UMass Boston Computer Science
CS450 High Level Languages (section 2)
Abstraction 2

Monday, October 16, 2023
Logistics

• HW 3 in
  • due: Sun 10/15 11:59 pm EST

• HW 4 out
  • due: Sun 10/22 11:59 pm EST
List (Recursive) Data Definition 1

;; A ListofInt is one of:
;; - empty
;; - (cons Int ListofInt)
List (Recursive) Data Definition 1: F.n Template

;; A ListofInt is one of:
;; - empty
;; - (cons Int ListofInt)

;;;; TEMPLATE for list-fn
;;;; list-fn : ListofInt -> ???
(define (list-fn lst)
  (cond
     [(empty? lst) ....]
     [(cons? lst) .... (first lst) ....
      .... (list-fn (rest lst)) ....]])
Recursive List Fn Example 1: \texttt{inc-list}

\begin{itemize}
  \item Function design recipe:
    \begin{enumerate}
      \item Name
      \item Signature
      \item Description
      \item Examples
      \item Template
      \end{enumerate}
\end{itemize}

\begin{verbatim}
;; inc-list : ListofInt -> ListofInt
;; increments each list element by 1
(define (inc-list lst)
  (cond
    [(empty? lst) ....]
    [(cons? lst) .... (first lst) ....
      .... (inc-list (rest lst)) ....]])
\end{verbatim}

(check-equal? (inc-list (list 1 2 3))
  (list 2 3 4))
Recursive List Fn Example 1: inc-list

;; inc-list : ListofInt -> ListofInt
;; increments each list element by 1
(define (inc-list lst)
  (cond
   [(empty? lst) empty]
   [(cons? lst) .... (first lst) .... .... (inc-list (rest lst)) ....]])

Empty input produces empty output
(look at signature for help if needed)
Recursive List Fn Example 1: inc-list

;; inc-list : ListofInt -> ListofInt
;; increments each list element by 1
(define (inc-list lst)
  (cond
    [(empty? lst) empty]
    [else .... (add1 (first lst)) ....
     .... (inc-list (rest lst)) ....]]))

Call another function to process (first) (Int) list element
Recursive List Fn Example 1: inc-list

;; inc-list : ListofInt -> ListofInt
;; increments each list element by 1
(define (inc-list lst)
  (cond
   [(empty? lst) empty
    [else (cons (add1 (first lst))
                (inc-list (rest lst))))]))

Figure out how to “combine” with recursive call result
(look at signature for help if needed)
List (Recursive) Data Definition 2

;; A ListofBall is one of:
;;  - empty
;;  - (cons Ball ListofBall)
List (Recursive) Data Definition 2: Fn Template

;; A ListofBall is one of:
;; - empty
;; - (cons Ball ListofBall)

;; TEMPLATE for list-fn
;; list-fn : ListofBall -> ???
(define (list-fn lst)
  (cond
[(empty? lst) .....]
[(cons? lst) ..... (first lst) ..... 
  ..... (list-fn (rest lst)) ..... ])))
Recursive List Fn Example 2: \texttt{next-world}

Function design recipe:
1. Name
2. Signature
3. Description
4. Examples
5. Template
...

\texttt{;; next-world: ListofBall -> ListofBall}
\texttt{;; Updates position each ball by one tick}

(\texttt{define \textbf{(next-world lst)}})
(\texttt{(cond}}
  \texttt{[(empty? lst) ....]}
  \texttt{[(cons? lst) .... (first lst) .... .... (next-world (rest lst)) ....]})
Recursive List Fn Example 2: `next-world`

```scheme
;; next-world: ListofBall -> ListofBall
;; Updates position each ball by one tick
(define (next-world lst)
  (cond
   [(empty? lst) empty]
   [(cons? lst) .... (first lst) .... .... (next-world (rest lst)) ....]])
```

Empty input produces empty output
(look at signature for help if needed)
Recursive List Fn Example 2: \texttt{next-world}

\begin{verbatim}
(check-equal? (next-world (list (make-ball 0 0 1 1)))
  (list (next-ball (make-ball 0 0 1 1))))
\end{verbatim}

\texttt{;; next-world: ListofBall \rightarrow ListofBall}
\texttt{;; Updates position each ball by one tick}
\texttt{(define \texttt{(next-world lst)})}
\texttt{(cond}
\texttt{  [(empty? lst) empty]}
\texttt{  [else .... (???) (first lst)) ....}
\texttt{    .... (next-world (rest lst)) ....]])
Recursive List Fn Example 2: `next-world`

```
;; next-world: ListofBall -> ListofBall
;; Updates position each ball by one tick
(define (next-world lst)
  (cond
   [(empty? lst) empty]
   [else .... (next-ball (first lst)) ....
     .... (next-world (rest lst)) ....]])
```

Call another function to process (first)(Ball) list element
Recursive List Fn Example 2: next-world

;;; next-world: ListofBall -> ListofBall
;;; Updates position each ball by one tick
(define (next-world lst)
  (cond
    [(empty? lst) empty]
    [else (cons (next-ball (first lst))
               (next-world (rest lst)))]))
Comparison 1

;; inc-lst: ListofInt -> ListofInt
;; Returns list with each element incremented
(define (inc-lst lst)
  (cond
   [(empty? lst) empty]
   [else (cons (add1 (first lst))
                (inc-lst (rest lst)))]))

;; next-world : ListofBall -> ListofBall
;; Updates position of each ball by one tick
(define (next-world lst)
  (cond
   [(empty? lst) empty]
   [else (cons (next-ball (first lst))
               (next-world (rest lst)))]))
Abstraction: Common List Function #1

(define (lst-fn1 fn lst)
  (cond
   [(empty? lst) empty]
   [else (cons (fn (first lst))
                (lst-fn1 (rest lst)))]))

Make the difference a parameter of a (function) abstraction
Abstraction Recipe

1. **Find similar patterns** in a program
   - Minimum: 2
   - Ideally: 3+

2. **Identify differences** and make them parameters

3. **Create a reusable abstraction** with the discovered parameters
   - E.g., a function(al) abstraction
Abstraction: Common List Function #1

```scheme
;; lst-fn1: (?? -> ??) Listof?? -> Listof??
;; Applies the given fn to each element of given lst

(define (lst-fn1 fn lst)
  (cond
   [(empty? lst) empty]
   [else (cons (fn (first lst))
                (lst-fn1 (rest lst)))]))
```
Abstraction of Data Definitions

;;; A ListofInt is one of
;;; - empty
;;; - (cons Int ListofInt)

;;; A ListofBall is one of
;;; - empty
;;; - (cons Ball ListofBall)
Abstraction Recipe

1. **Find similar patterns** in a program
   - Minimum: 2
   - Ideally: 3+

2. **Identify differences** and **make them parameters**

3. **Create** a reusable abstraction with the discovered parameters
   - E.g., a function(al) abstraction
Abstraction of Data Definitions

;; A ListofInt is one of
;; - empty
;; - (cons Int ListofInt)

;; A ListofBall is one of
;; - empty
;; - (cons Ball ListofBall)
Abstraction Recipe

1. Find similar patterns in a program
   • Minimum: 2
   • Ideally: 3+

2. Identify differences and make them parameters

3. Create a reusable abstraction with the discovered parameters
   • E.g., a function(al) abstraction
   • E.g., a data abstraction
Abstraction of Data Definitions

;; A ListofInt is one of
;; - empty
;; - (cons Int ListofInt)

;; A ListofBall is one of
;; - empty
;; - (cons Ball ListofBall)

;; A Listof<X> is one of
;; - empty
;; - (cons X Listof<X>)
Abstraction: Common List Function #1

```
;; lst-fn1: [X -> Y] [Listof X] -> [Listof Y]
;; Applies the given fn to each element of given lst

;; lst-fn1: (X -> Y) Listof<X> -> Listof<Y>
;; Applies the given fn to each element of given lst

(define (lst-fn1 fn lst)
  (cond
   [(empty? lst) empty]
   [else (cons (fn (first lst))
              (lst-fn1 (rest lst))))])
```
Abstraction Recipe

1. **Find similar patterns** in a program
   - Minimum: 2
   - Ideally: 3+

2. **Identify differences** and make them parameters

3. **Create** a reusable abstraction with the discovered parameters
   - E.g., a function(al) abstraction
   - E.g., a data abstraction

4. **Use** the abstraction by giving concrete “arguments” parameters
Abstraction: Common List Function #1

;;; lst-fn1: (X -> Y) Listof<X> -> Listof<Y>
;;; Applies the given fn to each element of given lst

(define (lst-fn1 fn lst)
  (cond
   [(empty? lst) empty]
   [else (cons (fn (first lst))
    (lst-fn1 (rest lst)))]))

(define (inc-lst lst) (lst-fn1 add1 lst)
(define (next-world lst) (lst-fn1 next-ball lst))
Q: Do these functions follow the design recipe (template)?

A: They do. Because “arithmetic” is always allowed.

```
(define (inc-lst lst) (lst-fn1 add1 lst))
(define (next-world lst) (lst-fn1 next-ball lst))
```
;; lst-fn1: (X -> Y) Listof<X> -> Listof<Y>
;; Applies the given fn to each element of given lst

(define (lst-fn1 fn lst)
  (cond
   [(empty? lst) empty]
   [else (cons (fn (first lst))
                (lst-fn1 (rest lst)))]))

(define (inc-lst lst) (lst-fn1 add1 lst)
(define (next-world lst) (lst-fn1 next-ball lst))
Common List Function #1: map

;; map: (X -> Y) Listof<X> -> Listof<Y>
;; Applies the given fn to each element of given lst

(define (map fn lst)
  (cond
    [(empty? lst) empty]
    [else (cons (fn (first lst))
                (map (rest lst))))]))

(define (inc-lst lst) (map add1 lst)
(define (next-world lst) (map next-ball lst)}
Abstraction Recipe

1. **Find similar patterns** in a program
   - Minimum: 2
   - Ideally: 3+

2. **Identify differences** and make them parameters

3. **Create a reusable abstraction** with the discovered parameters
   - E.g., a function(al) abstraction
   - E.g., a data abstraction
   - • The abstraction **must** have a short, clear name and “be logical”

4. **Use** the abstraction by giving concrete “arguments” parameters
Abstraction Recipe

1. Find similar patterns in a program
   - Minimum: 2
   - Ideally: 3+
   - Not all “similar patterns” should be abstracted

2. Identify differences and make them parameters

3. Create a reusable abstraction
   - E.g., a function(al) abstraction
   - E.g., a data abstraction
   - The abstraction must have a short, clear name and “be logical”

4. Use the abstraction by giving concrete “arguments” parameters
Abstraction Warning Story

I came to see the following pattern:

1. **Programmer A** sees duplication.
2. **Programmer A** extracts duplication and gives it a name. *This creates a new abstraction.*
3. **Programmer A** replaces the duplication with the new abstraction. *Ah, the code is perfect. Programmer A trots happily away.*
4. Time passes ...

This, a million times this! ”@BonzoESC: “Duplication is far cheaper than the wrong abstraction” @sandimetz @rbonales”

https://sandimetz.com/blog/2016/1/20/the-wrong-abstraction
Abstraction Warning Story

I came to see the following pattern:
1. **Programmer A** sees duplication.
2. **Programmer A** extracts duplication and gives it a name. *This creates a new abstraction.*
3. **Programmer A** replaces the duplication with the new abstraction. *Ah, the code is perfect. Programmer A trots happily away.*

4. Time passes …

5. A new requirement appears for which the current abstraction is *almost* perfect.
6. **Programmer B** gets tasked to implement this requirement. *Programmer B tries to retain the existing abstraction, but it’s not perfect, so they alter the code to take a parameter, and then add extra logic that is conditionally based on the value of that parameter.*

https://sandimetz.com/blog/2016/1/20/the-wrong-abstraction
Abstraction Warning Story

I came to see the following pattern:

1. Programmer A sees duplication.
2. Programmer A extracts duplication and gives it a name.
3. Programmer B replaces duplication with the new abstraction.
   Ah, the code is perfect! Programmer A trots happily away.
4. Time passes...

5. A new requirement appears for which the current abstraction is almost perfect.
6. Programmer B gets tasked to implement this requirement.

   **Programmer B** tries to retain the existing abstraction, but it’s not perfect, so they alter the code to take a parameter, and then **add extra logic** that is conditionally based on the value of that parameter.

7. Another new requirement arrives. And a new **Programmer X**, who adds an additional parameter and a new conditional. **Loop until code becomes incomprehensible.**

8. **You** appear in the story about here, and your life takes a dramatic turn for the worse.

---

https://sandimetz.com/blog/2016/1/20/the-wrong-abstraction
Program Design Recipe

Data Designs → Function Designs
Abstraction Warning Story

I came to see the following pattern:
1. Programmer A sees duplication.
2. Programmer A extracts duplication and gives it a name.
3. Programmer A incorporates the duplication with the new abstraction.
   Ah, the code is perfect. Programmer A trots happily away.
4. Time passes...
5. A new requirement appears for which the current code abstraction is almost perfect.
6. Programmer B gets tasked to implement this requirement.
   Programmer B tries to take a parameter, and then adds extra logic that is conditionally based on the value of that parameter.
7. Another new requirement arrives. And a new Programmer X, who adds an additional parameter and a new conditional. Loop until code becomes incomprehensible.
8. You appear in the story about here, and your life takes a dramatic turn for the worse.

How to avoid?

Always be thinking about the data

Don’t focus only on “getting the code working”

These programmers only cared about “getting the code working”
Common List Function #2: ???
Comparison #2

```lisp
;; sum-lst: ListofInt -> Int
(define (sum-lst lst)
  (cond
   [(empty? lst) 0]
   [else (+ (first lst)
            (sum-lst (rest lst)))])))

;; render-world : ListofBall -> Image
(define (render-world lst)
  (cond
   [(empty? lst) EMPTY-SCENE]
   [else (place-ball (first lst)
                     (render-world (rest lst)))]))
```
Abstraction Recipe

1. **Find similar patterns** in a program
   - Minimum: 2
   - Ideally: 3+

2. **Identify differences** and make them parameters

3. **Create** a reusable abstraction with the discovered parameters
   - E.g., a function(al) abstraction
   - E.g., a data abstraction
   - The abstraction **must** have a short, clear name and “be logical”

4. **Use** the abstraction by giving concrete “arguments” parameters
Comparison #2

;;; sum-lst: ListofInt -> Int
(define (sum-lst lst)
  (cond
    [(empty? lst) 0]
    [else (+ (first lst)
              (sum-lst (rest lst)))]))

;;; render-world : ListofBall -> Image
(define (render-world lst)
  (cond
    [(empty? lst) EMPTY-SCENE]
    [else (place-ball (first lst)
                      (render-world (rest lst)))]))
```scheme
;; list-fn2 : (X Y -> Y) Y Listof<X> -> Y

(define (lst-fn2 fn initial lst)
  (cond
   [(empty? lst) initial]
   [else (fn (first lst) (lst-fn2 fn initial (rest lst)))]))
```

- X = Type of list element
- Y = Result Type
Abstraction Recipe

1. Find similar patterns in a program
   - Minimum: 2
   - Ideally: 3+

2. Identify differences and make them parameters

3. Create a reusable abstraction with the discovered parameters
   - E.g., a function(al) abstraction
   - E.g., a data abstraction
   - The abstraction must have a short, clear name and “be logical”

4. Use the abstraction by giving concrete “arguments” parameters
Common List Function #2: \texttt{foldr}

Also called “reduce”
Because a list of values is “reduced” to one value

;; \texttt{foldr}: (X Y -> Y) Y Listof<X> -> Y
(define (foldr fn initial lst)
  (cond
   [(empty? lst) initial]
   [else (fn (first lst) (foldr fn initial (rest lst)))]))
Abstraction Recipe

1. **Find similar patterns** in a program
   - Minimum: 2
   - Ideally: 3+

2. **Identify differences** and make them parameters

3. **Create a reusable abstraction** with the discovered parameters
   - E.g., a function(al) abstraction
   - E.g., a data abstraction
   - The abstraction **must** have a short, clear name and “be logical”

4. **Use** the abstraction by giving concrete “arguments” parameters
Common List Function #2: $\text{foldr}$

$$\text{define} \ (\text{foldr} \ \text{fn} \ \text{initial} \ \text{lst})$$

$$\left(\text{cond}\right)$$

$$\left[[\text{(empty? lst) initial}]\right]$$

$$\left[[\text{else (fn (first lst) (foldr fn initial (rest lst))})]\right]\]$$

$$\text{define} \ (\text{sum-lst} \ \text{lst}) \ (\text{foldr} + 0 \ \text{lst})$$

$$\text{define} \ (\text{render-world} \ \text{lst}) \ (\text{foldr} \ \text{place-ball} \ \text{EMPTY-SCENE} \ \text{lst})$$
Do we always want to start at the right?

For some functions, order doesn’t matter, but for others, it does?

\[(\text{foldr } + 0 \ (\text{list } 1 \ 2 \ 3)) = (1 + (2 + (3 + 0)))\]

\[(1 + (2 + (3 + 0))) = (((1 + 0) + 2) + 3)\]

\[(1 - (2 - (3 - 0))) \neq (((1 - 0) - 2) - 3)\]

(Addition is associative)
Need List Function #2b: `foldl` (start from left)

Challenge:
- **Change `foldr` to `foldl`**
- so that the function is applied from the left (first element first)

\[
\begin{align*}
\text{(define (foldr fn initial lst)} &\quad (\text{cond}) \\
\quad &\quad [(\text{empty? lst) initial}] \\
\quad &\quad [\text{else (fn (first lst) (foldr fn initial (rest lst)))]})
\end{align*}
\]

\[
\begin{align*}
\text{(define (foldl fn initial lst)} &\quad (\text{cond}) \\
\quad &\quad [(\text{empty? lst) .... }] \\
\quad &\quad [\text{else .... (first lst) .... (foldl fn initial (rest lst))}]})
\end{align*}
\]
Need List Function #2b: \texttt{foldl} (start from left)

\begin{verbatim}
;;; foldr: (X Y -> Y) Y Listof<X> -> Y
(define (foldr fn initial lst)
  (cond
   [(empty? lst) initial]
   [else (fn (first lst) (foldr fn initial (rest lst)))]))
\end{verbatim}

Y = Result Type

Expressions with needed “result” type:
- initial
- \texttt{fn} call
- recursive call itself

(look at signature to help)

\begin{verbatim}
;;; foldl: (X Y -> Y) Y Listof<X> -> Y
(define (foldl fn initial lst)
  (cond
   [(empty? lst) ....]
   [else .... (first lst) .... (foldl fn initial (rest lst)) ....]])
\end{verbatim}
Need List Function #2b: **foldl** (start from left)

;;; foldl: \((X Y \to Y)\) \(Y\) Listof\(<X>\) \(\to Y\)

\[
\begin{align*}
\text{(define } \text{foldl fn initial lst)} \\
\text{(cond)} \\
\quad \text{[(empty? lst) initial]} \\
\quad \text{[else (fn (first lst) (foldl fn initial (rest lst)))]]}
\end{align*}
\]

Y = Result Type

Expressions with needed “result” type:
- initial
- fn call
- recursive call itself

(look at signature to help)

;;; foldr: \((X Y \to Y)\) \(Y\) Listof\(<X>\) \(\to Y\)

\[
\begin{align*}
\text{(define } \text{foldr fn initial lst)} \\
\text{(cond)} \\
\quad \text{[(empty? lst) initial]} \\
\quad \text{[else (fn (first lst) (foldr fn initial (rest lst)))]]}
\end{align*}
\]

Now fill in args to recursive call

Y = Result Type

(look at signature to help)
Need List Function #2b: **foldl** (start from left)

```scheme
;; foldr: (X Y -> Y) Y Listof<X> -> Y
(define (foldr fn initial lst)
  (cond
    [(empty? lst) initial]
    [else (fn (first lst) (foldr fn initial (rest lst)))]))

;; foldl: (X Y -> Y) Y Listof<X> -> Y
(define (foldl fn initial lst)
  (cond
    [(empty? lst) ....]
    [else (foldl fn .... (first lst) .... (rest lst))])))
```

- **foldr**: Function that folds a list from right to left.
- **foldl**: Function that folds a list from left to right.

In `foldl`, there is an additional argument that is only present for the first element of the list, indicating that the first argument is treated specially.
Need List Function #2b: foldl (start from left)

;; foldr: (X Y -> Y) Y Listof<X> -> Y
(define (foldr fn initial lst)
  (cond
    [(empty? lst) initial]
    [else (fn (first lst) (foldr fn initial (rest lst)))])))

Expressions with needed “result” Y type:
- initial
- fn call
- recursive call itself

;; foldl: (X Y -> Y) Y Listof<X> -> Y
(define (foldl fn initial lst)
  (cond
    [(empty? lst) ....]
    [else (foldl fn .... (first lst) .... (rest lst)))])))

Now just need middle arg (and need to use the “first” piece)

“rest” of list has proper “list” type
Need List Function #2b: foldl (start from left)

\[ \text{foldr: } (X Y \rightarrow Y) Y \text{ Listof}<X> \rightarrow Y \]

\[
\text{(define } \text{foldr fn initial lst)}
\text{ (cond}
\text{ [\text{empty? lst) initial]}
\text{ [else (fn \text{first lst) (foldr fn initial (rest lst))}]})
\]

\[ \text{foldl: } (X Y \rightarrow Y) Y \text{ Listof}<X> \rightarrow Y \]

\[
\text{(define } \text{foldl fn initial lst)}
\text{ (cond}
\text{ [\text{empty? lst) ....]}
\text{ [else (foldl fn (fn \text{first lst) ....) (rest lst))}]})
\]

Expressions with needed “result” Y type:
- initial
- fn call
- recursive call itself

Now just need middle arg (and need to use the “first” piece)

\[ (((1 + 0) + 2) + 3) \]

What goes here? (look at signature)
(\text{and examples})
Need List Function #2b: foldl (start from left)

;;; foldr: (X Y -> Y) Y Listof<X> -> Y
(define (foldr fn initial lst)
  (cond
   [(empty? lst) initial]
   [else (fn (first lst) (foldr fn initial (rest lst)))]))

;;; foldl: (X Y -> Y) Y Listof<X> -> Y
(define (foldl fn initial lst)
  (cond
   [(empty? lst) ....]
   [else (foldl fn (fn (first lst) initial) (rest lst))])))

Expressions with needed “result” Y type:
- initial
- fn call
- recursive call itself

(((1 + 0) + 2) + 3)
Need List Function #2b: \texttt{foldl} (start from left)

\begin{align*}
\text{;; foldl: (X Y -> Y) Y Listof<X> -> Y} \\
&\text{(define (foldl fn initial lst)} \\
&\text{ (cond)} \\
&\text{ [((empty? lst) initial]} \\
&\text{ [else (fn (first lst) (foldl fn initial (rest lst)))]})
\end{align*}

\begin{align*}
\text{;; foldr: (X Y -> Y) Y Listof<X> -> Y} \\
&\text{(define (foldr fn initial lst)} \\
&\text{ (cond)} \\
&\text{ [((empty? lst) initial]} \\
&\text{ [else (fn (first lst) (foldr fn initial (rest lst)))]})
\end{align*}
Need List Function #2b: foldl (start from left)

;; foldr: (X Y -> Y) Y Listof<X> -> Y
(define (foldr fn initial lst)
  (cond
   [(empty? lst) initial]
   [else (fn (first lst) (foldr fn initial (rest lst)))]))

;; foldl: (X Y -> Y) Y Listof<X> -> Y
(define (foldl fn result-so-far lst)
  (cond
   [(empty? lst) result-so-far]
   [else (foldl fn (fn (first lst) result-so-far) (rest lst))])))

Expressions with needed “result” Y type:
- initial
- fn call
- recursive call itself

“result so far”

(((1 + 0) + 2) + 3)
Common list function #3
Your tasks

Write the following functions:

**(smaller-than):** ListofInt Int -> ListofInt

- Returns a list containing elements of given list that are less than the given int

**(larger-than):** ListofInt Int -> ListofInt

- Returns a list containing elements of given list that are greater than the given int

**(quicksort):** ListofInt -> ListofInt

- Sorts a given list (with no dups) in ascending order

```scheme
(define (quicksort lst)
  (define pivot (random lst))
  (append (quicksort (smaller-than lst pivot)) pivot (quicksort (greater-than lst pivot))))
```
Your tasks

(define (smaller-than lst x)
  (cond
   [(empty? lst) empty]
   [else (if (< (first lst) x)
           (cons (first lst) (smaller-than (rest lst) x))
           (smaller-than (rest lst) x))])))

(define (larger-than lst x)
  (cond
   [(empty? lst) empty]
   [else (if (> (first lst) x)
           (cons (first lst) (larger-than (rest lst) x))
           (smaller-than (rest lst) x))])))
Abstraction Recipe

1. **Find similar patterns** in a program
   - Minimum: 2
   - Ideally: 3+

2. **Identify differences** and make them parameters

3. **Create** a reusable abstraction with the discovered parameters
   - E.g., a function(al) abstraction
   - E.g., a data abstraction
   - The abstraction **must** have a short, clear name and “be logical”

4. **Use** the abstraction by giving concrete “arguments” parameters
Your tasks

(define (smaller-than lst x)
  (cond
   [(empty? lst) empty]
   [else (if (< (first lst) x)
           (cons (first lst) (smaller-than (rest lst) x))
           (smaller-than (rest lst) x))]]
)

(define (larger-than lst x)
  (cond
   [(empty? lst) empty]
   [else (if (> (first lst) x)
           (cons (first lst) (larger-than (rest lst) x))
           (larger-than (rest lst) x))])))
Common list function #3?

Is this a “good” abstraction?

```
;; \texttt{lst-fn3}: \texttt{ListofInt Int (Int Int \to Boolean) \to ListofInt}
;; \text{Returns a list containing elements of given list}
;; \text{that are \underline{???} than the given int}

(define \texttt{(lst-fn3 \texttt{lst x fn})})
  (cond
    [(empty? \texttt{lst}) empty]
    [else (if (fn? (first \texttt{lst}) x)
        (cons (first \texttt{lst}) (\texttt{lst-fn3 (rest lst) x}))
        (\texttt{lst-fn3 (rest lst) x}))])
```
Abstraction Recipe

1. **Find similar patterns** in a program
   - Minimum: 2
   - Ideally: 3+

2. **Identify differences** and make them parameters

3. **Create a reusable abstraction** with the discovered parameters
   - E.g., a function(al) abstraction
   - E.g., a data abstraction
   - The abstraction **must** have a short, clear name and “be logical”

4. **Use** the abstraction by giving concrete “arguments” parameters
Abstraction Recipe

1. Find similar patterns in a program
   - Minimum: 2
   - Ideally: 3+

2. Identify differences and make them parameters

3. Create a reusable abstraction with the discovered parameters
   - E.g., a function(al) abstraction
   - E.g., a data abstraction
   - The abstraction must have a short, clear name and “be logical”

4. Use the abstraction by giving concrete “arguments” parameters
Common list function #3?

Is this a “good” abstraction?

What are possible use cases?

Should be more than just the two examples we are abstracting

`; lst-fn3: ListofInt Int (Int Int -> Boolean) -> ListofInt
`; Returns a list containing elements of given list
`; that are ?? than the given int

(define (lst-fn3 lst x fn?)
  (cond
   [(empty? lst) empty]
   [else (if (fn? (first lst) x)
           (cons (first lst) (lst-fn3 (rest lst) x))
           (lst-fn3 (rest lst) x))]]))
More tasks

Write the following functions:

;; shorter-than: ListofString Int -> ListofString
;; Returns a list containing elements of given list
;; that have \texttt{length} less than the given int

(check-equal?
  (shorter-than (list "a" "bc" "abc") 2)
  (list "a"))

;; shorter-than-str: ListofString String -> ListofString
;; Returns a list containing elements of given list
;; that have \texttt{length} less than the given \texttt{string}

(check-equal?
  (shorter-than-str (list "a" "bc" "abc") "xy")
  (list "a"))
Write the following functions:

`; lst-fn3: ListofInt Int (Int Int -> Boolean) -> ListofInt
; Returns a list containing elements of given list
; that are ?? than the given int`

`; shorter-than: ListofString Int -> ListofString
; Returns a list containing elements of given list
; that have length less than the given int`

Could these be implemented with our new abstraction?

`; shorter-than-str: ListofString String -> ListofString
; Returns a list containing elements of given list
; that have length less than the given string`
Abstraction Recipe

1. **Find similar patterns** in a program
   - Minimum: 2
   - Ideally: 3+

2. **Identify differences** and make them parameters

3. **Create a reusable abstraction** with the discovered parameters
   - E.g., a function(al) abstraction
   - E.g., a data abstraction
   - The abstraction **must** have a short, clear name and “be logical”

4. **Use** the abstraction by giving concrete “arguments” parameters
Abstraction Recipe

1. **Find similar patterns** in a program
   - Minimum: 2
   - Ideally: 3+

2. **Identify differences** and make them parameters

3. **Create** a reusable abstraction with the discovered parameters
   - E.g., a function(al) abstraction
   - E.g., a data abstraction
   - The abstraction **must** have a short, clear name and “be logical”

4. **Use** the abstraction by giving concrete “arguments” parameters

Remember: The Design Recipe (like good software development) is **iterative!**
Common list function #3?

```scheme
(define (lst-fn3 lst x fn?)
  (cond
    [(empty? lst) empty]
    [else (if (fn? (first lst) x)
              (cons (first lst) (lst-fn3 (rest lst) x))
              (lst-fn3 (rest lst) x))])
)
```

Is this a “good” abstraction?

```scheme
;; lst-fn3: ListofInt Int (Int Int -> Boolean) -> ListofInt
;; Returns a list containing elements of given list
;; that are ???> than the given int
```
A Better common list function #3?

;; lst-fn3: Listof<X> (X -> Boolean) -> Listof<X>
;; Returns a list containing elements of given list
;; for which the given predicate returns true

(define (lst-fn3 lst other-int-param general-pred?))
  (cond
    [(empty? lst) empty]
    [else (if (general-pred? (first lst))
      (cons (first lst) (lst-fn3 (rest lst)))))
      (lst-fn3 (rest lst)))]))
Common list function #3: \texttt{filter}

\begin{verbatim}
;; smaller-than: Listof<Int> Int -> Listof<Int>
;; Returns a list containing elements of given list \textbf{less} than the given int

(define (smaller-than lst thresh)
  (filter (lambda (x) (< x thresh)) lst))

lambda creates an anonymous “inline” function (expression)

;; filter: Listof<X> (X -> Boolean) -> Listof<X>
;; Returns a list containing elements of given list
;; for which the \textbf{given predicate returns true}

(define (filter lst pred?)
  (cond
   [(empty? lst) empty]
   [else (if (pred? (first lst))
            (cons (first lst) (filter (rest lst)))
            (filter (rest lst)))]]

\end{verbatim}
Common list function #3: filter

;; smaller-than: Listof<Int> Int -> Listof<Int>
;; Returns a list containing elements of given list less than the given int
(define (smaller-than lst thresh)
  (filter (lambda (x) (< x thresh)) lst))

(lambda creates an anonymous “inline” function (expression))

;; filter: Listof<X> (X -> Boolean) -> Listof<X>
;; Returns a list containing elements of for which the given predicate returns
(define (filter lst pred?)
  (cond
   [(empty? lst) empty]
   [else (if (pred? (first lst))
            (cons (first lst) (filter (filter (rest lst))))
            )]]))

(lambda rules:
- Can skip the design recipe steps, BUT
- name, description, and signature must be “obvious”
- code is arithmetic only
- otherwise, create standalone function define
Your Remaining tasks

Implement with filter

```haskell
;; smaller-than: ListofInt Int -> ListofInt
;; Returns list containing elements of given list less than the given int

;; larger-than: ListofInt Int -> ListofInt
;; Returns list containing elements of given list greater than the given int

;; shorter-than: ListofString Int -> ListofString
;; Returns list containing elements of given list with length less than given int

;; shorter-than-str: ListofString String -> ListofString
;; Returns list containing elements of given list with length less than given string
```
Check-In Quiz 10/16
on gradescope

(due 1 minute before midnight)