my-listof-node-fn lo-node)
  (foldr (lambda (x y)(\ldots x y)) (\ldots) lo-node)
\end{verbatim}
Mutual Recursion Templates

Finally, we need a template for Tree

;; my-tree-fn: Tree → Any
(define (my-tree-fn my-tree)
  (cond [(empty? my-tree) . . . ]
        [else (my-node-fn my-tree)]))
Unbounded Tree Traversal
Unbounded Tree Traversal

Recall our tree definition:

```
(define-struct node (key children))
;; A Node is a (make-node Nat (listof Node))

;; A Tree is (anyof Node empty)
```

Suppose we want to output all the keys in the tree in a single list. How can we apply our mutually recursive templates?
Unbounded Tree Traversal

There are two cases when dealing with trees:

- The tree is empty
- The tree is non-empty

;; my-tree-fn: Tree → Any
(define (my-tree-fn my-tree)
  (cond [(empty? my-tree) ...]
        [else (my-node-fn my-tree)]))
Unbounded Tree Traversal

There are two cases when dealing with trees:

- The tree is empty
- The tree is non-empty

;; traverse: Tree → (listof Num)
(define (traverse my-tree)
  (cond [(empty? my-tree) empty]
        [(else (traverse/node my-tree))])))
Unbounded Tree Traversal

Now we need a function to operate specifically on a Node. It needs to combine the key of the node with the keys of all its children.

;; my-node-fn: Node → Any
(define (my-node-fn my-node)
  (\ldots (node-key my-node)
    \ldots (node-children my-node)))
Unbounded Tree Traversal

Now we need a function to operate specifically on a Node. It needs to combine the key of the node with the keys of all its children.

;; traverse/node: Tree → (listof Num)
(define (traverse/node my-node)
  (cons (node-key my-node)
    (traverse/list (node-children my-node)))))
Unbounded Tree Traversal

To complete our mutual recursion, we need a traverse/list function that will process a list of Node. It will combine the results of multiple calls to traverse/node

;; my-listof-node-fn: (listof Node) → Any
(define (my-listof-node-fn lo-node)
  (cond [(empty? lo-node) . . .]
        [else (. . . (. . . (first lo-node))
                  (my-listof-node-fn (rest lo-node)))]))
Unbounded Tree Traversal

To complete our mutual recursion, we need a traverse/list function that will process a list of Node. It will combine the results of multiple calls to traverse/node

;; traverse/list: (listof Node) → (listof Num)
(define (traverse/list lo-node)
  (cond [(empty? lo-node) empty]
        [else (append (traverse/node (first lo-node))
                      (traverse/list (rest lo-node)))]))

Unbounded Tree Traversal

Alternatively with foldr:

;; traverse/list: (listof Node) → Any

(define (traverse/list lo-node)
  (foldr (lambda (x y)
           (append (traverse/node x) y))
         empty
         lo-node))
Stepping
Stepping Rules Summary: Module 03

1. Apply functions only when all arguments are values.

2. When given a choice, evaluate the leftmost expression first.

3. \((f \ v_1...v_n) \Rightarrow v\) when \(f\) is built-in...

4. \((f \ v_1...v_n) \Rightarrow exp'\) when \((\text{define} \ (f \ x_1...x_n) \ exp)\) occurs to the left...

5. \(id \Rightarrow val\) when \((\text{define} \ id \ val)\) occurs to the left.

6. \((\text{cond} \ [\text{false} \ exp] ... ) \Rightarrow (\text{cond} ... )\).

7. \((\text{cond} \ [\text{true} \ exp] ... ) \Rightarrow \text{exp}\).

8. \((\text{cond} \ [\text{else} \ exp]) \Rightarrow \text{exp}\).
9. \((\text{and} \ \text{false} \ldots) \Rightarrow \text{false}\). 

10. \((\text{and} \ \text{true} \ldots) \Rightarrow (\text{and} \ldots)\). 

11. \((\text{and}) \Rightarrow \text{true}\). 

12. \((\text{or} \ \text{true} \ldots) \Rightarrow \text{true}\). 

13. \((\text{or} \ \text{false} \ldots) \Rightarrow (\text{or} \ldots)\). 

14. \((\text{or}) \Rightarrow \text{false}\).
Stepping Rules Summary: Structures

If a structure is defined with the structure definition

\[(\text{define-struct } \text{sname} \ (\text{fname1} \ldots \ \text{fnamen})):\]

15. \((\text{sname-fnamei} \ (\text{make-sname} \ v1 \ldots vi \ldots vn)) \Rightarrow vi, \text{where } v1 \ldots vn \text{ are all values.}\)

16. \((\text{sname?} \ (\text{make-sname} \ v1 \ldots vn)) \Rightarrow \text{true, where } v1 \ldots vn \text{ are all values.}\)

17. \((\text{sname?} \ V) \Rightarrow \text{false for } V \text{ a value of any other type.}\)
Stepping Rules Summary: Lists

18. \((\text{first} \ (\text{cons} \ a \ b)) \Rightarrow a\), where \(a\) and \(b\) are values.

19. \((\text{rest} \ (\text{cons} \ a \ b)) \Rightarrow b\), where \(a\) and \(b\) are values.

20. \((\text{empty?} \ \text{empty}) \Rightarrow \text{true}\).

21. \((\text{empty?} \ a) \Rightarrow \text{false}\) where \(a\) is not empty

You can also use \((\text{list} \ . \ . \ . )\) and quoted list abbreviatons, but switching between notations does not count as a “step.”
Local Stepping Rule Review

An expression of the form

\[
\text{(local \[(define x_1 \exp_1) \ldots (define x_n \exp_n)\] bodyexp)}
\]

is handled as follows.

\(x_1\) is replaced with a fresh identifier (call it \(x_1_{\text{new}}\)) everywhere it’s used in the \text{local} expression.

The same thing is done with \(x_2\) through \(x_n\).

The definitions \((\text{define } x_{1\ldots n_{\text{new}} } \exp_{1\ldots n})\) are then lifted out (all at once) to the top level of the program, preserving their ordering.
Local Stepping Rule Review

When all the rewritten definitions have been lifted out, what remains looks like (local [] bodyexp′), where bodyexp′ is the rewritten version of bodyexp. This is just replaced with bodyexp′. All of this is a single step.

(local [(define x1 exp1) . . . (define xn expn)] bodyexp) 
⇒ (define x1_new exp1′) ; using _new as an example

... 
(define xn_new expn′)

bodyexp′
Local Stepping Example 1

When renaming local definitions append “.0” if possible, or else “.1”, “.2”, etc. **Do not** recopy any line that is already in simplest form.

(define x 5)
(define y 10)
(define (my-fn a b)
  (local [(define x (+ a b))
           (define y (+ x (local [(define x 20)] x)))]
          (list x y)))
(my-fn x y)
Local Stepping Example 1

\[(\text{my-fn } x \ y) \Rightarrow (\text{my-fn } 5 \ y) \Rightarrow (\text{my-fn } 5 \ 10)\]

\[\Rightarrow (\text{local }[(\text{define } x (\text{+ } 5 \ 10)) ; \text{Replace both a and b}\]
\[\quad \text{(define } y (\text{+ } x (\text{local }[(\text{define } x \ 20)] \ x)))\)]

\[\text{(list } x \ y)\]

\[\Rightarrow (\text{define } x_0 (\text{+ } 5 \ 10))\]

\[\text{(define } y_0 (\text{+ } x_0 (\text{local }[(\text{define } x \ 20)] \ x)))\]

\[\text{(list } x_0 \ y_0)\]

\[\Rightarrow (\text{define } x_0 \ 15)\]

\[\text{(define } y_0 (\text{+ } x_0 (\text{local }[(\text{define } x \ 20)] \ x)))\]

\[\text{(list } x_0 \ y_0)\]
Local Stepping Example 1

⇒ (define y_0 (+ 15 (local [(define x 20)] x)))
   (list x_0 y_0)
⇒ (define x_1 20)
   (define y_0 (+ 15 x_1))
   (list x_0 y_0)
⇒ (define y_0 (+ 15 20))
   (list x_0 y_0)
⇒ (define y_0 35)
   (list x_0 y_0)
Local Stepping Example 1

⇒ (list 15 y 0)
⇒ (list 15 35)
Local Stepping Example 2

When renaming local definitions append “.0” if possible, or else “.1”, “.2”, etc. **Do not** recopy any line that is already in simplest form.

```
(define x 5)
(define y 10)
(define (my-fn x y) ; parameters are now x and y
  (local [(define x y) ; this line is different too
    (define y (+ x (local [(define x 20)] x)))]
  (list x y)))

(my-fn x y)
```
Local Stepping Example 2

\[(\text{my-fn } x \ y)\]  
⇒  \[(\text{my-fn } 5 \ y)\]  
⇒  \[(\text{my-fn } 5 \ 10)\]  
⇒  \[(\text{local } [(\text{define } x \ y)\]  
\[\quad (\text{define } y (\ + \ x \ (\text{local } [(\text{define } x \ 20)] \ x)))\]  
\[\quad (\text{list } x \ y)\]  
⇒  \[(\text{define } x_0 \ y_0)\]  
\[\quad (\text{define } y_0 (\ + \ x_0 (\text{local } [(\text{define } x \ 20)] \ x)))\]  
\[\quad (\text{list } x_0 \ y_0)\]  
⇒  Error: \text{local} variable used before its definition: \text{y}
**Lambda Stepping Example 1**

Recall that the rule for evaluating anonymous lambda expression is

\[
((\text{lambda } (x_1 \ldots x_n) \text{ exp}) \; v_1 \ldots v_n) \Rightarrow \text{ exp}'
\]

where \( \text{exp}' \) is \( \text{exp} \) with all occurrences of \( x_1 \) replaced by \( v_1 \), all occurrences of \( x_2 \) replaced by \( v_2 \), and so on.
Lambda Stepping
Given that the following definition has been processed in the Intermediate Student with Lambda language:

```lambda
(define (a c)
  (lambda (n)
    (cond
      [(zero? n) 1]
      [(even? n) (sub1 n)]
      [else (+ n ((a c) (sub1 n)))])))
)
```

((a "quick brown fox jumps over the lazy dog") 5)

Produce a step-by-step evaluation of the following program.
Lambda Stepping Example 1

$$((a \ "quick brown fox jumps over the lazy dog") \ 3)$$

$$\Rightarrow ((\text{lambda} \ (n))$$

$$(\text{cond}$$

$$[\text{zero?} \ n \ 1]$$

$$[\text{even?} \ n \ (\text{sub1} \ n)]$$

$$[\text{else} \ (+ \ n \ ((a \ "quick brown fox jumps over the lazy dog") \ (\text{sub1} \ n)))]) \ 3)$$

$$\Rightarrow (\text{cond}$$

$$[\text{zero?} \ 3 \ 1]$$

$$[\text{even?} \ 3 \ (\text{sub1} \ 3)]$$

$$[\text{else} \ (+ \ 3 \ ((a \ "quick brown fox jumps over the lazy dog") \ (\text{sub1} \ 3)))])$$

$$\Rightarrow (\text{cond}$$

$$[\text{false} \ 1]$$

$$[\text{even?} \ 3 \ (\text{sub1} \ 3)]$$

$$[\text{else} \ (+ \ 3 \ ((a \ "quick brown fox jumps over the lazy dog") \ (\text{sub1} \ 3)))])$$
Lambda Stepping Example 1

⇒ (cond
    [(even? 3) (sub1 3)]
    [else (+ 3 ((a "quick brown fox jumps over the lazy dog") (sub1 3)))]
⇒ (cond
    [false (sub1 3)]
    [else (+ 3 ((a "quick brown fox jumps over the lazy dog") (sub1 3)))]
⇒ (cond
    [else (+ 3 ((a "quick brown fox jumps over the lazy dog") (sub1 3)))]
⇒ (+ 3 ((a "quick brown fox jumps over the lazy dog") (sub1 3)))
⇒ (+ 3 ((a "quick brown fox jumps over the lazy dog") 2))
Lambda Stepping Example 1

$$\Rightarrow (\ + \ 3 \ ((\text{lambda} \ (n)) \ \\
\quad \text{(cond} \ \\
\quad \quad [(\text{zero?} \ n) \ 1] \ \\
\quad \quad [(\text{even?} \ n) \ (\text{sub1} \ n)] \ \\
\quad \quad [\text{else} \ (+ \ n \ ((\text{a} \ \"\text{quick brown fox jumps over the lazy dog}\" \ \text{(sub1} \ n))))])) \ 2))$$

$$\Rightarrow (\ + \ 3 \ \text{(cond} \ \\
\quad [(\text{zero?} \ 2) \ 1] \ \\
\quad [(\text{even?} \ 2) \ (\text{sub1} \ 2)] \ \\
\quad [\text{else} \ (+ \ 2 \ ((\text{a} \ \"\text{quick brown fox jumps over the lazy dog}\" \ \text{(sub1} \ 2))))])$$

$$\Rightarrow (\ + \ 3 \ \text{(cond} \ \\
\quad \text{false} \ 1] \ \\
\quad [(\text{even?} \ 2) \ (\text{sub1} \ 2)] \ \\
\quad [\text{else} \ (+ \ 2 \ ((\text{a} \ \"\text{quick brown fox jumps over the lazy dog}\" \ \text{(sub1} \ 2))))]$$
Lambda Stepping Example 1

⇒ (+ 3 (cond

  [(even? 2) (sub1 2)]
  [else (+ 2 ((a "quick brown fox jumps over the lazy dog") (sub1 2)))]
)

⇒ (+ 3 (cond

  [true (sub1 2)]
  [else (+ 2 ((a "quick brown fox jumps over the lazy dog") (sub1 2)))]
)

⇒ (+ 3 (sub1 2))
⇒ (+ 3 1)
⇒ 4
Lambda Stepping Example 2

Given that the following definition has been processed in the Intermediate Student with Lambda language:

\[
\text{(define fun}
\text{(lambda (x y)}
\text{(lambda (y z)}
\text{(lambda (z) ( + x y z))))})
\]

Produce a step-by-step evaluation of the following program.
Lambda Stepping Example 2

```lisp
(((fun 1 2) 3 4) 5)
⇒ ((((lambda (x y)
    (lambda (y z)
      (lambda (z) (+ x y z))) 1 2) 3 4) 5)
⇒ ((((lambda (y z)
    (lambda (z) (+ y z))) 3 4) 5)
⇒ ((lambda (z) (+ 1 y z)) 3 4) 5)
⇒ ((lambda (z) (+ 1 3 z)) 5)
⇒ (+ 1 3 5)
⇒ 9
```
Accumulative and Generative Recursion
Accumulative and Generative Recursion

In **accumulative recursion**, all parameters to the recursive calls are the same, one step closer to base case, plus one or more parameters containing intermediate results. You can always distinguish accumulative recursion by checking the output when reaching the base case. Generally, in the base case accumulative recursion returns a parameter which is accumulative. **foldl** is an abstract list function for the “accumulative recursion on a list” pattern.

In **generative recursion**, parameters are freely calculated at each step (watch out for correctness and termination).
Accumulative Recursion - factorial

Implement a function `factorial` which takes a `Nat` and outputs the factorial of that number using explicit recursion here.
Accumulative Recursion - factorial

;; (factorial n) produces the factorial of n
;; factorial: Nat → Nat
;; Examples:
(check-expect (factorial 1) 1)
(check-expect (factorial 3) 6)
(define (factorial n)
    (local [(define (fact-helper n counter acc)
              (cond
                [(>= counter n) (* acc counter)]
                [else (fact-helper n (add1 counter) (* acc counter))]))
        (fact-helper n 1 1)))))
;; Tests:
(check-expect (factorial 0) 1)
(check-expect (factorial 5) 120)
(check-expect (factorial 6) 720)
factorial

Now implement the same function factorial using an abstract list function only this time

;; factorial: Nat → Nat

(define (factorial n)
  (foldl ∗ 1 (build-list n add1)))
Accumulative Recursion - fibonacci

Write a function called `fibonacci` which takes a `Nat n > 0` and produces the $n$th Fibonacci number using generative recursion here.
Accumulative Recursion - fibonacci

;; (fibonacci n) output the nth fibonacci number in the fibonacci sequence
;; fibonacci: Nat → Nat
;;   requires: n > 0
;; Examples:
(check-expect (fibonacci 1) 1)
(check-expect (fibonacci 2) 1)

(define (fibonacci n)
  (local [(define (fib-helper n acc1 acc2)
                 (cond ((or (= n 1) (= n 2)) acc2)
                       (else (fib-helper (sub1 n) acc2 (+ acc1 acc2))))]
    (fib-helper n 1 1)))
;; Tests:
(check-expect (fibonacci 3) 2)
(check-expect (fibonacci 4) 3)
Now implement the same function fibonacci using an abstract list function.

```
(define (fibonacci n)
  (second (foldr (lambda (x y)
                  (cond [(or (= x 1) (= x 2)) y]
                         [else (list (second y)
                                     (+ (first y) (second y))))]
                (list 1 1)
                (build-list n add1))))
```
Generative Recursion - Quicksort

Now let’s do a quick review for quicksort.

1. Pick an element, called a pivot, from the sequence.
2. Separate the unsorted list into three sublists so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). Also (list pivot) takes up the center.
3. Recursively apply the above steps to the sublist of sequence with smaller values and separately to the sublist of sequence with greater values.
We have seen quick-sort function from the course notes. We generalize quick-sort function here which takes a list and a compare function and outputs a sorted list in the order of comparing function.

;;; (quick-sort lst cmpfn) sorts lst in the order indicated by the cmpfn.
;;; quick-sort: (listof X) (X X → Bool) → (listof X)
(define (quick-sort lst cmpfn)
    (cond [(empty? lst) empty]
          [else (local [(define pivot (first lst))
                        (define less
                            (filter (lambda (x) (cmpfn x pivot)) (rest lst)))
                        (define greater
                            (filter (lambda (x) (not (cmpfn x pivot))) (rest lst)))]
                    (append (quick-sort less cmpfn) (list pivot)
                            (quick-sort greater cmpfn)))]))
Generative Recursion - Mergesort

Mergesort also uses generative recursion. Mergesort is a divide and conquer algorithm.

In the first several steps, mergesort divides the unsorted list into 2 sublists each step, and eventually get several 1-element list (a list of 1-element is considered sorted). After that, repeatedly merge sublists to produce new sorted sublists until there is only 1 sublist remaining which will be the sorted list.
;; (mergesort lst cmp) sorts the elements of lst using comparator cmp
;; mergesort: (listof X) (X X → Bool) → (listof X)
(define (mergesort lst cmp)
  (cond
    [(or (empty? lst) (empty? (rest lst))) lst]
    [else (local
              ;; (take-left l i) produces the left-most i items of l
              ;; take-left: (listof X) Nat → (listof X)
              (define (take-left l i)
                (cond [(zero? i) empty]
                      [else (cons (first l) (take-left (rest l) (sub1 i)))]))

              ;; (take-right l i) produces l, with the left-most i removed
              ;; take-right: (listof X) Nat → (listof X)
              (define (take-right l i)
                (cond [(zero? i) l]
                      [else (take-right (rest l) (sub1 i))])))]))
(merge slst1 slst2) merges slst1 slst2,
producing a new sorted list. Uses comparator cmp above
merge: (listof X) (listof X) → (listof X)
requires: slst1 and slst2 are sorted using cmp

(define (merge slst1 slst2)
  (cond [(empty? slst1) slst2]
        [(empty? slst2) slst1]
        [(cmp (first slst1) (first slst2))
         (cons (first slst1) (merge (rest slst1) slst2))]
        [else (cons (first slst2) (merge slst1 (rest slst2)))]))

(define cutoff (quotient (length lst) 2))
(define left (take-left lst cutoff))
(define right (take-right lst cutoff))

(merge (mergesort left cmp) (mergesort right cmp)))
Graphs
Adjacency List Representation

Recall the data definitions for Node and Graph from lectures.

;; A Node is a Sym

;; A Graph is a (listof (list Node (listof Node)))
Adjacency List Representation

'(A (B C D))
(B (E))
(C ()
(D (B C))
(E ()

How can we compute the number of nodes in a Graph G?
Adjacency List Representation

\[
\left((A \ (B \ C \ D)) \right) \\
(B \ (E)) \\
(C \ ()) \\
(D \ (B \ C)) \\
(E \ ()))
\]

How can we compute the number of nodes in a Graph G?

(length G)
Adjacency List Representation

\`
\((A \ (B \ C \ D))
(B \ (E))
(C \ ())
(D \ (B \ C))
(E \ ()))\`

How can we compute the number of edges in a Graph G?
How can we compute the number of edges in a Graph G?

Sum the lengths of the second list in each pair. One possible answer is:

\[
\text{(foldr } + 0 \text{ }
\text{(map (lambda (pair) (length (second pair)))))}
\]

\text{G})
Adding a Node to a Graph

Write a function `add-node` that consumes a Graph G, a Node node, list of in-neighbours, list of out-neighbours and produces a new Graph with new Node node added to a graph.

```
(add-node '((A (B C D))
            (B (E))
            (C ())
            (D (B C))
            (E ())
            'F '(A B) '(D E))

should produce

'((F (D E))
  (A (F B C D))
  (B (F E))
  (C ())
  (D (B C))
  (E ())
)
Adding a Node to a Graph

;; add-node: Graph Node (listof Node) (listof Node) → Graph

(define (add-node G node in-neighbours out-neighbours)
  (cons (list node out-neighbours)
        (map (lambda (x)
               (list (first x)
                   (cond[(member? (first x) in-neighbours)
                           (cons node (second x))]
                 [else (second x)])))
                G)))
Graph Union

Write a function `graph-union` that consumes two Graphs `g1` and `g2`, and produces a Graph that contains all the nodes and edges of `g1` and `g2`. The produced Graph should not contain any duplicate nodes or edges.

```
(graph-union '((A (B))
  (B ()))
  '((A (B C))
    (B (A C))
    '((A (B C)) should produce
    (B (A C))
    (B (A C))
    (C (A))))
```

Graph Union: Helpers We’ll Use

We will use two helper functions, both of which you’ve seen before:

- **dedup** from earlier in this review (and from Assignment 9)

- **neighbours** from Module 12 Slide 14
neighbours From Module 12 Slide 14

;; neighbours: Node Graph → (listof Node)
(define (neighbours v G)
  (cond [(empty? G) empty] ; minor change from Module 12
        [(symbol=? v (first (first G))) (second (first G))]
        [else (neighbours v (rest G))])))

(check-expect (neighbours 'B '((A (B C)) (B (A C)) (C (A))))
              '(A C))
Graph Union

;;; graph-union: Graph Graph → Graph
(define (graph-union g1 g2)
  (local [(define g1-nodes (map first g1))
           (define g2-nodes (map first g2))]
    (map (lambda (node)
           (list node
                 (dedup (append (neighbours node g1)
                               (neighbours node g2))))
                 (dedup (append g1-nodes g2-nodes))))))
Detecting Cycles

In lectures, we used a parameter to store the list of nodes we’ve visited so far. If the node we’re currently at is in the visited list, then there is a cycle.

Write a function has-cycle? that consumes a Graph G and produces true if G has a cycle in it and false otherwise.

We’ll use neighbours as a helper function again.

;; neighbours: Node Graph → (listof Node)
(define (neighbours v G)
  (cond [(empty? G) empty]
        [(symbol=? v (first (first G))) (second (first G))]
        [else (neighbours v (rest G))])))
Detecting Cycles

;;; has-cycle?: Graph → Bool
(define (has-cycle? G)
  (local [;;; has-cycle?/list: (listof Node) (listof Node) → Bool
    (define (has-cycle?/list lo-nodes visited)
      (cond [(empty? lo-nodes) false]
        [(member? (first lo-nodes) visited) true]
        [else (or (has-cycle?/list (neighbours (first lo-nodes) G)
                (cons (first lo-nodes) visited))
          (has-cycle?/list (rest lo-nodes) visited))])])
  (has-cycle?/list (map first G) empty)))
Length of a Longest Path
Recall from lectures that a sequence of nodes $v_1, v_2, \ldots, v_k$ is a path of length $k - 1$ if $(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k)$ are all edges.

Directed graphs without cycles are called directed acyclic graphs (DAGs).

Write a function `max-path-length` that consumes a DAG `dag` and produces the length of a longest path in `dag`. If `dag` has no paths, produce 0.
;;; max-path-length/list: (listof Node) Graph → Nat
(define (max-path-length/list l-nds dag)
  (cond
    [(empty? l-nds) 0]
    [else (local [(define nbrs (neighbours (first l-nds) dag))]
      (cond [(empty? nbrs)
        (max-path-length/list (rest l-nds) dag)]
        [else (max (add1 (max-path-length/list nbrs dag))
        (max-path-length/list (rest l-nds) dag))])))])

;;; max-path-length: Graph → Nat
(define (max-path-length dag)
  (max-path-length/list (map first dag) dag))
Other Graph Practise Questions

- Delete a Node from a Graph
- Find all nodes that can reach a given Node.
- Find a Node which is farthest from a given Node.
- Given a Graph G, produce a new Graph with the same nodes as G and all the edges that are not in G.