Part IV: Abstraction

Marco T. Morazán

Seton Hall University

Outline

Functional Abstraction

- Chapter 3 discusses how to abstract over similar expressions to eliminate repetitions
- Section 68 discusses how to eliminate repetitions among data definitions,

Functional Abstraction

- Chapter 3 discusses how to abstract over similar expressions to eliminate repetitions
- Section 68 discusses how to eliminate repetitions among data definitions,
- Now we discuss *functional abstraction*: how to eliminate repetitions among functions
- This leads to powerful *abstract functions* (or generic functions) that can perform many different jobs for us

Functional Abstraction

- Chapter 3 discusses how to abstract over similar expressions to eliminate repetitions
- Section 68 discusses how to eliminate repetitions among data definitions,
- Now we discuss *functional abstraction*: how to eliminate repetitions among functions
- This leads to powerful *abstract functions* (or generic functions) that can perform many different jobs for us
- It will also conceptually change the meaning of data in your mind. As we shall see functions are data just like a number, a list, and an image are data
- The conceptual line between data and functions is imaginary

Functional Abstraction

- Chapter 3 discusses how to abstract over similar expressions to eliminate repetitions
- Section 68 discusses how to eliminate repetitions among data definitions,
- Now we discuss *functional abstraction*: how to eliminate repetitions among functions
- This leads to powerful *abstract functions* (or generic functions) that can perform many different jobs for us
- It will also conceptually change the meaning of data in your mind. As we shall see functions are data just like a number, a list, and an image are data
- The conceptual line between data and functions is imaginary
- If you already have code that works, why would you want to perform abstraction to eliminate differences?

Functional Abstraction

- Chapter 3 discusses how to abstract over similar expressions to eliminate repetitions
- Section 68 discusses how to eliminate repetitions among data definitions,
- Now we discuss *functional abstraction*: how to eliminate repetitions among functions
- This leads to powerful *abstract functions* (or generic functions) that can perform many different jobs for us
- It will also conceptually change the meaning of data in your mind. As we shall see functions are data just like a number, a list, and an image are data
- The conceptual line between data and functions is imaginary
- If you already have code that works, why would you want to perform abstraction to eliminate differences?
- By eliminating repetition among functions you make your code easier to understand and maintain, shorter, and more elegant
- It allows you to think about solutions to problems more abstractly, thus, making problem solving and programming easier

Functional Abstraction

- Consider the two functions are nearly identical:

```
;; contains-laptop?: (listof symbol) → Boolean
;; Purpose: Determine if list contains 'laptop
(define (contains-laptop? a-los)
  (and (not (empty a-los))
       (or  (symbol=? (first a-los) 'laptop) (contains-laptop? (rest a-los)))))
;; contains-pen?: (listof symbol) → Boolean  Purpose: Determine if list contains 'pen
(define (contains-pen? a-los)
  (and (not (empty a-los))
       (or  (symbol=? (first a-los) 'pen) (contains-pen? (rest a-los)))))
```

- They only vary in the use of the symbol `'laptop` or `'pen`

Functional Abstraction

- Consider the two functions are nearly identical:

```
;; contains-laptop?: (listof symbol) → Boolean
;; Purpose: Determine if list contains 'laptop
(define (contains-laptop? a-los)
  (and (not (empty a-los))
       (or  (symbol=? (first a-los) 'laptop) (contains-laptop? (rest a-los)))))
;; contains-pen?: (listof symbol) → Boolean  Purpose: Determine if list contains 'pen
(define (contains-pen? a-los)
  (and (not (empty a-los))
       (or  (symbol=? (first a-los) 'pen) (contains-pen? (rest a-los)))))
```

- They only vary in the use of the symbol `'laptop` or `'pen`
- Use a variable to represent the symbol that varies and create a new *abstract function* that has a parameter for this varying symbol
- The other parameter in the original functions is still needed

Functional Abstraction

- Consider the two functions are nearly identical:

```
;; contains-laptop?: (listof symbol) → Boolean
;; Purpose: Determine if list contains 'laptop
(define (contains-laptop? a-los)
  (and (not (empty a-los))
       (or  (symbol=? (first a-los) 'laptop) (contains-laptop? (rest a-los)))))
;; contains-pen?: (listof symbol) → Boolean  Purpose: Determine if list contains 'pen
(define (contains-pen? a-los)
  (and (not (empty a-los))
       (or  (symbol=? (first a-los) 'pen) (contains-pen? (rest a-los)))))
```

- They only vary in the use of the symbol `'laptop` or `'pen`
- Use a variable to represent the symbol that varies and create a new *abstract function* that has a parameter for this varying symbol
- The other parameter in the original functions is still needed
- The abstraction step yields the following function:

```
;; contains?: symbol (listof symbol) → Boolean
;; Purpose: Determine if list contains symbol
(define (contains? a-symbol a-los)
  (and (not (empty a-los))
       (or  (symbol=? (first a-los) a-symbol) (contains? a-symbol (rest a-los)))))
```

Functional Abstraction

- Consider the two functions are nearly identical:

```
;; contains-laptop?: (listof symbol) → Boolean
;; Purpose: Determine if list contains 'laptop
(define (contains-laptop? a-los)
  (and (not (empty a-los))
       (or  (symbol=? (first a-los) 'laptop) (contains-laptop? (rest a-los)))))
;; contains-pen?: (listof symbol) → Boolean  Purpose: Determine if list contains 'pen
(define (contains-pen? a-los)
  (and (not (empty a-los))
       (or  (symbol=? (first a-los) 'pen) (contains-pen? (rest a-los)))))
```

- They only vary in the use of the symbol `'laptop` or `'pen`
- Use a variable to represent the symbol that varies and create a new *abstract function* that has a parameter for this varying symbol
- The other parameter in the original functions is still needed
- The abstraction step yields the following function:

```
;; contains?: symbol (listof symbol) → Boolean
;; Purpose: Determine if list contains symbol
(define (contains? a-symbol a-los)
  (and (not (empty a-los))
       (or  (symbol=? (first a-los) a-symbol) (contains? a-symbol (rest a-los)))))
```

- Refactor `contains-laptop?` and `contains-pen?`:

```
;; contains-laptop?: (listof symbol) → Boolean
;; Purpose: Determine if list contains 'laptop
(define (contains-laptop? a-los) (contains? 'laptop a-los))
;; contains-pen?: (listof symbol) → Boolean
;; Purpose: Determine if list contains 'pen
(define (contains-pen? a-los) (contains? 'pen a-los))
```

- Observe how much shorter and readable these functions are now
- There is only one recursive function instead of two

Functional Abstraction

- Used to write other functions:

```
;; contains-book?: (listof symbol) → Boolean
;; Purpose: Determine if the given list contains 'book
(define (contains-book? a-los) (contains? 'book a-los))

;; contains-keys?: (listof symbol) → Boolean
;; Purpose: Determine if the given list contains 'keys
(define (contains-keys? a-los) (contains? 'keys a-los))

;; contains-money?: (listof symbol) → Boolean
;; Purpose: Determine if the given list contains 'money
(define (contains-money? a-los) (contains? 'money a-los))
```

Functional Abstraction

- Used to write other functions:

```
;; contains-book?: (listof symbol) → Boolean
;; Purpose: Determine if the given list contains 'book
(define (contains-book? a-los) (contains? 'book a-los))

;; contains-keys?: (listof symbol) → Boolean
;; Purpose: Determine if the given list contains 'keys
(define (contains-keys? a-los) (contains? 'keys a-los))

;; contains-money?: (listof symbol) → Boolean
;; Purpose: Determine if the given list contains 'money
(define (contains-money? a-los) (contains? 'money a-los))
```

- This illustrates the power of abstract functions
- They can perform many different tasks which allows us to simplify code design and development

Functional Abstraction

Homework

- Problems: 226–227

Functional Abstraction

DR for Abstraction

- The goal for functional abstraction is to abstract over similar expressions in the body of different functions with a similar number of parameters

Functional Abstraction

DR for Abstraction

- The goal for functional abstraction is to abstract over similar expressions in the body of different functions with a similar number of parameters
- The design recipe for functional abstraction:

  Mark Differences Compare and mark the differences in the bodies of the functions

  Create the Abstraction Abstraction Step

  1. Define an abstract function that has the same number of parameters as the functions abstracted over plus a parameter for every difference identified in the previous step
  2. The body of the abstract function references the parameters for the differences instead of the values found in the functions abstracted over
  3. Make sure the signature takes into account that the new parameters' types may vary making the function generic

  Refactor Refactor the functions abstracted over to use the abstract function

Functional Abstraction

Functions as Values

- Consider computing the slope of a secant line
- A secant line crosses the graph of a function, `f(x)`, at exactly two points: `(x1 , f(x1))` and `(x2 , f(x2))`
- When the distance between `x1` and `x2` is very small, the slope of the secant line may be used as an approximation for the slope of `f(x)` at the midpoint between the two points
- Slope of a (secant) line:

$$\texttt{m} = \frac{f(x_2) - f(x_1)}{x_2 - x_1}$$

Functional Abstraction

Functions as Values

```
;; number number → number
;; Purpose: Compute f(x) = x^2 slope of secant line for given x-values
(define (x^2-sec-slope x1 x2) (/ (- (sqr x2) (sqr x1)) (- x2 x1)))

;; Sample expressions for x^2-sec-slope
(define MSEC1 (/ (- (sqr 4) (sqr 2)) (- 4 2)))
(define MSEC2 (/ (- (sqr 3.75) (sqr -2.03)) (- 3.75 -2.03)))
;; Tests using sample computations for x^2-sec-slope
(check-within (x^2-sec-slope 2 4)        MSEC1 0.01)
(check-within (x^2-sec-slope -2.03 3.75) MSEC2 0.01)
;; Tests using sample values for x^2-sec-slope
(check-within (x^2-sec-slope 1.5 3.2) 4.7 0.01)

;; number number → number
;; Purpose: Compute f(x) = x^3 slope of the secant line for given x-values
(define (x^3-sec-slope x1 x2) (/ (- (cube x2) (cube x1)) (- x2 x1)))

;; Sample expressions for x^2-sec-slope
(define MSEC3 (/ (- (cube 5)    (cube 1))    (- 5 1)))
(define MSEC4 (/ (- (cube 10.1) (cube -4.6)) (- 10.1 -4.6)))
;; Tests using sample computations for x^3-sec-slope
(check-within (x^3-sec-slope 1 5)       MSEC3 0.01)
(check-within (x^3-sec-slope -4.6 10.1) MSEC4 0.01)
;; Tests using sample values for x^3-sec-slope
(check-within (x^3-sec-slope 1.5 3.2) 17.29 0.01)
```

Functional Abstraction

Functions as Values

- ```
  ;; number number → number
  ;; Purpose: Compute f(x) = x^2 slope of secant line for given x-values
  (define (x^2-sec-slope x1 x2) (/ (- (sqr x2) (sqr x1)) (- x2 x1)))

  ;; Sample expressions for x^2-sec-slope
  (define MSEC1 (/ (- (sqr 4) (sqr 2)) (- 4 2)))
  (define MSEC2 (/ (- (sqr 3.75) (sqr -2.03)) (- 3.75 -2.03)))
  ;; Tests using sample computations for x^2-sec-slope
  (check-within (x^2-sec-slope 2 4)        MSEC1 0.01)
  (check-within (x^2-sec-slope -2.03 3.75) MSEC2 0.01)
  ;; Tests using sample values for x^2-sec-slope
  (check-within (x^2-sec-slope 1.5 3.2) 4.7 0.01)

  ;; number number → number
  ;; Purpose: Compute f(x) = x^3 slope of the secant line for given x-values
  (define (x^3-sec-slope x1 x2) (/ (- (cube x2) (cube x1)) (- x2 x1)))

  ;; Sample expressions for x^2-sec-slope
  (define MSEC3 (/ (- (cube 5)    (cube 1))    (- 5 1)))
  (define MSEC4 (/ (- (cube 10.1) (cube -4.6)) (- 10.1 -4.6)))
  ;; Tests using sample computations for x^3-sec-slope
  (check-within (x^3-sec-slope 1 5)       MSEC3 0.01)
  (check-within (x^3-sec-slope -4.6 10.1) MSEC4 0.01)
  ;; Tests using sample values for x^3-sec-slope
  (check-within (x^3-sec-slope 1.5 3.2) 17.29 0.01)
  ```
- `x^2-sec-slope` and `x^3-sec-slope` are very similar
  ```

Functional Abstraction

Functions as Values

```
;; number number → number
;; Purpose: Compute f(x) = x^2 slope of secant line for given x-values
(define (x^2-sec-slope x1 x2) (/ (- (sqr x2) (sqr x1)) (- x2 x1)))

;; Sample expressions for x^2-sec-slope
(define MSEC1 (/ (- (sqr 4) (sqr 2)) (- 4 2)))
(define MSEC2 (/ (- (sqr 3.75) (sqr -2.03)) (- 3.75 -2.03)))
;; Tests using sample computations for x^2-sec-slope
(check-within (x^2-sec-slope 2 4)        MSEC1 0.01)
(check-within (x^2-sec-slope -2.03 3.75) MSEC2 0.01)
;; Tests using sample values for x^2-sec-slope
(check-within (x^2-sec-slope 1.5 3.2) 4.7 0.01)

;; number number → number
;; Purpose: Compute f(x) = x^3 slope of the secant line for given x-values
(define (x^3-sec-slope x1 x2) (/ (- (cube x2) (cube x1)) (- x2 x1)))

;; Sample expressions for x^2-sec-slope
(define MSEC3 (/ (- (cube 5)    (cube 1))    (- 5 1)))
(define MSEC4 (/ (- (cube 10.1) (cube -4.6)) (- 10.1 -4.6)))
;; Tests using sample computations for x^3-sec-slope
(check-within (x^3-sec-slope 1 5)       MSEC3 0.01)
(check-within (x^3-sec-slope -4.6 10.1) MSEC4 0.01)
;; Tests using sample values for x^3-sec-slope
(check-within (x^3-sec-slope 1.5 3.2) 17.29 0.01)
```

- `x^2-sec-slope` and `x^3-sec-slope` are very similar
- Only significant difference is the function that is applied to an `x`-value

Functional Abstraction

Functions as Values

```
;; number number → number
;; Purpose: Compute f(x) = x^2 slope of secant line for given x-values
(define (x^2-sec-slope x1 x2) (/ (- (sqr x2) (sqr x1)) (- x2 x1)))

;; Sample expressions for x^2-sec-slope
(define MSEC1 (/ (- (sqr 4) (sqr 2)) (- 4 2)))
(define MSEC2 (/ (- (sqr 3.75) (sqr -2.03)) (- 3.75 -2.03)))
;; Tests using sample computations for x^2-sec-slope
(check-within (x^2-sec-slope 2 4)        MSEC1 0.01)
(check-within (x^2-sec-slope -2.03 3.75) MSEC2 0.01)
;; Tests using sample values for x^2-sec-slope
(check-within (x^2-sec-slope 1.5 3.2) 4.7 0.01)

;; number number → number
;; Purpose: Compute f(x) = x^3 slope of the secant line for given x-values
(define (x^3-sec-slope x1 x2) (/ (- (cube x2) (cube x1)) (- x2 x1)))

;; Sample expressions for x^2-sec-slope
(define MSEC3 (/ (- (cube 5)    (cube 1))    (- 5 1)))
(define MSEC4 (/ (- (cube 10.1) (cube -4.6)) (- 10.1 -4.6)))
;; Tests using sample computations for x^3-sec-slope
(check-within (x^3-sec-slope 1 5)       MSEC3 0.01)
(check-within (x^3-sec-slope -4.6 10.1) MSEC4 0.01)
;; Tests using sample values for x^3-sec-slope
(check-within (x^3-sec-slope 1.5 3.2) 17.29 0.01)
```

- `x^2-sec-slope` and `x^3-sec-slope` are very similar
- Only significant difference is the function that is applied to an `x`-value
- New: the difference is a function
- This means that functions must be data just like a natural number

Functions as Values

DR for Abstraction

- The next step of the design recipe is to create an abstract function
- The abstract function needs a parameter for the single difference, say `f`, and two parameters for the two `x`-values

Functions as Values

DR for Abstraction

- The next step of the design recipe is to create an abstract function
- The abstract function needs a parameter for the single difference, say `f`, and two parameters for the two `x`-values
- To write this function we need to denote a function in the signature
- This is accomplished by writing a signature for the type of function the parameter denotes
- In this case both `f(x) = x`^{2} and `f(x) = x`^{3} have the same signature: `number number` $\rightarrow$ `number`:

```
;; (number → number) number number → number
;; Purpose: Approximate the slope of the secant line between
;;          the given x-values for the given function
(define (f-sec-slope f x1 x2) (/ (- (f x2) (f x1)) (- x2 x1)))
```

Functions as Values

DR for Abstraction

- The next step of the design recipe is to create an abstract function
- The abstract function needs a parameter for the single difference, say `f`, and two parameters for the two `x`-values
- To write this function we need to denote a function in the signature
- This is accomplished by writing a signature for the type of function the parameter denotes
- In this case both `f(x) = x`^{2} and `f(x) = x`^{3} have the same signature: `number number` $\rightarrow$ `number`:

```
;; (number → number) number number → number
;; Purpose: Approximate the slope of the secant line between
;;          the given x-values for the given function
(define (f-sec-slope f x1 x2) (/ (- (f x2) (f x1)) (- x2 x1)))
```

- The third step of the design recipe has us refactor:

```
;; number number → number
;; Purpose: Approximate the slope of the secant line between
;;          the given x-values for f(x) = x^2
(define (x^2-secant-slope x1 x2) (f-sec-slope sqr x1 x2))

;; number number → number
;; Purpose: Approximate the slope of the secant line between
;;          the given x-values for f(x) = x^3
(define (x^3-secant-slope x1 x2) (f-sec-slope cube x1 x2))
```

- Observe that nothing changes except the bodies of the functions

Functional Abstraction

DR for Abstraction

- By now you must find it strange that this discussion has ignored writing tests for abstract functions
- There are two good reasons for this

Functional Abstraction

DR for Abstraction

- By now you must find it strange that this discussion has ignored writing tests for abstract functions
- There are two good reasons for this
- The first is the refactoring performed on the functions that are abstracted
- These refactored functions use the abstract function and have tests
- If the testing is thorough bugs in the abstract function will be caught by the existing tests

Functional Abstraction

DR for Abstraction

- By now you must find it strange that this discussion has ignored writing tests for abstract functions
- There are two good reasons for this
- The first is the refactoring performed on the functions that are abstracted
- These refactored functions use the abstract function and have tests
- If the testing is thorough bugs in the abstract function will be caught by the existing tests
- The second is that first-class functions push programmers against the limits of what is possible
- To directly test such a function the returned function and another function must be proven equivalent
- Testing the equivalence of functions is an *unsolvable problem*

Functional Abstraction

Abstraction Over List-Processing Functions

- Chapter 13 discusses common operations over lists
- It is reasonable to expect functions that perform the same operation for different types of lists to be almost identical
- Makes them good candidates for abstraction
- Sometimes refactoring is needed

Functional Abstraction

List Summarizing

```
;; (listof quizgrade) → number Purpose: Compute sum of (listof quizgrade)
(define (sum-loq a-loq)
  (if (empty? a-loq) 0 (+ (first a-loq) (sum-loq (rest a-loq)))))

;; (listof quizgrade) → number Purpose: Compute length of (listof quizgrade)
(define (length-loq a-loq)
  (if (empty? a-loq) 0 (+ 1 (length-loq (rest a-loq)))))

;; (listof number) → number ;; Purpose: Compute product of (listof number)
(define (product-lon a-lon)
  (if (empty? a-lon) 1 (* (first a-lon) (product-lon (rest a-lon)))))

;; (listof string) → string ;; Purpose: Append strings in (listof string)
(define (append-los a-lostr)
  (if (empty? a-lostr) ""
      (string-append  (first a-lostr) (append-los (rest a-lostr)))))
```

Functional Abstraction

List Summarizing

- ```
  ;; (listof quizgrade) → number Purpose: Compute sum of (listof quizgrade)
  (define (sum-loq a-loq)
    (if (empty? a-loq) 0 (+ (first a-loq) (sum-loq (rest a-loq)))))

  ;; (listof quizgrade) → number Purpose: Compute length of (listof quizgrade)
  (define (length-loq a-loq)
    (if (empty? a-loq) 0 (+ 1 (length-loq (rest a-loq)))))

  ;; (listof number) → number ;; Purpose: Compute product of (listof number)
  (define (product-lon a-lon)
    (if (empty? a-lon) 1 (* (first a-lon) (product-lon (rest a-lon)))))

  ;; (listof string) → string ;; Purpose: Append strings in (listof string)
  (define (append-los a-lostr)
    (if (empty? a-lostr) ""
        (string-append  (first a-lostr) (append-los (rest a-lostr)))))
  ```
- Very similar, but not structurally identical
- `length-loq` does not explicitly use the first element of the list
- Instead of having an application expression involving the first list element it has the constant expression 1
  ```

Functional Abstraction

List Summarizing

```
;; (listof quizgrade) → number Purpose: Compute sum of (listof quizgrade)
(define (sum-loq a-loq)
  (if (empty? a-loq) 0 (+ (first a-loq) (sum-loq (rest a-loq)))))

;; (listof quizgrade) → number Purpose: Compute length of (listof quizgrade)
(define (length-loq a-loq)
  (if (empty? a-loq) 0 (+ 1 (length-loq (rest a-loq)))))

;; (listof number) → number ;; Purpose: Compute product of (listof number)
(define (product-lon a-lon)
  (if (empty? a-lon) 1 (* (first a-lon) (product-lon (rest a-lon)))))

;; (listof string) → string ;; Purpose: Append strings in (listof string)
(define (append-los a-lostr)
  (if (empty? a-lostr) ""
      (string-append  (first a-lostr) (append-los (rest a-lostr)))))
```

- Very similar, but not structurally identical
- `length-loq` does not explicitly use the first element of the list
- Instead of having an application expression involving the first list element it has the constant expression 1
- Code refactoring may be useful to create similarities
- Need an application expression that always evaluates to 1

Functional Abstraction

List Summarizing

- What function always yields 1?

Functional Abstraction

List Summarizing

- What function always yields 1?
- `f(x) = 1`

Functional Abstraction

List Summarizing

- What function always yields 1?
- `f(x) = 1`
- `length-loq` may be refactored to:

```
;; X → 1
;; Purpose: Return 1
(define (constant-f1 an-x) 1)

;; loq → number
;; Purpose: To compute the length of the given loq
(define (length-loq a-loq)
  (if (empty? a-loq)
      0
      (+ (constant-f1 (first a-loq))
         (length-loq (rest a-loq)))))
```

Functional Abstraction

List Summarizing

```
;; (listof quizgrade) → number Purpose: Compute sum of (listof quizgrade)
(define (sum-loq a-loq)
  (if (empty? a-loq) 0 (+ (first a-loq) (sum-loq (rest a-loq)))))

;; X → 1  Purpose: Return 1
(define (constant-f1 an-x) 1)

;; loq → number Purpose: Compute length of loq
(define (length-loq a-loq)
 (if (empty? a-loq) 0 (+ (constant-f1 (first a-loq)) (length-loq (rest a-loq)))))

;; (listof number) → number ;; Purpose: Compute product of (listof number)
(define (product-lon a-lon)
  (if (empty? a-lon) 1 (* (first a-lon) (product-lon (rest a-lon)))))

;; (listof string) → string ;; Purpose: Append strings in (listof string)
(define (append-los a-lostr)
  (if (empty? a-lostr) "" (string-append (first a-lostr)
                                         (append-los (rest a-lostr)))))
```

Functional Abstraction

List Summarizing

- ```
  ;; (listof quizgrade) → number Purpose: Compute sum of (listof quizgrade)
  (define (sum-loq a-loq)
    (if (empty? a-loq) 0 (+ (first a-loq) (sum-loq (rest a-loq)))))

  ;; X → 1  Purpose: Return 1
  (define (constant-f1 an-x) 1)

  ;; loq → number Purpose: Compute length of loq
  (define (length-loq a-loq)
   (if (empty? a-loq) 0 (+ (constant-f1 (first a-loq)) (length-loq (rest a-loq)))))

  ;; (listof number) → number ;; Purpose: Compute product of (listof number)
  (define (product-lon a-lon)
    (if (empty? a-lon) 1 (* (first a-lon) (product-lon (rest a-lon)))))

  ;; (listof string) → string ;; Purpose: Append strings in (listof string)
  (define (append-los a-lostr)
    (if (empty? a-lostr) "" (string-append (first a-lostr)
                                           (append-los (rest a-lostr)))))
  ```
- Now, `length-loq` uses its first element
- It is still, however, different from the other 3 functions: applies a function to its first element
  ```

Functional Abstraction

List Summarizing

- ```
  ;; (listof quizgrade) → number Purpose: Compute sum of (listof quizgrade)
  (define (sum-loq a-loq)
    (if (empty? a-loq) 0 (+ (first a-loq) (sum-loq (rest a-loq)))))

  ;; X → 1  Purpose: Return 1
  (define (constant-f1 an-x) 1)

  ;; loq → number Purpose: Compute length of loq
  (define (length-loq a-loq)
   (if (empty? a-loq) 0 (+ (constant-f1 (first a-loq)) (length-loq (rest a-loq)))))

  ;; (listof number) → number ;; Purpose: Compute product of (listof number)
  (define (product-lon a-lon)
    (if (empty? a-lon) 1 (* (first a-lon) (product-lon (rest a-lon)))))

  ;; (listof string) → string ;; Purpose: Append strings in (listof string)
  (define (append-los a-lostr)
    (if (empty? a-lostr) "" (string-append (first a-lostr)
                                           (append-los (rest a-lostr)))))
  ```
- Now, `length-loq` uses its first element
- It is still, however, different from the other 3 functions: applies a function to its first element
- To perform functional abstraction we need all 4 functions to apply a function to the first element
- Need a function that returns the value it gets as input.
  ```

Functional Abstraction

List Summarizing

- ```
  ;; (listof quizgrade) → number Purpose: Compute sum of (listof quizgrade)
  (define (sum-loq a-loq)
    (if (empty? a-loq) 0 (+ (first a-loq) (sum-loq (rest a-loq)))))

  ;; X → 1  Purpose: Return 1
  (define (constant-f1 an-x) 1)

  ;; loq → number Purpose: Compute length of loq
  (define (length-loq a-loq)
   (if (empty? a-loq) 0 (+ (constant-f1 (first a-loq)) (length-loq (rest a-loq)))))

  ;; (listof number) → number ;; Purpose: Compute product of (listof number)
  (define (product-lon a-lon)
    (if (empty? a-lon) 1 (* (first a-lon) (product-lon (rest a-lon)))))

  ;; (listof string) → string ;; Purpose: Append strings in (listof string)
  (define (append-los a-lostr)
    (if (empty? a-lostr) "" (string-append (first a-lostr)
                                           (append-los (rest a-lostr)))))
  ```
- Now, `length-loq` uses its first element
- It is still, however, different from the other 3 functions: applies a function to its first element
- To perform functional abstraction we need all 4 functions to apply a function to the first element
- Need a function that returns the value it gets as input.
- Such a function is the identity function: `f(x) = x`
  ```

Functional Abstraction

List Summarizing

```
;; X → X  Purpose: Return the given input
(define (id an-x) an-x)

;; (listof quizgrade) → number Purpose: Compute sum of (listof quizgrade)
(define (sum-loq a-loq)
  (if (empty? a-loq) 0 (+ (id (first a-loq)) (sum-loq (rest a-loq)))))

;; X → 1  Purpose: Return 1
(define (constant-f1 an-x) 1)

;; loq → number Purpose: Compute length of loq
(define (length-loq a-loq)
 (if (empty? a-loq) 0 (+ (constant-f1 (first a-loq)) (length-loq (rest a-loq)))))

;; (listof number) → number ;; Purpose: Compute product of (listof number)
(define (product-lon a-lon)
  (if (empty? a-lon) 1 (* (id (first a-lon)) (product-lon (rest a-lon)))))

;; (listof string) → string ;; Purpose: Append strings in (listof string)
(define (append-los a-lostr)
  (if (empty? a-lostr) "" (string-append (id (first a-lostr))
                                         (append-los (rest a-lostr)))))
```

Functional Abstraction

List Summarizing

- ```
  ;; X → X  Purpose: Return the given input
  (define (id an-x) an-x)

  ;; (listof quizgrade) → number Purpose: Compute sum of (listof quizgrade)
  (define (sum-loq a-loq)
    (if (empty? a-loq) 0 (+ (id (first a-loq)) (sum-loq (rest a-loq)))))

  ;; X → 1  Purpose: Return 1
  (define (constant-f1 an-x) 1)

  ;; loq → number Purpose: Compute length of loq
  (define (length-loq a-loq)
   (if (empty? a-loq) 0 (+ (constant-f1 (first a-loq)) (length-loq (rest a-loq)))))

  ;; (listof number) → number ;; Purpose: Compute product of (listof number)
  (define (product-lon a-lon)
    (if (empty? a-lon) 1 (* (id (first a-lon)) (product-lon (rest a-lon)))))

  ;; (listof string) → string ;; Purpose: Append strings in (listof string)
  (define (append-los a-lostr)
    (if (empty? a-lostr) "" (string-append (id (first a-lostr))
                                           (append-los (rest a-lostr)))))
  ```

- There are three difference among the four functions:
  - the base value
  - the function applied to the first element
  - the combinator
  ```

Functional Abstraction

List Summarizing

- Three ddifferences: base-val, ffirst, comb
- ;; (listof quizgrade) → number Purpose: Compute sum of loq

```
(define (sum-loq a-loq)
  (if (empty? a-loq)
      0
      (+ (id (first a-loq)) (sum-loq (rest a-loq)))))
```

Functional Abstraction

List Summarizing

- Three ddifferences: base-val, ffirst, comb
-
```
;; (listof quizgrade) → number Purpose: Compute sum of loq
(define (sum-loq a-loq)
  (if (empty? a-loq)
      0
      (+ (id (first a-loq)) (sum-loq (rest a-loq)))))
```
-
```
;; ? (? → ?) (? ? → ?) (listof ?) → ?
;; Purpose: Summarize given list
(define (accum base-val ffirst comb a-lox)
  (if (empty? a-lox)
      base-val
      (comb (ffirst (first a-lox))
            (accum base-val ffirst comb (rest a-lox)))))
```

Functional Abstraction

List Summarizing

```
;; ? (? → ?) (? ? → ?) (listof ?) → ?
;; Purpose: Summarize given list
(define (accum base-val ffirst comb a-lox)
  (if (empty? a-lox)
      base-val
      (comb (ffirst (first a-lox))
            (accum base-val ffirst comb (rest a-lox)))))
```

Functional Abstraction

List Summarizing

- ```
  ;; ? (? → ?) (? ? → ?) (listof ?) → ?
  ;; Purpose: Summarize given list
  (define (accum base-val ffirst comb a-lox)
    (if (empty? a-lox)
        base-val
        (comb (ffirst (first a-lox))
              (accum base-val ffirst comb (rest a-lox)))))
  ```
- Types of lists processed vary:

  ```
  ? (? → ?) (? ? → ?) (listof X) → ?
  ```
  ```

Functional Abstraction

List Summarizing

- ```
  ;; ? (? → ?) (? ? → ?) (listof ?) → ?
  ;; Purpose: Summarize given list
  (define (accum base-val ffirst comb a-lox)
    (if (empty? a-lox)
        base-val
        (comb (ffirst (first a-lox))
              (accum base-val ffirst comb (rest a-lox)))))
  ```
- Types of lists processed vary:

  ```
  ? (? → ?) (? ? → ?) (listof X) → ?
  ```
- The input to the function applied to the first element must be also of type `X`:

  ```
  ? (X → ?) (? ? → ?) (listof X) → ?
  ```
  ```

Functional Abstraction

List Summarizing

- ```
  ;; ? (? → ?) (? ? → ?) (listof ?) → ?
  ;; Purpose: Summarize given list
  (define (accum base-val ffirst comb a-lox)
    (if (empty? a-lox)
        base-val
        (comb (ffirst (first a-lox))
              (accum base-val ffirst comb (rest a-lox)))))
  ```
- Types of lists processed vary:

  `? (? → ?) (? ? → ?) (listof X) → ?`
- The input to the function applied to the first element must be also of type `X`:

  `? (X → ?) (? ? → ?) (listof X) → ?`
- The returned type of the functions abstracted over varies:

  `? (X → ?) (? ? → ?) (listof X) → Z`
  ```

Functional Abstraction

List Summarizing

- ```
  ;; ? (? → ?) (? ? → ?) (listof ?) → ?
  ;; Purpose: Summarize given list
  (define (accum base-val ffirst comb a-lox)
    (if (empty? a-lox)
        base-val
        (comb (ffirst (first a-lox))
              (accum base-val ffirst comb (rest a-lox)))))
  ```
- Types of lists processed vary:

  `? (? → ?) (? ? → ?) (listof X) → ?`
- The input to the function applied to the first element must be also of type `X`:

  `? (X → ?) (? ? → ?) (listof X) → ?`
- The returned type of the functions abstracted over varies:

  `? (X → ?) (? ? → ?) (listof X) → Z`
- The base value and the value returned by the combining function must also be of type `Z`:

  `Z (X → ?) (? ? → Z) (listof X) → Z`
  ```

Functional Abstraction

List Summarizing

- ```
  ;; ? (? → ?) (? ? → ?) (listof ?) → ?
  ;; Purpose: Summarize given list
  (define (accum base-val ffirst comb a-lox)
    (if (empty? a-lox)
        base-val
        (comb (ffirst (first a-lox))
              (accum base-val ffirst comb (rest a-lox)))))
  ```
- Types of lists processed vary:

  `? (? → ?) (? ? → ?) (listof X) → ?`
- The input to the function applied to the first element must be also of type `X`:

  `? (X → ?) (? ? → ?) (listof X) → ?`
- The returned type of the functions abstracted over varies:

  `? (X → ?) (? ? → ?) (listof X) → Z`
- The base value and the value returned by the combining function must also be of type Z:

  `Z (X → ?) (? ? → Z) (listof X) → Z`
- Result of the recursive call is the second input to the combining function:

  `Z (X → ?) (? Z → Z) (listof X) → Z`
  ```

Functional Abstraction

List Summarizing

- ```
  ;; ? (? → ?) (? ? → ?) (listof ?) → ?
  ;; Purpose: Summarize given list
  (define (accum base-val ffirst comb a-lox)
    (if (empty? a-lox)
        base-val
        (comb (ffirst (first a-lox))
              (accum base-val ffirst comb (rest a-lox)))))
  ```
- Types of lists processed vary:

  `? (? → ?) (? ? → ?) (listof X) → ?`
- The input to the function applied to the first element must be also of type `X`:

  `? (X → ?) (? ? → ?) (listof X) → ?`
- The returned type of the functions abstracted over varies:

  `? (X → ?) (? ? → ?) (listof X) → Z`
- The base value and the value returned by the combining function must also be of type Z:

  `Z (X → ?) (? ? → Z) (listof X) → Z`
- Result of the recursive call is the second input to the combining function:

  `Z (X → ?) (? Z → Z) (listof X) → Z`
- The type of value returned by the function applied to the first element of the list varies:

  `Z (X → Y) (Y Z → Z) (listof X) → Z`
  ```

Functional Abstraction

List Summarizing

- ```
  ;; ? (? → ?) (? ? → ?) (listof ?) → ?
  ;; Purpose: Summarize given list
  (define (accum base-val ffirst comb a-lox)
    (if (empty? a-lox)
        base-val
        (comb (ffirst (first a-lox))
              (accum base-val ffirst comb (rest a-lox)))))
  ```
- Types of lists processed vary:

  `? (? → ?) (? ? → ?) (listof X) → ?`
- The input to the function applied to the first element must be also of type `X`:

  `? (X → ?) (? ? → ?) (listof X) → ?`
- The returned type of the functions abstracted over varies:

  `? (X → ?) (? ? → ?) (listof X) → Z`
- The base value and the value returned by the combining function must also be of type Z:

  `Z (X → ?) (? ? → Z) (listof X) → Z`
- Result of the recursive call is the second input to the combining function:

  `Z (X → ?) (? Z → Z) (listof X) → Z`
- The type of value returned by the function applied to the first element of the list varies:

  `Z (X → Y) (Y Z → Z) (listof X) → Z`
- Signature defines a generic function
  ```

Functional Abstraction

List Summarizing

- ```
  ;; ? (? → ?) (? ? → ?) (listof ?) → ?
  ;; Purpose: Summarize given list
  (define (accum base-val ffirst comb a-lox)
    (if (empty? a-lox)
        base-val
        (comb (ffirst (first a-lox))
              (accum base-val ffirst comb (rest a-lox)))))
  ```
- Types of lists processed vary:

  `? (? → ?) (? ? → ?) (listof X) → ?`
- The input to the function applied to the first element must be also of type `X`:

  `? (X → ?) (? ? → ?) (listof X) → ?`
- The returned type of the functions abstracted over varies:

  `? (X → ?) (? ? → ?) (listof X) → Z`
- The base value and the value returned by the combining function must also be of type `Z`:

  `Z (X → ?) (? ? → Z) (listof X) → Z`
- Result of the recursive call is the second input to the combining function:

  `Z (X → ?) (? Z → Z) (listof X) → Z`
- The type of value returned by the function applied to the first element of the list varies:

  `Z (X → Y) (Y Z → Z) (listof X) → Z`
- Signature defines a generic function
- Just like functions have parameters signatures may also have parameters
- To declare type parameters for signatures we shall write the type variables inside angled brackets:

  `<X Y Z> Z (X → Y) (Y Z → Z) (listof X) → Z`
- Signature informs any reader that the types represented by `X`, `Y`, and `Z` only become known when arguments are provided to `accum`
  ```

Functional Abstraction

List Summarizing

- Refactor the functions abstracted over:

```
;; (listof quizgrade) → number
;; Purpose: To compute the length of the given loq
(define (length-loq a-loq) (accum 0 constant-f1 + a-loq))

;; (listof quizgrade) → number
;; Purpose: To compute the sum of the given loq
(define (sum-loq a-loq) (accum 0 id + a-loq))

;; (listof number) → number
;; Purpose: To compute the product of the given lon
(define (product-lon a-lon) (accum 1 id * a-lon))

;; (listof string) → string
;; Purpose: To append the strings in the given lostr
(define (append-los a-lostr) (accum "" id string-append a-lostr))
```

Functional Abstraction

List Summarizing

- Refactor the functions abstracted over:

```
;; (listof quizgrade) → number
;; Purpose: To compute the length of the given loq
(define (length-loq a-loq) (accum 0 constant-f1 + a-loq))

;; (listof quizgrade) → number
;; Purpose: To compute the sum of the given loq
(define (sum-loq a-loq) (accum 0 id + a-loq))

;; (listof number) → number
;; Purpose: To compute the product of the given lon
(define (product-lon a-lon) (accum 1 id * a-lon))

;; (listof string) → string
;; Purpose: To append the strings in the given lostr
(define (append-los a-lostr) (accum "" id string-append a-lostr))
```

- Take time to appreciate what has been achieved by using abstraction
- Instead of having 4 recursive functions there is only one recursive function

Functional Abstraction

List Summarizing

- Refactor the functions abstracted over:

```
;; (listof quizgrade) → number
;; Purpose: To compute the length of the given loq
(define (length-loq a-loq) (accum 0 constant-f1 + a-loq))

;; (listof quizgrade) → number
;; Purpose: To compute the sum of the given loq
(define (sum-loq a-loq) (accum 0 id + a-loq))

;; (listof number) → number
;; Purpose: To compute the product of the given lon
(define (product-lon a-lon) (accum 1 id * a-lon))

;; (listof string) → string
;; Purpose: To append the strings in the given lostr
(define (append-los a-lostr) (accum "" id string-append a-lostr))
```

- Take time to appreciate what has been achieved by using abstraction
- Instead of having 4 recursive functions there is only one recursive function
- Code is shorter and more elegant

Functional Abstraction

List Summarizing

- Refactor the functions abstracted over:

```
;; (listof quizgrade) → number
;; Purpose: To compute the length of the given loq
(define (length-loq a-loq) (accum 0 constant-f1 + a-loq))

;; (listof quizgrade) → number
;; Purpose: To compute the sum of the given loq
(define (sum-loq a-loq) (accum 0 id + a-loq))

;; (listof number) → number
;; Purpose: To compute the product of the given lon
(define (product-lon a-lon) (accum 1 id * a-lon))

;; (listof string) → string
;; Purpose: To append the strings in the given lostr
(define (append-los a-lostr) (accum "" id string-append a-lostr))
```

- Take time to appreciate what has been achieved by using abstraction
- Instead of having 4 recursive functions there is only one recursive function
- Code is shorter and more elegant
- After refactoring, the sample expressions no longer reflect the expressions abstracted over to write the bodies of the functions
- We are now using a new way (i.e., the abstract function) to compute the same value

Functional Abstraction

List Summarizing

- The most important achievement: a powerful function to perform list-summarizing computations

Functional Abstraction

List Summarizing

- The most important achievement: a powerful function to perform list-summarizing computations
- Extracting the x-values from a `(listof posn)`:

```
;; Sample instances of (listof posn)
(define ELOP '())
(define LOP1 (list (make-posn 1 9) (make-posn 2 8)
                   (make-posn 3 7) (make-posn 4 6)))
```

Functional Abstraction

List Summarizing

- The most important achievement: a powerful function to perform list-summarizing computations
- Extracting the x-values from a `(listof posn)`:

```
;; Sample instances of (listof posn)
(define ELOP '())
(define LOP1 (list (make-posn 1 9) (make-posn 2 8)
                   (make-posn 3 7) (make-posn 4 6)))
```

-
```
;; Sample expressions for get-xs-lop
(define ELOP-VAL (accum '() posn-x cons ELOP))
(define LOP1-VAL (accum '() posn-x cons LOP1))
```

- Observe that the sample expressions use `accum`

Functional Abstraction

List Summarizing

- The most important achievement: a powerful function to perform list-summarizing computations
- Extracting the x-values from a `(listof posn)`:

```
;; Sample instances of (listof posn)
(define ELOP '())
(define LOP1 (list (make-posn 1 9) (make-posn 2 8)
                   (make-posn 3 7) (make-posn 4 6)))
```

- `;; (listof posn)` → `lon Purpose: Extract x-values in (listof posn)`

```
(define (get-xs-lop a-lop)
```

-

```
;; Sample expressions for get-xs-lop
(define ELOP-VAL (accum '() posn-x cons ELOP))
(define LOP1-VAL (accum '() posn-x cons LOP1))
```

- Observe that the sample expressions use `accum`

Functional Abstraction

List Summarizing

- The most important achievement: a powerful function to perform list-summarizing computations
- Extracting the x-values from a `(listof posn)`:

```
;; Sample instances of (listof posn)
(define ELOP '())
(define LOP1 (list (make-posn 1 9) (make-posn 2 8)
                   (make-posn 3 7) (make-posn 4 6)))
```

- `;; (listof posn) → lon Purpose: Extract x-values in (listof posn)`
`(define (get-xs-lop a-lop)`

-
```
;; Sample expressions for get-xs-lop
(define ELOP-VAL (accum '() posn-x cons ELOP))
(define LOP1-VAL (accum '() posn-x cons LOP1))
```

-
```
;; Tests using sample computations for get-xs-lop
(check-expect (get-xs-lop ELOP) ELOP-VAL)
(check-expect (get-xs-lop LOP1) LOP1-VAL)

;; Tests using sample values for get-xs-lop
(check-expect (get-xs-lop (list (make-posn -10 2) (make-posn -5 7))
              '(-10 -5))
```

- Observe that the sample expressions use `accum`

Functional Abstraction

List Summarizing

- The most important achievement: a powerful function to perform list-summarizing computations
- Extracting the x-values from a `(listof posn)`:

```
;; Sample instances of (listof posn)
(define ELOP '())
(define LOP1 (list (make-posn 1 9) (make-posn 2 8)
                   (make-posn 3 7) (make-posn 4 6)))
```

- `;; (listof posn) → lon Purpose: Extract x-values in (listof posn)`
  `(define (get-xs-lop a-lop)`
- `  (accum '() posn-x cons a-lop))`
-
```
;; Sample expressions for get-xs-lop
(define ELOP-VAL (accum '() posn-x cons ELOP))
(define LOP1-VAL (accum '() posn-x cons LOP1))
```

-
```
;; Tests using sample computations for get-xs-lop
(check-expect (get-xs-lop ELOP) ELOP-VAL)
(check-expect (get-xs-lop LOP1) LOP1-VAL)

;; Tests using sample values for get-xs-lop
(check-expect (get-xs-lop (list (make-posn -10 2) (make-posn -5 7))
                          '(-10 -5))
```

- Observe that the sample expressions use `accum`

Functional Abstraction

List Summarizing

- Problems: 228–230

Functional Abstraction

List Searching

- Section 73 discusses the design of the following function:

```
;; number (listof number) → (listof number)
;; Purpose: To return the sublist of the given lon that starts
;;          with the first instance of given number.
(define (xsublist-lon x a-lon)
  (cond [(empty? a-lon)           '()]
        [(equal? x (first a-lon)) a-lon]
        [else (xsublist-lon x (rest a-lon))]))
```

Functional Abstraction

List Searching

- Section 73 discusses the design of the following function:

```
;; number (listof number) → (listof number)
;; Purpose: To return the sublist of the given lon that starts
;;          with the first instance of given number.
(define (xsublist-lon x a-lon)
  (cond [(empty? a-lon)            '()]
        [(equal? x (first a-lon)) a-lon]
        [else (xsublist-lon x (rest a-lon))]))
```

- It is not difficult to now design a function that returns the sublist of a given list of symbols that starts with the first instance of a symbol that is not equal to a given symbol:

```
;; symbol (listof symbol) → (listof symbol)
;; Purpose: To return the sublist of the given list that starts
;;          with the first symbol not equal to given symbol.
(define (nonxsublist-los x a-los)
  (cond [(empty? a-los)                  '()]
        [(not (equal? x (first a-los))) a-los]
        [else (notxsublist-los x (rest a-los))]))
```

Functional Abstraction

List Searching

- Section 73 discusses the design of the following function:

```
;; number (listof number) → (listof number)
;; Purpose: To return the sublist of the given lon that starts
;;          with the first instance of given number.
(define (xsublist-lon x a-lon)
  (cond [(empty? a-lon)            '()]
        [(equal? x (first a-lon)) a-lon]
        [else (xsublist-lon x (rest a-lon))]))
```

- It is not difficult to now design a function that returns the sublist of a given list of symbols that starts with the first instance of a symbol that is not equal to a given symbol:

```
;; symbol (listof symbol) → (listof symbol)
;; Purpose: To return the sublist of the given list that starts
;;          with the first symbol not equal to given symbol.
(define (nonxsublist-los x a-los)
  (cond [(empty? a-los)                  '()]
        [(not (equal? x (first a-los))) a-los]
        [else (notxsublist-los x (rest a-los))]))
```

- Structurally these functions are almost the same
- The only structural difference is the function composition in `nonxsublist-los`'s second condition
- We need to refactor the code, if possible, to make it a two-argument application expression as in `xsublist-lon`

Functional Abstraction

List Searching

- Recall from Mathematics that `(g ∘ h)(x) = g(h(x))`
- This means that any expression for a function composition may be refactored into an expression that applies a new function (namely `(g ∘ h)(x)`)
- For this problem we need to design a function to compute the negation of the `equal?` function

Functional Abstraction

List Searching

- Recall from Mathematics that (g ∘ h)(x) = g(h(x))
- This means that any expression for a function composition may be refactored into an expression that applies a new function (namely (g ∘ h)(x))
- For this problem we need to design a function to compute the negation of the equal? function
-
```
;; <A B> A B → Boolean
;; Purpose: To determine is the given values are not equal?
(define (not-equal? v1 v2) (not (eq? v1 v2)))

;; Sample expressions for not-equal?
(define VAL1 (not (equal? 5 3)))
(define VAL2 (not (equal? '(1 2) '(1 2))))

;; Tests using sample computations for not-equal?
(check-expect (not-equal? 5 3)           VAL1)
(check-expect (not-equal? '(1 2) '(1 2)) VAL2)

;; Tests using sample values for not-equal?
(check-expect (not-equal? "MATTHIAS" "SHRIRAM") #true)
(check-expect (not-equal? (make-posn (sub1 1) 10) (make-posn 0 (+
              #false)
```

Functional Abstraction

List Searching

- Refactor `nonxsublist-lon`:

```
;; number (listof number) → (listof number)
;; Purpose: To return the sublist of the given lon that starts
;;          with the first instance of given number.
(define (xsublist-lon x a-lon)
  (cond [(empty? a-lon)            '()]
        [(equal? x (first a-lon)) a-lon]
        [else (xsublist-lon x (rest a-lon))]))
;; symbol (listof symbol) → (listof symbol)
;; Purpose: To return the sublist of the given list that starts
;;          with the first symbol not equal to given symbol.
(define (nonxsublist-los x a-los)
  (cond [(empty? a-los)       '()]
        [(not-equal? x (first a-los)) a-los]
        [else (notxsublist-los x (rest a-los))]))
```

Functional Abstraction

List Searching

- Refactor nonxsublist-lon:

```
;; number (listof number) → (listof number)
;; Purpose: To return the sublist of the given lon that starts
;;          with the first instance of given number.
(define (xsublist-lon x a-lon)
  (cond [(empty? a-lon)              '()]
        [(equal? x (first a-lon)) a-lon]
        [else (xsublist-lon x (rest a-lon))]))
;; symbol (listof symbol) → (listof symbol)
;; Purpose: To return the sublist of the given list that starts
;;          with the first symbol not equal to given symbol.
(define (nonxsublist-los x a-los)
  (cond [(empty? a-los)          '()]
        [(not-equal? x (first a-los)) a-los]
        [else (notxsublist-los x (rest a-los))]))
```

- The abstraction step results in:

```
;; (? → Boolean) ? (listof ?) → (listof ?)
;; Purpose: To return the sublist of the given list that starts
;;          with first element that satisfies given predicate.
(define (pred-sublist pred x a-lox)
  (cond [(empty? a-lox) '()]
        [(pred x (first a-lox)) a-lox]
        [else (pred-sublist pred x (rest a-lox))]))
```

Functional Abstraction

List Searching

- ```
  ;; (? → Boolean) ? (listof ?) → (listof ?)
  ;; Purpose: To return the sublist of the given list that starts
  ;;          with first element that satisfies given predicate.
  (define (pred-sublist pred x a-lox)
    (cond [(empty? a-lox) '()]
          [(pred x (first a-lox)) a-lox]
          [else (pred-sublist pred x (rest a-lox))]))
  ```
- Once again, care must be taken to write the signature of the abstract function
  ```

Functional Abstraction

List Searching

- ```
  ;; (? → Boolean) ? (listof ?) → (listof ?)
  ;; Purpose: To return the sublist of the given list that starts
  ;;          with first element that satisfies given predicate.
  (define (pred-sublist pred x a-lox)
    (cond [(empty? a-lox) '()]
          [(pred x (first a-lox)) a-lox]
          [else (pred-sublist pred x (rest a-lox))]))
  ```
- Once again, care must be taken to write the signature of the abstract function
- List type processed varies among the functions abstracted over:

  ```
  <X> (? ? → Boolean) ? (listof X) → (listof ?)
  ```
  ```

Functional Abstraction

List Searching

- ```
  ;; (? → Boolean) ? (listof ?) → (listof ?)
  ;; Purpose: To return the sublist of the given list that starts
  ;;          with first element that satisfies given predicate.
  (define (pred-sublist pred x a-lox)
    (cond [(empty? a-lox) '()]
          [(pred x (first a-lox)) a-lox]
          [else (pred-sublist pred x (rest a-lox))]))
  ```
- Once again, care must be taken to write the signature of the abstract function
- List type processed varies among the functions abstracted over:

  ```
  <X> (? ? → Boolean) ? (listof X) → (listof ?)
  ```
- The second argument to the given predicate is a list element:

  ```
  <X> (? X → Boolean) ? (listof X) → (listof ?)
  ```
  ```

Functional Abstraction

List Searching

- ```
  ;; (? → Boolean) ? (listof ?) → (listof ?)
  ;; Purpose: To return the sublist of the given list that starts
  ;;          with first element that satisfies given predicate.
  (define (pred-sublist pred x a-lox)
    (cond [(empty? a-lox) '()]
          [(pred x (first a-lox)) a-lox]
          [else (pred-sublist pred x (rest a-lox))]))
  ```
- Once again, care must be taken to write the signature of the abstract function
- List type processed varies among the functions abstracted over:

  ```
  <X> (? ? → Boolean) ? (listof X) → (listof ?)
  ```
- The second argument to the given predicate is a list element:

  ```
  <X> (? X → Boolean) ? (listof X) → (listof ?)
  ```
- Second line of the conditional the given list is returned which means return type for `pred-sublist` is: Typo in textbook!

  ```
  <X> (? X → Boolean) ? (listof X) → (listof X)
  ```
  ```

Functional Abstraction

List Searching

- ```
  ;; (? → Boolean) ? (listof ?) → (listof ?)
  ;; Purpose: To return the sublist of the given list that starts
  ;;          with first element that satisfies given predicate.
  (define (pred-sublist pred x a-lox)
    (cond [(empty? a-lox) '()]
          [(pred x (first a-lox)) a-lox]
          [else (pred-sublist pred x (rest a-lox))]))
  ```
- Once again, care must be taken to write the signature of the abstract function
- List type processed varies among the functions abstracted over:

  ```
  <X> (? ? → Boolean) ? (listof X) → (listof ?)
  ```
- The second argument to the given predicate is a list element:

  ```
  <X> (? X → Boolean) ? (listof X) → (listof ?)
  ```
- Second line of the conditional the given list is returned which means return type for `pred-sublist` is: Typo in textbook!

  ```
  <X> (? X → Boolean) ? (listof X) → (listof X)
  ```
- First argument to the functions abstracted is always the same as the list elements' type:

  ```
  <X> (X X → Boolean) X (listof X) → (listof X)
  ```
  ```

Functional Abstraction

List Searching

- Refactor `xsublist-lon` and `nonxsublist-lon`:

```
;; number (listof number) → (listof number)
;; Purpose: To return the sublist of the given lon that starts
;;          with the first instance of the given number.
(define (xsublist-lon x a-lon)
  (pred-sublist equal? x a-lon))

;; symbol (listof symbol) → (listof symbol)
;; Purpose: To return the sublist of the given los that starts
;;          with first symbol not equal to given symbol.
(define (nonxsublist-los x a-los)
  (pred-sublist not-equal? x a-los))
```

- Once again, observe that the difference among the functions are part of the input to the abstract function

Functional Abstraction

List Searching

- We now have a powerful function that can return the sublist that starts with the first element that satisfies any predicate

Functional Abstraction

List Searching

- We now have a powerful function that can return the sublist that starts with the first element that satisfies any predicate
- Design a function that returns the sublist of a list of strings that starts with the given string

Functional Abstraction

List Searching

- We now have a powerful function that can return the sublist that starts with the first element that satisfies any predicate
- Design a function that returns the sublist of a list of strings that starts with the given string

- ```
  ;; Sample expressions for strsublist-lostr
  (define STRSUBLIST-LOS1-VAL (pred-sublist string=? "Janice"   LOS1))
  (define STRSUBLIST-LOS2-VAL (pred-sublist string=? "Matthias" LOS2))
  ```
  ```

Functional Abstraction

List Searching

- We now have a powerful function that can return the sublist that starts with the first element that satisfies any predicate
- Design a function that returns the sublist of a list of strings that starts with the given string
- ```
  ;; string (listof string) → (listof string)
  ;; Purpose: To return the sublist of the given (listof string) that starts
  ;;          with the first instance of the given string.
  (define (strsublist-lostr str a-lostr)
  ```

- ```
  ;; Sample expressions for strsublist-lostr
  (define STRSUBLIST-LOS1-VAL (pred-sublist string=? "Janice"   LOS1))
  (define STRSUBLIST-LOS2-VAL (pred-sublist string=? "Matthias" LOS2))
  ```

Functional Abstraction

List Searching

- We now have a powerful function that can return the sublist that starts with the first element that satisfies any predicate
- Design a function that returns the sublist of a list of strings that starts with the given string
- ```
  ;; string (listof string)→ (listof string)
  ;; Purpose: To return the sublist of the given (listof string) that starts
  ;;          with the first instance of the given string.
  (define (strsublist-lostr str a-lostr)
  ```
- ```
  ;; Sample expressions for strsublist-lostr
  (define STRSUBLIST-LOS1-VAL (pred-sublist string=? "Janice"   LOS1))
  (define STRSUBLIST-LOS2-VAL (pred-sublist string=? "Matthias" LOS2))
  ```
- ```
  ;; Tests using sample computations for strsublist-lostr
  (check-expect (strsublist-lostr "Janice"   LOS1) STRSUBLIST-LOS1-VAL)
  (check-expect (strsublist-lostr "Matthias" LOS2) STRSUBLIST-LOS2-VAL)

  ;; Tests using sample values for strsublist-lostr
  (check-expect (strsublist-los "Eladio" '("Rob" "Eladio" "Juan" "Eladio"))
                '("Eladio" "Juan" "Eladio"))
  (check-expect (strsublist-los "zed" '("u" "v" "x" "y" "z" "ZED")) '())
  ```
  ```

Functional Abstraction

List Searching

- We now have a powerful function that can return the sublist that starts with the first element that satisfies any predicate
- Design a function that returns the sublist of a list of strings that starts with the given string
- ```
  ;; string (listof string)→ (listof string)
  ;; Purpose: To return the sublist of the given (listof string) that starts
  ;;          with the first instance of the given string.
  (define (strsublist-lostr str a-lostr)
  ```
- ```
    (pred-sublist string=? str a-lostr))
  ```
- ```
  ;; Sample expressions for strsublist-lostr
  (define STRSUBLIST-LOS1-VAL (pred-sublist string=? "Janice"   LOS1))
  (define STRSUBLIST-LOS2-VAL (pred-sublist string=? "Matthias" LOS2))
  ```
- ```
  ;; Tests using sample computations for strsublist-lostr
  (check-expect (strsublist-lostr "Janice"   LOS1) STRSUBLIST-LOS1-VAL)
  (check-expect (strsublist-lostr "Matthias" LOS2) STRSUBLIST-LOS2-VAL)

  ;; Tests using sample values for strsublist-lostr
  (check-expect (strsublist-los "Eladio" '("Rob" "Eladio" "Juan" "Eladio"))
                '("Eladio" "Juan" "Eladio"))
  (check-expect (strsublist-los "zed" '("u" "v" "x" "y" "z" "ZED")) '())
  ```

Functional Abstraction

Homework

- 234–235

Functional Abstraction

List SearchingList ORing

- Consider:

```
;; (listof alien) → Boolean
;; Purpose: To determine if any alien is at scene's left edge
(define (any-alien-at-left-edge? a-loa)
  (and (not (empty? a-loa))
       (or (alien-at-left-edge? (first a-loa))
           (any-alien-at-left-edge? (rest a-loa)))))

;; (listof number) → Boolean
;; Purpose: To determine if lon contains an even number
(define (has-even-lon? a-lon)
  (and (not (empty? a-lon))
       (or (even? (first a-lon))
           (has-even-lon? (rest a-lon)))))
```

Functional Abstraction

List SearchingList ORing

- Consider:

```
;; (listof alien) → Boolean
;; Purpose: To determine if any alien is at scene's left edge
(define (any-alien-at-left-edge? a-loa)
  (and (not (empty? a-loa))
       (or (alien-at-left-edge? (first a-loa))
           (any-alien-at-left-edge? (rest a-loa)))))

;; (listof number) → Boolean
;; Purpose: To determine if lon contains an even number
(define (has-even-lon? a-lon)
  (and (not (empty? a-lon))
       (or (even? (first a-lon))
           (has-even-lon? (rest a-lon)))))
```

- Structurally the same and, therefore, candidates for abstraction

Functional Abstraction

List SearchingList ORing

- Consider:

```
;; (listof alien) → Boolean
;; Purpose: To determine if any alien is at scene's left edge
(define (any-alien-at-left-edge? a-loa)
  (and (not (empty? a-loa))
       (or (alien-at-left-edge? (first a-loa))
           (any-alien-at-left-edge? (rest a-loa)))))

;; (listof number) → Boolean
;; Purpose: To determine if lon contains an even number
(define (has-even-lon? a-lon)
  (and (not (empty? a-lon))
       (or (even? (first a-lon))
           (has-even-lon? (rest a-lon)))))
```

- Structurally the same and, therefore, candidates for abstraction
- One difference: predicate that is applied to the first element

Functional Abstraction

List SearchingList ORing

- Consider:

```
;; (listof alien) → Boolean
;; Purpose: To determine if any alien is at scene's left edge
(define (any-alien-at-left-edge? a-loa)
  (and (not (empty? a-loa))
       (or (alien-at-left-edge? (first a-loa))
           (any-alien-at-left-edge? (rest a-loa)))))

;; (listof number) → Boolean
;; Purpose: To determine if lon contains an even number
(define (has-even-lon? a-lon)
  (and (not (empty? a-lon))
       (or (even? (first a-lon))
           (has-even-lon? (rest a-lon)))))
```

- Structurally the same and, therefore, candidates for abstraction
- One difference: predicate that is applied to the first element
- The abstraction step:

```
;; <X> (X → Boolean) (listof X) → Boolean
;; Purpose: Determine if any list element satisfies predicate
(define (ormap-pred pred? a-lox)
  (and (not (empty? a-lox))
       (or (pred? (first a-lox)) (ormap-pred pred? (rest a-lox)))))
```

Functional Abstraction

List SearchingList ORing

- Refactoring `any-alien-at-left-edge?` and `has-even-lon?`:

```
;; (listof alien) → Boolean
;; Purpose: To determine if any alien is at scene's left edge
(define (any-alien-at-left-edge? a-loa)
  (ormap-pred alien-at-left-edge? a-loa))

;; (listof number) → Boolean
;; Purpose: Determine if lon contains an even number
(define (has-even-lon? a-lon)
  (ormap-pred even? a-lon))
```

Functional Abstraction

List SearchingList ORing

- We now have a powerful function to perform list oring operations

Functional Abstraction

List SearchingList ORing

- We now have a powerful function to perform list oring operations
- `any-alien-reached-earth?` from Section 73.2 and `any-alien-at-right-edge?` from Section 74.2 may be refactored:

```
;; (listof alien) → Boolean
;; Purpose: Determine if any alien has reached earth
(define (any-alien-reached-earth? a-loa)
  (ormap-pred alien-reached-earth? a-loa))

;; (listof alien) → Boolean
;; Purpose: To determine if any alien is at scene's right edge
(define (any-alien-at-right-edge? a-loa)
  (ormap-pred alien-at-right-edge? a-loa))
```

Functional Abstraction

List SearchingList ORing

- We now have a powerful function to perform list oring operations
- `any-alien-reached-earth?` from Section 73.2 and `any-alien-at-right-edge?` from Section 74.2 may be refactored:

```
;; (listof alien) → Boolean
;; Purpose: Determine if any alien has reached earth
(define (any-alien-reached-earth? a-loa)
  (ormap-pred alien-reached-earth? a-loa))

;; (listof alien) → Boolean
;; Purpose: To determine if any alien is at scene's right edge
(define (any-alien-at-right-edge? a-loa)
  (ormap-pred alien-at-right-edge? a-loa))
```

- So powerful and useful that ISL+ provides it and is called `ormap`

Functional Abstraction

List SearchingList ORing

- Problems: 238–240, 242

Functional Abstraction

List ANDing

- Consider:

```
;; (listof number) → Boolean
;; Purpose: Determine if lon only has even numbers
(define (all-even-lon? a-lon)
  (or (empty? a-lon)
      (and (even? (first a-lon)) (all-even-lon? (rest a-lon)))))
                                   Good candidates for abstraction
;; string → Boolean
;; Purpose: Determine if the length of the given string is ≤ 5
(define (string-len<5? a-str) (<= (string-length a-str) 5))
;; (listof string) → Boolean
;; Purpose: Determine if lostr only has strings of length ≤ 5
(define (string-all-len<5? a-lostr)
 (or (empty? a-lostr)
     (and (string-len<5? (first a-lostr))
     (string-all-len<5? (rest a-lostr)))))
```

Functional Abstraction

List ANDing

- Consider:

```
;; (listof number) → Boolean
;; Purpose: Determine if lon only has even numbers
(define (all-even-lon? a-lon)
  (or (empty? a-lon)
      (and (even? (first a-lon)) (all-even-lon? (rest a-lon)))))
```

Good candidates for abstraction

```
;; string → Boolean
;; Purpose: Determine if the length of the given string is ≤ 5
(define (string-len<5? a-str) (<= (string-length a-str) 5))
;; (listof string) → Boolean
;; Purpose: Determine if lostr only has strings of length ≤ 5
(define (string-all-len<5? a-lostr)
 (or (empty? a-lostr)
     (and (string-len<5? (first a-lostr))
     (string-all-len<5? (rest a-lostr)))))
```

- The abstraction:

```
;; <X> (X → Boolean) (listof X) → Boolean
;; Purpose: Determine if all list elements satisfy predicate
(define (andmap-pred pred a-lox)
  (or (empty? a-lox)
      (and (pred (first a-lox)) (andmap-pred pred (rest a-lox)))))
```

Functional Abstraction

List ANDing

- The functions refactored:

```
;; lon → Boolean
;; Purpose: Determine if the given lon only has even numbers
(define (all-even-lon? a-lon)
  (andmap-pred even? a-lon))

;; (listof string) → Boolean
;; Purpose: Determine if the given (listof string) only
;;          has strings of length <= 5
(define (string-all-len<5? a-lostr)
  (andmap-pred string-len<5? a-lostr))
```

- Elegance and clarity provided by the abstract function is truly remarkable
- We have a powerful abstract function to perform list anding operations

Functional Abstraction

List ANDing

- The functions refactored:

```
;; lon → Boolean
;; Purpose: Determine if the given lon only has even numbers
(define (all-even-lon? a-lon)
  (andmap-pred even? a-lon))

;; (listof string) → Boolean
;; Purpose: Determine if the given (listof string) only
;;          has strings of length <= 5
(define (string-all-len<5? a-lostr)
  (andmap-pred string-len<5? a-lostr))
```

- Elegance and clarity provided by the abstract function is truly remarkable
- We have a powerful abstract function to perform list anding operations
- This list operation is so useful that ISL provides it and it is called andmap

Functional Abstraction

Homework

- Problems: 244–246

Functional Abstraction

List Mapping

- Consider:

```
;; (listof shot) → (listof shot) Purpose: Move the list of shots
(define (move-los a-los)
  (if (empty? a-los)
      E-LOS
      (cons (move-shot (first a-los)) (move-los  (rest a-los)))))
;; (listof string) → (listof number)
;; Purpose: Return the string lengths in the given (listof string)
(define (lengths-lostr a-lostr)
  (if (empty? a-lostr)
      '()
      (cons (string-length (first a-lostr))
            (lengths-lostr (rest a-lostr)))))
```

Functional Abstraction

List Mapping

- Consider:

```
;; (listof shot) → (listof shot) Purpose: Move the list of shots
(define (move-los a-los)
  (if (empty? a-los)
      E-LOS
      (cons (move-shot (first a-los)) (move-los  (rest a-los)))))
;; (listof string) → (listof number)
;; Purpose: Return the string lengths in the given (listof string)
(define (lengths-lostr a-lostr)
  (if (empty? a-lostr)
      '()
      (cons (string-length (first a-lostr))
            (lengths-lostr (rest a-lostr)))))
```

- Good candidates for abstraction
- The only significant difference: function applied to first element

Functional Abstraction

List Mapping

- Consider:

```
;; (listof shot) → (listof shot) Purpose: Move the list of shots
(define (move-los a-los)
  (if (empty? a-los)
      E-LOS
      (cons (move-shot (first a-los)) (move-los  (rest a-los)))))
;; (listof string) → (listof number)
;; Purpose: Return the string lengths in the given (listof string)
(define (lengths-lostr a-lostr)
  (if (empty? a-lostr)
      '()
      (cons (string-length (first a-lostr))
            (lengths-lostr (rest a-lostr)))))
```

- Good candidates for abstraction
- The only significant difference: function applied to first element
- The abstraction:

```
;; <X Y> (X → Y) (listof X) → (listof Y)
;; Purpose: Return values applying function to list's elements
(define (map-f f a-lox)
  (if (empty? a-lox)
      '()
      (cons (f (first a-lox)) (map-f f (rest a-lox)))))
```

Functional Abstraction

List Mapping

- Refactoring:

```
;; los → los
;; Purpose: To move the given list of shots
(define (move-los a-los) (map-f move-shot a-los))

;; (listof string) → lon
;; Purpose: Return the string lengths in the given (listof string)
(define (lengths-lostr a-lostr) (map-f string-length a-lostr))
```

- Many experienced programmers argue that the use of abstract functions is the poetry of programming
- Named `map` in ISL+

Functional Abstraction

Homework

- Problems: 248–253
- Typo in 248: `move-loa` (not `move-alien`)
- Typo in 249: Design and implement a nonrecursive function to add 1 to the elements of a list of numbers.

Functional Abstraction

List Filtering

- Consider:

```
;; lon → lon  Purpose: Return evens in list
(define (extract-evens a-lon)
  (cond [(empty? a-lon) '()]
        [(even? (first a-lon))
         (cons (first a-lon) (extract-evens (rest a-lon)))]
        [else (extract-evens (rest a-lon))]))

;; (listof shot) → (listof shot)  Purpose: Return posn shots list
(define (extract-posn-shots a-los)
 (cond [(empty? a-los) '()]
       [(posn? (first a-los))
        (cons (first a-los) (extract-posn-shots (rest a-los)))]
       [else (extract-posn-shots (rest a-los))]))
```

- Candidates for abstraction and only difference: predicate applied

Functional Abstraction

List Filtering

- Consider:

```
;; lon → lon  Purpose: Return evens in list
(define (extract-evens a-lon)
  (cond [(empty? a-lon) '()]
        [(even? (first a-lon))
         (cons (first a-lon) (extract-evens (rest a-lon)))]
        [else (extract-evens (rest a-lon))]))

;; (listof shot) → (listof shot)  Purpose: Return posn shots list
(define (extract-posn-shots a-los)
 (cond [(empty? a-los) '()]
       [(posn? (first a-los))
        (cons (first a-los) (extract-posn-shots (rest a-los)))]
       [else (extract-posn-shots (rest a-los))]))
```

- Candidates for abstraction and only difference: predicate applied
- The abstraction:

```
;; <X> (X → Boolean) (listof X) → (listof X)
;; Purpose: Return list elements that satisfy predicate
(define (filter-pred pred a-lox)
  (cond [(empty? a-lox) '()]
        [(pred (first a-lox))
         (cons (first a-lox) (filter-pred pred (rest a-lox)))]
        [else (filter-pred pred (rest a-lox))]))
```

Functional Abstraction

List Filtering

- Refactored functions:

```
;; lon → lon
;; Purpose: Return list of the even numbers in the given list
(define (extract-evens a-lon) (filter-pred even? a-lon))

;; los → los
;; Purpose: Return list of posn shots in the given list
(define (extract-posn-shots a-los) (filter-pred posn? a-los))
```

Functional Abstraction

List Filtering

- Refactored functions:

```
;; lon → lon
;; Purpose: Return list of the even numbers in the given list
(define (extract-evens a-lon) (filter-pred even? a-lon))

;; los → los
;; Purpose: Return list of posn shots in the given list
(define (extract-posn-shots a-los) (filter-pred posn? a-los))
```

- This abstraction is so powerful and useful that ISL+ provides it: `filter`

Functional Abstraction

List Filtering

- Problems: 255–256, 258, 260

Functional Abstraction

List Sorting

- Consider:

```
;; number (listof number) → (listof number)
(define (insert a-num a-lon)
  (cond [(empty? a-lon) (list a-num)]
        [(<= a-num (first a-lon)) (cons a-num a-lon)]
        [else (cons (first a-lon) (insert a-num (rest a-lon)))]))
;; (listof number) → (listof number)
(define (sort-lon a-lon)
  (cond [(empty? a-lon) '()]
        [else (insert (first a-lon) (sort-lon (rest a-lon)))]))
```

-

```
(define-struct item (name price quantity))

;; item item → Boolean
(define (leq-item item1 item2) (<= (item-price item1) (item-price item2)))
;; item (listof item) → (listof item)
(define (insert-loi an-item a-loi)
  (cond [(empty? a-loi) (list an-item)]
        [(leq-item an-item (first a-loi)) (cons an-item a-loi)]
        [else (cons (first a-loi)
                    (insert-loi an-item (rest a-loi)))]))
;; loi → loi
;; Purpose: Sort the given loi in nonincreasing order by price
(define (sort-loi a-loi)
  (cond [(empty? a-loi) '()]
        [else (insert-loi (first a-loi)
                          (sort-loi (rest a-loi)))]))
```

Functional Abstraction

List Sorting

- Consider:

```
;; number (listof number) → (listof number)
(define (insert a-num a-lon)
  (cond [(empty? a-lon) (list a-num)]
        [(<= a-num (first a-lon)) (cons a-num a-lon)]
        [else (cons (first a-lon) (insert a-num (rest a-lon)))]))
;; (listof number) → (listof number)
(define (sort-lon a-lon)
  (cond [(empty? a-lon) '()]
        [else (insert (first a-lon) (sort-lon (rest a-lon)))]))
```

- `(define-struct item (name price quantity))`

```
;; item item → Boolean
(define (leq-item item1 item2) (<= (item-price item1) (item-price item2)))
;; item (listof item) → (listof item)
(define (insert-loi an-item a-loi)
  (cond [(empty? a-loi) (list an-item)]
        [(leq-item an-item (first a-loi)) (cons an-item a-loi)]
        [else (cons (first a-loi)
                    (insert-loi an-item (rest a-loi)))]))
;; loi → loi
;; Purpose: Sort the given loi in nonincreasing order by price
(define (sort-loi a-loi)
  (cond [(empty? a-loi) '()]
        [else (insert-loi (first a-loi)
                          (sort-loi (rest a-loi)))]))
```

- inserting functions only difference: predicate used to compare two values
- Sorting functions only difference: inserting function called

Functional Abstraction

List Sorting

- The abstraction for inserting:

```
;; <X> (X → Boolean) X (listof X) → (listof X)
;; Purpose: Insert the given number in the given lox to
;;          create sorted lox using the given predicate
;; ASSUMPTION: Given lon is sorted as defined by the given
;;             predicate
(define (insert-pred pred an-x a-lox)
  (cond [(empty? a-lox) (list an-x)]
        [(pred an-x (first a-lox)) (cons an-x a-lox)]
        [else (cons (first a-lox)
                    (insert-pred pred an-x (rest a-lox)))]))
```

Functional Abstraction

List Sorting

- The abstraction for inserting:

```
;; <X> (X → Boolean) X (listof X) → (listof X)
;; Purpose: Insert the given number in the given lox to
;;          create sorted lox using the given predicate
;; ASSUMPTION: Given lon is sorted as defined by the given
;;             predicate
(define (insert-pred pred an-x a-lox)
  (cond [(empty? a-lox) (list an-x)]
        [(pred an-x (first a-lox)) (cons an-x a-lox)]
        [else (cons (first a-lox)
                    (insert-pred pred an-x (rest a-lox)))]))
```

- Refactored inserting functions: Refactoring the inserting functions yields:

```
;; insert: number (listof number) → (listof number)
(define (insert a-num a-lon)
  (insert-pred <= a-num a-lon))

;; item (listof item) → (listof item)
(define (insert-loi an-item a-loi)
  (insert-pred leq-item an-item a-loi))
```

Functional Abstraction

List Sorting

- ```
  ;; (listof number) → (listof number)
  (define (sort-lon a-lon)
    (cond [(empty? a-lon) '()]
          [else (insert (first a-lon) (sort-lon (rest a-lon)))]))

  ;; loi → loi
  ;; Purpose: Sort the given loi in nonincreasing order by price
  (define (sort-loi a-loi)
    (cond [(empty? a-loi) '()]
          [else (insert-loi (first a-loi) (sort-loi (rest a-loi)))]))
  ```
  ```

Functional Abstraction

List Sorting

- ```
  ;; (listof number) → (listof number)
  (define (sort-lon a-lon)
    (cond [(empty? a-lon) '()]
          [else (insert (first a-lon) (sort-lon (rest a-lon)))]))

  ;; loi → loi
  ;; Purpose: Sort the given loi in nonincreasing order by price
  (define (sort-loi a-loi)
    (cond [(empty? a-loi) '()]
          [else (insert-loi (first a-loi) (sort-loi (rest a-loi)))]))
  ```
- The abstraction:

  ```
  ;; <X> (X (listof X) → (listof X)) (listof X) → (listof X)
  ;; Purpose: Sort given lox using the given inserting function
  (define (sort-lox insert-f a-lox)
    (cond [(empty? a-lox) '()]
          [else (insert-f (first a-lox) (sort-lox (rest a-lox)))]))
  ```
  ```

Functional Abstraction

List Sorting

- ```
  ;; (listof number) → (listof number)
  (define (sort-lon a-lon)
    (cond [(empty? a-lon) '()]
          [else (insert (first a-lon) (sort-lon (rest a-lon)))]))

  ;; loi → loi
  ;; Purpose: Sort the given loi in nonincreasing order by price
  (define (sort-loi a-loi)
    (cond [(empty? a-loi) '()]
          [else (insert-loi (first a-loi) (sort-loi (rest a-loi)))]))
  ```
- The abstraction:

  ```
  ;; <X> (X (listof X) → (listof X)) (listof X) → (listof X)
  ;; Purpose: Sort given lox using the given inserting function
  (define (sort-lox insert-f a-lox)
    (cond [(empty? a-lox) '()]
          [else (insert-f (first a-lox) (sort-lox (rest a-lox)))]))
  ```
- **Wrong abstraction!**
- Users are not interested in nor inclined to develop the proper inserting function even if they can use `insert-pred` to do so
  ```

Functional Abstraction

List Sorting

- The proper abstraction has the user provides a predicate and a list to sort
- How can we develop an abstract sorting function that provides the right abstraction?

Functional Abstraction

List Sorting

- The proper abstraction has the user provides a predicate and a list to sort
- How can we develop an abstract sorting function that provides the right abstraction?
- Eliminate inserting function as difference, make predicate the difference

Functional Abstraction

List Sorting

- The proper abstraction has the user provides a predicate and a list to sort
- How can we develop an abstract sorting function that provides the right abstraction?
- Eliminate inserting function as difference, make predicate the difference
- Done by refactoring the sorting functions before performing the abstraction step
- More concretely, we can inline the proper use of `insert-pred` into the bodies of the sorting functions

Functional Abstraction

List Sorting

- The proper abstraction has the user provides a predicate and a list to sort
- How can we develop an abstract sorting function that provides the right abstraction?
- Eliminate inserting function as difference, make predicate the difference
- Done by refactoring the sorting functions before performing the abstraction step
- More concretely, we can inline the proper use of `insert-pred` into the bodies of the sorting functions
- The refactored sorting functions:

```
;; (listof number) → (listof number)
;; Purpose: Sort lon in nondecreasing order
(define (sort-lon a-lon)
 (cond [(empty? a-lon) '()]
       [else
        (insert-pred <= (first a-lon) (sort-lon (rest a-lon)))]))
;; (listof item) → (listof item)
;; Purpose: Sort list of items in nonincreasing order by price
(define (sort-loi a-loi)
 (cond [(empty? a-loi) '()]
       [else
        (insert-pred leq-item (first a-loi) (sort-loi (rest a-loi)))]))
```

Functional Abstraction

List Sorting

- The proper abstraction has the user provides a predicate and a list to sort
- How can we develop an abstract sorting function that provides the right abstraction?
- Eliminate inserting function as difference, make predicate the difference
- Done by refactoring the sorting functions before performing the abstraction step
- More concretely, we can inline the proper use of `insert-pred` into the bodies of the sorting functions
- The refactored sorting functions:

```
;; (listof number) → (listof number)
;; Purpose: Sort lon in nondecreasing order
(define (sort-lon a-lon)
 (cond [(empty? a-lon) '()]
       [else
        (insert-pred <= (first a-lon) (sort-lon (rest a-lon)))]))
;; (listof item) → (listof item)
;; Purpose: Sort list of items in nonincreasing order by price
(define (sort-loi a-loi)
 (cond [(empty? a-loi) '()]
       [else
        (insert-pred leq-item (first a-loi) (sort-loi (rest a-loi)))]))
```

- Two important consequences of inlining the use of `insert-pred`:
  - Only difference is the predicate used
  - `insert` and `insert-loi` are no longer needed

Functional Abstraction

List Sorting

- The proper abstraction has the user provides a predicate and a list to sort
- How can we develop an abstract sorting function that provides the right abstraction?
- Eliminate inserting function as difference, make predicate the difference
- Done by refactoring the sorting functions before performing the abstraction step
- More concretely, we can inline the proper use of `insert-pred` into the bodies of the sorting functions
- The refactored sorting functions:

```
;; (listof number) → (listof number)
;; Purpose: Sort lon in nondecreasing order
(define (sort-lon a-lon)
 (cond [(empty? a-lon) '()]
       [else
        (insert-pred <= (first a-lon) (sort-lon (rest a-lon)))]))
;; (listof item) → (listof item)
;; Purpose: Sort list of items in nonincreasing order by price
(define (sort-loi a-loi)
 (cond [(empty? a-loi) '()]
       [else
        (insert-pred leq-item (first a-loi) (sort-loi (rest a-loi)))]))
```

- Two important consequences of inlining the use of `insert-pred`:
  - Only difference is the predicate used
  - `insert` and `insert-loi` are no longer needed
- The abstraction:

```
;; <X> (X → Boolean) (listof X) → (listof X)
;; Purpose: To sort the given lox using the given predicate
(define (sort-pred pred a-lox)
 (cond [(empty? a-lox) '()]
       [else (insert-pred pred (first a-lox) (sort-pred (rest a-lox)))]))
```

Functional Abstraction

List Sorting

- Refactored sorting functions:

```
;; (listof number) → (listof number)
;; Purpose: Sort lon in non-decreasing order
(define (sort-lon a-lon) (sort-pred <= a-lon))

;; (listof item) → (listof item)
;; Purpose: Sort list of items in nonincreasing order by price
     (define (sort-loi a-loi) (sort-pred leq-item a-loi))
```

Functional Abstraction

List Sorting

- Refactored sorting functions:

```
;; (listof number) → (listof number)
;; Purpose: Sort lon in non-decreasing order
(define (sort-lon a-lon) (sort-pred <= a-lon))

;; (listof item) → (listof item)
;; Purpose: Sort list of items in nonincreasing order by price
     (define (sort-loi a-loi) (sort-pred leq-item a-loi))
```

- What do you think about how simple it is now to write sorting functions?
- This powerful abstract function to write sorting functions is called `sort` in ISL+.

Functional Abstraction

List Sorting

- Refactored sorting functions:

```
;; (listof number) → (listof number)
;; Purpose: Sort lon in non-decreasing order
(define (sort-lon a-lon) (sort-pred <= a-lon))

;; (listof item) → (listof item)
;; Purpose: Sort list of items in nonincreasing order by price
     (define (sort-loi a-loi) (sort-pred leq-item a-loi))
```

- What do you think about how simple it is now to write sorting functions?
- This powerful abstract function to write sorting functions is called `sort` in ISL+.
- Important lesson: not all possible abstractions are useful or make code easier to understand
- The first attempt to abstract the sorting functions yielded an abstraction that is unlikely to be useful
- It is important to be mindful about the usefulness of the abstractions we create

Functional Abstraction

Homework

- Problems: 261–262

Functional Abstraction

Abstraction Over Interval-Processing Functions

- Consider:

```
;; [int..int] → int
;; Purpose: Compute the product of the squares of the ints
;;          in the given interval
(define (interval-product-sqrs low high)
  (if (> low high)
      1
      (* (sqr low) (interval-product-sqrs (add1 low) high))))

;; [int..int] → (listof string)
;; Purpose: Construct the list of strings for the ints in
;;          the given interval
(define (interval->lostr low high)
  (if (> low high)
      '()
      (cons (number->string low)
            (interval->lostr (add1 low) high))))
```

Functional Abstraction

Abstraction Over Interval-Processing Functions

- Consider:

```
;; [int..int] → int
;; Purpose: Compute the product of the squares of the ints
;;          in the given interval
(define (interval-product-sqrs low high)
  (if (> low high)
      1
      (* (sqr low) (interval-product-sqrs (add1 low) high))))

;; [int..int] → (listof string)
;; Purpose: Construct the list of strings for the ints in
;;          the given interval
(define (interval->lostr low high)
  (if (> low high)
      '()
      (cons (number->string low)
            (interval->lostr (add1 low) high))))
```

- Good candidates for abstraction
- Three differences: the base value returned, the function applied to the first element of the interval, and the combining function

Functional Abstraction

Abstraction Over Interval-Processing Functions

- Consider:

```
;; [int..int] → int
;; Purpose: Compute the product of the squares of the ints
;;          in the given interval
(define (interval-product-sqrs low high)
  (if (> low high)
      1
      (* (sqr low) (interval-product-sqrs (add1 low) high))))

;; [int..int] → (listof string)
;; Purpose: Construct the list of strings for the ints in
;;          the given interval
(define (interval->lostr low high)
  (if (> low high)
      '()
      (cons (number->string low)
            (interval->lostr (add1 low) high))))
```

- Good candidates for abstraction
- Three differences: the base value returned, the function applied to the first element of the interval, and the combining function
- The abstraction:

```
;; <X Y> X (int → Y) (Y X → X) [int..int] → X
;; Purpose: Compute a value by traversing the given interval
;;          from low to high using the given base value,
;;          function to apply to low, and combining function.
(define (interval-accum-l2h base-val ffirst comb low high)
  (if (> low high)
      base-val
     (comb (ffirst low)
           (interval-accum-l2h  base-val ffirst comb (add1 low) high))))
```

Functional Abstraction

Abstraction Over Interval-Processing Functions

- Refactored functions:

```
;; [int..int] → (listof string)
;; Purpose: Construct the list of strings for the ints in
;;          the given interval
(define (interval->lostr low high)
  (interval-accum-l2h '() number->string cons low high))

;; [int..int] → int
;; Purpose: Compute the product of the squares of the ints
;;          in the given interval
(define (interval-product-sqrs low high)
  (interval-accum-l2h 1 sqr * low high))
```

- Once again, observe that the functions abstracted over have been greatly simplified
- This abstract function, like the others in this chapter, allow the programmers to spend more time on problem solving and less time on mundane typing of repetitions, which usually means fewer bugs in the software developed

Functional Abstraction

Homework

- Problems: 265–266, 269

Encapsulation

- As the use of auxiliary functions increases the size of programs grows
- You may have already noticed in Aliens Attack, for example, that you have related functions scattered all over your program's file
- This makes it difficult to understand a design and to make refinements

Encapsulation

- As the use of auxiliary functions increases the size of programs grows
- You may have already noticed in Aliens Attack, for example, that you have related functions scattered all over your program's file
- This makes it difficult to understand a design and to make refinements
- It is therefore desirable to *encapsulate* the definitions in our programs
- Encapsulation means that we package together all related definitions as a single piece of software
- This allows any reader of our code to more easily understand the design and makes the process of program refinement easier.

Encapsulation

- As the use of auxiliary functions increases the size of programs grows
- You may have already noticed in Aliens Attack, for example, that you have related functions scattered all over your program's file
- This makes it difficult to understand a design and to make refinements
- It is therefore desirable to *encapsulate* the definitions in our programs
- Encapsulation means that we package together all related definitions as a single piece of software
- This allows any reader of our code to more easily understand the design and makes the process of program refinement easier.
- New syntax is required to encapsulate definitions
- Need definitions inside of functionsfor auxiliary functions

Encapsulation

Local Expressions

- A `local`-expression is used to group together an arbitrary number of related definitions

Encapsulation

Local Expressions

- A `local`-expression is used to group together an arbitrary number of related definitions
- The `local`-expression syntax is:
  `expr` ::= (local [`def`*] `expr`)
- Inside parenthesis the keyword `local`
- Inside square brackets zero or more definitions
- An expression, called the body, for the value of the `local`-expression
- We say that the local definitions are encapsulated inside the `local`-expression.

Encapsulation

Local Expressions

- ```
  ;; los → los  Purpose: To move the given list of shots
  (define (move-los a-los)
    (if (empty? a-los)
        E-LOS     Same one-input function is applied to every shot
        (cons (move-shot (first a-los)) (move-los (rest a-los)))))
  ```
  ```

Encapsulation

Local Expressions

- ```
  ;; los → los  Purpose: To move the given list of shots
  (define (move-los a-los)
    (if (empty? a-los)
        E-LOS     Same one-input function is applied to every shot
        (cons (move-shot (first a-los)) (move-los (rest a-los)))))
  ```
- May be refactored using `map`:
  ```
  ;; los → los  Purpose: To move the given list of shots
  (define (move-los a-los) (map move-shot a-los))
  ```
- Can any programmer easily make refinements to this code?
  ```

Encapsulation

Local Expressions

- ```
  ;; los → los  Purpose: To move the given list of shots
  (define (move-los a-los)
    (if (empty? a-los)
        E-LOS     Same one-input function is applied to every shot
        (cons (move-shot (first a-los)) (move-los (rest a-los)))))
  ```
- May be refactored using `map`:
  ```
  ;; los → los  Purpose: To move the given list of shots
  (define (move-los a-los) (map move-shot a-los))
  ```
- Can any programmer easily make refinements to this code?
- Not fully: Where is `move-shot`?
  ```

Encapsulation

Local Expressions

- ```
  ;; los → los  Purpose: To move the given list of shots
  (define (move-los a-los)
    (if (empty? a-los)
        E-LOS     Same one-input function is applied to every shot
        (cons (move-shot (first a-los)) (move-los (rest a-los)))))
  ```
- May be refactored using `map`:
  ```
  ;; los → los  Purpose: To move the given list of shots
  (define (move-los a-los) (map move-shot a-los))
  ```
- Can any programmer easily make refinements to this code?
- Not fully: Where is `move-shot`?
- Encapsulate both of these functions into a single package:
  ```
  ;; los → los Purpose: To move the given list of shots
  (define (move-los a-los)
    (local
     [;; shot → shot  Purpose: To move the given shot
      (define (local-move-shot a-shot)
       (cond [(eq? a-shot NO-SHOT) a-shot]
             [else
              (cond
               [(= (posn-y a-shot) MIN-IMG-Y) NO-SHOT]
               [else (make-posn (posn-x a-shot)
                                (move-up-image-y (posn-y a-shot)))]))]
      (map local-move-shot a-los))
  ```
  ```

Encapsulation

Local Expressions

- `local`-expressions are syntactic sugar

Encapsulation

Local Expressions

- `local`-expressions are syntactic sugar
- Any `local`-expression may be eliminated as follows:
  1. Give every local definition a unique name and change all references using the old name to be references using the new names
  2. Move the renamed definitions out of the `local`-expression
  3. Replace the `local`-expression with its body
- A (partial) *refactoring recipe* to eliminate `local`-expressions

Encapsulation

Local Expressions

- `local`-expressions are syntactic sugar
- Any `local`-expression may be eliminated as follows:
  1. Give every local definition a unique name and change all references using the old name to be references using the new names
  2. Move the renamed definitions out of the `local`-expression
  3. Replace the `local`-expression with its body
- A (partial) *refactoring recipe* to eliminate `local`-expressions
- Consider:

```
(define DELTA 10)
(+ DELTA (local [(define DELTA 100) (define VAL (+ DELTA DELTA))]
           VAL))
```

- What is the value of the sum?

Encapsulation

Local Expressions

- `local`-expressions are syntactic sugar
- Any `local`-expression may be eliminated as follows:
  1. Give every local definition a unique name and change all references using the old name to be references using the new names
  2. Move the renamed definitions out of the `local`-expression
  3. Replace the `local`-expression with its body
- A (partial) *refactoring recipe* to eliminate `local`-expressions
- Consider:

```
(define DELTA 10)
(+ DELTA (local [(define DELTA 100) (define VAL (+ DELTA DELTA))]
           VAL))
```

- What is the value of the sum?
- Refactor to eliminate the `local`-expression. After Step 1:

```
(define DELTA 10)
(+ DELTA (local [(define DLT 100) (define VAL (+ DLT DLT))] VAL))
```

Encapsulation

Local Expressions

- `local`-expressions are syntactic sugar
- Any `local`-expression may be eliminated as follows:
  1. Give every local definition a unique name and change all references using the old name to be references using the new names
  2. Move the renamed definitions out of the `local`-expression
  3. Replace the `local`-expression with its body
- A (partial) *refactoring recipe* to eliminate `local`-expressions
- Consider:

```
(define DELTA 10)
(+ DELTA (local [(define DELTA 100) (define VAL (+ DELTA DELTA))]
           VAL))
```

- What is the value of the sum?
- Refactor to eliminate the `local`-expression. After Step 1:

```
(define DELTA 10)
(+ DELTA (local [(define DLT 100) (define VAL (+ DLT DLT))] VAL))
```

- Step 2 yields:

```
(define DELTA 10)  (define DLT 100)  (define VAL (+ DLT DLT))
(+ DELTA (local [] VAL))
```

Encapsulation

Local Expressions

- `local`-expressions are syntactic sugar
- Any `local`-expression may be eliminated as follows:
  1. Give every local definition a unique name and change all references using the old name to be references using the new names
  2. Move the renamed definitions out of the `local`-expression
  3. Replace the `local`-expression with its body
- A (partial) *refactoring recipe* to eliminate `local`-expressions
- Consider:

```
(define DELTA 10)
(+ DELTA (local [(define DELTA 100) (define VAL (+ DELTA DELTA))]
           VAL))
```

- What is the value of the sum?
- Refactor to eliminate the `local`-expression. After Step 1:

```
(define DELTA 10)
(+ DELTA (local [(define DLT 100) (define VAL (+ DLT DLT))] VAL))
```

- Step 2 yields:

```
(define DELTA 10)  (define DLT 100)  (define VAL (+ DLT DLT))
(+ DELTA (local [] VAL))
```

- Step 3 yields:

```
(define DELTA 10)  (define DLT 100) (define VAL (+ DLT DLT))
(+ DELTA VAL)
```

- In this form it becomes clear that the value of the sum is 210

Encapsulation

Lexical Scoping

- Why is it necessary to rename definitions before lifting them outside a `local`-expression?
- How do we know which references need to be updated to use the new name?

Encapsulation

Lexical Scoping

- Why is it necessary to rename definitions before lifting them outside a `local`-expression?
- How do we know which references need to be updated to use the new name?
- The answer is *lexical scoping* (or *static scoping*)
- The lexical scope of a definition or a variable (as the parameters of a function) is the part of the program where the definition or variable is valid

Encapsulation

Lexical Scoping

- Why is it necessary to rename definitions before lifting them outside a `local`-expression?
- How do we know which references need to be updated to use the new name?
- The answer is *lexical scoping* (or *static scoping*)
- The lexical scope of a definition or a variable (as the parameters of a function) is the part of the program where the definition or variable is valid
- Consider:

```
;; natnum natnum → natnum
;; Purpose: Multiply the given natural numbers
(define (mult x y)
  (if (= x 0)
      0
      (+ y (mult (sub1 x) y))))
```

- Three names declared: `mult`, `x`, and `y`

Encapsulation

Lexical Scoping

- Why is it necessary to rename definitions before lifting them outside a `local`-expression?
- How do we know which references need to be updated to use the new name?
- The answer is *lexical scoping* (or *static scoping*)
- The lexical scope of a definition or a variable (as the parameters of a function) is the part of the program where the definition or variable is valid
- Consider:

```
;; natnum natnum → natnum
;; Purpose: Multiply the given natural numbers
(define (mult x y)
  (if (= x 0)
      0
      (+ y (mult (sub1 x) y))))
```

- Three names declared: `mult`, `x`, and `y`
- Do not have the same lexical scope
- `mult` has *global scope* (or simply is global) because it is not locally defined nor is it a parameter

Encapsulation

Lexical Scoping

- Why is it necessary to rename definitions before lifting them outside a `local`-expression?
- How do we know which references need to be updated to use the new name?
- The answer is *lexical scoping* (or *static scoping*)
- The lexical scope of a definition or a variable (as the parameters of a function) is the part of the program where the definition or variable is valid
- Consider:

```
;; natnum natnum → natnum
;; Purpose: Multiply the given natural numbers
(define (mult x y)
  (if (= x 0)
      0
      (+ y (mult (sub1 x) y))))
```

- Three names declared: `mult`, `x`, and `y`
- Do not have the same lexical scope
- `mult` has *global scope* (or simply is global) because it is not locally defined nor is it a parameter
- The lexical scope of `x` and `y` is the body of the function
- `x` and `y` are bound variables in the body of `mult`

Encapsulation

Lexical Scoping

- consider this larger fragment of the program:

```
(define x 2)

(define (add-x-y-and-z y) (+ x y z))

(define (fact x)
  (if (= x 0)
      1
      (* x (fact (sub1 x)))))

(define (mult x y)
  (if (= x 0)
      0
      (+ y (mult (sub1 x) y))))
```

- 4 declarations that have global scope: `x` (defined as 2), `add-x`, `fact`, and `mult`
- There are four declarations that do not have global scope: the parameters

Encapsulation

Lexical Scoping

- consider this larger fragment of the program:

```
(define x 2)

(define (add-x-y-and-z y) (+ x y z))

(define (fact x)
  (if (= x 0)
      1
      (* x (fact (sub1 x)))))

(define (mult x y)
  (if (= x 0)
      0
      (+ y (mult (sub1 x) y))))
```

- 4 declarations that have global scope: `x` (defined as 2), `add-x`, `fact`, and `mult`
- There are four declarations that do not have global scope: the parameters
- Lexical scoping allows programmers to use the same variable names multiple times
- The scoping rules make it possible to understand where a variable is bound or if it is *free*
- In the body of `addd-x-y-z` we say that `y` and `z` are *free variables*
- A free variable is one that is not bound to a declaration

Encapsulation

Lexical Scoping

- Most programming languages use lexical scoping
- Lexical scoping determines where a variable is bound based on the syntax of the programming language

Encapsulation

Lexical Scoping

- Most programming languages use lexical scoping
- Lexical scoping determines where a variable is bound based on the syntax of the programming language
- Global definitions are valid everywhere unless they are *shadowed* by a local declaration
- If the same variable name is reused in a program the local declaration is the bounding declaration.
- The closest declaration is always the bounding declaration

Encapsulation

Lexical Scoping

- Most programming languages use lexical scoping
- Lexical scoping determines where a variable is bound based on the syntax of the programming language
- Global definitions are valid everywhere unless they are *shadowed* by a local declaration
- If the same variable name is reused in a program the local declaration is the bounding declaration.
- The closest declaration is always the bounding declaration
- The lexical scoping rules are:
  1. The definitions not typed inside a `local`-expression have global lexical scope and are valid everywhere unless shadowed
  2. The lexical scope of parameters is the function that declares them unless shadowed
  3. The lexical scope of a local definition is the declarations and the body of the `local`-expression unless shadowed

Encapsulation

Lexical Scoping

- ```
  (define DELTA 10)
  (+ DELTA (local [(define DELTA 100)
                   (define VAL (+ DELTA DELTA))]
             VAL))
  ```
- What is the value of the sum?
  ```

Encapsulation

Lexical Scoping

- ```
  (define DELTA 10)
  (+ DELTA (local [(define DELTA 100)
                   (define VAL (+ DELTA DELTA))]
             VAL))
  ```
- What is the value of the sum?
- ```
  (define X 2)
  (define Y 3)

  (+ X Y (local [(define X 10)]
           (+ X (local [(define Y 10)]
                  (+ X Y)))))
  ```
- What is the value of the sum?

Encapsulation

Using Local Expressions

- Important to understand when and how to use a `local`-expression

Encapsulation

Using Local Expressions

- Important to understand when and how to use a `local`-expression
- There are four reasons to use a `local`-expression
  1. Encapsulation: package together related functions
  2. Make code more readable
  3. Make the use of abstract functions possible
  4. Eliminate multiple evaluations of the same expression

Encapsulation

Using Local Expressions

- Encapsulation is a good practice to develop because it makes refinements easier
- When a refinement is needed all the code is located in one place

Encapsulation

Using Local Expressions

- Encapsulation is a good practice to develop because it makes refinements easier
- When a refinement is needed all the code is located in one place
- To encapsulate code use these steps:
  1. Develop your program using any of the design recipes
  2. Test your functions thoroughly
  3. Identify related definitions to package together
  4. Identify the main function among the functions to be packaged
  5. Place all the functions inside a `local`-expression
  6. Place the `local`-expression inside a new function that has the same signature, purpose, and header as the main function
  7. In the body of the `local`-expression call the main function with the given arguments
  8. Comment out the sample expressions and the tests for all functions except the main function and comment out the sample expressions and the tests using sample computations for the main function

Encapsulation

Using Local Expressions

- Encapsulation is a good practice to develop because it makes refinements easier
- When a refinement is needed all the code is located in one place
- To encapsulate code use these steps:
  1. Develop your program using any of the design recipes
  2. Test your functions thoroughly
  3. Identify related definitions to package together
  4. Identify the main function among the functions to be packaged
  5. Place all the functions inside a `local`-expression
  6. Place the `local`-expression inside a new function that has the same signature, purpose, and header as the main function
  7. In the body of the `local`-expression call the main function with the given arguments
  8. Comment out the sample expressions and the tests for all functions except the main function and comment out the sample expressions and the tests using sample computations for the main function
- It is ***important*** to remember not to encapsulate functions before thorough testing
- Localized functions cannot be tested due to scoping rules
- **Test before encapsulation**

Encapsulation

Using Local Expressions

- Consider:

```
;; world → scene
;; Purpose: To draw the world in E-SCENE
(define (draw-world a-world)
  (draw-los (world-shots a-world)
            (draw-loa (world-aliens a-world)
                      (draw-rocket (world-rocket a-world) E-SCENE))))
```

- 3 auxiliary functions and a constant are used

Encapsulation

Using Local Expressions

- Consider:

```
;; world → scene
;; Purpose: To draw the world in E-SCENE
(define (draw-world a-world)
  (draw-los (world-shots a-world)
            (draw-loa (world-aliens a-world)
                      (draw-rocket (world-rocket a-world) E-SCENE))))
```

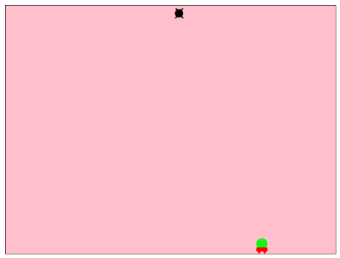
- 3 auxiliary functions and a constant are used
-

```
;; world → scene     Purpose: To draw the world in E-SCENE
(define (draw-world a-world)
  (local
    [(define E-SCENE-COLOR 'pink)
     (define E-SCENE (empty-scene E-SCENE-W E-SCENE-H E-SCENE-COLOR))
     ;; los scene → scene     Purpose: Draw given los in given scene
     (define (draw-los a-los scn)
      (if (empty? a-los)
          scn
          (draw-shot (first a-los) (draw-los (rest a-los) scn))))
     ;; loa scene → scene     Purpose: Draw given loa in given scene
     (define (draw-loa a-loa scn)
      (if (empty? a-loa) scn
          (draw-alien (first a-loa) (draw-loa (rest a-loa) scn))))
      ;; rocket scene → scene     Purpose: Draw rocket in given scene
      (define (draw-rocket a-rocket a-scene)
           (draw-ci ROCKET-IMG a-rocket ROCKET-Y a-scene))
      ;; world → scene    Purpose: To draw the world in E-SCENE
      (define (draw-world a-world)
       (draw-los (world-shots a-world)
                 (draw-loa
                   (world-aliens a-world)
                   (draw-rocket (world-rocket a-world) E-SCENE))))]
   (draw-world a-world)))
```

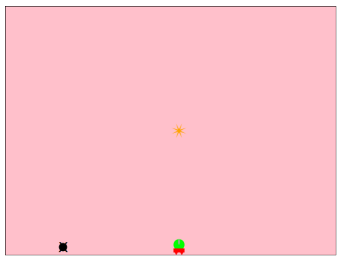
Encapsulation

Using Local Expressions

- ```
  ;; Tests using sample values for draw-world
  (check-expect
   (draw-world (make-world INIT-ROCKET2 (list INIT-ALIEN) DIR3 '())
  ```

  ```
  )
  (check-expect
   (draw-world (make-world INIT-ROCKET
                           (list INIT-ALIEN2)
                           DIR2
                           (list SHOT2)))
  ```

  ```
  )
  ```
  ```

Encapsulation

Using Local Expressions

- Observe that `draw-shot` is an auxiliary function that is only needed by `draw-los`
- The same observation may be made for `draw-alien` and `draw-loa`
- These ought to also be localized
- See textbook for refactored `draw-world` after all encapsulation

Encapsulation

Using Local Expressions

- Observe that `draw-shot` is an auxiliary function that is only needed by `draw-los`
- The same observation may be made for `draw-alien` and `draw-loa`
- These ought to also be localized
- See textbook for refactored `draw-world` after all encapsulation
- The code is organized in a manner that allows any reader (including yourself six months from now) to easily find related functions
- It also means that when a refinement is necessary we know where the new tested functions must be placed

Encapsulation

Homework

- Problems: 270–271

Encapsulation

Using Local Expressions

- In Section 106:

```
;; (listof sexpr) arrow (listof number) throws error
;; Purpose: To evaluate the sexprs in the given list
(define (eval-args a-losexpr)
  (if (empty? a-losexpr)
      '()
      (cons (eval-sexpr (first a-losexpr))
            (eval-args (rest a-losexpr)))))

;; function (listof number) → number throws error
;; Purpose: To apply the given function to the given numbers
(define (apply-f a-function a-lon)
  (cond [(eq? a-function '+) (sum-lon  a-lon)]
        [(eq? a-function '-) (subt-lon a-lon)]
        [else (mult-lon a-lon)]))

;; slist → number
;; Purpose: To evaluate the given slist
(define (eval-slist an-slist)
  (apply-f (first an-slist) (eval-args (rest an-slist))))

;; sexpr → number throws error
;; Purpose: To evaluate the given sexpr
(define (eval-sexpr a-sexpr)
  (if (number? a-sexpr)
      a-sexpr
      (eval-slist a-sexpr)))
```

- What does `(first a-losexpr)` represent? Do you remember?
- Is it reasonable to expect others reading this code for the first time to know or easily figure it out?

Encapsulation

Using Local Expressions

- In Section 106:

```
;; (listof sexpr) arrow (listof number) throws error
;; Purpose: To evaluate the sexprs in the given list
(define (eval-args a-losexpr)
  (if (empty? a-losexpr)
      '()
      (cons (eval-sexpr (first a-losexpr))
            (eval-args (rest a-losexpr)))))

;; function (listof number) → number throws error
;; Purpose: To apply the given function to the given numbers
(define (apply-f a-function a-lon)
  (cond [(eq? a-function '+) (sum-lon  a-lon)]
        [(eq? a-function '-) (subt-lon a-lon)]
        [else (mult-lon a-lon)]))

;; slist → number
;; Purpose: To evaluate the given slist
(define (eval-slist an-slist)
  (apply-f (first an-slist) (eval-args (rest an-slist))))

;; sexpr → number throws error
;; Purpose: To evaluate the given sexpr
(define (eval-sexpr a-sexpr)
  (if (number? a-sexpr)
      a-sexpr
      (eval-slist a-sexpr)))
```

- What does `(first a-losexpr)` represent? Do you remember?
- Is it reasonable to expect others reading this code for the first time to know or easily figure it out?
- To mitigate this problem a `local`-expression may be used to define intermediate values using descriptive variable names

Encapsulation

Using Local Expressions

- Consider:

```
;; slist → number
;; Purpose: To evaluate the given slist
(define (eval-slist an-slist)
  (apply-f (first an-slist) (eval-args (rest an-slist))))
```

- We can define local variables with names that elicit the meaning of the expressions

Encapsulation

Using Local Expressions

- Consider:

```
;; slist → number
;; Purpose: To evaluate the given slist
(define (eval-slist an-slist)
  (apply-f (first an-slist) (eval-args (rest an-slist))))
```

- We can define local variables with names that elicit the meaning of the expressions
- Local variables may be used to easily discern that `(first an-slist)` represents the arithmetic operator and that `(eval-args (rest an-slist))` the arguments for the arithmetic operator

Encapsulation

Using Local Expressions

- Consider:

```
;; slist → number
;; Purpose: To evaluate the given slist
(define (eval-slist an-slist)
  (apply-f (first an-slist) (eval-args (rest an-slist))))
```

- We can define local variables with names that elicit the meaning of the expressions
- Local variables may be used to easily discern that `(first an-slist)` represents the arithmetic operator and that `(eval-args (rest an-slist))` the arguments for the arithmetic operator
- Refactored function:

```
;; slist → number
;; Purpose: To evaluate the given slist
(define (eval-slist an-slist)
  (local [(define operator  (first an-slist))
          (define arg-exprs (rest an-slist))]
    (apply-f operator (eval-args arg-exprs))))
```

Encapsulation

Using Local Expressions

- Consider:

```
;; (listof sexpr) arrow (listof number) throws error
;; Purpose: To evaluate the sexprs in the given list
(define (eval-args a-losexpr)
 (if (empty? a-losexpr)
     '()
     (cons (eval-sexpr (first a-losexpr))
           (eval-args (rest a-losexpr)))))
```

Encapsulation

Using Local Expressions

- Consider:

```
;; (listof sexpr) arrow (listof number) throws error
;; Purpose: To evaluate the sexprs in the given list
(define (eval-args a-losexpr)
 (if (empty? a-losexpr)
     '()
     (cons (eval-sexpr (first a-losexpr))
           (eval-args (rest a-losexpr)))))
```

- Refactored function:

```
;; (listof sexpr) arrow (listof number) throws error
;; Purpose: To evaluate the sexprs in the given list
(define (eval-args a-losexpr)
  (if (empty? a-losexpr)
      '()
      (local [(define first-arg (first a-losexpr))
              (define rest-args (rest a-losexpr))]
        (cons (eval-sexpr first-arg)
              (eval-args  rest-args)))))
```

Encapsulation

Homework

- Problems: 272–274

Encapsulation

Using Local Expressions

- Consider the function to move a `(listof alien)`:

```
;; (listof alien) dir → (listof alien) Purpose: Move loa in given direction
(define (move-loa a-loa dir)
  (if (empty? a-loa)
      E-LOA
      (cons (move-alien (first a-loa) dir) (move-loa (rest a-loa) dir))))
```

- The same function, `move-alien`, is applied to every `alien`
- Suggests using `map`

Encapsulation

Using Local Expressions

- Consider the function to move a `(listof alien)`:

```
;; (listof alien) dir → (listof alien) Purpose: Move loa in given direction
(define (move-loa a-loa dir)
  (if (empty? a-loa)
      E-LOA
      (cons (move-alien (first a-loa) dir) (move-loa (rest a-loa) dir))))
```

- The same function, `move-alien`, is applied to every `alien`
- Suggests using `map`
- Recall, however, the signature for `map`:

  `(X → Y) (listof X) → (listof Y)`

- Expects a one-input function as input
- Do you see the problem?

Encapsulation

Using Local Expressions

- Consider the function to move a `(listof alien)`:

```
;; (listof alien) dir → (listof alien) Purpose: Move loa in given direction
(define (move-loa a-loa dir)
  (if (empty? a-loa)
      E-LOA
      (cons (move-alien (first a-loa) dir) (move-loa (rest a-loa) dir))))
```

- The same function, `move-alien`, is applied to every `alien`
- Suggests using `map`
- Recall, however, the signature for `map`:

  `(X → Y) (listof X) → (listof Y)`

- Expects a one-input function as input
- Do you see the problem?
- The signature for `move-alien`:

  `alien direction → alien`

- A two-input function cannot be given as input to `map`

Encapsulation

Using Local Expressions

- Consider the function to move a `(listof alien)`:

```
;; (listof alien) dir → (listof alien) Purpose: Move loa in given direction
(define (move-loa a-loa dir)
  (if (empty? a-loa)
      E-LOA
      (cons (move-alien (first a-loa) dir) (move-loa (rest a-loa) dir))))
```

- The same function, `move-alien`, is applied to every `alien`
- Suggests using `map`
- Recall, however, the signature for `map`:

  `(X → Y) (listof X) → (listof Y)`

- Expects a one-input function as input
- Do you see the problem?
- The signature for `move-alien`:

  `alien direction → alien`

- A two-input function cannot be given as input to `map`
- Observe that to move an alien the only thing that varies is the alien
- The direction is the same for all aliens

Encapsulation

Using Local Expressions

- Consider the function to move a `(listof alien)`:

```
;; (listof alien) dir → (listof alien) Purpose: Move loa in given direction
(define (move-loa a-loa dir)
  (if (empty? a-loa)
      E-LOA
      (cons (move-alien (first a-loa) dir) (move-loa (rest a-loa) dir))))
```

- The same function, `move-alien`, is applied to every `alien`
- Suggests using `map`
- Recall, however, the signature for `map`:

  `(X → Y) (listof X) → (listof Y)`

- Expects a one-input function as input
- Do you see the problem?
- The signature for `move-alien`:

  `alien direction → alien`

- A two-input function cannot be given as input to `map`
- Observe that to move an alien the only thing that varies is the alien
- The direction is the same for all aliens
- Suggests that inside `move-loa` the moving of an alien may be implemented as a one-input function

Encapsulation

Using Local Expressions

- Consider the function to move a `(listof alien)`:

```
;; (listof alien) dir → (listof alien) Purpose: Move loa in given direction
(define (move-loa a-loa dir)
  (if (empty? a-loa)
      E-LOA
      (cons (move-alien (first a-loa) dir) (move-loa (rest a-loa) dir))))
```

- The same function, `move-alien`, is applied to every `alien`
- Suggests using `map`
- Recall, however, the signature for `map`:

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

- Expects a one-input function as input
- Do you see the problem?
- The signature for `move-alien`:

```
alien direction → alien
```

- A two-input function cannot be given as input to `map`
- Observe that to move an alien the only thing that varies is the alien
- The direction is the same for all aliens
- Suggests that inside `move-loa` the moving of an alien may be implemented as a one-input function
- Refactored function:

```
;; (listof alien) dir → (listof alien)
;; Purpose: To move the given list of aliens in the
;;          given direction
(define (move-loa a-loa dir)
  (local [(define (alien-mover an-alien)
            (move-alien an-alien dir))]
    (map alien-mover a-loa)))
```

Encapsulation

Using Local Expressions

- ```
  ;; (listof alien) dir → (listof alien) Purpose: Move loa in given
  (define (move-loa a-loa dir)
    (if (empty? a-loa)
        E-LOA
        (cons (move-alien (first a-loa) dir)
              (move-loa (rest a-loa) dir))))
  ```
- ```
  ;; (listof alien) dir → (listof alien)
  ;; Purpose: To move the given list of aliens in the
  ;;          given direction
  (define (move-loa a-loa dir)
    (local [(define (alien-mover an-alien)
              (move-alien an-alien dir))]
      (map alien-mover a-loa)))
  ```
- A note of caution is in order
- The `local`-expression in `move-loa` cannot be eliminated using the draft refactoring recipe presented
- The problem is that `alien-mover` has a free variable, `dir`, that only exists inside `move-loa`
- Therefore, `alien-mover` cannot simply be moved outside the scope of `dir`
- We shall discuss how to solve this problem in Chapter 22

Encapsulation

Homework

- Problems: 275–277

Encapsulation

Using Local Expressions

- The use of `local`-expressions may make a program faster
- Programs are slowed down evaluating an expression more than once
- Abstract running time may be worsened

Encapsulation

Using Local Expressions

- The use of `local`-expressions may make a program faster
- Programs are slowed down evaluating an expression more than once
- Abstract running time may be worsened
- Consider:

```
;; number (listof item) → result Purpose: Get last item < a-price
(define (get-last a-price a-loi)
  (cond [(empty? a-loi) '()]
        [(< (item-price (first a-loi)) a-price)
         (if (empty? (get-last a-price (rest a-loi)))
             (first a-loi) same expr may be evaluated twice
             (get-last a-price (rest a-loi)))]
        [else (get-last a-price (rest a-loi))]))
```

Encapsulation

Using Local Expressions

- The use of `local`-expressions may make a program faster
- Programs are slowed down evaluating an expression more than once
- Abstract running time may be worsened
- Consider:

```
;; number (listof item) → result Purpose: Get last item < a-price
(define (get-last a-price a-loi)
  (cond [(empty? a-loi) '()]
        [(< (item-price (first a-loi)) a-price)
         (if (empty? (get-last a-price (rest a-loi)))
             (first a-loi) same expr may be evaluated twice
             (get-last a-price (rest a-loi)))]
        [else (get-last a-price (rest a-loi))]))
```

- Locally define a variable and only evaluate the expression once
- Reference the local variable instead of evaluating the expression again
- Refactored to:

```
;; number (listof item) → result
;; Purpose: Get last item < given price
(define (get-last a-price a-loi)
 (cond[(empty? a-loi) '()]
      [(< (item-price (first a-loi)) a-price)
       (local[(define rest-result (get-last a-price (rest a-loi)))]
        (if (empty? rest-result) (first a-loi) rest-result))]
      [else (get-last a-price (rest a-loi))]))
```

Encapsulation

Using Local Expressions

- Is this type of refactoring nitpicking or can it have a significant impact on performance?

Encapsulation

Using Local Expressions

- Is this type of refactoring nitpicking or can it have a significant impact on performance?
- Worst possible performance: price given to `get-last` is larger than any price in list
- An item with a smaller price exists in the rest oflist

Encapsulation

Using Local Expressions

- Is this type of refactoring nitpicking or can it have a significant impact on performance?
- Worst possible performance: price given to `get-last` is larger than any price in list
- An item with a smaller price exists in the rest oflist
- A list of with all identical items using:

```
;; int → item
;; Purpose: Returns a doll item that costs 30
(define (make-doll-item n) (make-item "doll" 30))
;; Sample expressions for make-doll-item
(define DOLL1 (make-item "doll" 30))
(define DOLL2 (make-item "doll" 30))
;; Tests using sample computations for make-doll-item
(check-expect (make-doll-item 4) DOLL1)
(check-expect (make-doll-item 8) DOLL2)
;; Tests using sample values for make-doll-item
(check-expect (make-doll-item -5) (make-item "doll" 30))
(check-expect (make-doll-item 0)  (make-item "doll" 30))
(define LOIL (interval-accum-l2h '() make-doll-item cons 0 19))
```

- `LOIL` is a (short) list of 20 identical items

Encapsulation

Using Local Expressions

- Rename second version as `get-last2` and experiment:

```
(time (get-last  1200 LOIL))
(time (get-last2 1200 LOIL))
```

Encapsulation

Using Local Expressions

- Rename second version as `get-last2` and experiment:

```
(time (get-last  1200 LOIL))
(time (get-last2 1200 LOIL))
```

- The results obtained for CPU time are:

Function	CPU Time
get-last	1547
get-last2	0

Encapsulation

Using Local Expressions

- Why is the `get-last2` much faster?

Encapsulation

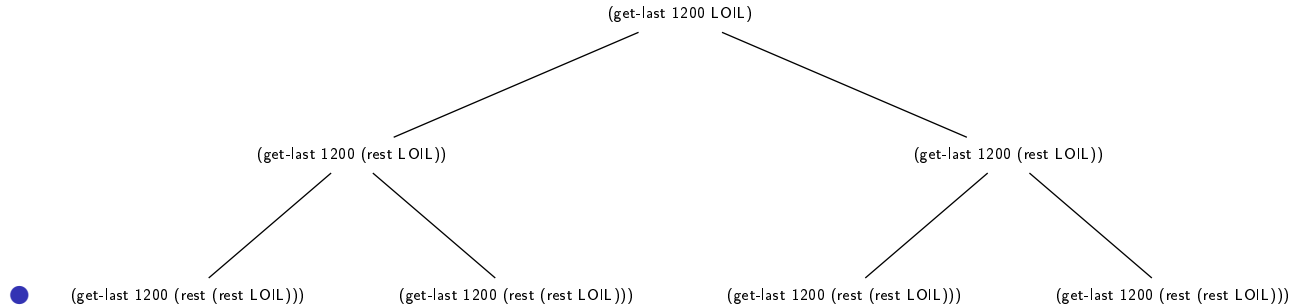
Using Local Expressions

- Why is the `get-last2` much faster?
- To answer this let us look at the abstract running time of both functions

Encapsulation

Using Local Expressions

- Why is the `get-last2` much faster?
- To answer this let us look at the abstract running time of both functions

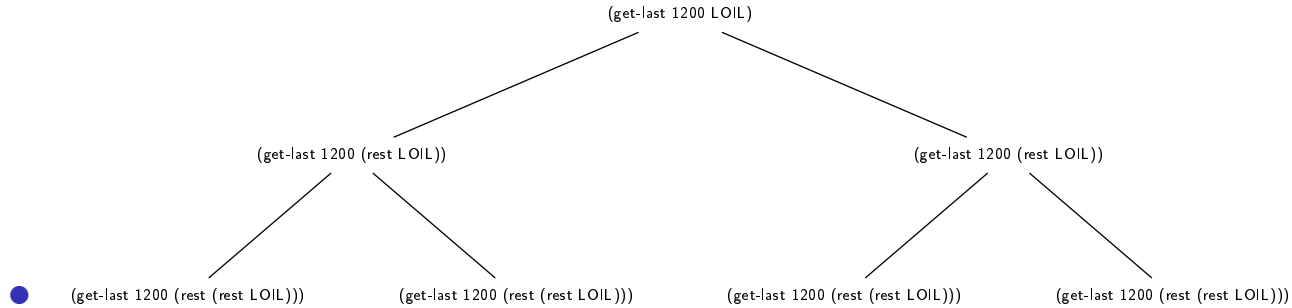


- `get-last` generates two calls for every element until `'()`
- How many calls are made for a list of length `n`?

Encapsulation

Using Local Expressions

- Why is the `get-last2` much faster?
- To answer this let us look at the abstract running time of both functions

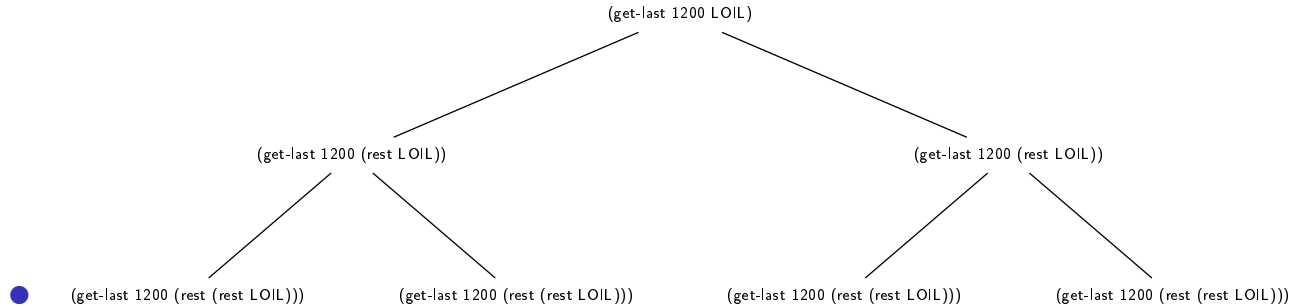


- `get-last` generates two calls for every element until `'()`
- How many calls are made for a list of length `n`?
- The binary tree of calls has height `n` and full
- The number of calls is equal to the number of nodes in a full binary tree

Encapsulation

Using Local Expressions

- Why is the `get-last2` much faster?
- To answer this let us look at the abstract running time of both functions



- `get-last` generates two calls for every element until `'()`
- How many calls are made for a list of length `n`?
- The binary tree of calls has height `n` and full
- The number of calls is equal to the number of nodes in a full binary tree
- Number of nodes in a full binary tree of height `h`: $2^{h+1} - 1$ nodes

h	Number of Nodes
0	0
1	3
2	7
3	15
4	31
...	...

- This means that the number of function calls for a list of length `n` is $2^{n+1} - 1$
- `get-last` is $O(2^n)$– grows exponentially

Encapsulation

Using Local Expressions

- What is the abstract running time of `get-last2`?

Encapsulation

Using Local Expressions

- What is the abstract running time of `get-last2`?
- For every function call there is at most one recursive
- The total number of function calls for a list of length `n` is `n`

Encapsulation

Using Local Expressions

- What is the abstract running time of `get-last2`?
- For every function call there is at most one recursive
- The total number of function calls for a list of length `n` is `n`
- `get-last2` `O(n)`.
- Explains why `get-last2` is much faster than `get-last`.

Encapsulation

Using Local Expressions

- What is the abstract running time of `get-last2`?
- For every function call there is at most one recursive
- The total number of function calls for a list of length `n` is `n`
- `get-last2` `O(n)`.
- Explains why `get-last2` is much faster than `get-last`.
- **LESSON**: evaluating the same expression multiple times may have a profound impact on abstract and actual running time
- It is not the case, however, that eliminating the repetitive evaluation of the same expression always leads to better abstract running time

Encapsulation

Using Local Expressions

- What is the abstract running time of `get-last2`?
- For every function call there is at most one recursive
- The total number of function calls for a list of length `n` is `n`
- `get-last2` `O(n)`.
- Explains why `get-last2` is much faster than `get-last`.
- **LESSON**: evaluating the same expression multiple times may have a profound impact on abstract and actual running time
- It is not the case, however, that eliminating the repetitive evaluation of the same expression always leads to better abstract running time
- A program that has an exponential abstract running time is not a practical solution
- The amount of work done grows too quickly and makes the computation of a solution unfeasible even for input of moderate size

Encapsulation

Using Local Expressions

- Problems: 278–280

Lambda Expressions

- Functions are a type of value just like numbers, strings, and images

Lambda Expressions

- Functions are a type of value just like numbers, strings, and images
- How do we create a function value? Return a function?
- This has profound implications for problem solving and program design
- Leads to functions that compute functions and to object oriented programming
- Why would you want to compute a function in the first place?

Lambda Expressions

- Functions are a type of value just like numbers, strings, and images
- How do we create a function value? Return a function?
- This has profound implications for problem solving and program design
- Leads to functions that compute functions and to object oriented programming
- Why would you want to compute a function in the first place?
- The short answer is that it is a common operation in human life

Lambda Expressions

- Functions are a type of value just like numbers, strings, and images
- How do we create a function value? Return a function?
- This has profound implications for problem solving and program design
- Leads to functions that compute functions and to object oriented programming
- Why would you want to compute a function in the first place?
- The short answer is that it is a common operation in human life
- Consider: composition of two functions as you have studied in HS Math:

  $(f \circ g)(x) = f(g(x))$

- Although subtle and not emphasized in a high school Mathematics course, the composition of `f` and `g` is a function
- ```
  ;; number → number   Purpose: Add 4 to the given number
  (define (add4 x) (+ x 4))

  ;; number → number   Purpose: Double the given number
  (define (double x) (* 2 x))
  ```
  ```

Lambda Expressions

- Functions are a type of value just like numbers, strings, and images
- How do we create a function value? Return a function?
- This has profound implications for problem solving and program design
- Leads to functions that compute functions and to object oriented programming
- Why would you want to compute a function in the first place?
- The short answer is that it is a common operation in human life
- Consider: composition of two functions as you have studied in HS Math:

  $(f \circ g)(x) = f(g(x))$

- Although subtle and not emphasized in a high school Mathematics course, the composition of f and g is a function
- ```
  ;; number → number   Purpose: Add 4 to the given number
  (define (add4 x) (+ x 4))

  ;; number → number   Purpose: Double the given number
  (define (double x) (* 2 x))
  ```
- ```
  ;; number → number   Purpose: Add 4 to double of the given number
  (define (add4-o-double x) (add4 (double x)))
  ```

Lambda Expressions

- Functions are a type of value just like numbers, strings, and images
- How do we create a function value? Return a function?
- This has profound implications for problem solving and program design
- Leads to functions that compute functions and to object oriented programming
- Why would you want to compute a function in the first place?
- The short answer is that it is a common operation in human life
- Consider: composition of two functions as you have studied in HS Math:

  $(f \circ g)(x) = f(g(x))$

- Although subtle and not emphasized in a high school Mathematics course, the composition of `f` and `g` is a function
- ```
  ;; number → number   Purpose: Add 4 to the given number
  (define (add4 x) (+ x 4))

  ;; number → number   Purpose: Double the given number
  (define (double x) (* 2 x))
  ```
- ```
  ;; number → number   Purpose: Add 4 to double of the given number
  (define (add4-o-double x) (add4 (double x)))
  ```
- How do we automate this?

Lambda Expressions

- Two ways of composing `add4` and `double`:

```
;; number → number Purpose: Add 4 to double of given number
(define (add4-o-double x) (add4 (double x)))
;; number → number Purpose: Double sum of given number and 4
(define (double-o-add4 x) (double (add4 x)))
```

- The result is two different functions

Lambda Expressions

- Two ways of composing `add4` and `double`:

```
;; number → number Purpose: Add 4 to double of given number
(define (add4-o-double x) (add4 (double x)))
;; number → number Purpose: Double sum of given number and 4
(define (double-o-add4 x) (double (add4 x)))
```

- The result is two different functions
- Good candidates for abstraction. Differences: the composed functions

Lambda Expressions

- Two ways of composing add4 and double:

```
;; number → number Purpose: Add 4 to double of given number
(define (add4-o-double x) (add4 (double x)))
;; number → number Purpose: Double sum of given number and 4
(define (double-o-add4 x) (double (add4 x)))
```

- The result is two different functions
- Good candidates for abstraction. Differences: the composed functions
-

```
;; (number → number) (number → number) number → number
;; Purpose: Return f(g(x))
(define (f-o-g f g x) (f (g x)))
;; number → number
;; Purpose: Add 4 to the double of the given number
(define (add4-o-double x) (f-o-g add4 double x))
;; number → number
;; Purpose: Double the sum of the given number and 4
(define (double-o-add4 x) (f-o-g double add4 x))
```

Lambda Expressions

- Two ways of composing `add4` and `double`:

```
;; number → number Purpose: Add 4 to double of given number
(define (add4-o-double x) (add4 (double x)))
;; number → number Purpose: Double sum of given number and 4
(define (double-o-add4 x) (double (add4 x)))
```

- The result is two different functions
- Good candidates for abstraction. Differences: the composed functions
-

```
;; (number → number) (number → number) number → number
;; Purpose: Return f(g(x))
(define (f-o-g f g x) (f (g x)))
;; number → number
;; Purpose: Add 4 to the double of the given number
(define (add4-o-double x) (f-o-g add4 double x))
;; number → number
;; Purpose: Double the sum of the given number and 4
(define (double-o-add4 x) (f-o-g double add4 x))
```

- The abstract function, however, is not quite what we want
- `x` is a parameter, but not a difference in original functions
- `f-o-g` should return a function, not the value given `x`
- This means `x` should not be a parameter in the abstract function

Lambda Expressions

- Two ways of composing `add4` and `double`:

```
;; number → number Purpose: Add 4 to double of given number
(define (add4-o-double x) (add4 (double x)))
;; number → number Purpose: Double sum of given number and 4
(define (double-o-add4 x) (double (add4 x)))
```

- The result is two different functions
- Good candidates for abstraction. Differences: the composed functions
-

```
;; (number → number) (number → number) number → number
;; Purpose: Return f(g(x))
(define (f-o-g f g x) (f (g x)))
;; number → number
;; Purpose: Add 4 to the double of the given number
(define (add4-o-double x) (f-o-g add4 double x))
;; number → number
;; Purpose: Double the sum of the given number and 4
(define (double-o-add4 x) (f-o-g double add4 x))
```

- The abstract function, however, is not quite what we want
- `x` is a parameter, but not a difference in original functions
- `f-o-g` should return a function, not the value given `x`
- This means `x` should not be a parameter in the abstract function
- We want an abstract function with the following signature and purpose:

```
;; (number → number) (number → number) → (number → number)
;; Purpose: Return the composition of the given functions
(define (f-o-g f g) ...)
```

Lambda Expressions

Anonymous Functions

- You probably suspect already that returning a function is easy
- Simply need to return the function you want

Lambda Expressions

Anonymous Functions

- You probably suspect already that returning a function is easy
- Simply need to return the function you want
- Consider:

```
;; number → (number → number)
;; Purpose: Return + if the given number is even.
;;          Otherwise, return -.
(define (choose-f x) (if (even? x) + -))

(check-expect ((choose-f 4) 1 5) 6)
(check-expect ((choose-f 3) 1 5) -4)
```

- The function `choose-f` returns a function
- `((choose-f 4) 1 5)` is `(+ 1 5)` `((choose-f 3) 1 5)` is `(- 1 5)`

Lambda Expressions

Anonymous Functions

- You probably suspect already that returning a function is easy
- Simply need to return the function you want
- Consider:

```
;; number → (number → number)
;; Purpose: Return + if the given number is even.
;;          Otherwise, return -.
(define (choose-f x) (if (even? x) + -))

(check-expect ((choose-f 4) 1 5) 6)
(check-expect ((choose-f 3) 1 5) -4)
```

- The function `choose-f` returns a function
- `((choose-f 4) 1 5)` is `(+ 1 5)` `((choose-f 3) 1 5)` is `(- 1 5)`
- Works because it handles functions that exist and have a name
- A function name is simply a variable whose value is a function

Lambda Expressions

Anonymous Functions

- You probably suspect already that returning a function is easy
- Simply need to return the function you want
- Consider:

```
;; number → (number → number)
;; Purpose: Return + if the given number is even.
;;          Otherwise, return -.
(define (choose-f x) (if (even? x) + -))

(check-expect ((choose-f 4) 1 5) 6)
(check-expect ((choose-f 3) 1 5) -4)
```

- The function `choose-f` returns a function
- `((choose-f 4) 1 5)` is `(+ 1 5)` `((choose-f 3) 1 5)` is `(- 1 5)`
- Works because it handles functions that exist and have a name
- A function name is simply a variable whose value is a function
- In contrast, we want `f-o-g` to return a function that does not have a name
- We need an expression that returns a function when it is evaluated

Lambda Expressions

Anonymous Functions

- You probably suspect already that returning a function is easy
- Simply need to return the function you want
- Consider:

```
;; number → (number → number)
;; Purpose: Return + if the given number is even.
;;          Otherwise, return -.
(define (choose-f x) (if (even? x) + -))

(check-expect ((choose-f 4) 1 5) 6)
(check-expect ((choose-f 3) 1 5) -4)
```

- The function `choose-f` returns a function
- `((choose-f 4) 1 5)` is `(+ 1 5)` `((choose-f 3) 1 5)` is `(- 1 5)`
- Works because it handles functions that exist and have a name
- A function name is simply a variable whose value is a function
- In contrast, we want `f-o-g` to return a function that does not have a name
- We need an expression that returns a function when it is evaluated
- Such an expression is a `lambda`-expression (or λ-expression):

```
expr   ::=   (lambda (symbol+) expr)
       ::=   (λ (symbol+) expr)
```

Lambda Expressions

Anonymous Functions

- You probably suspect already that returning a function is easy
- Simply need to return the function you want
- Consider:

```
;; number → (number → number)
;; Purpose: Return + if the given number is even.
;;          Otherwise, return -.
(define (choose-f x) (if (even? x) + -))

(check-expect ((choose-f 4) 1 5) 6)
(check-expect ((choose-f 3) 1 5) -4)
```

- The function `choose-f` returns a function
- `((choose-f 4) 1 5)` is `(+ 1 5)` `((choose-f 3) 1 5)` is `(- 1 5)`
- Works because it handles functions that exist and have a name
- A function name is simply a variable whose value is a function
- In contrast, we want `f-o-g` to return a function that does not have a name
- We need an expression that returns a function when it is evaluated
- Such an expression is a `lambda`-expression (or λ-expression):

```
expr   ::=   (lambda (symbol+) expr)
       ::=   (λ (symbol+) expr)
```

- The following expression returns a function that adds 10 to its input:

```
(λ (a-number) (+ a-number 10))
```

- We may apply this function to a number:

```
((λ (a-number) (+ a-number 10)) 25)
```

Lambda Expressions

Anonymous Functions

- A λ-expression allows us to create an *anonymous function*
- If we wish to give such a function a name then we use `define`:

```
(define add10 (λ (a-number) (+ a-number 10)))
```

Lambda Expressions

Anonymous Functions

- A λ-expression allows us to create an *anonymous function*
- If we wish to give such a function a name then we use `define`:

```
(define add10 (λ (a-number) (+ a-number 10)))
```

- It is now possible for you to understand that:

```
(define (<name> <name>$^+$) expr)
```

  is syntactic sugar for:

```
(define <name> (lambda (<name>$^+$) expr))
```

Lambda Expressions

Anonymous Functions

- A λ-expression allows us to create an *anonymous function*
- If we wish to give such a function a name then we use `define`:

```
(define add10 (λ (a-number) (+ a-number 10)))
```

- It is now possible for you to understand that:

```
(define (<name> <name>$^+$) expr)
```

  is syntactic sugar for:

```
(define <name> (lambda (<name>$^+$) expr))
```

- You should consider using a λ-expression on its own (i.e., without binding its value to a variable) when:
  - A new function is needed.
  - The needed function is not recursive.
  - The needed function is only used once.
- Keep in mind that a λ-expression evaluates to an anonymous function
- It cannot recursively call itself
- In order for a function to be recursive it must have a name
- Commonly, a λ-expression is used to provide an argument to a function or to return a function as the value of a function call

Lambda Expressions

Homework

- Problems: 281–285

Lambda Expressions

Revisiting Function Composition

- Given `add4` and `double`, we need two functions: `(add4 ∘ double)(x)` and `(double ∘ add4)(x)`

Lambda Expressions

Revisiting Function Composition

- Given `add4` and `double`, we need two functions: `(add4 ∘ double)(x)` and `(double ∘ add4)(x)`
- We can define these functions using a λ-expression as follows:

```
;; number → number
;; Purpose: Add 4 to the double of the given number
(define add4-o-double (λ (x) (add4 (double x))))

;; number → number
;; Purpose: Double the sum of the given number and 4
(define double-o-add4 (λ (x) (double (add4 x))))
```

Lambda Expressions

Revisiting Function Composition

- Given `add4` and `double`, we need two functions: `(add4 ∘ double)(x)` and `(double ∘ add4)(x)`
- We can define these functions using a λ-expression as follows:

```
;; number → number
;; Purpose: Add 4 to the double of the given number
(define add4-o-double (λ (x) (add4 (double x))))

;; number → number
;; Purpose: Double the sum of the given number and 4
(define double-o-add4 (λ (x) (double (add4 x))))
```

- The two expressions are nearly identical
- Strongly suggests abstracting over the expressions

Lambda Expressions

Revisiting Function Composition

- Given `add4` and `double`, we need two functions: `(add4 ∘ double)(x)` and `(double ∘ add4)(x)`
- We can define these functions using a λ-expression as follows:

```
;; number → number
;; Purpose: Add 4 to the double of the given number
(define add4-o-double (λ (x) (add4 (double x))))

;; number → number
;; Purpose: Double the sum of the given number and 4
(define double-o-add4 (λ (x) (double (add4 x))))
```

- The two expressions are nearly identical
- Strongly suggests abstracting over the expressions
- The abstraction step yields:

```
;; (number → number) (number → number) → (number → number)
;; Purpose: Return f(g(x))
(define (f-o-g f g) (λ (x) (f (g x))))
```

- Now have the correct abstraction
- A function that given two functions returns a function for the composition of the two functions

Lambda Expressions

Revisiting Function Composition

- ```
  ;; number → number
  ;; Purpose: Add 4 to the double of the given number
  (define add4-o-double (λ (x) (add4 (double x))))
  ;; number → number
  ;; Purpose: Double the sum of the given number and 4
  (define double-o-add4 (λ (x) (double (add4 x))))
  ```
- We can now refactor:

  ```
  ;; (number → number) (number → number) → (number → number)
  ;; Purpose: Return f(g(x))
  (define (f-o-g f g) (λ (x) (f (g x))))
  ;; number → number
  ;; Purpose: Add 4 to the double of the given number
  (define add4-o-double (f-o-g add4 double))
  ;; number → number
  ;; Purpose: Double the sum of the given number and 4
  (define double-o-add4 (f-o-g double add4))
  ```
  ```

Lambda Expressions

Revisiting Function Composition

- ```
  ;; number → number
  ;; Purpose: Add 4 to the double of the given number
  (define add4-o-double (λ (x) (add4 (double x))))
  ;; number → number
  ;; Purpose: Double the sum of the given number and 4
  (define double-o-add4 (λ (x) (double (add4 x))))
  ```
- We can now refactor:
  ```
  ;; (number → number) (number → number) → (number → number)
  ;; Purpose: Return f(g(x))
  (define (f-o-g f g) (λ (x) (f (g x))))
  ;; number → number
  ;; Purpose: Add 4 to the double of the given number
  (define add4-o-double (f-o-g add4 double))
  ;; number → number
  ;; Purpose: Double the sum of the given number and 4
  (define double-o-add4 (f-o-g double add4))
  ```
- We can design functions that compute functions
- We can now use functions that we did not write, but computed for us
- This is a powerful tool in your design arsenal
- A program can create specialized functions as needed
- `f-o-g` is so useful that it is provided by ISL+ as `compose`
  ```

Lambda Expressions

Revisiting Function Composition

- ```
  ;; number → number
  ;; Purpose: Add 4 to the double of the given number
  (define add4-o-double (λ (x) (add4 (double x))))
  ;; number → number
  ;; Purpose: Double the sum of the given number and 4
  (define double-o-add4 (λ (x) (double (add4 x))))
  ```
- We can now refactor:
  ```
  ;; (number → number) (number → number) → (number → number)
  ;; Purpose: Return f(g(x))
  (define (f-o-g f g) (λ (x) (f (g x))))
  ;; number → number
  ;; Purpose: Add 4 to the double of the given number
  (define add4-o-double (f-o-g add4 double))
  ;; number → number
  ;; Purpose: Double the sum of the given number and 4
  (define double-o-add4 (f-o-g double add4))
  ```
- We can design functions that compute functions
- We can now use functions that we did not write, but computed for us
- This is a powerful tool in your design arsenal
- A program can create specialized functions as needed
- `f-o-g` is so useful that it is provided by ISL+ as `compose`
- Used incorrectly C3PO is right:

  Shut me down. Machines building machines. How perverse.
  ```

Lambda Expressions

Homework

- Problems: 287–288

Lambda Expressions

Curried Functions

- Creating specialized recursive functions as a program is evaluated may be implemented using both `local`- and λ-expressions
- Consider:

```
;; loa scene → scene Purpose: Draw loa in scene
(define (draw-loa a-loa scn)
  (if (empty? a-loa)
      scn
      (draw-alien (first a-loa) (draw-loa (rest a-loa) scn))))

;; los scene → scene Purpose: Draw los in scene
(define (draw-los a-los scn)
  (if (empty? a-los)
      scn
      (draw-shot (first a-los) (draw-los (rest a-los) scn))))
```

- Clear candidates for abstraction

Lambda Expressions

Curried Functions

- Creating specialized recursive functions as a program is evaluated may be implemented using both `local`- and λ-expressions
- Consider:

```
;; loa scene → scene Purpose: Draw loa in scene
(define (draw-loa a-loa scn)
  (if (empty? a-loa)
      scn
      (draw-alien (first a-loa) (draw-loa (rest a-loa) scn))))

;; los scene → scene Purpose: Draw los in scene
(define (draw-los a-los scn)
  (if (empty? a-los)
      scn
      (draw-shot (first a-los) (draw-los (rest a-los) scn))))
```

- Clear candidates for abstraction
- Difference: function used to draw the first list element

Lambda Expressions

Curried Functions

- Creating specialized recursive functions as a program is evaluated may be implemented using both `local`- and λ-expressions
- Consider:

```
;; loa scene → scene Purpose: Draw loa in scene
(define (draw-loa a-loa scn)
  (if (empty? a-loa)
      scn
      (draw-alien (first a-loa) (draw-loa (rest a-loa) scn))))

;; los scene → scene Purpose: Draw los in scene
(define (draw-los a-los scn)
  (if (empty? a-los)
      scn
      (draw-shot (first a-los) (draw-los (rest a-los) scn))))
```

- Clear candidates for abstraction
- Difference: function used to draw the first list element
- The abstraction step yields:

```
;; (listof X) image → image Purpose: Draw list in scene
(define (draw-lox draw-x a-lox scn)
  (if (empty? a-lox)
      scn
      (draw-x (first a-lox) (draw-lox draw-x (rest a-lox) scn))))
```

Lambda Expressions

Curried Functions

- ```
  ;; (listof X) image → image
  ;; Purpose: To draw the given list in the given scene
  (define (draw-lox draw-x a-lox scn)
    (if (empty? a-lox)
        scn
        (draw-x (first a-lox)
                (draw-lox draw-x (rest a-lox) scn))))
  ```
- Has a structure very similar to `accum` from Section 116.1
  ```

Lambda Expressions

Curried Functions

- ```
  ;; (listof X) image → image
  ;; Purpose: To draw the given list in the given scene
  (define (draw-lox draw-x a-lox scn)
    (if (empty? a-lox)
        scn
        (draw-x (first a-lox)
                (draw-lox draw-x (rest a-lox) scn))))
  ```
- Has a structure very similar to `accum` from Section 116.1
- Refactored to have the same structure using `id`:

```
;; (listof X) image → image
;; Purpose: To draw the given list in the given scene
(define (draw-lox draw-x a-lox scn)
  (if (empty? a-lox)
      scn
      (draw-x (id (first a-lox))
              (draw-lox draw-x (rest a-lox) scn))))
```
```

Lambda Expressions

Curried Functions

- ```
  ;; (listof X) image → image
  ;; Purpose: To draw the given list in the given scene
  (define (draw-lox draw-x a-lox scn)
    (if (empty? a-lox)
        scn
        (draw-x (first a-lox)
                (draw-lox draw-x (rest a-lox) scn))))
  ```
- Has a structure very similar to `accum` from Section 116.1
- Refactored to have the same structure using `id`:

```
;; (listof X) image → image
;; Purpose: To draw the given list in the given scene
(define (draw-lox draw-x a-lox scn)
  (if (empty? a-lox)
      scn
      (draw-x (id (first a-lox))
              (draw-lox draw-x (rest a-lox) scn))))
```

- `draw-lox` may be refactored using `accum`:

```
;; (listof X) image → image
;; Purpose: To draw the given list in the given scene
(define (draw-lox draw-x a-lox scn)
  (accum scn id draw-x a-lox))
```
```

Lambda Expressions

Curried Functions

- Refactoring the functions abstracted over yields:

```
;; (listof alien) scene → scene
;; Purpose: To draw the given loa in the given scene
(define (draw-loa a-loa scn) (draw-lox draw-alien a-loa scn))
;; (listof shot) scene → scene
;; Purpose: To draw the given los in the given scene
(define (draw-los a-los scn) (draw-lox draw-shot a-los scn))
```

- Still a lot of repetition

Lambda Expressions

Curried Functions

- Refactoring the functions abstracted over yields:

```
;; (listof alien) scene → scene
;; Purpose: To draw the given loa in the given scene
(define (draw-loa a-loa scn) (draw-lox draw-alien a-loa scn))
;; (listof shot) scene → scene
;; Purpose: To draw the given los in the given scene
(define (draw-los a-los scn) (draw-lox draw-shot a-los scn))
```

- Still a lot of repetition
- Difference is the function used to draw either an `alien` or a `shot`

Lambda Expressions

Curried Functions

- Refactoring the functions abstracted over yields:

```
;; (listof alien) scene → scene
;; Purpose: To draw the given loa in the given scene
(define (draw-loa a-loa scn) (draw-lox draw-alien a-loa scn))
;; (listof shot) scene → scene
;; Purpose: To draw the given los in the given scene
(define (draw-los a-los scn) (draw-lox draw-shot a-los scn))
```

- Still a lot of repetition
- Difference is the function used to draw either an `alien` or a `shot`
- Abstraction suggests a function outlined as follows:

```
;; (X → image) → ???    ;; Purpose: ???
(define (draw-lox-maker draw-x) (...))
```

- What should such a function return?

Lambda Expressions

Curried Functions

- Refactoring the functions abstracted over yields:

```
;; (listof alien) scene → scene
;; Purpose: To draw the given loa in the given scene
(define (draw-loa a-loa scn) (draw-lox draw-alien a-loa scn))
;; (listof shot) scene → scene
;; Purpose: To draw the given los in the given scene
(define (draw-los a-los scn) (draw-lox draw-shot a-los scn))
```

- Still a lot of repetition
- Difference is the function used to draw either an `alien` or a `shot`
- Abstraction suggests a function outlined as follows:

```
;; (X → image) → ???    ;; Purpose: ???
(define (draw-lox-maker draw-x) (...))
```

- What should such a function return?
- This proposed abstract function only takes as input a drawing function
- Of little use without the list of elements to draw and the scene
- This means that this list and scene are still needed as input

Lambda Expressions

Curried Functions

- Refactoring the functions abstracted over yields:

```
;; (listof alien) scene → scene
;; Purpose: To draw the given loa in the given scene
(define (draw-loa a-loa scn) (draw-lox draw-alien a-loa scn))
;; (listof shot) scene → scene
;; Purpose: To draw the given los in the given scene
(define (draw-los a-los scn) (draw-lox draw-shot a-los scn))
```

- Still a lot of repetition
- Difference is the function used to draw either an `alien` or a `shot`
- Abstraction suggests a function outlined as follows:

```
;; (X → image) → ???    ;; Purpose: ???
(define (draw-lox-maker draw-x) (...))
```

- What should such a function return?
- This proposed abstract function only takes as input a drawing function
- Of little use without the list of elements to draw and the scene
- This means that this list and scene are still needed as input
- Needed ⇒ `draw-lox` must return a function that consumes these inputs
- Returned function must capture similarities in `draw-loa` and `draw-los`

Lambda Expressions

Curried Functions

- Refactoring the functions abstracted over yields:

```
;; (listof alien) scene → scene
;; Purpose: To draw the given loa in the given scene
(define (draw-loa a-loa scn) (draw-lox draw-alien a-loa scn))
;; (listof shot) scene → scene
;; Purpose: To draw the given los in the given scene
(define (draw-los a-los scn) (draw-lox draw-shot a-los scn))
```

- Still a lot of repetition
- Difference is the function used to draw either an `alien` or a `shot`
- Abstraction suggests a function outlined as follows:

```
;; (X → image) → ???    ;; Purpose: ???
(define (draw-lox-maker draw-x) (...))
```

- What should such a function return?
- This proposed abstract function only takes as input a drawing function
- Of little use without the list of elements to draw and the scene
- This means that this list and scene are still needed as input
- Needed ⇒ `draw-lox` must return a function that consumes these inputs
- Returned function must capture similarities in `draw-loa` and `draw-los`
- The abstraction step yields:

```
;; (X image → image) → ((listof X) image → image)
;; Purpose: Return a function to draw a (listof X)
(define (draw-lox-maker draw-x)
  (λ (a-lox scn) (draw-lox draw-x a-lox scn)))
```

Lambda Expressions

Curried Functions

- ```
  ;; (X image → image) → ((listof X) image → image)
  ;; Purpose: Return a function to draw a (listof X)
  (define (draw-lox-maker draw-x)
    (λ (a-lox scn) (draw-lox draw-x a-lox scn)))
  ```
  ```

Lambda Expressions

Curried Functions

- ```
  ;; (X image → image) → ((listof X) image → image)
  ;; Purpose: Return a function to draw a (listof X)
  (define (draw-lox-maker draw-x)
    (λ (a-lox scn) (draw-lox draw-x a-lox scn)))
  ```
- Example of a `curried function`
- Consumes part of the input & returns function to consume rest of the input
  ```

Lambda Expressions

Curried Functions

- ```
  ;; (X image → image) → ((listof X) image → image)
  ;; Purpose: Return a function to draw a (listof X)
  (define (draw-lox-maker draw-x)
    (λ (a-lox scn) (draw-lox draw-x a-lox scn)))
  ```
- Example of a `curried function`
- Consumes part of the input & returns function to consume rest of the input
- The returned function is specialized using the value given as input
  ```

Lambda Expressions

Curried Functions

- ```
  ;; (X image → image) → ((listof X) image → image)
  ;; Purpose: Return a function to draw a (listof X)
  (define (draw-lox-maker draw-x)
    (λ (a-lox scn) (draw-lox draw-x a-lox scn)))
  ```
- Example of a `curried function`
- Consumes part of the input & returns function to consume rest of the input
- The returned function is specialized using the value given as input
- Consider refactoring the functions abstracted over:

  ```
  ;; loa scene → scene Purpose: Draw loa in scene
  (define draw-loa (draw-lox-maker draw-alien))
  ;; los scene → scene Purpose: Draw los in scene
  (define draw-los (draw-lox-maker draw-shot))
  ```
- What are `draw-loa` and `draw-los`?
  ```

Lambda Expressions

Curried Functions

- ```
  ;; (X image → image) → ((listof X) image → image)
  ;; Purpose: Return a function to draw a (listof X)
  (define (draw-lox-maker draw-x)
    (λ (a-lox scn) (draw-lox draw-x a-lox scn)))
  ```
- Example of a `curried function`
- Consumes part of the input & returns function to consume rest of the input
- The returned function is specialized using the value given as input
- Consider refactoring the functions abstracted over:

  ```
  ;; loa scene → scene Purpose: Draw loa in scene
  (define draw-loa (draw-lox-maker draw-alien))
  ;; los scene → scene Purpose: Draw los in scene
  (define draw-los (draw-lox-maker draw-shot))
  ```
- What are `draw-loa` and `draw-los`?
- Plug-in the given argument to `draw-lox-maker` and substitute the result into the above definitions:

  ```
  ;; loa scene → scene
  ;; Purpose: To draw the given loa in the given scene
  (define draw-loa (λ (a-lox scn) (draw-lox draw-alien a-lox scn)))
  ;; los scene → scene
  ;; Purpose: To draw the given los in the given scene
  (define draw-los (λ (a-lox scn) (draw-lox draw-shot a-lox scn)))
  ```
- If we use the `define` syntactic sugar and change the name of `a-lox` we obtain exactly the functions that were abstracted over
  ```

Lambda Expressions

Curried Functions

- Curried functions allow programmers to stage providing the inputs
- Allows for the partial evaluation of functions to produce specialized functions
- Do you feel this is strange?

Lambda Expressions

Curried Functions

- Curried functions allow programmers to stage providing the inputs
- Allows for the partial evaluation of functions to produce specialized functions
- Do you feel this is strange?
- Consider that without knowing it you have been using curried functions

Lambda Expressions

Curried Functions

- Curried functions allow programmers to stage providing the inputs
- Allows for the partial evaluation of functions to produce specialized functions
- Do you feel this is strange?
- Consider that without knowing it you have been using curried functions
- The `big-bang` expression used to create video games and simulations is an example of currying in action

Lambda Expressions

Curried Functions

- Curried functions allow programmers to stage providing the inputs
- Allows for the partial evaluation of functions to produce specialized functions
- Do you feel this is strange?
- Consider that without knowing it you have been using curried functions
- The `big-bang` expression used to create video games and simulations is an example of currying in action
- As a programmer you provide the functions, for example, to process keystrokes and to process clock ticks
- Later a player and the computer provide the rest of the input (i.e., the keystrokes and the world)

Lambda Expressions

Curried Functions

- The power of curried functions allows us to revisit `move-loa`:

```
;; (listof alien) dir → (listof alien)
;; Purpose: To move the given list of aliens in the
;;          given direction
(define (move-loa a-loa dir)
  (local [(define (alien-mover an-alien)
            (move-alien an-alien dir))]
    (map alien-mover a-loa)))
```

- Recall that we were unable to remove the `local`-expression because `alien-mover` has a free variable

Lambda Expressions

Curried Functions

- The power of curried functions allows us to revisit `move-loa`:

```
;; (listof alien) dir → (listof alien)
;; Purpose: To move the given list of aliens in the
;;          given direction
(define (move-loa a-loa dir)
  (local [(define (alien-mover an-alien)
            (move-alien an-alien dir))]
    (map alien-mover a-loa)))
```

- Recall that we were unable to remove the `local`-expression because `alien-mover` has a free variable
- To remove the `local`-expression an `alien-mover` function specialized with the value of `dir` is needed
- This can be done using a curried function

Lambda Expressions

Curried Functions

- The power of curried functions allows us to revisit `move-loa`:

```
;; (listof alien) dir → (listof alien)
;; Purpose: To move the given list of aliens in the
;;          given direction
(define (move-loa a-loa dir)
  (local [(define (alien-mover an-alien)
            (move-alien an-alien dir))]
    (map alien-mover a-loa)))
```

- Recall that we were unable to remove the `local`-expression because `alien-mover` has a free variable
- To remove the `local`-expression an `alien-mover` function specialized with the value of `dir` is needed
- This can be done using a curried function
- When a local function has free variables a function that takes as input the free variables and that returns a specialized function is needed

Lambda Expressions

Curried Functions

- Refactored code:

```
;; dir → (alien → alien)
;; Purpose: Return a function to move an alien in the
;;          given direction
(define (alien-mover-maker a-dir)
  (λ (an-alien) (move-alien an-alien a-dir)))

;; (listof alien) dir → (listof alien)
;; Purpose: To move the given list of aliens in the
;;          given direction
(define (move-loa a-loa dir)
  (map (alien-mover-maker dir) a-loa))
```

Lambda Expressions

Curried Functions

- Refactored code:

```
;; dir → (alien → alien)
;; Purpose: Return a function to move an alien in the
;;          given direction
(define (alien-mover-maker a-dir)
  (λ (an-alien) (move-alien an-alien a-dir)))

;; (listof alien) dir → (listof alien)
;; Purpose: To move the given list of aliens in the
;;          given direction
(define (move-loa a-loa dir)
  (map (alien-mover-maker dir) a-loa))
```

- Cautionary note: computing a function's complete set of free variables and lifting it out of its lexical scope is a nontrivial process
- Such lifting becomes harder when local functions are mutually recursive
- The process to lift functions outside of their scope is called *λ-lifting*
- You may study it in a future course on the implementation of programming languages or on your own

Lambda Expressions

Homework

- Problems: 289–291

Lambda Expressions

Designing Using Existing Abstractions

- You may have the impression that designing abstract functions is a lot of work
- You may feel that it always requires abstracting over similar functions and refactoring the functions abstracted over

Lambda Expressions

Designing Using Existing Abstractions

- You may have the impression that designing abstract functions is a lot of work
- You may feel that it always requires abstracting over similar functions and refactoring the functions abstracted over
- Without a doubt this is an effective way to write more elegant and easier to maintain code
- An abstract function may be directly designed just like any other function

Lambda Expressions

Designing Using Existing Abstractions

- In Mathematics, a set of ordered terms is called a *sequence*
- The sequence of the first 5 odd natural numbers is: `1 3 5 7 9`
- This sequence is finite
- A sequence may also be infinite

Lambda Expressions

Designing Using Existing Abstractions

- In Mathematics, a set of ordered terms is called a *sequence*
- The sequence of the first 5 odd natural numbers is: `1 3 5 7 9`
- This sequence is finite
- A sequence may also be infinite
- You have probably seen the sequence of odd natural numbers written as follows:

  `1 3 5 7 9 ...`

Lambda Expressions

Designing Using Existing Abstractions

- In Mathematics, a set of ordered terms is called a *sequence*
- The sequence of the first 5 odd natural numbers is: `1 3 5 7 9`
- This sequence is finite
- A sequence may also be infinite
- You have probably seen the sequence of odd natural numbers written as follows:

  `1 3 5 7 9 ...`
- Each term in a sequence may be computed using its index in the sequence with indexing starts at 0:

```
;; natunum → natnum     Purpose: Return ith odd number
(define (ith-odd i) (add1 (* 2 i)))

;; Sample expressions for ith-odd
(define 0TH-ODD (add1 (* 2 0)))
(define 9TH-ODD (add1 (* 2 9)))

;; Tests using sample computations for ith-odd
(check-expect (ith-odd 0) 0TH-ODD)
(check-expect (ith-odd 9) 9TH-ODD)

;; Tests using sample values for ith-odd
(check-expect (ith-odd 2) 5)
(check-expect (ith-odd 4) 9)
```

Lambda Expressions

Designing Using Existing Abstractions

- The sum of the elements of a sequence is called a *series*
- Summing up the elements of a sequence may provide the means by which to compute a number

Lambda Expressions

Designing Using Existing Abstractions

- The sum of the elements of a sequence is called a *series*
- Summing up the elements of a sequence may provide the means by which to compute a number
- The sum of the first `n` elements of the sequence of odd natural numbers is equal to n^2:

$$0^2 = 0$$

$$3^2 = 1 + 3 + 5$$

$$5^2 = 1 + 3 + 5 + 7 + 9$$

Lambda Expressions

Designing Using Existing Abstractions

- The sum of the elements of a sequence is called a *series*
- Summing up the elements of a sequence may provide the means by which to compute a number
- The sum of the first `n` elements of the sequence of odd natural numbers is equal to n^2:

0^2 = 0

3^2 = 1 + 3 + 5

5^2 = 1 + 3 + 5 + 7 + 9

- Mathematicians have created syntax for summing the terms of a sequence:

0^2 = 0

3^2 = $\Sigma_{i=0}^{2}$(ith-odd i)

5^2 = $\Sigma_{i=0}^{4}$(ith-odd i)

Lambda Expressions

Designing Using Existing Abstractions

- The sum of the elements of a sequence is called a *series*
- Summing up the elements of a sequence may provide the means by which to compute a number
- The sum of the first `n` elements of the sequence of odd natural numbers is equal to n^2:

  0^2 = 0

  3^2 = 1 + 3 + 5

  5^2 = 1 + 3 + 5 + 7 + 9
- Mathematicians have created syntax for summing the terms of a sequence:

  0^2 = 0

  3^2 = $\Sigma_{i=0}^{2}$(ith-odd i)

  5^2 = $\Sigma_{i=0}^{4}$(ith-odd i)
- A function for the terms for any sequence may be used:

  series_f(n) = $\Sigma_{i=0}^{n-1}$(f i)
- Observe that this defines a curried function
- It first takes as input the term function and then to actually compute a value it takes the number of terms to add

Lambda Expressions

Designing Using Existing Abstractions

- ```
  ;; (natnum → number) → (natnum → number)
  ;; Purpose: Return function for series defined by given function
  (define (series term-f)
  ```
  ```

Lambda Expressions

Designing Using Existing Abstractions

- ```
  ;; (natnum → number) → (natnum → number)
  ;; Purpose: Return function for series defined by given function
  (define (series term-f)
  ```
- ```
   (lambda (n)
  ```

Lambda Expressions

Designing Using Existing Abstractions

- ```
  ;; (natnum → number) → (natnum → number)
  ;; Purpose: Return function for series defined by given function
  (define (series term-f)
  ```
- ```
   (lambda (n)
  ```
- ```
    (local [;; natnum → number Purpose: Compute series using n terms
            (define (add-terms n)
              (if (= n 0)
                  (term-f 0)
                  (+ (term-f n) (add-terms (sub1 n)))))]
      (if (= n 0)
          (term-f 0)
          (add-terms (sub1 n))))))
  ```
- `(sub1 n)` because the number of integers in `[0..n-1]` is `n`
  ```

Lambda Expressions

Designing Using Existing Abstractions

- ```
  ;; (natnum → number) → (natnum → number)
  ;; Purpose: Return function for series defined by given function
  (define (series term-f)
  ```
- ```
   (lambda (n)
  ```
- ```
    (local [;; natnum → number Purpose: Compute series using n terms
            (define (add-terms n)
              (if (= n 0)
                  (term-f 0)
                  (+ (term-f n) (add-terms (sub1 n)))))]
      (if (= n 0)
          (term-f 0)
          (add-terms (sub1 n))))))
  ```
- `(sub1 n)` because the number of integers in `[0..n-1]` is `n`
- Function to compute n^2:

```
;; natnum → natnum  Purpose: Compute square of number
(define n-square (series ith-odd))

;; Sample expressions for nth-square
(define ZERO-SQ  (n-square 0))
(define THREE-SQ (n-square 3))

;; Tests using sample computations for nth-square
(check-expect (n-square 0) ZERO-SQ)
(check-expect (n-square 3) THREE-SQ)

;; Tests using sample values for nth-square
(check-expect (n-square 10) 100)
(check-expect (n-square 4)   16)
(check-expect (n-square 325) (sqr 325))
(check-expect (n-square 57)  (sqr 57))
```
```

Lambda Expressions

Designing Using Existing Abstractions

- An irrational number, like π, cannot be exactly represented by a fraction
- How then can the value of π be represented?

Lambda Expressions

Designing Using Existing Abstractions

- An irrational number, like π, cannot be exactly represented by a fraction
- How then can the value of π be represented?
- The answer is that it cannot
- All that we can do is approximate the value of π

Lambda Expressions

Designing Using Existing Abstractions

- An irrational number, like π, cannot be exactly represented by a fraction
- How then can the value of π be represented?
- The answer is that it cannot
- All that we can do is approximate the value of π
- In the 14^{th} century the Indian mathematician Madhava of Sangamagrama discovered an infinite series to approximate π:

$$\pi = \sum_{i=0}^{\infty} (4 * \frac{(-1)^i}{(\texttt{ith-odd } i)})$$

- This formula states that if we add an infinite number of terms the value of π is obtained
- It is, of course, impossible to add an infinite number of terms
- π is approximated by adding the first `n` terms of the series.

Lambda Expressions

Designing Using Existing Abstractions

- An irrational number, like π, cannot be exactly represented by a fraction
- How then can the value of π be represented?
- The answer is that it cannot
- All that we can do is approximate the value of π
- In the 14^{th} century the Indian mathematician Madhava of Sangamagrama discovered an infinite series to approximate π:

$$\pi = \sum_{i=0}^{\infty} (4 * \frac{(-1)^i}{(\texttt{ith-odd } i)})$$

- This formula states that if we add an infinite number of terms the value of π is obtained
- It is, of course, impossible to add an infinite number of terms
- π is approximated by adding the first `n` terms of the series.
- Using the abstract `series` function:

```
;; natnum → number
;; Purpose: Approximate pi using the given number of terms
(define pi-series (series (lambda (i)
                            (* 4 (/ (expt -1 i) (ith-odd i))))))
```

Lambda Expressions

Designing Using Existing Abstractions

- ```
  ;; natnum → number
  ;; Purpose: Approximate pi using the given number of terms
  (define pi-series (series (lambda (i)
                              (* 4 (/ (expt -1 i) (ith-odd i))))))
  ```
- How to test this function?
  ```

Lambda Expressions

Designing Using Existing Abstractions

- ```
  ;; natnum → number
  ;; Purpose: Approximate pi using the given number of terms
  (define pi-series (series (lambda (i)
                              (* 4 (/ (expt -1 i) (ith-odd i))))))
  ```
- How to test this function?
- Tests need to illustrate that as the number of terms used to approximate π increases the approximation gets better
- This requires experimenting with the function to make sure it *converges*
- That is, the approximation error gets smaller as the number of terms grows
  ```

Lambda Expressions

Designing Using Existing Abstractions

- ```
  ;; natnum → number
  ;; Purpose: Approximate pi using the given number of terms
  (define pi-series (series (lambda (i)
                              (* 4 (/ (expt -1 i) (ith-odd i))))))
  ```
- How to test this function?
- Tests need to illustrate that as the number of terms used to approximate π increases the approximation gets better
- This requires experimenting with the function to make sure it *converges*
- That is, the approximation error gets smaller as the number of terms grows
- The following tests achieve this goal:

  ```
  (check-within (pi-series 500)   3.1415 0.01)   ;; 3.139592
  (check-within (pi-series 3000)  3.1415 0.001)  ;; 3.141259
  (check-within (pi-series 12000) 3.1415 0.0001) ;; 3.141509
  ```
- Clearly illustrate that as the number of terms increases the accuracy of the results get better
- The values returned by `pi-series` are written as comments so that any reader can easily see how the approximation of π is converging
  ```

Lambda Expressions

Designing Using Existing Abstractions

- Developing your design and abstraction skills using λ-expressions and first-class functions is important
- They are becoming more prevalent across all programming languages
- Virtually all functional programming languages (e.g., `ISL+`, `Racket`, `Haskell`, `Clean`, and `F#`) have λ-expressions and support first-class functions
- This support is now also found in object-oriented programming languages (e.g., `Java`, `Kotlin`, and `Scala`)
- You may even use λ-expressions in `Excel`

Lambda Expressions

Homework

- Problems: 293–296

Aliens Attack Version 5

- Let's refactor Aliens Attack version 4
- The goal is group together related functions and to streamline the implementation of functions
- The refactored code will have a set of constants needed by multiple functions or for testing, 6 global functions, and tests for the global functions
- The global functions needed are `run` and the functions used in the `big-bang` expression

Aliens Attack Version 5

- The structure of the refactored program:

```
(require 2htdp/image)
(require 2htdp/universe)
;; Data Definitions
      .
      .
      .
;; Constants
      .
      .
      .
(define-struct world (rocket aliens dir shots))
;; Sample world instances
      .
      .
      .
<draw-world code>
      .
      .
      .
<process-key code>
      .
      .
      .
<process-tick code>
      .
      .
      .
<game-over? code>
      .
      .
      .
<draw-last-world code>
      .
      .
      .
<run code>
```

Aliens Attack Version 5

Constants

- Two varieties of constants: those needed globally and those needed locally
- Global constants: needed by more than one global function or by the tests of a global function
- All other constants ought to be encapsulated

Aliens Attack Version 5

Constants

- Two varieties of constants: those needed globally and those needed locally
- Global constants: needed by more than one global function or by the tests of a global function
- All other constants ought to be encapsulated
- Three categories: general constants needed to define elements of the video game, constants to define images, and constants to define an initial world

Aliens Attack Version 5

Constants

- Two varieties of constants: those needed globally and those needed locally
- Global constants: needed by more than one global function or by the tests of a global function
- All other constants ought to be encapsulated
- Three categories: general constants needed to define elements of the video game, constants to define images, and constants to define an initial world
- The general constants needed to define elements of the video game include those needed to define a scene, to define coordinates inside a scene, and NO-SHOT:

```
;; General Constants
(define MAX-CHARS-HORIZONTAL 20)
(define MAX-CHARS-VERTICAL 15)
(define IMAGE-WIDTH 30)
(define IMAGE-HEIGHT 30)
(define MIN-IMG-X 0)
(define MAX-IMG-X (sub1 MAX-CHARS-HORIZONTAL))
(define MIN-IMG-Y 0)
(define MAX-IMG-Y (sub1 MAX-CHARS-VERTICAL))
(define E-SCENE-W (* MAX-CHARS-HORIZONTAL IMAGE-WIDTH))
(define E-SCENE-H (* MAX-CHARS-VERTICAL IMAGE-HEIGHT))
(define NO-SHOT 'no-shot)
```

- Each these constants is either needed by more than one function or test or is needed by another constant definition

Aliens Attack Version 5

Constants

- The constants needed to define images that must remain global are those used only to test the `draw-world` function
- Image constants used while the game is running are encapsulated in `draw-world`

Aliens Attack Version 5

Constants

- The constants needed to define images that must remain global are those used only to test the `draw-world` function
- Image constants used while the game is running are encapsulated in `draw-world`
- The global image constants used in testing are:

```
;; Drawing Testing Constants
(define SHOT-COLOR2     'skyblue)
(define ALIEN-COLOR2    'orange)
(define WINDOW2-COLOR   'white)
(define FUSELAGE2-COLOR 'orange)
(define NACELLE2-COLOR  'brown)
(define ALIEN-IMG2   ...)
(define SHOT-IMG2    ...)
(define FUSELAGE2    ...)
(define WINDOW2 (ellipse 3 10 'solid WINDOW2-COLOR))
(define SINGLE-BOOSTER2 ...)
(define BOOSTER2 (beside SINGLE-BOOSTER2 SINGLE-BOOSTER2))
(define ROCKET-MAIN2 ...)
(define NACELLE2     ...)
(define ROCKET-IMG2  ...)
```

- The values of all these constants remains unchanged
- In the interest of clarity and brevity some of the code above is omitted as indicated by the ellipsis

Aliens Attack Version 5

Constants

- The constants to define initial worlds are those needed to construct the initial world used in the `run` function or in the testing of the handlers:

```
;; Constants for INIT-WORLD
(define AN-IMG-X (/ MAX-CHARS-HORIZONTAL 2))
(define INIT-ROCKET  AN-IMG-X)
(define INIT-DIR  'right)
(define INIT-SHOT NO-SHOT)
(define INIT-LOA ...)
(define INIT-LOS '())

;; Constants for INIT-WORLD2
(define INIT-ROCKET2 15)
(define INIT-ALIEN2 (make-posn 3 MAX-IMG-Y))
(define DIR2 'left)
(define SHOT2 (make-posn AN-IMG-X
                         (/ (sub1 MAX-CHARS-VERTICAL) 2)))
```

- These constants remain mostly unchanged
- To build `SHOT2`, `AN-IMG-X` is used to eliminate having to evaluate `(/ MAX-CHARS-HORIZONTAL 2)` twice

Aliens Attack Version 5

Constants

- We shall now focus on the definition of `INIT-LOA`
- Section 87 discusses the design of `create-alien-army` using interval processing to define `INIT-LOA`
- We discuss a new approach using ISL+'s `build-list` abstract function whose signature is:

```
<X> natnum (natnum → X) → (listof X)
```

Aliens Attack Version 5

Constants

- We shall now focus on the definition of `INIT-LOA`
- Section 87 discusses the design of `create-alien-army` using interval processing to define `INIT-LOA`
- We discuss a new approach using ISL+'s `build-list` abstract function whose signature is:

  `<X> natnum (natnum → X) → (listof X)`

- Let us call the given natural number `k`
- This abstract function builds a list of size `k` with the results of applying the given function to every integer, `n`, in the interval `[0..k-1]` from low to high

Aliens Attack Version 5

Constants

- Four constants may be defined:

```
(define NUM-ALIENS 18)
(define NUM-ALIEN-LINES 3)
(define ALIENS-PER-LINE (/ NUM-ALIENS NUM-ALIEN-LINES))
(define STARTING-X-LINE (add1 (/ (- MAX-CHARS-HORIZONTAL
                                    ALIENS-PER-LINE)
                                 2)))
```

- The alien army ought to start in the middle of the horizontal axis and at the top of the scene
- `STARTING-X-LINE`: Number of empty `image-x` coordinates before and after a row of aliens + 1

Aliens Attack Version 5

Constants

- Four constants may be defined:

```
(define NUM-ALIENS 18)
(define NUM-ALIEN-LINES 3)
(define ALIENS-PER-LINE (/ NUM-ALIENS NUM-ALIEN-LINES))
(define STARTING-X-LINE (add1 (/ (- MAX-CHARS-HORIZONTAL
                                    ALIENS-PER-LINE)
                                 2)))
```

- The alien army ought to start in the middle of the horizontal axis and at the top of the scene
- `STARTING-X-LINE`: Number of empty `image-x` coordinates before and after a row of aliens + 1
- Modular arithmetic used to compute the `image-x` and `image-y` coordinates of each alien:

```
x = (+ STARTING-X-LINE (remainder n ALIENS-PER-LINE))
y = (quotient n ALIENS-PER-LINE)
```

- `n` is the number of the alien to be created

Aliens Attack Version 5

Constants

- Four constants may be defined:

```
(define NUM-ALIENS 18)
(define NUM-ALIEN-LINES 3)
(define ALIENS-PER-LINE (/ NUM-ALIENS NUM-ALIEN-LINES))
(define STARTING-X-LINE (add1 (/ (- MAX-CHARS-HORIZONTAL
                                    ALIENS-PER-LINE)
                                 2)))
```

- The alien army ought to start in the middle of the horizontal axis and at the top of the scene
- `STARTING-X-LINE`: Number of empty `image-x` coordinates before and after a row of aliens + 1
- Modular arithmetic used to compute the `image-x` and `image-y` coordinates of each alien:

```
x = (+ STARTING-X-LINE (remainder n ALIENS-PER-LINE))
y = (quotient n ALIENS-PER-LINE)
```

- `n` is the number of the alien to be created
- Define `INIT-LOA` as follows:

```
(define INIT-LOA
        (build-list NUM-ALIENS
                    (λ (n)
                     (make-posn (+ STARTING-X-LINE (remainder n ALIENS-PER-LINE))
                                (quotient n ALIENS-PER-LINE)))))
```

- Take time to appreciate how much more concise and elegant the code is now

Aliens Attack Version 5

Homework

- Problems: 297–298

Aliens Attack Version 5

Structure Definitions

- No changes made to the `world` structure nor to the sample worlds used in testing

Aliens Attack Version 5

Structure Definitions

- No changes made to the `world` structure nor to the sample worlds used in testing
- Remain as:

```
;; A world is a structure: (make-world rocket loa dir los)
(define-struct world (rocket aliens dir shots))

(define INIT-WORLD  (make-world INIT-ROCKET  INIT-LOA
                                INIT-DIR     INIT-LOS))

(define INIT-WORLD2 (make-world INIT-ROCKET2 (list INIT-ALIEN2)
                                DIR2         (list SHOT2)))

(define WORLD3 (make-world 7         (list (make-posn 3 3))
                           'right    (list (make-posn 3 3))))
```

Aliens Attack Version 5

The `draw-world` Handler

- Encapsulate everything only the drawing handler uses
- image-creating functions do not require refactoring

Aliens Attack Version 5

The `draw-world` Handler

- Encapsulate everything only the drawing handler uses
- image-creating functions do not require refactoring
-
```
(define SHOT-COLOR 'orange) (define ALIEN-COLOR    'black)
(define WINDOW-COLOR   'darkgray) (define FUSELAGE-COLOR 'green)
(define NACELLE-COLOR  'red) (define E-SCENE-COLOR  'pink)
(define E-SCENE (empty-scene E-SCENE-W E-SCENE-H E-SCENE-COLOR))
(define ROCKET-Y (sub1 MAX-CHARS-VERTICAL))
;; color → image  Purpose: Create alien image of given color
(define (mk-alien-img a-color) ...) (define ALIEN-IMG (mk-alien-img ALIEN-COLOR))
;; color → image  Purpose: Create shot image of given color
(define (mk-shot-img a-color) ...) (define SHOT-IMG (mk-shot-img SHOT-COLOR))
;; color → image  Purpose: Create fuselage image of given color
(define (mk-fuselage-img a-color) ...)
(define FUSELAGE (mk-fuselage-img FUSELAGE-COLOR))
(define FUSELAGE-W (image-width FUSELAGE)) (define FUSELAGE-H (image-width FUSELAGE))
;; color → image  Purpose: Create rocket window image
(define (mk-window-img a-color) ...) (define WINDOW  (mk-window-img WINDOW-COLOR))
;; color → image  Purpose: Create single booster image
(define (mk-single-booster-img a-color) ...)
(define SINGLE-BOOSTER (mk-single-booster-img NACELLE-COLOR))
;; image → image  Purpose: Create booster image
(define (mk-booster-img a-sb-img) ...) (define BOOSTER (mk-booster-img SINGLE-BOOSTER))
;; image image image → image  Purpose: Create main rocket image
(define (mk-rocket-main-img a-window a-fuselage a-booster) ...)
(define ROCKET-MAIN (mk-rocket-main-img WINDOW FUSELAGE BOOSTER))
;; image color → image  Purpose: Create rocket nacelle image
(define (mk-nacelle-img a-rocket-main-img a-color) ...)
(define NACELLE (mk-nacelle-img ROCKET-MAIN NACELLE-COLOR))
;; image image → ci     Purpose: Create a rocket ci
(define (mk-rocket-ci a-rocket-main-img a-nacelle-img) ...)
(define ROCKET-IMG (mk-rocket-ci ROCKET-MAIN NACELLE))
```

Aliens Attack Version 5

The `draw-world` Handler

- In Section 125 the functions to draw a `loa` and a `los` were refactored:

```
;; loa scene → scene
;; Purpose: To draw the given loa in the given scene
(define draw-loa (λ (a-lox scn) (draw-lox draw-alien a-lox scn)))

;; los scene → scene
;; Purpose: To draw the given los in the given scene
(define draw-los (λ (a-lox scn) (draw-lox draw-shot a-lox scn)))
```

Aliens Attack Version 5

The `draw-world` Handler

- In Section 125 the functions to draw a `loa` and a `los` were refactored:

```
;; loa scene → scene
;; Purpose: To draw the given loa in the given scene
(define draw-loa (λ (a-lox scn) (draw-lox draw-alien a-lox scn)))

;; los scene → scene
;; Purpose: To draw the given los in the given scene
(define draw-los (λ (a-lox scn) (draw-lox draw-shot a-lox scn)))
```

- `draw-alien` and `draw-shot`) are not recursive and only referenced once
- Suggests using a λ-expression:

```
;; loa scene → scene
;; Purpose: To draw the given loa in the given scene
(define draw-loa
  (draw-lox-maker (λ (an-alien scn)
                    (draw-ci ALIEN-IMG (posn-x an-alien) (posn-y an-alien) scn))))

;; los scene → scene
;; Purpose: To draw the given los in the given scene
(define draw-los
  (draw-lox-maker (λ (a-shot scn)
                    (if (eq? a-shot NO-SHOT)
                        scn
                        (draw-ci SHOT-IMG (posn-x a-shot) (posn-y a-shot) scn)))))
```

Aliens Attack Version 5

The `draw-world` Handler

- In Section 125 the functions to draw a `loa` and a `los` were refactored:

```
;; loa scene → scene
;; Purpose: To draw the given loa in the given scene
(define draw-loa (λ (a-lox scn) (draw-lox draw-alien a-lox scn)))

;; los scene → scene
;; Purpose: To draw the given los in the given scene
(define draw-los (λ (a-lox scn) (draw-lox draw-shot a-lox scn)))
```

- `draw-alien` and `draw-shot`) are not recursive and only referenced once
- Suggests using a λ-expression:

```
;; loa scene → scene
;; Purpose: To draw the given loa in the given scene
(define draw-loa
  (draw-lox-maker (λ (an-alien scn)
                    (draw-ci ALIEN-IMG (posn-x an-alien) (posn-y an-alien) scn))))

;; los scene → scene
;; Purpose: To draw the given los in the given scene
(define draw-los
  (draw-lox-maker (λ (a-shot scn)
                    (if (eq? a-shot NO-SHOT)
                        scn
                        (draw-ci SHOT-IMG (posn-x a-shot) (posn-y a-shot) scn)))))
```

- Implemented in this manner there is no longer a need for `draw-alien` and for `draw-shot` to be defined
- They may be deleted from the game's program
- Encapsulating the above functions means that `draw-ci`, `draw-lox-maker`, and `draw-lox` may also be encapsulated inside `draw-world` given that they are not referenced elsewhere.

Aliens Attack Version 5

The `draw-world` Handler

- To complete the refactoring of `draw-world` add `draw-rocket` and `draw-world` as local functions
- The body of the local ought to call the locally defined `draw-world`
- The tests for (the new global) `draw-world` remain unchanged

Aliens Attack Version 5

The `process-key` Handler

- Only one that uses `process-shooting`, `move-rckt-left`, and `move-rckt-right` as auxiliary functions: encapsulate

Aliens Attack Version 5

The process-key Handler

- Only one that uses `process-shooting`, `move-rckt-left`, and `move-rckt-right` as auxiliary functions: encapsulate
- `move-rckt-left` and `move-rckt-right` are very similar:

```
;; rocket → rocket Purpose: Move the given rocket right
(define (move-rckt-right a-rocket)
  (if (< a-rocket (sub1 MAX-CHARS-HORIZONTAL)) (add1 a-rocket) a-rocket))

;; rocket → rocket Purpose: Move the given rocket left
(define (move-rckt-left a-rocket)
  (if (> a-rocket 0) (sub1 a-rocket) a-rocket))
```

- Suggests using a curried function to eliminate the repetitions

Aliens Attack Version 5

The process-key Handler

- Only one that uses `process-shooting`, `move-rckt-left`, and `move-rckt-right` as auxiliary functions: encapsulate
- `move-rckt-left` and `move-rckt-right` are very similar:

```
;; rocket → rocket Purpose: Move the given rocket right
(define (move-rckt-right a-rocket)
  (if (< a-rocket (sub1 MAX-CHARS-HORIZONTAL)) (add1 a-rocket) a-rocket))

;; rocket → rocket Purpose: Move the given rocket left
(define (move-rckt-left a-rocket)
  (if (> a-rocket 0) (sub1 a-rocket) a-rocket))
```

- Suggests using a curried function to eliminate the repetitions
- Three differences: the comparing predicate, the second value given to this predicate, and the function used to create a new rocket

Aliens Attack Version 5

The process-key Handler

- Only one that uses `process-shooting`, `move-rckt-left`, and `move-rckt-right` as auxiliary functions: encapsulate
- `move-rckt-left` and `move-rckt-right` are very similar:

```
;; rocket → rocket Purpose: Move the given rocket right
(define (move-rckt-right a-rocket)
  (if (< a-rocket (sub1 MAX-CHARS-HORIZONTAL)) (add1 a-rocket) a-rocket))

;; rocket → rocket Purpose: Move the given rocket left
(define (move-rckt-left a-rocket)
  (if (> a-rocket 0) (sub1 a-rocket) a-rocket))
```

- Suggests using a curried function to eliminate the repetitions
- Three differences: the comparing predicate, the second value given to this predicate, and the function used to create a new rocket
- The abstraction:

```
(define (make-rocket-mover cmp val f)
  (λ (rckt) (if (cmp rckt val) (f rckt) rckt)))
```

Aliens Attack Version 5

The process-key Handler

- Only one that uses `process-shooting`, `move-rckt-left`, and `move-rckt-right` as auxiliary functions: encapsulate
- `move-rckt-left` and `move-rckt-right` are very similar:

```
;; rocket → rocket Purpose: Move the given rocket right
(define (move-rckt-right a-rocket)
  (if (< a-rocket (sub1 MAX-CHARS-HORIZONTAL)) (add1 a-rocket) a-rocket))

;; rocket → rocket Purpose: Move the given rocket left
(define (move-rckt-left a-rocket)
  (if (> a-rocket 0) (sub1 a-rocket) a-rocket))
```

- Suggests using a curried function to eliminate the repetitions
- Three differences: the comparing predicate, the second value given to this predicate, and the function used to create a new rocket
- The abstraction:

```
(define (make-rocket-mover cmp val f)
  (λ (rckt) (if (cmp rckt val) (f rckt) rckt)))
```

- Refactored function:

```
;; rocket → rocket  Purpose: Move the given rocket left
(define move-rckt-left (make-rocket-mover > 0 sub1))

;; rocket → rocket  Purpose: Move the given rocket right
(define move-rckt-right (mk-rckt-mvr < (sub1 MAX-CHARS-HORIZONTAL) add1))
```

Aliens Attack Version 5

The process-key Handler

- Only one that uses process-shooting, move-rckt-left, and move-rckt-right as auxiliary functions: encapsulate
- move-rckt-left and move-rckt-right are very similar:

```
;; rocket → rocket Purpose: Move the given rocket right
(define (move-rckt-right a-rocket)
  (if (< a-rocket (sub1 MAX-CHARS-HORIZONTAL)) (add1 a-rocket) a-rocket))

;; rocket → rocket Purpose: Move the given rocket left
(define (move-rckt-left a-rocket)
  (if (> a-rocket 0) (sub1 a-rocket) a-rocket))
```

- Suggests using a curried function to eliminate the repetitions
- Three differences: the comparing predicate, the second value given to this predicate, and the function used to create a new rocket
- The abstraction:

```
(define (make-rocket-mover cmp val f)
  (λ (rckt) (if (cmp rckt val) (f rckt) rckt)))
```

- Refactored function:

```
;; rocket → rocket  Purpose: Move the given rocket left
(define move-rckt-left (make-rocket-mover > 0 sub1))

;; rocket → rocket  Purpose: Move the given rocket right
(define move-rckt-right (mk-rckt-mvr < (sub1 MAX-CHARS-HORIZONTAL) add1))
```

- Encapsulate the three functions using a local-expression
- The body of the local-expression calls the local process-key
- The tests for this handler remain unchanged

Aliens Attack Version 5

The `process-tick` Handler

- Has the most auxiliary functions

Aliens Attack Version 5

The `process-tick` Handler

- Has the most auxiliary functions
- We start with the refactored function to move a list of aliens:

```
;; dir → (alien → alien)
;; Purpose: Return function to move alien in given direction
(define (alien-mover-maker a-dir)
  (λ (an-alien) (move-alien an-alien a-dir)))

;; (listof alien) dir → (listof alien)
;; Purpose: Move loa in given direction
(define (move-loa a-loa dir)
  (map (alien-mover-maker dir) a-loa))
```

- `alien-mover-maker` refers to a function only referenced once
- Suggests encapsulating inside `alien-mover-maker`

Aliens Attack Version 5

The process-tick Handler

- Has the most auxiliary functions
- We start with the refactored function to move a list of aliens:

```
;; dir → (alien → alien)
;; Purpose: Return function to move alien in given direction
(define (alien-mover-maker a-dir)
  (λ (an-alien) (move-alien an-alien a-dir)))

;; (listof alien) dir → (listof alien)
;; Purpose: Move loa in given direction
(define (move-loa a-loa dir)
  (map (alien-mover-maker dir) a-loa))
```

- `alien-mover-maker` refers to a function only referenced once
- Suggests encapsulating inside `alien-mover-maker`
- `move-alien` is the only function that refers to `move-down-image-y`, `move-left-image-x`, and `move-right-image-x`
- These functions ought to be encapsulated inside `move-alien`

Aliens Attack Version 5

The process-tick Handler

- ```
  ;; dir → (alien → alien)
  ;; Purpose: Return a function to move alien in given direction
  (define (move-alien-maker a-dir)
   (local
    [;; alien → alien
     ;; Purpose: Move the given alien in a-dir
     (define (move-alien an-alien)
      (local [;; image-y<max → image-y Purpose: Move given image-y<max down
              (define (move-down-image-y an-img-y<max)
                (add1 an-img-y<max))
              ;; image-x>min → image-x Purpose: Move given image-x>min left
              (define (move-left-image-x an-img-x>min)
                (sub1 an-img-x>min))
              ;; image-x<max → image-x Purpose: Move given image-x<max right
              (define (move-right-image-x an-img-x<max)
                (add1 an-img-x<max))]
        (cond [(eq? a-dir 'right)
               (make-posn (move-right-image-x (posn-x an-alien))
                          (posn-y an-alien))]
              [(eq? a-dir 'left)
               (make-posn (move-left-image-x (posn-x an-alien))
                          (posn-y an-alien))]
              [else (make-posn
                     (posn-x an-alien)
                     (move-down-image-y (posn-y an-alien)))])))]
    move-alien))
  ```
  ```

Aliens Attack Version 5

The process-tick Handler

```
;; dir → (alien → alien)
;; Purpose: Return a function to move alien in given direction
(define (move-alien-maker a-dir)
 (local
  [;; alien → alien
   ;; Purpose: Move the given alien in a-dir
   (define (move-alien an-alien)
    (local [;; image-y<max → image-y Purpose: Move given image-y<max down
            (define (move-down-image-y an-img-y<max)
              (add1 an-img-y<max))
            ;; image-x>min → image-x Purpose: Move given image-x>min left
            (define (move-left-image-x an-img-x>min)
              (sub1 an-img-x>min))
            ;; image-x<max → image-x Purpose: Move given image-x<max right
            (define (move-right-image-x an-img-x<max)
              (add1 an-img-x<max))]
      (cond [(eq? a-dir 'right)
             (make-posn (move-right-image-x (posn-x an-alien))
                        (posn-y an-alien))]
            [(eq? a-dir 'left)
             (make-posn (move-left-image-x (posn-x an-alien))
                        (posn-y an-alien))]
            [else (make-posn
                   (posn-x an-alien)
                   (move-down-image-y (posn-y an-alien)))])))]
  move-alien))
```

- `move-alien` is locally defined
- Only referenced once and it is not recursive
- Suggests using a λ-expression

Aliens Attack Version 5

The process-tick Handler

- ```
  ;; dir → (alien → alien)
  ;; Purpose: Return a function to move an alien in the
  ;;          given direction
  (define (move-alien-maker a-dir)
   (lambda (an-alien)
    (local [;; image-y<max → image-y
            ;; Purpose: To move the given image-y<max down
            (define (move-down-image-y an-img-y<max) (add1 an-img-y<max))

            ;; image-x>min → image-x
            ;; Purpose: Move the given image-x>min left
            (define (move-left-image-x an-img-x>min) (sub1 an-img-x>min))

            ;; image-x<max → image-x
            ;; Purpose: Move the given image-x<max right
            (define (move-right-image-x an-img-x<max) (add1 an-img-x<max))]
      (cond [(eq? a-dir 'right)
             (make-posn (move-right-image-x (posn-x an-alien))
                        (posn-y an-alien))]
            [(eq? a-dir 'left)
             (make-posn (move-left-image-x (posn-x an-alien))
                        (posn-y an-alien))]
            [else (make-posn
                    (posn-x an-alien)
                    (move-down-image-y (posn-y an-alien)))]))))
  ```
  ```

Aliens Attack Version 5

The process-tick Handler

- ```
  ;; dir → (alien → alien)
  ;; Purpose: Return a function to move an alien in the
  ;;          given direction
  (define (move-alien-maker a-dir)
   (lambda (an-alien)
    (local [;; image-y<max → image-y
            ;; Purpose: To move the given image-y<max down
            (define (move-down-image-y an-img-y<max) (add1 an-img-y<max))

            ;; image-x>min → image-x
            ;; Purpose: Move the given image-x>min left
            (define (move-left-image-x an-img-x>min) (sub1 an-img-x>min))

            ;; image-x<max → image-x
            ;; Purpose: Move the given image-x<max right
            (define (move-right-image-x an-img-x<max) (add1 an-img-x<max))]
      (cond [(eq? a-dir 'right)
             (make-posn (move-right-image-x (posn-x an-alien))
                        (posn-y an-alien))]
            [(eq? a-dir 'left)
             (make-posn (move-left-image-x (posn-x an-alien))
                        (posn-y an-alien))]
            [else (make-posn
                    (posn-x an-alien)
                    (move-down-image-y (posn-y an-alien)))]))))
  ```
- Easier to see that `alien-mover-maker` is a curried function
- This is the function to encapsulate inside `process-tick`
  ```

Aliens Attack Version 5

The process-tick Handler

- Move a los:

```
;; los → los Purpose: To move the given list of shots
(define (move-los a-los)
 (local [;; shot → shot
         ;; Purpose: To move the given shot
         (define (local-move-shot a-shot)
          (cond [(eq? a-shot NO-SHOT) a-shot]
                [else
                 (cond [(= (posn-y a-shot) MIN-IMG-Y) NO-SHOT]
                       [else
                        (make-posn (posn-x a-shot)
                                   (move-up-image-y (posn-y a-shot)))]))]
   (map local-move-shot a-los))
```

- Local function defined that is not recursive and only referenced once This
- Suggests using a λ-expression

Aliens Attack Version 5

The process-tick Handler

- Move a los:

```
;; los → los Purpose: To move the given list of shots
(define (move-los a-los)
 (local [;; shot → shot
         ;; Purpose: To move the given shot
         (define (local-move-shot a-shot)
          (cond [(eq? a-shot NO-SHOT) a-shot]
                [else
                 (cond [(= (posn-y a-shot) MIN-IMG-Y) NO-SHOT]
                       [else
                        (make-posn (posn-x a-shot)
                                   (move-up-image-y (posn-y a-shot)))]))
   (map local-move-shot a-los))
```

- Local function defined that is not recursive and only referenced once This
- Suggests using a λ-expression
- Refactored move-los:

```
;; los → los
;; Purpose: To move the given list of shots
(define (move-los a-los)
  (map (lambda (a-shot)
         (cond [(eq? a-shot NO-SHOT) a-shot]
               [(= (posn-y a-shot) MIN-IMG-Y) NO-SHOT]
               [else (make-posn (posn-x a-shot) (sub1 (posn-y a-shot)))]))
       a-los))
```

- move-up-image is inlined instead of making it local to the λ-expression

Aliens Attack Version 5

The `process-tick` Handler

- Another auxiliary function used by `process-tick` is:

```
;; loa dir → dir
;; Purpose: Return new aliens direction
(define (new-dir-after-tick a-loa old-dir)
  (cond [(eq? old-dir 'down)
         (new-dir-after-down a-loa)]
        [(eq? old-dir 'left)
         (new-dir-after-left a-loa)]
        [else (new-dir-after-right a-loa)]))
```

- References three auxiliary functions that are simple and may be inlined without sacrificing readability

Aliens Attack Version 5

The process-tick Handler

- ```
  ;; loa dir → dir
  ;; Purpose: Return new aliens direction
  (define (new-dir-after-tick a-loa old-dir)
    (cond [(eq? old-dir 'down)
           (if (any-alien-at-left-edge? a-loa) 'right 'left)]
          [(eq? old-dir 'left)
           (if (any-alien-at-left-edge? a-loa) 'down 'left)]
          [else
           (if (any-alien-at-right-edge? a-loa) 'down 'right)]))
  ```
  ```

Aliens Attack Version 5

The process-tick Handler

- ```
  ;; loa dir → dir
  ;; Purpose: Return new aliens direction
  (define (new-dir-after-tick a-loa old-dir)
    (cond [(eq? old-dir 'down)
           (if (any-alien-at-left-edge? a-loa) 'right 'left)]
          [(eq? old-dir 'left)
           (if (any-alien-at-left-edge? a-loa) 'down 'left)]
          [else
           (if (any-alien-at-right-edge? a-loa) 'down 'right)]))
  ```
- Now this function is the only function that references `any-alien-at-left-edge?` and `any-alien-at-right-edge?`
- These functions ought to be encapsulated
- Before doing so, we refactor them to use `ormap`
  ```

Aliens Attack Version 5

The process-tick Handler

- ```
  ;; loa dir → dir
  ;; Purpose: Return new aliens direction
  (define (new-dir-after-tick a-loa old-dir)
    (local [;; loa → Boolean
            ;; Purpose: Determine if any alien is at scene's right edge
            (define (any-alien-at-right-edge? a-loa)
              (ormap (lambda (an-alien)
                       (= (posn-x an-alien) MAX-IMG-X))
                     a-loa))
            ;; loa → Boolean
            ;; Purpose: Determine if any alien is at scene's left edge
            (define (any-alien-at-left-edge? a-loa)
              (ormap (lambda (an-alien) (= (posn-x an-alien) MIN-IMG-X))
                     a-loa))]
      (cond [(eq? old-dir 'down)
             (if (any-alien-at-left-edge? a-loa) 'right 'left)]
            [(eq? old-dir 'left)
             (if (any-alien-at-left-edge? a-loa) 'down 'left)]
            [else
             (if (any-alien-at-right-edge? a-loa) 'down 'right)])))
  ```
  ```

Aliens Attack Version 5

The process-tick Handler

- ```
  ;; loa dir → dir
  ;; Purpose: Return new aliens direction
  (define (new-dir-after-tick a-loa old-dir)
    (local [;; loa → Boolean
            ;; Purpose: Determine if any alien is at scene's right edge
            (define (any-alien-at-right-edge? a-loa)
              (ormap (lambda (an-alien)
                       (= (posn-x an-alien) MAX-IMG-X))
                     a-loa))
            ;; loa → Boolean
            ;; Purpose: Determine if any alien is at scene's left edge
            (define (any-alien-at-left-edge? a-loa)
              (ormap (lambda (an-alien) (= (posn-x an-alien) MIN-IMG-X))
                     a-loa))]
      (cond [(eq? old-dir 'down)
             (if (any-alien-at-left-edge? a-loa) 'right 'left)]
            [(eq? old-dir 'left)
             (if (any-alien-at-left-edge? a-loa) 'down 'left)]
            [else
             (if (any-alien-at-right-edge? a-loa) 'down 'right)])))
  ```
- Streamline the function by inlining the two local functions in the body of the cond-expression?
  ```

Aliens Attack Version 5

The process-tick Handler

```
;; loa dir → dir
;; Purpose: Return new aliens direction
(define (new-dir-after-tick a-loa old-dir)
  (local [;; loa → Boolean
          ;; Purpose: Determine if any alien is at scene's right edge
          (define (any-alien-at-right-edge? a-loa)
            (ormap (lambda (an-alien)
                     (= (posn-x an-alien) MAX-IMG-X))
                   a-loa))
          ;; loa → Boolean
          ;; Purpose: Determine if any alien is at scene's left edge
          (define (any-alien-at-left-edge? a-loa)
            (ormap (lambda (an-alien) (= (posn-x an-alien) MIN-IMG-X))
                   a-loa))]
    (cond [(eq? old-dir 'down)
           (if (any-alien-at-left-edge? a-loa) 'right 'left)]
          [(eq? old-dir 'left)
           (if (any-alien-at-left-edge? a-loa) 'down 'left)]
          [else
           (if (any-alien-at-right-edge? a-loa) 'down 'right)])))
```

- Streamline the function by inlining the two local functions in the body of the cond-expression?
- No technical difficulty in doing so
- Likely to reduce the readability of the code
- In the interest of making sure our code clearly communicates how the problem is solved we refrain from further inlining
- You may, of course, disagree and perform the inlining of the local functions

Aliens Attack Version 5

The process-tick Handler

- remove-hit-aliens and remove-shots are only used by process-tick:

```
;; loa los → loa Purpose: Remove aliens hit by any shot
(define (remove-hit-aliens a-loa a-los)
  (cond [(empty? a-loa) '()]
        [(hit-by-any-shot? (first a-loa) a-los)
         (remove-hit-aliens (rest a-loa) a-los)]
        [else (cons (first a-loa)
                    (remove-hit-aliens (rest a-loa) a-los))]))
;; los loa → los Purpose: Remove NO-SHOTs and hit shots
(define (remove-shots a-los a-loa)
  (cond [(empty? a-los) a-los]
        [(or (eq? (first a-los) NO-SHOT)
             (hit-any-alien? (first a-los) a-loa))
         (remove-shots (rest a-los) a-loa)]
        [else (cons (first a-los)
                    (remove-shots (rest a-los) a-loa))]))
```

- Immediately encapsulation comes to mind but can they be simplified?

Aliens Attack Version 5

The process-tick Handler

- remove-hit-aliens and remove-shots are only used by process-tick:

```
;; loa los → loa Purpose: Remove aliens hit by any shot
(define (remove-hit-aliens a-loa a-los)
  (cond [(empty? a-loa) '()]
        [(hit-by-any-shot? (first a-loa) a-los)
         (remove-hit-aliens (rest a-loa) a-los)]
        [else (cons (first a-loa)
                    (remove-hit-aliens (rest a-loa) a-los))]))
;; los loa → los Purpose: Remove NO-SHOTs and hit shots
(define (remove-shots a-los a-loa)
  (cond [(empty? a-los) a-los]
        [(or (eq? (first a-los) NO-SHOT)
             (hit-any-alien? (first a-los) a-loa))
         (remove-shots (rest a-los) a-loa)]
        [else (cons (first a-los)
                    (remove-shots (rest a-los) a-loa))]))
```

- Immediately encapsulation comes to mind but can they be simplified?
- These functions are good candidates to be simplified using filter

Aliens Attack Version 5

The process-tick Handler

- remove-hit-aliens and remove-shots are only used by process-tick:

```
;; loa los → loa Purpose: Remove aliens hit by any shot
(define (remove-hit-aliens a-loa a-los)
  (cond [(empty? a-loa) '()]
        [(hit-by-any-shot? (first a-loa) a-los)
         (remove-hit-aliens (rest a-loa) a-los)]
        [else (cons (first a-loa)
                    (remove-hit-aliens (rest a-loa) a-los))]))
;; los loa → los Purpose: Remove NO-SHOTs and hit shots
(define (remove-shots a-los a-loa)
  (cond [(empty? a-los) a-los]
        [(or (eq? (first a-los) NO-SHOT)
             (hit-any-alien? (first a-los) a-loa))
         (remove-shots (rest a-los) a-loa)]
        [else (cons (first a-los)
                    (remove-shots (rest a-los) a-loa))]))
```

- Immediately encapsulation comes to mind but can they be simplified?
- These functions are good candidates to be simplified using `filter`
- For `remove-hit-aliens` `filter`: predicate that negates the result of testing if an alien is hit by any shot
- For `remove-shots` `filter`: predicate that negates the result of testing if a shot is `NO-SHOT` or has been hit by any alien
- Both may be implemented using a λ-expression and `ormap`

Aliens Attack Version 5

The process-tick Handler

- ```
  ;; loa los → loa Purpose: Remove aliens hit by any shot
  (define (remove-hit-aliens a-loa a-los)
   (filter (lambda (an-alien)
             (not (ormap (lambda (a-shot) (hit? a-shot an-alien))
                         a-los)))
           a-loa))
  ;; los loa → los Purpose: Remove NO-SHOTs and hit shots
  (define (remove-shots a-los a-loa)
   (filter (lambda (a-shot)
             (not (or (eq? a-shot NO-SHOT)
                      (ormap (lambda (an-alien)
                               (hit? a-shot an-alien))
                             a-loa))))
           a-los))
  ```
- These functions may now be encapsulated in `process-tick`
  ```

Aliens Attack Version 5

The process-tick Handler

- ```
  ;; loa los → loa Purpose: Remove aliens hit by any shot
  (define (remove-hit-aliens a-loa a-los)
   (filter (lambda (an-alien)
             (not (ormap (lambda (a-shot) (hit? a-shot an-alien))
                         a-los)))
           a-loa))
  ;; los loa → los Purpose: Remove NO-SHOTs and hit shots
  (define (remove-shots a-los a-loa)
   (filter (lambda (a-shot)
             (not (or (eq? a-shot NO-SHOT)
                      (ormap (lambda (an-alien)
                               (hit? a-shot an-alien))
                             a-loa))))
           a-los))
  ```
- These functions may now be encapsulated in `process-tick`
- Observe that `hit?` is referenced only by these two functions
- `hit?` may not be encapsulated in either
- `hit?` may be encapsulated in `process-tick` because it will be in scope for the functions that need them
  ```

Aliens Attack Version 5

The process-tick Handler

- ```
  ;; loa los → loa Purpose: Remove aliens hit by any shot
  (define (remove-hit-aliens a-loa a-los)
   (filter (lambda (an-alien)
             (not (ormap (lambda (a-shot) (hit? a-shot an-alien))
                         a-los)))
           a-loa))
  ;; los loa → los Purpose: Remove NO-SHOTs and hit shots
  (define (remove-shots a-los a-loa)
   (filter (lambda (a-shot)
             (not (or (eq? a-shot NO-SHOT)
                      (ormap (lambda (an-alien)
                               (hit? a-shot an-alien))
                             a-loa))))
           a-los))
  ```
- These functions may now be encapsulated in `process-tick`
- Observe that `hit?` is referenced only by these two functions
- `hit?` may not be encapsulated in either
- `hit?` may be encapsulated in `process-tick` because it will be in scope for the functions that need them
- This is the last function that needs to be encapsulated in `process-tick`
- The body of the `local`-expression in `process-tick` calls the local `process-tick`
  ```

Aliens Attack Version 5

The `process-tick` Handler

- Observe that running the existing tests after encapsulation fails to tests all the code
- Not all the code in `hit?` and in `move-los` is covered by the existing tests
- None of the tests for `process-tick` have an alien hit nor test moving a `NO-SHOT`

Aliens Attack Version 5

The `process-tick` Handler

- Observe that running the existing tests after encapsulation fails to tests all the code
- Not all the code in `hit?` and in `move-los` is covered by the existing tests
- None of the tests for `process-tick` have an alien hit nor test moving a `NO-SHOT`
- To remedy this we add the following test:

```
(check-expect (process-tick
                (make-world INIT-ROCKET
                            (list (make-posn 2 5))
                            'left
                            (list (make-posn 1 6) NO-SHOT)))
              (make-world INIT-ROCKET
                          '()
                          'left
                          '()))
```

- The alien is hit by the first shot
- The second shot is `NO-SHOT`

Aliens Attack Version 5

The `process-tick` Handler

- Observe that running the existing tests after encapsulation fails to tests all the code
- Not all the code in `hit?` and in `move-los` is covered by the existing tests
- None of the tests for `process-tick` have an alien hit nor test moving a `NO-SHOT`
- To remedy this we add the following test:

```
(check-expect (process-tick
                (make-world INIT-ROCKET
                            (list (make-posn 2 5))
                            'left
                            (list (make-posn 1 6) NO-SHOT)))
              (make-world INIT-ROCKET
                          '()
                          'left
                          '()))
```

- The alien is hit by the first shot
- The second shot is `NO-SHOT`
- **Lesson**: Encapsulation may remove tests that cover code not covered by the tests for the function encapsulated into and, thus, requires new tests to be developed

Aliens Attack Version 5

The `game-over?` Handler

- References to `any-alien-reached-earth?` and `any-aliens-alive?` are only in this handler and in `draw-last-world`
- Suggests that these two functions should not be encapsulated.
- Let us not decide a priori that these functions should not be encapsulated and engage in speculative encapsulation

Aliens Attack Version 5

The game-over? Handler

- References to any-alien-reached-earth? and any-aliens-alive? are only in this handler and in draw-last-world
- Suggests that these two functions should not be encapsulated.
- Let us not decide a priori that these functions should not be encapsulated and engage in speculative encapsulation
- Refactored code:

```
;; world → Boolean
;; Purpose: Detect if the game is over
(define (game-over? a-world)
  (local [;; (listof alien) → Boolean
          ;; Purpose: Determine if any alien has reached earth
          (define (any-alien-reached-earth? a-loa)
            (local [(define (alien-reached-earth? an-alien)
                      (= (posn-y an-alien) MAX-IMG-Y))]
              (ormap alien-reached-earth? a-loa)))
          ;; loa Boolean → Boolean
          ;; Purpose: Determine if there is an alien in the
          ;;          given loa
          (define (any-aliens-alive? a-loa) (cons? a-loa))]
    (or (any-alien-reached-earth? (world-aliens a-world))
        (not (any-aliens-alive? (world-aliens a-world))))))
```

- Negating the value returned by any-aliens-alive? is the same as testing if the given list of aliens is empty
- any-alien-reached-earth? is easily inlined into the local-expression's body
- alien-reached-earth? may be represented using a λ-expression

Aliens Attack Version 5

The game-over? Handler

- Refactored handler:

```
;; world → Boolean
;; Purpose: Detect if the game is over
(define (game-over? a-world)
  (or (ormap (λ (an-alien) (= (posn-y an-alien) MAX-IMG-Y))
             (world-aliens a-world))
      (empty? (world-aliens a-world))))
```

Aliens Attack Version 5

The game-over? Handler

- Refactored handler:

```
;; world → Boolean
;; Purpose: Detect if the game is over
(define (game-over? a-world)
  (or (ormap (λ (an-alien) (= (posn-y an-alien) MAX-IMG-Y))
             (world-aliens a-world))
      (empty? (world-aliens a-world))))
```

- The tests for this handler are unchanged
- Lessons:
  - Logic is a useful tool to simplify functions
  - Speculative encapsulation may make us aware of possible simplifications

Aliens Attack Version 5

The `draw-last-world` Handler

- This handler determines why `game-over?` returns `#true`.
- References the same auxiliary functions as `game-over?`
- Employ the same logic and inlining

Aliens Attack Version 5

The `draw-last-world` Handler

- This handler determines why `game-over?` returns `#true`.
- References the same auxiliary functions as `game-over?`
- Employ the same logic and inlining
- The refactored function is:

```
;; world → scene throws error
;; Purpose: To draw the game's final scene
(define (draw-last-world a-world)
  (cond [(ormap (lambda (an-alien)
                  (= (posn-y an-alien) MAX-IMG-Y))
                (world-aliens a-world))
         (place-image (text "EARTH WAS CONQUERED!" 36 'red)
                      (/ E-SCENE-W 2)
                      (/ E-SCENE-H 4)
                      (draw-world a-world))]
        [(empty? (world-aliens a-world))
         (place-image (text "EARTH WAS SAVED!" 36 'green)
                      (/ E-SCENE-W 2)
                      (/ E-SCENE-H 4)
                      (draw-world a-world))]
        [else
         (error (format "draw-last-world: Given world has ~s
                         aliens and none have reached earth."
                        (length (world-aliens a-world))))]))
```

Aliens Attack Version 5

Refactoring `run`

- Only function that references `TICK-RATE`:

```
;; string → world
;; Purpose: To run the game
(define (run a-name)
  (local [(define TICK-RATE 1/4)]
    (big-bang INIT-WORLD
              [on-draw draw-world]
              [name a-name]
              [on-key process-key]
              [on-tick process-tick TICK-RATE]
              [stop-when game-over? draw-last-world])))
```

Aliens Attack Version 5

Refactoring `run`

- Only function that references `TICK-RATE`:

```
;; string → world
;; Purpose: To run the game
(define (run a-name)
  (local [(define TICK-RATE 1/4)]
    (big-bang INIT-WORLD
              [on-draw draw-world]
              [name a-name]
              [on-key process-key]
              [on-tick process-tick TICK-RATE]
              [stop-when game-over? draw-last-world])))
```

- The only task remaining is to decide where to place `accum` and `id`
- These functions are only referenced by `draw-lox`, which is encapsulated in `draw-world`
- These functions may also be encapsulated inside `draw-world`

Aliens Attack Version 5

Refactoring `run`

- Only function that references `TICK-RATE`:

```
;; string → world
;; Purpose: To run the game
(define (run a-name)
  (local [(define TICK-RATE 1/4)]
    (big-bang INIT-WORLD
              [on-draw draw-world]
              [name a-name]
              [on-key process-key]
              [on-tick process-tick TICK-RATE]
              [stop-when game-over? draw-last-world])))
```

- The only task remaining is to decide where to place `accum` and `id`
- These functions are only referenced by `draw-lox`, which is encapsulated in `draw-world`
- These functions may also be encapsulated inside `draw-world`
- This ends the development of Aliens Attack version 5
- Take time to appreciate how much better organized and clearer the game's code is
- Do you feel that the code for Aliens Attack version 5 better communicates how the game works than the code for Aliens Attack version 4?
- Have you developed a bigger appreciation for encapsulation, anonymous functions, and abstract functions?

Aliens Attack Version 5

Homework

- Problems: 300–301

For-Loops and Pattern Matching

- Abstract functions traverse data of arbitrary size to compute a value
- They *iterate* through the elements contained in an instance of data of arbitrary size
- Every time a function that traverses data of arbitrary size is written a conditional expression is needed
- This repetition suggests an abstraction is needed

For-Loops and Pattern Matching

- Abstract functions traverse data of arbitrary size to compute a value
- They *iterate* through the elements contained in an instance of data of arbitrary size
- Every time a function that traverses data of arbitrary size is written a conditional expression is needed
- This repetition suggests an abstraction is needed
- In `ISL+`, this abstraction is provided by *`for` loops*
- A `for` loop generates the sequences of values to be iterated over
- Combines the values obtained from evaluating an expression (known as its body) to return a value

For-Loops and Pattern Matching

- Abstract functions traverse data of arbitrary size to compute a value
- They *iterate* through the elements contained in an instance of data of arbitrary size
- Every time a function that traverses data of arbitrary size is written a conditional expression is needed
- This repetition suggests an abstraction is needed
- In `ISL+`, this abstraction is provided by *`for` loops*
- A `for` loop generates the sequences of values to be iterated over
- Combines the values obtained from evaluating an expression (known as its body) to return a value
- Allow programmers to dispense with the coding of conditional expressions
- In addition, they allow programmers iterate over multiple pieces data.

For-Loops and Pattern Matching

- The conditionals written to process data of arbitrary size distinguish between the subtypes and use selector functions

For-Loops and Pattern Matching

- The conditionals written to process data of arbitrary size distinguish between the subtypes and use selector functions
- In `ISL+`, a `match`-expression provides programmers with an abstraction to eliminate this repetitive practice
- A `match`-expression dispatches on the type of the data and may introduce local variables to capture the values in a compound piece of data
- The use of such expressions can also significantly improve code readability
- Determining a data type to introduce local variables or control program evaluation is called *pattern matching*

For-Loops and Pattern Matching

- The conditionals written to process data of arbitrary size distinguish between the subtypes and use selector functions
- In `ISL+`, a `match`-expression provides programmers with an abstraction to eliminate this repetitive practice
- A `match`-expression dispatches on the type of the data and may introduce local variables to capture the values in a compound piece of data
- The use of such expressions can also significantly improve code readability
- Determining a data type to introduce local variables or control program evaluation is called *pattern matching*
- Neither of these increase our computational power, just make programming easier
- In order to use them you must require the `2htdp/abstraction` teachpack:

```
(require 2htdp/abstraction)
```

For-Loops and Pattern Matching

For Loops

- For loops come in two general varieties in `ISL+`:
  - those that extend the variables in scope for the body of the `for`-loop
  - those that extend the variables in scope for the body and each subsequent variable introduced by the `for`-loop
- The first we shall call `for`-loops and the second we shall call `for*`-loops
- Both varieties require *comprehension clauses* that declare local variables and create the values iterated over

For-Loops and Pattern Matching

For Loops

- ISL+ offers 6 different `for`-loops with the following syntax:

```
      expr   ::=   (for/list          (clause+)   expr)
             ::=   (for/and           (clause+)   expr)
             ::=   (for/or            (clause+)   expr)
             ::=   (for/sum           (clause+)   expr)
             ::=   (for/product       (clause+)   expr)
             ::=   (for/string        (clause+)   expr)
   clause   ::=   [variable expr]
```

- Keyword for loop type, comprehension clauses, and a body
- A comprehension clause declares a variable for the sequence of values to iterate over
- For each iteration step the variable becomes next value sequence
- The scope of a comprehension variable is the body of the `for`-loop
- For each value a variable takes on the body of is evaluated

For-Loops and Pattern Matching

For Loops

- ISL+ offers 6 different `for`-loops with the following syntax:

```
expr   ::=  (for/list     (clause+)  expr)
       ::=  (for/and      (clause+)  expr)
       ::=  (for/or       (clause+)  expr)
       ::=  (for/sum      (clause+)  expr)
       ::=  (for/product  (clause+)  expr)
       ::=  (for/string   (clause+)  expr)
clause ::=  [variable expr]
```

- Keyword for loop type, comprehension clauses, and a body
- A comprehension clause declares a variable for the sequence of values to iterate over
- For each iteration step the variable becomes next value sequence
- The scope of a comprehension variable is the body of the `for`-loop
- For each value a variable takes on the body of is evaluated
- The type of `for`-loop used defines how the values obtained from evaluating the loop's body are combined:

for/list	`cons`es all the values of type `X`
for/and	`and`s all the `Booleans`
for/or	`or`s all the `Booleans`
for/sum	adds all the `numbers`
for/product	multiplies all the `numbers`
for/string	appends all the `strings`

- The body of the loop must always evaluate to a value of the expected type

For-Loops and Pattern Matching

For Loops

- A comprehension clause declares a variable that shadows previous declarations of the same variable
- Generates the values the variable iterates over
- The type and value of the embedded expression determines the values generated as follows:

List	the elements of the given list from the first to the last element
Natural number	the integers in [0..n-1], where `n` is the given natural number
String	the substrings of length 1 from left to right of the given string

For-Loops and Pattern Matching

For Loops

- A comprehension clause declares a variable that shadows previous declarations of the same variable
- Generates the values the variable iterates over
- The type and value of the embedded expression determines the values generated as follows:

List	the elements of the given list from the first to the last element
Natural number	the integers in [0..n-1], where `n` is the given natural number
String	the substrings of length 1 from left to right of the given string

- The comprehension `[i (build-list 4 (λ (x) x))]` means that `i` is used to iterate over the list elements 0, 1, 2, and 3

For-Loops and Pattern Matching

For Loops

- A comprehension clause declares a variable that shadows previous declarations of the same variable
- Generates the values the variable iterates over
- The type and value of the embedded expression determines the values generated as follows:

List	the elements of the given list from the first to the last element
Natural number	the integers in [0..n-1], where `n` is the given natural number
String	the substrings of length 1 from left to right of the given string

- The comprehension `[i (build-list 4 (λ (x) x))]` means that `i` is used to iterate over the list elements 0, 1, 2, and 3
- The comprehension `[s "dog"]` means that `s` is used to iterate over the substrings of length one: `"d"`, `"o"`, and `"g"`

For-Loops and Pattern Matching

For Loops

- Consider the list-mapping function from Section 116.5:

```
;; <X Y> (X → Y) (listof X) → (listof Y)
;; Purpose: Return values from applying funct to list's elements
(define (map-f f a-lox)
  (if (empty? a-lox)
      '()
      (cons (f (first a-lox))
            (map-f f (rest a-lox)))))
```

- Distinguishes between the subtypes of a list
- Uses list selector functions to access the components of a non-empty list
- Returns a list
- Returns the empty list when the given list is empty
- Otherwise, `cons`es all the results obtained from applying `f` to the first list element

For-Loops and Pattern Matching

For Loops

- Consider the list-mapping function from Section 116.5:

```
;; <X Y> (X → Y) (listof X) → (listof Y)
;; Purpose: Return values from applying funct to list's elements
(define (map-f f a-lox)
  (if (empty? a-lox)
      '()
      (cons (f (first a-lox))
            (map-f f (rest a-lox)))))
```

- Distinguishes between the subtypes of a list
- Uses list selector functions to access the components of a non-empty list
- Returns a list
- Returns the empty list when the given list is empty
- Otherwise, `cons`es all the results obtained from applying `f` to the first list element
- This makes it an ideal candidate for refactoring using `for/list`
- The body must only specify what do with each of the values in the loop: apply the given `f` to it

For-Loops and Pattern Matching

For Loops

- Consider the list-mapping function from Section 116.5:

```
;; <X Y> (X → Y) (listof X) → (listof Y)
;; Purpose: Return values from applying funct to list's elements
(define (map-f f a-lox)
  (if (empty? a-lox)
      '()
      (cons (f (first a-lox))
            (map-f f (rest a-lox)))))
```

- Distinguishes between the subtypes of a list
- Uses list selector functions to access the components of a non-empty list
- Returns a list
- Returns the empty list when the given list is empty
- Otherwise, `cons`es all the results obtained from applying `f` to the first list element
- This makes it an ideal candidate for refactoring using `for/list`
- The body must only specify what do with each of the values in the loop: apply the given `f` to it
- Refactored function:

```
;; <X Y> (X → Y) (listof X) → (listof Y)
;; Purpose: Return values from applying funct to list's elements
(define (map-f f a-lox)
  (for/list ([an-x a-lox])
    (f an-x)))
```

For-Loops and Pattern Matching

For Loops

- ```
  ;; los → los
  ;; Purpose: To move the given list of shots
  (define (move-los a-los)
    (map (lambda (a-shot)
           (cond [(eq? a-shot NO-SHOT) a-shot]
                 [(= (posn-y a-shot) MIN-IMG-Y) NO-SHOT]
                 [else (make-posn
                         (posn-x a-shot)
                         (sub1 (posn-y a-shot)))]))
         a-los))
  ```
  ```

For-Loops and Pattern Matching

For Loops

- ```
  ;; los → los
  ;; Purpose: To move the given list of shots
  (define (move-los a-los)
    (map (lambda (a-shot)
           (cond [(eq? a-shot NO-SHOT) a-shot]
                 [(= (posn-y a-shot) MIN-IMG-Y) NO-SHOT]
                 [else (make-posn
                         (posn-x a-shot)
                         (sub1 (posn-y a-shot)))]))
         a-los))
  ```
- Refactored to use a `for`-loop:

  ```
  (define (move-los a-los)
    (for/list ([a-shot a-los])
      (cond [(eq? a-shot NO-SHOT) a-shot]
            [(= (posn-y a-shot) MIN-IMG-Y) NO-SHOT]
            [else (make-posn
                    (posn-x a-shot)
                    (sub1 (posn-y a-shot)))])))
  ```
  ```

For-Loops and Pattern Matching

For Loops

- Loops may traverse multiple values at the same time
- At each step the next value in each sequence used to evaluate the body
- The loop stops when the end of any sequence is reached
- Whatever remains of the other sequences is ignored

For-Loops and Pattern Matching

For Loops

- Loops may traverse multiple values at the same time
- At each step the next value in each sequence used to evaluate the body
- The loop stops when the end of any sequence is reached
- Whatever remains of the other sequences is ignored
- Consider:

```
(define (mlist L1 L2)
  (if (empty? L1)
      '()
      (cons (* (first L1) (first L2))
            (mlist (rest L1) (rest L2)))))
```

- Both lists are simultaneously traversed

For-Loops and Pattern Matching

For Loops

- Loops may traverse multiple values at the same time
- At each step the next value in each sequence used to evaluate the body
- The loop stops when the end of any sequence is reached
- Whatever remains of the other sequences is ignored
- Consider:

```
(define (mlist L1 L2)
  (if (empty? L1)
      '()
      (cons (* (first L1) (first L2))
            (mlist (rest L1) (rest L2)))))
```

- Both lists are simultaneously traversed
- To refactor this function using a `for`-loop two comprehensions are needed: one for each list
- Body multiplies corresponding elements as values are iterated over

For-Loops and Pattern Matching

For Loops

- Loops may traverse multiple values at the same time
- At each step the next value in each sequence used to evaluate the body
- The loop stops when the end of any sequence is reached
- Whatever remains of the other sequences is ignored
- Consider:

```
(define (mlist L1 L2)
  (if (empty? L1)
      '()
      (cons (* (first L1) (first L2))
            (mlist (rest L1) (rest L2)))))
```

- Both lists are simultaneously traversed
- To refactor this function using a `for`-loop two comprehensions are needed: one for each list
- Body multiplies corresponding elements as values are iterated over
- Refactored function:

```
(define (mlist L1 L2)
  (for/list ([v1 L1] [v2 L2])
    (* v1 v2)))
```

For-Loops and Pattern Matching

For Loops

- Consider writing a predicate to determine if all the elements of a given list of numbers are less than a given threshold
- There are several options: use structural recursion on a list of `numbers`, use `andmap`, or use a `for/and` loop
- Let's a `for/and` loop

For-Loops and Pattern Matching

For Loops

```
;; Sample instances of lon
(define L0 '())
(define L1 '(89 33 77 56 12 8 7))
(define L2 '(8  31 37 44 12 2 4))
```

For-Loops and Pattern Matching

For Loops

- ```
  ;; Sample instances of lon
  (define L0 '())
  (define L1 '(89 33 77 56 12 8 7))
  (define L2 '(8  31 37 44 12 2 4))
  ```

- ```
  ;; Sample expressions for all-lt
  (define L0-VAL (for/and ([v L0]) (< v 887)))
  (define L1-VAL (for/and ([v L1]) (< v 100)))
  (define L2-VAL (for/and ([v L2]) (< v 20)))
  ```

For-Loops and Pattern Matching

For Loops

- ```
  ;; Sample instances of lon
  (define L0 '())
  (define L1 '(89 33 77 56 12 8 7))
  (define L2 '(8  31 37 44 12 2 4))
  ```
- ```
  ;; all-lt: (listof number) number → boolean
  ;; Purpose: Determine if numbers given list are < given number
  (define (all-lt L threshold)
  ```

- ```
  ;; Sample expressions for all-lt
  (define L0-VAL (for/and ([v L0]) (< v 887)))
  (define L1-VAL (for/and ([v L1]) (< v 100)))
  (define L2-VAL (for/and ([v L2]) (< v 20)))
  ```
  ```

For-Loops and Pattern Matching

For Loops

- ```
  ;; Sample instances of lon
  (define L0 '())
  (define L1 '(89 33 77 56 12 8 7))
  (define L2 '(8  31 37 44 12 2 4))
  ```
- ```
  ;; all-lt: (listof number) number → boolean
  ;; Purpose: Determine if numbers given list are < given number
  (define (all-lt L threshold)
  ```

- ```
  ;; Sample expressions for all-lt
  (define L0-VAL (for/and ([v L0]) (< v 887)))
  (define L1-VAL (for/and ([v L1]) (< v 100)))
  (define L2-VAL (for/and ([v L2]) (< v 20)))
  ```
- ```
  ;; Tests using sample computations for all-lt
  (check-expect (all-lt L0 887) L0-VAL)
  (check-expect (all-lt L1 100) L1-VAL)
  (check-expect (all-lt L2 20)  L2-VAL)

  ;; Tests using sample values for all-lt
  (check-expect (all-lt '() 5)            #true)
  (check-expect (all-lt (list -1 2 -3) 0)  #false)
  (check-expect (all-lt (list 3 -8 -4) 10) #true)
  ```

For-Loops and Pattern Matching

For Loops

```
;; Sample instances of lon
(define L0 '())
(define L1 '(89 33 77 56 12 8 7))
(define L2 '(8  31 37 44 12 2 4))
```

```
;; all-lt: (listof number) number → boolean
;; Purpose: Determine if numbers given list are < given number
(define (all-lt L threshold)
  (for/and ([v L])
    (< v threshold)))
```

```
;; Sample expressions for all-lt
(define L0-VAL (for/and ([v L0]) (< v 887)))
(define L1-VAL (for/and ([v L1]) (< v 100)))
(define L2-VAL (for/and ([v L2]) (< v 20)))
```

```
;; Tests using sample computations for all-lt
(check-expect (all-lt L0 887) L0-VAL)
(check-expect (all-lt L1 100) L1-VAL)
(check-expect (all-lt L2 20)  L2-VAL)

;; Tests using sample values for all-lt
(check-expect (all-lt '() 5)            #true)
(check-expect (all-lt (list -1 2 -3) 0)  #false)
(check-expect (all-lt (list 3 -8 -4) 10) #true)
```

For-Loops and Pattern Matching

Homework

- Problems: 302–307

For-Loops and Pattern Matching

for*-loops

- The second kind of ISL+ loop, for*-loop
- Similar syntax and the same 6 varieties as for-loops:

```
expr  ::=  (for*/list     (clause+)  expr)
      ::=  (for*/and      (clause+)  expr)
      ::=  (for*/or       (clause+)  expr)
      ::=  (for*/sum      (clause+)  expr)
      ::=  (for*/product  (clause+)  expr)
      ::=  (for*/string   (clause+)  expr)
```

For-Loops and Pattern Matching

`for*`-loops

- The second kind of ISL+ loop, `for*`-loop
- Similar syntax and the same 6 varieties as `for`-loops:

```
expr ::= (for*/list    (clause+) expr)
     ::= (for*/and     (clause+) expr)
     ::= (for*/or      (clause+) expr)
     ::= (for*/sum     (clause+) expr)
     ::= (for*/product (clause+) expr)
     ::= (for*/string  (clause+) expr)
```

- The difference with `for`-loops is the scope of the comprehension variables
- The scope of a comprehension variable is the remaining comprehension clauses and the loop's body

For-Loops and Pattern Matching

for*-loops

- The second kind of ISL+ loop, for*-loop
- Similar syntax and the same 6 varieties as for-loops:

```
expr  ::=  (for*/list     (clause+)  expr)
      ::=  (for*/and      (clause+)  expr)
      ::=  (for*/or       (clause+)  expr)
      ::=  (for*/sum      (clause+)  expr)
      ::=  (for*/product  (clause+)  expr)
      ::=  (for*/string   (clause+)  expr)
```

- The difference with for-loops is the scope of the comprehension variables
- The scope of a comprehension variable is the remaining comprehension clauses and the loop's body
- In addition, the sequences of values are iterated over differently
- For every value taken on by the first comprehension variable the second sequence of values is iterated over
- For every value taken on by the second comprehension variable the third sequence of values is iterated
- This pattern is repeated for each subsequent comprehension
- Observe that there is no difference between a for-loop and a for*-loop if there is only one comprehension clause

For-Loops and Pattern Matching

`for*`-loops

- Consider the problem of computing the cross product of two lists
- The cross product of two lists, `L1` and `L2`, is all pairs, `(a b)`, where `a` is a member of `L1` and `b` is a member of `L2`

For-Loops and Pattern Matching

`for*`-loops

- Consider the problem of computing the cross product of two lists
- The cross product of two lists, `L1` and `L2`, is all pairs, `(a b)`, where `a` is a member of `L1` and `b` is a member of `L2`
- The cross product of '(a b) and '(7 8 9) is:

```
'((a 7) (a 8) (a 9)
  (b 7) (b 8) (b 9))
```

- How can such a list be created?

For-Loops and Pattern Matching

`for*`-loops

- Consider the problem of computing the cross product of two lists
- The cross product of two lists, `L1` and `L2`, is all pairs, `(a b)`, where `a` is a member of `L1` and `b` is a member of `L2`
- The cross product of '(a b) and '(7 8 9) is:

```
'((a 7) (a 8) (a 9)
  (b 7) (b 8) (b 9))
```

- How can such a list be created?
- All the values in the first list must be iterated over
- What needs to be done for each of these values?

For-Loops and Pattern Matching

`for*`-loops

- Consider the problem of computing the cross product of two lists
- The cross product of two lists, `L1` and `L2`, is all pairs, `(a b)`, where `a` is a member of `L1` and `b` is a member of `L2`
- The cross product of '(a b) and '(7 8 9) is:

```
'((a 7) (a 8) (a 9)
  (b 7) (b 8) (b 9))
```

- How can such a list be created?
- All the values in the first list must be iterated over
- What needs to be done for each of these values?
- Must be matched with every value in the second list
- The second list must be iterated over to form the needed pairs

For-Loops and Pattern Matching

`for*`-loops

- Consider the problem of computing the cross product of two lists
- The cross product of two lists, `L1` and `L2`, is all pairs, `(a b)`, where `a` is a member of `L1` and `b` is a member of `L2`
- The cross product of '(a b) and '(7 8 9) is:

```
'((a 7) (a 8) (a 9)
  (b 7) (b 8) (b 9))
```

- How can such a list be created?
- All the values in the first list must be iterated over
- What needs to be done for each of these values?
- Must be matched with every value in the second list
- The second list must be iterated over to form the needed pairs
- All of the pairs must be `cons`ed together to create the needed list
- Suggests using a `for*/list`
- The loop's body creates a pair by `list`ing the current values of the two lists

For-Loops and Pattern Matching

for*-loops

- ```
  ;; Sample instances of (listof X)
  (define L0 '())
  (define L1 '(i j c d))
  (define L2 '(7 6 3))
  ```
  ```

For-Loops and Pattern Matching

for*-loops

- ```
  ;; Sample instances of (listof X)
  (define L0 '())
  (define L1 '(i j c d))
  (define L2 '(7 6 3))
  ```

- ```
  ;; Sample expressions for cross-product
  (define L0-L0-VAL (for*/list ([v1 L0] [v2 L0])
                      (list v1 v2)))
  (define L0-L1-VAL (for*/list ([v1 L0] [v2 L1])
                      (list v1 v2)))
  (define L2-L0-VAL (for*/list ([v1 L2] [v2 L0])
                      (list v1 v2)))
  (define L1-L2-VAL (for*/list ([v1 L1] [v2 L2])
                      (list v1 v2)))
  ```

For-Loops and Pattern Matching

for*-loops

- ```
  ;; Sample instances of (listof X)
  (define L0 '())
  (define L1 '(i j c d))
  (define L2 '(7 6 3))
  ```
- ```
  ;; <X Y> (listof X) (listof Y) → (listof (list X Y))
  ;; Purpose: To compute the cross product of the two lists
  (define (cross-product a-lox a-loy)
  ```
- ```
  ;; Sample expressions for cross-product
  (define L0-L0-VAL (for*/list ([v1 L0] [v2 L0])
                      (list v1 v2)))
  (define L0-L1-VAL (for*/list ([v1 L0] [v2 L1])
                      (list v1 v2)))
  (define L2-L0-VAL (for*/list ([v1 L2] [v2 L0])
                      (list v1 v2)))
  (define L1-L2-VAL (for*/list ([v1 L1] [v2 L2])
                      (list v1 v2)))
  ```
  ```

For-Loops and Pattern Matching

for*-loops

```
;; Sample instances of (listof X)
(define L0 '())
(define L1 '(i j c d))
(define L2 '(7 6 3))
```

```
;; <X Y> (listof X) (listof Y) → (listof (list X Y))
;; Purpose: To compute the cross product of the two lists
(define (cross-product a-lox a-loy)
```

```
;; Sample expressions for cross-product
(define L0-L0-VAL (for*/list ([v1 L0] [v2 L0])
                    (list v1 v2)))
(define L0-L1-VAL (for*/list ([v1 L0] [v2 L1])
                    (list v1 v2)))
(define L2-L0-VAL (for*/list ([v1 L2] [v2 L0])
                    (list v1 v2)))
(define L1-L2-VAL (for*/list ([v1 L1] [v2 L2])
                    (list v1 v2)))
```

```
;; Tests using sample computations for cross-product
(check-expect (cross-product L0 L0) L0-L0-VAL)
(check-expect (cross-product L0 L1) L0-L1-VAL)
(check-expect (cross-product L2 L0) L2-L0-VAL)
(check-expect (cross-product L1 L2) L1-L2-VAL)

;; Tests using sample values for cross-product
(check-expect (cross-product '() '()) '())
(check-expect (cross-product '() '(x y z)) '())
(check-expect (cross-product '(x y z) '()) '())
(check-expect (cross-product '(a b) '(7 8 9))
              (list (list 'a 7) (list 'a 8) (list 'a 9)
                    (list 'b 7) (list 'b 8) (list 'b 9)))
```

For-Loops and Pattern Matching

for*-loops

```
;; Sample instances of (listof X)
(define L0 '())
(define L1 '(i j c d))
(define L2 '(7 6 3))
```

```
;; <X Y> (listof X) (listof Y) → (listof (list X Y))
;; Purpose: To compute the cross product of the two lists
(define (cross-product a-lox a-loy)
  (for*/list ([v1 a-lox] [v2 a-loy])
    (list v1 v2)))
```

```
;; Sample expressions for cross-product
(define L0-L0-VAL (for*/list ([v1 L0] [v2 L0])
                    (list v1 v2)))
(define L0-L1-VAL (for*/list ([v1 L0] [v2 L1])
                    (list v1 v2)))
(define L2-L0-VAL (for*/list ([v1 L2] [v2 L0])
                    (list v1 v2)))
(define L1-L2-VAL (for*/list ([v1 L1] [v2 L2])
                    (list v1 v2)))
```

```
;; Tests using sample computations for cross-product
(check-expect (cross-product L0 L0) L0-L0-VAL)
(check-expect (cross-product L0 L1) L0-L1-VAL)
(check-expect (cross-product L2 L0) L2-L0-VAL)
(check-expect (cross-product L1 L2) L1-L2-VAL)

;; Tests using sample values for cross-product
(check-expect (cross-product '() '()) '())
(check-expect (cross-product '() '(x y z)) '())
(check-expect (cross-product '(x y z) '()) '())
(check-expect (cross-product '(a b) '(7 8 9))
              (list (list 'a 7) (list 'a 8) (list 'a 9)
                    (list 'b 7) (list 'b 8) (list 'b 9)))
```

For-Loops and Pattern Matching

`for*`-loops

- Consider flattening a `(listof (listof symbol))`
- Every element of the given list is a list of symbols
- A list containing all the elements of all the sublists needs to be computed
- Given `'((o p) (q r s t) (u))` the result is `'(o p q r s t u)`
- How can this be accomplished?

For-Loops and Pattern Matching

`for*`-loops

- Consider flattening a `(listof (listof symbol))`
- Every element of the given list is a list of symbols
- A list containing all the elements of all the sublists needs to be computed
- Given `'((o p) (q r s t) (u))` the result is `'(o p q r s t u)`
- How can this be accomplished?
- Each sublist must be processed
- This may be done by iterating over the elements of the given list
- What needs to be done with each sublist?

For-Loops and Pattern Matching

`for*`-loops

- Consider flattening a `(listof (listof symbol))`
- Every element of the given list is a list of symbols
- A list containing all the elements of all the sublists needs to be computed
- Given `'((o p) (q r s t) (u))` the result is `'(o p q r s t u)`
- How can this be accomplished?
- Each sublist must be processed
- This may be done by iterating over the elements of the given list
- What needs to be done with each sublist?
- Each element of a sublist must be added as the member of the result
- This may be accomplished by iterating through the sublist and returning each element
- Suggests using a `for*/list` loop

For-Loops and Pattern Matching

for*-loops

```
;; Sample instances of (listof (listof symbol))
(define L0 '())
(define L1 '((a b c) (d e f g h) (i j)))
```

For-Loops and Pattern Matching

for*-loops

- ```
  ;; Sample instances of (listof (listof symbol))
  (define L0 '())
  (define L1 '((a b c) (d e f g h) (i j)))
  ```

- ```
  ;; Sample expressions for flatten
  (define L0-VAL (for*/list ([a-los L0] (a-symbol a-los))
                   a-symbol))
  (define L1-VAL (for*/list ([a-los L1] (a-symbol a-los))
                   a-symbol))
  ```

For-Loops and Pattern Matching

for*-loops

- ```
  ;; Sample instances of (listof (listof symbol))
  (define L0 '())
  (define L1 '((a b c) (d e f g h) (i j)))
  ```
- ```
  ;; (listof (listof symbol)) → (listof symbol)
  ;; Purpose: Flatten the given (listof (listof symbol))
  (define (flatten-lolos a-lolos)
  ```

- ```
  ;; Sample expressions for flatten
  (define L0-VAL (for*/list ([a-los L0] (a-symbol a-los))
                   a-symbol))
  (define L1-VAL (for*/list ([a-los L1] (a-symbol a-los))
                   a-symbol))
  ```
  ```

For-Loops and Pattern Matching

for*-loops

```
;; Sample instances of (listof (listof symbol))
(define L0 '())
(define L1 '((a b c) (d e f g h) (i j)))
```

```
;; (listof (listof symbol)) → (listof symbol)
;; Purpose: Flatten the given (listof (listof symbol))
(define (flatten-lolos a-lolos)
```

```
;; Sample expressions for flatten
(define L0-VAL (for*/list ([a-los L0] (a-symbol a-los))
                 a-symbol))
(define L1-VAL (for*/list ([a-los L1] (a-symbol a-los))
                 a-symbol))
```

```
;; Tests using sample computations for flatten-lolos
(check-expect (flatten-lolos L0) L0-VAL)
(check-expect (flatten-lolos L1) L1-VAL)

;; Tests using sample values for flatten-lolos
(check-expect (flatten-lolos '((x) (y) (z))) '(x y z))
(check-expect (flatten-lolos '((k l m n))) '(k l m n))
(check-expect (flatten-lolos '((o p) () (q r s t) (u)))
              '(o p q r s t u))
```

For-Loops and Pattern Matching

for*-loops

```
;; Sample instances of (listof (listof symbol))
(define L0 '())
(define L1 '((a b c) (d e f g h) (i j)))
```

```
;; (listof (listof symbol)) → (listof symbol)
;; Purpose: Flatten the given (listof (listof symbol))
(define (flatten-lolos a-lolos)
```

```
  (for*/list ([a-los a-lolos] [a-symbol a-los])
    a-symbol))
```

```
;; Sample expressions for flatten
(define L0-VAL (for*/list ([a-los L0] (a-symbol a-los))
                 a-symbol))
(define L1-VAL (for*/list ([a-los L1] (a-symbol a-los))
                 a-symbol))
```

```
;; Tests using sample computations for flatten-lolos
(check-expect (flatten-lolos L0) L0-VAL)
(check-expect (flatten-lolos L1) L1-VAL)

;; Tests using sample values for flatten-lolos
(check-expect (flatten-lolos '((x) (y) (z))) '(x y z))
(check-expect (flatten-lolos '((k l m n))) '(k l m n))
(check-expect (flatten-lolos '((o p) () (q r s t) (u)))
              '(o p q r s t u))
```

For-Loops and Pattern Matching

Homework

- Problems: 309–310

For-Loops and Pattern Matching

Pattern Matching

- Whenever there is variety in the data we have used a conditional expression
- If it is compound data selector functions are used in every function
- This repetition calls for an abstraction

For-Loops and Pattern Matching

Pattern Matching

- Whenever there is variety in the data we have used a conditional expression
- If it is compound data selector functions are used in every function
- This repetition calls for an abstraction
- Pattern matching eliminates this repetition based on the type of data processed
- In ISL+:

```
expr    ::=  (match expr
               [pattern expr]+)
        ::=  (match expr
               [pattern expr]+)
               [else expr]+)

pattern ::=  literal
        ::=  variable
        ::=  (cons pattern pattern)
        ::=  (structure-name pattern*)
        ::=  (? predicate-name)
```

For-Loops and Pattern Matching

Pattern Matching

- Whenever there is variety in the data we have used a conditional expression
- If it is compound data selector functions are used in every function
- This repetition calls for an abstraction
- Pattern matching eliminates this repetition based on the type of data processed
- In ISL+:

```
expr    ::=  (match expr
               [pattern expr]+)
        ::=  (match expr
               [pattern expr]+)
               [else expr]+)

pattern ::=  literal
        ::=  variable
        ::=  (cons pattern pattern)
        ::=  (structure-name pattern*)
        ::=  (? predicate-name)
```

- A pattern may match a constant

For-Loops and Pattern Matching

Pattern Matching

- Whenever there is variety in the data we have used a conditional expression
- If it is compound data selector functions are used in every function
- This repetition calls for an abstraction
- Pattern matching eliminates this repetition based on the type of data processed
- In ISL+:

```
expr    ::=  (match expr
               [pattern expr]+)
        ::=  (match expr
               [pattern expr]+)
               [else expr]+)

pattern ::=  literal
        ::=  variable
        ::=  (cons pattern pattern)
        ::=  (structure-name pattern*)
        ::=  (? predicate-name)
```

- A pattern may match a constant
- A pattern may match a variable and take the value of the matching expression

For-Loops and Pattern Matching

Pattern Matching

- Whenever there is variety in the data we have used a conditional expression
- If it is compound data selector functions are used in every function
- This repetition calls for an abstraction
- Pattern matching eliminates this repetition based on the type of data processed
- In ISL+:

```
expr    ::=  (match expr
               [pattern expr]+)
        ::=  (match expr
               [pattern expr]+)
               [else expr]+)

pattern ::=  literal
        ::=  variable
        ::=  (cons pattern pattern)
        ::=  (structure-name pattern*)
        ::=  (? predicate-name)
```

- A pattern may match a constant
- A pattern may match a variable and take the value of the matching expression
- A pattern may match a list: match patterns for the first and rest

For-Loops and Pattern Matching

Pattern Matching

- Whenever there is variety in the data we have used a conditional expression
- If it is compound data selector functions are used in every function
- This repetition calls for an abstraction
- Pattern matching eliminates this repetition based on the type of data processed
- In ISL+:

```
expr    ::=  (match expr
               [pattern expr]+)
        ::=  (match expr
               [pattern expr]+)
               [else expr]+)

pattern ::=  literal
        ::=  variable
        ::=  (cons pattern pattern)
        ::=  (structure-name pattern*)
        ::=  (? predicate-name)
```

- A pattern may match a constant
- A pattern may match a variable and take the value of the matching expression
- A pattern may match a list: match patterns for the first and rest
- A pattern may match a structure: a pattern for each of its components

For-Loops and Pattern Matching

Pattern Matching

- Whenever there is variety in the data we have used a conditional expression
- If it is compound data selector functions are used in every function
- This repetition calls for an abstraction
- Pattern matching eliminates this repetition based on the type of data processed
- In ISL+:

```
expr    ::=  (match expr
               [pattern expr]+)
        ::=  (match expr
               [pattern expr]+)
               [else expr]+)

pattern ::=  literal
        ::=  variable
        ::=  (cons pattern pattern)
        ::=  (structure-name pattern*)
        ::=  (? predicate-name)
```

- A pattern may match a constant
- A pattern may match a variable and take the value of the matching expression
- A pattern may match a list: match patterns for the first and rest
- A pattern may match a structure: a pattern for each of its components
- A pattern may match by testing a predicate

For-Loops and Pattern Matching

Pattern Matching

- Whenever there is variety in the data we have used a conditional expression
- If it is compound data selector functions are used in every function
- This repetition calls for an abstraction
- Pattern matching eliminates this repetition based on the type of data processed
- In ISL+:

```
expr    ::=  (match expr
               [pattern expr]+)
        ::=  (match expr
               [pattern expr]+)
               [else expr]+)

pattern ::=  literal
        ::=  variable
        ::=  (cons pattern pattern)
        ::=  (structure-name pattern*)
        ::=  (? predicate-name)
```

- A pattern may match a constant
- A pattern may match a variable and take the value of the matching expression
- A pattern may match a list: match patterns for the first and rest
- A pattern may match a structure: a pattern for each of its components
- A pattern may match by testing a predicate
- The scope of a variable is the corresponding expression in the match stanza

For-Loops and Pattern Matching

Pattern Matching

- ```
  (require 2htdp/abstraction)
  (define-struct student (name year gpa))  (define X 42)  (define B "Basia Mucha")
  (define DIMITROVA (make-student "Rositsa Abrasheva" 'senior 3.94))
  (define BLST '(#true #true #true))  (define NLST '(1 2 3))  (define ELST '())
  ;; Any → string Purpose: Illustrate the semantics of match-expressions
  (define (f a-value)
    (match a-value
      [42 "Matched 42"]
      [(? string?) (string-append "Matched a string: " a-value)]
      [(student n y g)
       (string-append "Matched a student whose gpa is: " (number->string g))]
      [(cons (? boolean?) r)
       (string-append "Matched a list that starts with a Boolean: "
                      (boolean->string (first a-value)))]
      [(cons B C) (format "Matched a list whose first element is: ~s" B)]
      [else "Nothing is matched."]))
  ;; Sample expressions for f
  (define X-VAL "Matched 42")  (define B-VAL (string-append "Matched a string: " B))
  (define DIMITROVA-VAL
    (string-append "Matched a student whose gpa is: "
                   (number->string (student-gpa DIMITROVA))))
  (define BLST-VAL
   (string-append "Matched a list that starts with a Boolean: "
                  (boolean->string (first BLST))))
  (define NLST-VAL (format "Matched a list whose first element is: ~s" (first NLST)))
  (define ELST-VAL "Nothing is matched.")
  ```
  ```

For-Loops and Pattern Matching

Pattern Matching

- ```
  (require 2htdp/abstraction)
  (define-struct student (name year gpa))  (define X 42)  (define B "Basia Mucha")
  (define DIMITROVA (make-student "Rositsa Abrasheva" 'senior 3.94))
  (define BLST '(#true #true #true))  (define NLST '(1 2 3))  (define ELST '())
  ;; Any → string Purpose: Illustrate the semantics of match-expressions
  (define (f a-value)
    (match a-value
      [42 "Matched 42"]
      [(? string?) (string-append "Matched a string: " a-value)]
      [(student n y g)
       (string-append "Matched a student whose gpa is: " (number->string g))]
      [(cons (? boolean?) r)
       (string-append "Matched a list that starts with a Boolean: "
                      (boolean->string (first a-value)))]
      [(cons B C) (format "Matched a list whose first element is: ~s" B)]
      [else "Nothing is matched."]))
  ;; Sample expressions for f
  (define X-VAL "Matched 42")  (define B-VAL (string-append "Matched a string: " B))
  (define DIMITROVA-VAL
    (string-append "Matched a student whose gpa is: "
                   (number->string (student-gpa DIMITROVA))))
  (define BLST-VAL
   (string-append "Matched a list that starts with a Boolean: "
                  (boolean->string (first BLST))))
  (define NLST-VAL (format "Matched a list whose first element is: ~s" (first NLST)))
  (define ELST-VAL "Nothing is matched.")
  ```
- ```
  (check-expect (f X) X-VAL) Matched in first stanza
  (check-expect (f B) B-VAL) Matched in second stanza
  (check-expect (f "Matthew Flatt") Matched in second stanza
                "Matched a string: Matthew Flatt")
  ```

For-Loops and Pattern Matching

Pattern Matching

- ```
  (require 2htdp/abstraction)
  (define-struct student (name year gpa))  (define X 42)  (define B "Basia Mucha")
  (define DIMITROVA (make-student "Rositsa Abrasheva" 'senior 3.94))
  (define BLST '(#true #true #true))  (define NLST '(1 2 3))  (define ELST '())
  ;; Any → string Purpose: Illustrate the semantics of match-expressions
  (define (f a-value)
    (match a-value
      [42 "Matched 42"]
      [(? string?) (string-append "Matched a string: " a-value)]
      [(student n y g)
       (string-append "Matched a student whose gpa is: " (number->string g))]
      [(cons (? boolean?) r)
       (string-append "Matched a list that starts with a Boolean: "
                      (boolean->string (first a-value)))]
      [(cons B C) (format "Matched a list whose first element is: ~s" B)]
      [else "Nothing is matched."]))
  ;; Sample expressions for f
  (define X-VAL "Matched 42")  (define B-VAL (string-append "Matched a string: " B))
  (define DIMITROVA-VAL
    (string-append "Matched a student whose gpa is: "
                   (number->string (student-gpa DIMITROVA))))
  (define BLST-VAL
   (string-append "Matched a list that starts with a Boolean: "
                  (boolean->string (first BLST))))
  (define NLST-VAL (format "Matched a list whose first element is: ~s" (first NLST)))
  (define ELST-VAL "Nothing is matched.")
  ```
- ```
  (check-expect (f DIMITROVA) DIMITROVA-VAL) Matched in third stanza
  (check-expect (f (make-student "Robby Findler" 'freshman 4))
                "Matched a student whose gpa is: 4") Matched in third stanza
  ```

For-Loops and Pattern Matching

Pattern Matching

- ```
  (require 2htdp/abstraction)
  (define-struct student (name year gpa))  (define X 42)  (define B "Basia Mucha")
  (define DIMITROVA (make-student "Rositsa Abrasheva" 'senior 3.94))
  (define BLST '(#true #true #true))  (define NLST '(1 2 3))  (define ELST '())
  ;; Any → string Purpose: Illustrate the semantics of match-expressions
  (define (f a-value)
    (match a-value
      [42 "Matched 42"]
      [(? string?) (string-append "Matched a string: " a-value)]
      [(student n y g)
       (string-append "Matched a student whose gpa is: " (number->string g))]
      [(cons (? boolean?) r)
       (string-append "Matched a list that starts with a Boolean: "
                      (boolean->string (first a-value)))]
      [(cons B C) (format "Matched a list whose first element is: ~s" B)]
      [else "Nothing is matched."]))
  ;; Sample expressions for f
  (define X-VAL "Matched 42")  (define B-VAL (string-append "Matched a string: " B))
  (define DIMITROVA-VAL
    (string-append "Matched a student whose gpa is: "
                   (number->string (student-gpa DIMITROVA))))
  (define BLST-VAL
   (string-append "Matched a list that starts with a Boolean: "
                  (boolean->string (first BLST))))
  (define NLST-VAL (format "Matched a list whose first element is: ~s" (first NLST)))
  (define ELST-VAL "Nothing is matched.")
  ```
- ```
  (check-expect (f BLST) BLST-VAL) Matched in fourth stanza
  (check-expect
    (f '(#false)) Matched in fourth stanza
    "Matched a list that starts with a Boolean: #false")
  ```

For-Loops and Pattern Matching

Pattern Matching

- ```
  (require 2htdp/abstraction)
  (define-struct student (name year gpa))  (define X 42)  (define B "Basia Mucha")
  (define DIMITROVA (make-student "Rositsa Abrasheva" 'senior 3.94))
  (define BLST '(#true #true #true))  (define NLST '(1 2 3))  (define ELST '())
  ;; Any → string Purpose: Illustrate the semantics of match-expressions
  (define (f a-value)
    (match a-value
      [42 "Matched 42"]
      [(? string?) (string-append "Matched a string: " a-value)]
      [(student n y g)
       (string-append "Matched a student whose gpa is: " (number->string g))]
      [(cons (? boolean?) r)
       (string-append "Matched a list that starts with a Boolean: "
                      (boolean->string (first a-value)))]
      [(cons B C) (format "Matched a list whose first element is: ~s" B)]
      [else "Nothing is matched."]))
  ;; Sample expressions for f
  (define X-VAL "Matched 42")  (define B-VAL (string-append "Matched a string: " B))
  (define DIMITROVA-VAL
    (string-append "Matched a student whose gpa is: "
                   (number->string (student-gpa DIMITROVA))))
  (define BLST-VAL
   (string-append "Matched a list that starts with a Boolean: "
                  (boolean->string (first BLST))))
  (define NLST-VAL (format "Matched a list whose first element is: ~s" (first NLST)))
  (define ELST-VAL "Nothing is matched.")
  ```
- ```
  (check-expect (f NLST) NLST-VAL) Matched in fifth stanza
  (check-expect Matched in fifth stanza
    (f '("P. Achten" "J. Hughes" "P. Koopman"))
    "Matched a list whose first element is: \"P. Achten\"")
  ```

For-Loops and Pattern Matching

Pattern Matching

- ```
  (require 2htdp/abstraction)
  (define-struct student (name year gpa))  (define X 42)  (define B "Basia Mucha")
  (define DIMITROVA (make-student "Rositsa Abrasheva" 'senior 3.94))
  (define BLST '(#true #true #true))  (define NLST '(1 2 3))  (define ELST '())
  ;; Any → string Purpose: Illustrate the semantics of match-expressions
  (define (f a-value)
    (match a-value
      [42 "Matched 42"]
      [(? string?) (string-append "Matched a string: " a-value)]
      [(student n y g)
       (string-append "Matched a student whose gpa is: " (number->string g))]
      [(cons (? boolean?) r)
       (string-append "Matched a list that starts with a Boolean: "
                      (boolean->string (first a-value)))]
      [(cons B C) (format "Matched a list whose first element is: ~s" B)]
      [else "Nothing is matched."]))
  ;; Sample expressions for f
  (define X-VAL "Matched 42")  (define B-VAL (string-append "Matched a string: " B))
  (define DIMITROVA-VAL
    (string-append "Matched a student whose gpa is: "
                   (number->string (student-gpa DIMITROVA))))
  (define BLST-VAL
   (string-append "Matched a list that starts with a Boolean: "
                  (boolean->string (first BLST))))
  (define NLST-VAL (format "Matched a list whose first element is: ~s" (first NLST)))
  (define ELST-VAL "Nothing is matched.")
  ```
- ```
  (check-expect (f '())   ELST-VAL) no match, processed by else
  (check-expect (f (square 10 "solid" "green")) no match, processed by else
                "Nothing is matched.")
  ```

For-Loops and Pattern Matching

Pattern Matching

- Recall:

```
;; <X Y Z> Z (X → Y) (Y Z → Z) (listof X) → Z  Purpose: Summarize given list
(define (accum base-val ffirst comb L)
 (if (empty? L) base-val
     (comb (ffirst (first L)) (accum base-val ffirst comb (rest L)))))
;; X → X     Purpose: Return the given X
(define (id an-x) an-x)
;; (listof sexpr) → (listof number) throws error
;; Purpose: To evaluate the sexprs in the given list
(define (eval-args a-losexpr)
 (if (empty? a-losexpr) '()
     (cons (eval-sexpr (first a-losexpr)) (eval-args (rest a-losexpr)))))
;; slist → number throws error  Purpose: To evaluate the given slist
(define (eval-slist sl) (apply-f (first sl) (eval-args (rest sl))))
;; sexpr → number throws error  Purpose: To evaluate the given sexpr
(define (eval-sexpr a-sexpr)
 (cond [(number? a-sexpr) a-sexpr]
       [else (eval-slist a-sexpr)]))
;; function (listof number) → number throws error  Purpose: Apply funct to nums
(define (apply-f a-function a-lon)
 (cond [(eq? a-function '+) (sum-lon  a-lon)]
       [(eq? a-function '-) (subt-lon a-lon)]
       [else (mult-lon a-lon)]))
;; lon → number     Purpose: To sum the given lon
(define (sum-lon a-lon)
 (if (empty? a-lon) 0 (+ (first a-lon) (sum-lon (rest a-lon)))))
;; lon → number throws error  Purpose: To subtract the given lon
(define (subt-lon a-lon)
(if (empty? a-lon) (error "No numbers provided to -.")
    (- (first a-lon) (sum-lon (rest a-lon)))))
;; lon → number                Purpose: To multiply the given lon
(define (mult-lon a-lon) (if (empty? a-lon) 1 (* (first a-lon) (mult-lon (rest a-lon)))))
```

For-Loops and Pattern Matching

Pattern Matching

- ```
  ;; lon → number     Purpose: To sum the given lon
  (define (sum-lon a-lon)
   (if (empty? a-lon) 0 (+ (first a-lon) (sum-lon (rest a-lon)))))

  ;; lon → number throws error  Purpose: To subtract the given lon
  (define (subt-lon a-lon)
  (if (empty? a-lon) (error "No numbers provided to -.")
      (- (first a-lon) (sum-lon (rest a-lon)))))

  ;; lon → number                Purpose: To multiply the given lon
  (define (mult-lon a-lon) (if (empty? a-lon) 1 (* (first a-lon) (mult-lon (rest a-lon)))))
  ```
- `sum-lon`, `subt-lon`, and `mult-lon` are all very similar because they are list-summarizing operations
- This suggest refactoring using `accum`
  ```

For-Loops and Pattern Matching

Pattern Matching

- ```
  ;; lon → number     Purpose: To sum the given lon
  (define (sum-lon a-lon)
   (if (empty? a-lon) 0 (+ (first a-lon) (sum-lon (rest a-lon)))))

  ;; lon → number throws error  Purpose: To subtract the given lon
  (define (subt-lon a-lon)
  (if (empty? a-lon) (error "No numbers provided to -.")
      (- (first a-lon) (sum-lon (rest a-lon)))))

  ;; lon → number                Purpose: To multiply the given lon
  (define (mult-lon a-lon) (if (empty? a-lon) 1 (* (first a-lon) (mult-lon (rest a-lon)))))
  ```
- sum-lon, subt-lon, and mult-lon are all very similar because they are list-summarizing operations
- This suggest refactoring using accum
- ```
  ;; lon → number
  ;; Purpose: To sum the given lon
  (define (sum-lon a-lon) (accum 0 id + a-lon))

  ;; lon → number     Purpose: To multiply the given lon
  (define (mult-lon a-lon) (accum 1 id * a-lon))
  ```

For-Loops and Pattern Matching

Pattern Matching

- ```
  ;; lon → number     Purpose: To sum the given lon
  (define (sum-lon a-lon)
   (if (empty? a-lon) 0 (+ (first a-lon) (sum-lon (rest a-lon)))))

  ;; lon → number throws error  Purpose: To subtract the given lon
  (define (subt-lon a-lon)
  (if (empty? a-lon) (error "No numbers provided to -.")
      (- (first a-lon) (sum-lon (rest a-lon)))))

  ;; lon → number                Purpose: To multiply the given lon
  (define (mult-lon a-lon) (if (empty? a-lon) 1 (* (first a-lon) (mult-lon (rest a-lon)))))
  ```
- sum-lon, subt-lon, and mult-lon are all very similar because they are list-summarizing operations
- This suggest refactoring using accum
- ```
  ;; lon → number
  ;; Purpose: To sum the given lon
  (define (sum-lon a-lon) (accum 0 id + a-lon))

  ;; lon → number     Purpose: To multiply the given lon
  (define (mult-lon a-lon) (accum 1 id * a-lon))
  ```
- subt-lon requires a little algebraic manipulation
- (- a b c) = (* -1 (+ (* -1 a) b c))
- A match-expression may be used to eliminate the use of first and rest

For-Loops and Pattern Matching

Pattern Matching

- ```
  ;; lon → number     Purpose: To sum the given lon
  (define (sum-lon a-lon)
   (if (empty? a-lon) 0 (+ (first a-lon) (sum-lon (rest a-lon)))))

  ;; lon → number throws error  Purpose: To subtract the given lon
  (define (subt-lon a-lon)
  (if (empty? a-lon) (error "No numbers provided to -.")
      (- (first a-lon) (sum-lon (rest a-lon)))))

  ;; lon → number                Purpose: To multiply the given lon
  (define (mult-lon a-lon) (if (empty? a-lon) 1 (* (first a-lon) (mult-lon (rest a-lon)))))
  ```
- sum-lon, subt-lon, and mult-lon are all very similar because they are list-summarizing operations
- This suggest refactoring using accum
- ```
  ;; lon → number
  ;; Purpose: To sum the given lon
  (define (sum-lon a-lon) (accum 0 id + a-lon))

  ;; lon → number     Purpose: To multiply the given lon
  (define (mult-lon a-lon) (accum 1 id * a-lon))
  ```
- subt-lon requires a little algebraic manipulation
- (- a b c) = (* -1 (+ (* -1 a) b c))
- A match-expression may be used to eliminate the use of first and rest
- ```
  ;; lon → number throws error
  ;; Purpose: To subtract the given lon
  (define (subt-lon a-lon)
    (match a-lon
      ['() (error "No numbers provided to -.")]
      [(cons fnum rnums)
       (* -1 (accum 0 id + (cons (* -1 fnum) rnums)))]))
  ```
  ```

For-Loops and Pattern Matching

Pattern Matching

- ```
  ;; lon → number  Purpose: To sum the given lon
  (define (sum-lon a-lon) (accum 0 id + a-lon))
  ;; lon → number throws error  Purpose: To subtract the given lon
  (define (subt-lon a-lon)
    (match a-lon
      ['() (error "No numbers provided to -.")]
      [(cons fnum rnums)
       (* -1 (accum 0 id + (cons (* -1 fnum) rnums)))]))
  ```
- `sum-lon` and `subt-lon` have an expression to sum a list of numbers
  ```

For-Loops and Pattern Matching

Pattern Matching

- ```
  ;; lon → number  Purpose: To sum the given lon
  (define (sum-lon a-lon) (accum 0 id + a-lon))
  ;; lon → number throws error  Purpose: To subtract the given lon
  (define (subt-lon a-lon)
    (match a-lon
      ['() (error "No numbers provided to -.")]
      [(cons fnum rnums)
       (* -1 (accum 0 id + (cons (* -1 fnum) rnums)))]))
  ```
- sum-lon and subt-lon have an expression to sum a list of numbers
- To eliminate mostly repeated code we abstract to create a new abstract sum-lon:

  ```
  ;; X → X              Purpose: Return the given input
  (define (id x) x)
  ;; (listof number) → number   Purpose: Sum given lon
  (define (abs-sum-lon a-lon) (accum 0 id + a-lon)) Typo in textbook
  ;; lon → number       Purpose: To sum the given lon
  (define (sum-lon a-lon) (sum-lon a-lon))
  ;; lon → number throws error  Purpose: To subtract the given lon
  (define (subt-lon a-lon)
    (match a-lon
      ['() (error "No numbers provided to -.")]
      [(cons fnum rnums) (* -1 (sum-lon (cons (* -1 fnum) rnums)))
  ```
  ```

For-Loops and Pattern Matching

Pattern Matching

- Bodies of `subt-lon` and `mult-lon` are easily inlined into the body of `apply-f`

For-Loops and Pattern Matching

Pattern Matching

- Bodies of `subt-lon` and `mult-lon` are easily inlined into the body of `apply-f`
-
```
;; sexpr → number throws error  Purpose: To evaluate the given sexpr
(define (eval-sexpr a-sexpr)
  (local
   [;; <X Y Z> Z (X → Y) (Y Z → Z) (listof X) → Z Purpose: Summarize given list
    (define (accum base-val ffirst comb a-lox)
      (match a-lox
        ['() base-val]
        [(cons x xs) (comb (ffirst x) (accum base-val ffirst comb xs))]))
    ;; function (listof number) → number throws error Purpose: Apply funct to nums
    (define (apply-f a-function a-lon)
      (local [;; X → X     Purpose: Return the given input
              (define (id x) x)
              ;; (listof number) → number Purpose: Sum given lon
              (define (sum-lon a-lon) (accum 0 id + a-lon))]
        (cond [(eq? a-function '+) (sum-lon a-lon)]
              [(eq? a-function '-)
               (match a-lon
                 ['() (error "No numbers provided to -.")]
                 [(cons fnum rnums) (* -1 (sum-lon (cons (* -1 fnum) rnums)))])]
              [else (accum 1 id * a-lon)])))
    ;; sexpr → number throws error  Purpose: Evaluate the given sexpr
    (define (eval-sexpr a-sexpr)
      (match a-sexpr
        [(cons op args)
          (apply-f op (accum '()  (λ (a-sexpr) (eval-sexpr a-sexpr)) cons  args))]
        [else a-sexpr]))]
   (eval-sexpr a-sexpr)))
```

For-Loops and Pattern Matching

Pattern Matching

```
;; (listof sexpr) → (listof number) throws error
;; Purpose: To evaluate the sexprs in the given list
(define (eval-args a-losexpr)
 (if (empty? a-losexpr)     List-summarizing operation
     '()
     (cons (eval-sexpr (first a-losexpr))
           (eval-args (rest a-losexpr)))))

;; slist → number throws error  Purpose: Evaluate given slist
(define (eval-slist sl)
  (apply-f (first sl) (eval-args (rest sl))))
```

For-Loops and Pattern Matching

Pattern Matching

- ```
  ;; (listof sexpr) → (listof number) throws error
  ;; Purpose: To evaluate the sexprs in the given list
  (define (eval-args a-losexpr)
   (if (empty? a-losexpr)     List-summarizing operation
       '()
       (cons (eval-sexpr (first a-losexpr))
             (eval-args (rest a-losexpr)))))

  ;; slist → number throws error  Purpose: Evaluate given slist
  (define (eval-slist sl)
    (apply-f (first sl) (eval-args (rest sl))))
  ```
- ```
  ;; (listof sexpr) → (listof number) throws error
  ;; Purpose: To evaluate the sexprs in the given list
  (define (eval-args a-losexpr)
    (accum '()
           (λ (a-sexpr) (eval-sexpr a-sexpr)) cons args))
  ```

For-Loops and Pattern Matching

Pattern Matching

- ```
  ;; (listof sexpr) → (listof number) throws error
  ;; Purpose: To evaluate the sexprs in the given list
  (define (eval-args a-losexpr)
   (if (empty? a-losexpr)     List-summarizing operation
       '()
       (cons (eval-sexpr (first a-losexpr))
             (eval-args (rest a-losexpr)))))

  ;; slist → number throws error  Purpose: Evaluate given slist
  (define (eval-slist sl)
    (apply-f (first sl) (eval-args (rest sl))))
  ```
- ```
  ;; (listof sexpr) → (listof number) throws error
  ;; Purpose: To evaluate the sexprs in the given list
  (define (eval-args a-losexpr)
    (accum '()
           (λ (a-sexpr) (eval-sexpr a-sexpr)) cons args))
  ```
- Is easily inlined:

  ```
  ;; slist → number throws error  Purpose: Evaluate given slist
  (define (eval-slist sl)
    (apply-f
      (first sl)
      (accum '() (λ (a-sexpr) (eval-sexpr a-sexpr)) cons (rest sl))))
  ```

For-Loops and Pattern Matching

Pattern Matching

```
;; sexpr → number throws error  Purpose: To evaluate the given sexpr
(define (eval-sexpr a-sexpr)
  (local
    [;; slist → number throws error  Purpose: Evaluate given slist
     (define (eval-slist sl)
       (apply-f (first sl)
                (accum '() (λ (a-sexpr) (eval-sexpr a-sexpr)) cons (rest sl))))
     ;; <X Y Z> Z (X → Y) (Y Z → Z) (listof X) → Z Purpose: Summarize given list
     (define (accum base-val ffirst comb a-lox)
       (match a-lox
         ['() base-val]
         [(cons x xs) (comb (ffirst x) (accum base-val ffirst comb xs))]))
     ;; function (listof number) → number throws error Purpose: Apply funct to nums
     (define (apply-f a-function a-lon)
       (local [;; X → X Purpose: Return the given input
               (define (id x) x)
               ;; (listof number) → number Purpose: Sum given lon
               (define (sum-lon a-lon) (accum 0 id + a-lon))]
         (cond [(eq? a-function '+) (sum-lon a-lon)]
               [(eq? a-function '-)
                (match a-lon
                  ['() (error "No numbers provided to -.")]
                  [(cons fnum rnums) (* -1 (sum-lon (cons (* -1 fnum) rnums)))])]
               [else (accum 1 id * a-lon)])))
     ;; sexpr → number throws error  Purpose: Evaluate the given sexpr
     (define (eval-sexpr a-sexpr)
       (match a-sexpr
         [(cons op args)
          (apply-f op (accum '()  (λ (a-sexpr) (eval-sexpr a-sexpr)) cons  args))]
         [else a-sexpr]))]
    (eval-sexpr a-sexpr)))
```

For-Loops and Pattern Matching

Pattern Matching

- Consider a function that consumes a list of numbers and that returns a list of the even numbers in the given list doubled
- How can this problem be solved?

For-Loops and Pattern Matching

Pattern Matching

- Consider a function that consumes a list of numbers and that returns a list of the even numbers in the given list doubled
- How can this problem be solved?
- Reason about the varieties of `(listof number)`

For-Loops and Pattern Matching

Pattern Matching

- Consider a function that consumes a list of numbers and that returns a list of the even numbers in the given list doubled
- How can this problem be solved?
- Reason about the varieties of `(listof number)`
- If the given list is empty then the result is the empty

For-Loops and Pattern Matching

Pattern Matching

- Consider a function that consumes a list of numbers and that returns a list of the even numbers in the given list doubled
- How can this problem be solved?
- Reason about the varieties of `(listof number)`
- If the given list is empty then the result is the empty
- If the given list is not empty then distinguish between two subtypes: those that start with an even number and those that start with an odd number

For-Loops and Pattern Matching

Pattern Matching

- Consider a function that consumes a list of numbers and that returns a list of the even numbers in the given list doubled
- How can this problem be solved?
- Reason about the varieties of `(listof number)`
- If the given list is empty then the result is the empty
- If the given list is not empty then distinguish between two subtypes: those that start with an even number and those that start with an odd number
- ```
  ;; Data Definition
  ;; A (listof number) (lon) is either:
  ;;  1. '()
  ;;  2. (cons even-number lon)
  ;;  3. (cons odd-number  lon)
  ```
  ```

For-Loops and Pattern Matching

Pattern Matching

- Consider a function that consumes a list of numbers and that returns a list of the even numbers in the given list doubled
- How can this problem be solved?
- Reason about the varieties of `(listof number)`
- If the given list is empty then the result is the empty
- If the given list is not empty then distinguish between two subtypes: those that start with an even number and those that start with an odd number
- ```
  ;; Data Definition
  ;; A (listof number) (lon) is either:
  ;;  1. '()
  ;;  2. (cons even-number lon)
  ;;  3. (cons odd-number  lon)
  ```
- If the given list starts with an even number the double of the first number is consed to the result of recursively processing the rest of the given list
- If the given list starts with an odd number the answer is obtained by recursively processing the rest of the given list
  ```

For-Loops and Pattern Matching

Pattern Matching

- ```
  ;; Sample instances of lon
  (define ELON '())  (define LON1 '(0 1 2 3 4 5))
  (define LON2 '(7 8 9))
  ```
  ```

For-Loops and Pattern Matching

Pattern Matching

- ```
  ;; Sample instances of lon
  (define ELON '())  (define LON1 '(0 1 2 3 4 5))
  (define LON2 '(7 8 9))
  ```

- ```
  ;; Sample expressions for extract-double-evens
  (define ELON-VAL '())
  (define LON1-VAL (cons (* 2 (first LON1))
                         (extract-double-evens (rest LON1))))
  (define LON2-VAL (extract-double-evens (rest LON2)))
  ```

For-Loops and Pattern Matching

Pattern Matching

- ```
  ;; Sample instances of lon
  (define ELON '())  (define LON1 '(0 1 2 3 4 5))
  (define LON2 '(7 8 9))
  ```
- ```
  ;; (listof number) → (listof number) Purpose: Double evens in list
  (define (extract-double-evens a-lon)
  ```

- ```
  ;; Sample expressions for extract-double-evens
  (define ELON-VAL '())
  (define LON1-VAL (cons (* 2 (first LON1))
                         (extract-double-evens (rest LON1))))
  (define LON2-VAL (extract-double-evens (rest LON2)))
  ```
  ```

For-Loops and Pattern Matching

Pattern Matching

- ```
  ;; Sample instances of lon
  (define ELON '())  (define LON1 '(0 1 2 3 4 5))
  (define LON2 '(7 8 9))
  ```
- ```
  ;; (listof number) → (listof number) Purpose: Double evens in list
  (define (extract-double-evens a-lon)
  ```

- ```
  ;; Sample expressions for extract-double-evens
  (define ELON-VAL '())
  (define LON1-VAL (cons (* 2 (first LON1))
                         (extract-double-evens (rest LON1))))
  (define LON2-VAL (extract-double-evens (rest LON2)))
  ```
- ```
  ;; Tests using sample computations for extract-double-evens
  (check-expect (extract-double-evens ELON) ELON-VAL)
  (check-expect (extract-double-evens LON1) LON1-VAL)
  (check-expect (extract-double-evens LON2) LON2-VAL)
  ;; Tests using sample values for extract-double-evens
  (check-expect (extract-double-evens '(-3 -7 -11)) '())
  (check-expect (extract-double-evens '(88 10 120)) '(176 20 240))
  (check-expect (extract-double-evens '(1 2 4 8)) '(4 8 16))
  ```

For-Loops and Pattern Matching

Pattern Matching

```
;; Sample instances of lon
(define ELON '())  (define LON1 '(0 1 2 3 4 5))
(define LON2 '(7 8 9))
```

```
;; (listof number) → (listof number) Purpose: Double evens in list
(define (extract-double-evens a-lon)
```

```
  (match a-lon
    ['() '()]
    [(cons (? even?) rlon)
     (cons (* 2 (first a-lon)) (extract-double-evens rlon))]
    [(cons flon rlon) (extract-double-evens rlon)]))
```

```
;; Sample expressions for extract-double-evens
(define ELON-VAL '())
(define LON1-VAL (cons (* 2 (first LON1))
                       (extract-double-evens (rest LON1))))
(define LON2-VAL (extract-double-evens (rest LON2)))
```

```
;; Tests using sample computations for extract-double-evens
(check-expect (extract-double-evens ELON) ELON-VAL)
(check-expect (extract-double-evens LON1) LON1-VAL)
(check-expect (extract-double-evens LON2) LON2-VAL)
;; Tests using sample values for extract-double-evens
(check-expect (extract-double-evens '(-3 -7 -11)) '())
(check-expect (extract-double-evens '(88 10 120)) '(176 20 240))
(check-expect (extract-double-evens '(1 2 4 8)) '(4 8 16))
```

For-Loops and Pattern Matching

Homework

- Problems: 311–315

Interfaces and Objects

- Functions are data
- Is data a function?

Interfaces and Objects

- Functions are data
- Is data a function?
- Intuitively, perhaps, the immediate answer that comes to mind is an unequivocal no
- A `posn`, for example, is not a function
- Indeed, it may not be intuitive to think of data as a function

Interfaces and Objects

- Functions are data
- Is data a function?
- Intuitively, perhaps, the immediate answer that comes to mind is an unequivocal no
- A `posn`, for example, is not a function
- Indeed, it may not be intuitive to think of data as a function
- This lack of intuition is more than anything else a product of training.
- Has it ever been suggested in a high school Mathematics textbook that a number is a function?

Interfaces and Objects

- Functions are data
- Is data a function?
- Intuitively, perhaps, the immediate answer that comes to mind is an unequivocal no
- A `posn`, for example, is not a function
- Indeed, it may not be intuitive to think of data as a function
- This lack of intuition is more than anything else a product of training.
- Has it ever been suggested in a high school Mathematics textbook that a number is a function?
- It turns out that data can be a function
- That is, data may be represented using a function

Interfaces and Objects

- Functions are data
- Is data a function?
- Intuitively, perhaps, the immediate answer that comes to mind is an unequivocal no
- A `posn`, for example, is not a function
- Indeed, it may not be intuitive to think of data as a function
- This lack of intuition is more than anything else a product of training.
- Has it ever been suggested in a high school Mathematics textbook that a number is a function?
- It turns out that data can be a function
- That is, data may be represented using a function
- Traditionally, data is thought of separately from functions
- This occurs despite the fact that whenever we create a data definition we have in mind functions that are valid on the defined data
- An `alien` is expectedto move right, left, or down
- It becomes clear that an instance of a type is information and the functions that are valid on it

Interfaces and Objects

- Functions are data
- Is data a function?
- Intuitively, perhaps, the immediate answer that comes to mind is an unequivocal no
- A `posn`, for example, is not a function
- Indeed, it may not be intuitive to think of data as a function
- This lack of intuition is more than anything else a product of training.
- Has it ever been suggested in a high school Mathematics textbook that a number is a function?
- It turns out that data can be a function
- That is, data may be represented using a function
- Traditionally, data is thought of separately from functions
- This occurs despite the fact that whenever we create a data definition we have in mind functions that are valid on the defined data
- An `alien` is expectedto move right, left, or down
- It becomes clear that an instance of a type is information and the functions that are valid on it
- Why should anyone strictly think of a data definition and the operations on the defined type as two separate entities?

Interfaces and Objects

- Functions are data
- Is data a function?
- Intuitively, perhaps, the immediate answer that comes to mind is an unequivocal no
- A `posn`, for example, is not a function
- Indeed, it may not be intuitive to think of data as a function
- This lack of intuition is more than anything else a product of training.
- Has it ever been suggested in a high school Mathematics textbook that a number is a function?
- It turns out that data can be a function
- That is, data may be represented using a function
- Traditionally, data is thought of separately from functions
- This occurs despite the fact that whenever we create a data definition we have in mind functions that are valid on the defined data
- An `alien` is expectedto move right, left, or down
- It becomes clear that an instance of a type is information and the functions that are valid on it
- Why should anyone strictly think of a data definition and the operations on the defined type as two separate entities?
- If valid type operations are encapsulated then there is no technical impediment to also encapsulating the values that define an instance of a type
- To do so we must first learn how to define the operations valid on a type

Interfaces and Objects

Interfaces

- A data definition defines a type
- In the case of compound data it also defines the type of the components
- States nothing about the valid type operations

Interfaces and Objects

Interfaces

- A data definition defines a type
- In the case of compound data it also defines the type of the components
- States nothing about the valid type operations
- An *interface* defines the behavior of a defined type
- Specifies the operations that are valid on a type

Interfaces and Objects

Interfaces

- Consider computing the distance to the origin for a 3Dposn:

```
;; A 3Dposn is a structure: (make-3Dposn number number number)
(define-struct 3Dposn (x y z))

;; 3Dposn → number Purpose: Return the distance to the origin
(define (dist-origin a-3dposn)
  (sqrt (+ (sqr (3Dposn-x a-3dposn))
           (sqr (3Dposn-y a-3dposn))
           (sqr (3Dposn-z a-3dposn)))))
```

Interfaces and Objects

Interfaces

- Consider computing the distance to the origin for a `3Dposn`:

```
;; A 3Dposn is a structure: (make-3Dposn number number number)
(define-struct 3Dposn (x y z))

;; 3Dposn → number Purpose: Return the distance to the origin
(define (dist-origin a-3dposn)
  (sqrt (+ (sqr (3Dposn-x a-3dposn))
           (sqr (3Dposn-y a-3dposn))
           (sqr (3Dposn-z a-3dposn)))))
```

- Unstated assumptions
  - There is a function to construct a `3Dposn`
  - There are selector functions
  - The values of `x`, `y`, `z`, and the selectors are stored separately from the function `dist-origin`
- Would all this had been clear to you before reading the preceding chapters?
- Would you have known that there is a function `3Dposn-x` to extract the `x` value of an instance of a `3Dposn`?

Interfaces and Objects

Interfaces

- In addition, an interface defines the expected behavior of a type
- Outlines the valid operations and the returned type

Interfaces and Objects

Interfaces

- In addition, an interface defines the expected behavior of a type
- Outlines the valid operations and the returned type
- Request `x`: `number`
  Request `y`: `number`
  Request `z`: `number`
  Request `distance`: `number`
- Makes it clear to any reader which are the valid operations on a `3Dposn`
- Says nothing about how a data type and its valid operations are implemented

Interfaces and Objects

Interfaces

- In addition, an interface defines the expected behavior of a type
- Outlines the valid operations and the returned type
- ```
  Request x: number
  Request y: number
  Request z: number
  Request distance: number
  ```
- Makes it clear to any reader which are the valid operations on a `3Dposn`
- Says nothing about how a data type and its valid operations are implemented
- Relate `x`, `y`, `z`, `3Dposn-x`, `3Dposn-y`, `3Dposn-z`, and `dist-origin`
- Ought to be able to encapsulate them into a single package
- Whenever a `3Dposn` is constructed the package returned ought to be able to perform all the operations in the interface
  ```

Interfaces and Objects

Interfaces

- In addition, an interface defines the expected behavior of a type
- Outlines the valid operations and the returned type
- ```
  Request x: number
  Request y: number
  Request z: number
  Request distance: number
  ```
- Makes it clear to any reader which are the valid operations on a `3Dposn`
- Says nothing about how a data type and its valid operations are implemented
- Relate `x`, `y`, `z`, `3Dposn-x`, `3Dposn-y`, `3Dposn-z`, and `dist-origin`
- Ought to be able to encapsulate them into a single package
- Whenever a `3Dposn` is constructed the package returned ought to be able to perform all the operations in the interface
- How can an `3Dposn` instance perform many operations?
- How does it know what operations to perform?
  ```

Interfaces and Objects

Interfaces

- *message-passing* is used
- An interface is implemented by a constructor function that returns a message-processing function
- This is a curried function that receives as input a message requesting a service

Interfaces and Objects

Interfaces

- *message-passing* is used
- An interface is implemented by a constructor function that returns a message-processing function
- This is a curried function that receives as input a message requesting a service
- For example, this function may get the message `'getx` requesting the `x` value

Interfaces and Objects

Interfaces

- *message-passing* is used
- An interface is implemented by a constructor function that returns a message-processing function
- This is a curried function that receives as input a message requesting a service
- For example, this function may get the message `'getx` requesting the `x` value
- An interface must specify the messages used to request a service

Interfaces and Objects

Interfaces

- *message-passing* is used
- An interface is implemented by a constructor function that returns a message-processing function
- This is a curried function that receives as input a message requesting a service
- For example, this function may get the message `'getx` requesting the `x` value
- An interface must specify the messages used to request a service
- We can now refine the `3Dposn` interface to be:

```
'getx: number
'gety: number
'getz: number
 'd2o: number
```

- A unique message associated with each service
- The message-processing function determines what value to compute by examining the message it gets as input
- Observe that embedded in the interface definition is a data definition for a `message`

Interfaces and Objects

Interfaces

- A constructor function that implements an interface is called a *class*
- A class encapsulates the values of and the operations on a type
- Defines a constructor for instances of a type
- The value returned by a class is called an *object*
- An object is an instance of an interface and knows how to perform all the services in the interface using message-passing

Interfaces and Objects

Interfaces

- A constructor function that implements an interface is called a *class*
- A class encapsulates the values of and the operations on a type
- Defines a constructor for instances of a type
- The value returned by a class is called an *object*
- An object is an instance of an interface and knows how to perform all the services in the interface using message-passing
- The message-passing function is an object
- It knows how to compute all the services in the interface
- The `local`-expression returns it
- Testing interfaces requires defining one or more objects and writing tests to check services are correctly provided

Interfaces and Objects

Interfaces

- `(require 2htdp/abstraction)`

```
;; number number number → 3Dposn  Purpose: Return a 3Dposn object
(define (make-3Dposn x y z)
```

Interfaces and Objects

Interfaces

- ```
  (require 2htdp/abstraction)

  ;; number number number → 3Dposn  Purpose: Return a 3Dposn object
  (define (make-3Dposn x y z)
  ```

- ```
  ;; Sample 3Dposn objects
  (define ORIGIN  (make-3Dposn 0 0 0))   (define A3DPOSN (make-3Dposn 2 3 5))
  ```

Interfaces and Objects

Interfaces

- ```
  (require 2htdp/abstraction)

  ;; number number number → 3Dposn  Purpose: Return a 3Dposn object
  (define (make-3Dposn x y z)
  ```
- ```
    (local [;; 3Dposn → number Purpose: Return distance to origin
            (define (dist-origin x y z)
              (sqrt (+ (sqr x) (sqr y) (sqr z))))

            ;; message → 3Dposn service throws error
            ;; Purpose: To manage messages for a 3Dposn
            (define (manager m)
              (match m
                ['getx x]
                ['gety y]
                ['getz z]
                ['d2o  (dist-origin x y z)]
                [else (error (string-append "Unknown message to 3Dposn: "
                                            (symbol->string m)))]))]
      manager))
  ```
- ```
  ;; Sample 3Dposn objects
  (define ORIGIN  (make-3Dposn 0 0 0))   (define A3DPOSN (make-3Dposn 2 3 5))
  ```
  ```

Interfaces and Objects

Interfaces

- ```
  (require 2htdp/abstraction)

  ;; number number number → 3Dposn  Purpose: Return a 3Dposn object
  (define (make-3Dposn x y z)
  ```
- ```
    (local [;; 3Dposn → number Purpose: Return distance to origin
            (define (dist-origin x y z)
              (sqrt (+ (sqr x) (sqr y) (sqr z))))

            ;; message → 3Dposn service throws error
            ;; Purpose: To manage messages for a 3Dposn
            (define (manager m)
              (match m
                ['getx x]
                ['gety y]
                ['getz z]
                ['d2o  (dist-origin x y z)]
                [else (error (string-append "Unknown message to 3Dposn: "
                                            (symbol->string m)))]))]
      manager))
  ```
- ```
  ;; Sample 3Dposn objects
  (define ORIGIN  (make-3Dposn 0 0 0))   (define A3DPOSN (make-3Dposn 2 3 5))
  ```
- ```
  ;; Tests using sample computations for 3Dposn
  (check-within (ORIGIN  'getx) 0    0.01)
  (check-within (A3DPOSN 'gety) 3    0.01)
  (check-within (ORIGIN  'getz) 0    0.01)
  (check-within (A3DPOSN 'd2o)  6.16 0.01)

  ;; Tests using sample values for 3Dposn
  (check-within ((make-3Dposn 10 20 30) 'd2o) 37.42 0.01)
  (check-error (A3DPOSN 'move-r) "Unknown message to 3Dposn: move-r")
  ```

Interfaces and Objects

Interfaces

- A `3Dposn` is a function!
- An instance of the curried function `manager` (i.e., an object) that is specialized for the values of `x`, `y`, `z`
- Functions are data and data are functions

Interfaces and Objects

Improving the Human Interface

- Message-passing may reduces readability of the code
- Does `(A3DPOSN 'd20)` communicate to others that this expression represents `A3DPOSN`'s distance to the origin?

Interfaces and Objects

Improving the Human Interface

- Message-passing may reduces readability of the code
- Does `(A3DPOSN 'd20)` communicate to others that this expression represents `A3DPOSN`'s distance to the origin?
- Unless you are intimately familiar with the message-passing protocol it is likely that this expression is meaningless
- It is unlikely that any programmer (including yourself) will permanently remember the message-passing protocol making it unnecessarily more difficult to refine the program.

Interfaces and Objects

Improving the Human Interface

- Message-passing may reduces readability of the code
- Does `(A3DPOSN 'd20)` communicate to others that this expression represents `A3DPOSN`'s distance to the origin?
- Unless you are intimately familiar with the message-passing protocol it is likely that this expression is meaningless
- It is unlikely that any programmer (including yourself) will permanently remember the message-passing protocol making it unnecessarily more difficult to refine the program.
- To mitigate this problem `wrapper functions` are written
- A wrapper function hides the details of the implementation: message-passing
- Allow programmers to use `3Dposns` without forcing them to know how they are implemented

Interfaces and Objects

Improving the Human Interface

- Message-passing may reduces readability of the code
- Does `(A3DPOSN 'd20)` communicate to others that this expression represents `A3DPOSN`'s distance to the origin?
- Unless you are intimately familiar with the message-passing protocol it is likely that this expression is meaningless
- It is unlikely that any programmer (including yourself) will permanently remember the message-passing protocol making it unnecessarily more difficult to refine the program.
- To mitigate this problem `wrapper functions` are written
- A wrapper function hides the details of the implementation: message-passing
- Allow programmers to use `3Dposns` without forcing them to know how they are implemented
- A wrapper function is needed for each service in the interface
- It takes as input an object (and any additional inputs if any)
- Its body applies the interface to the appropriate message

Interfaces and Objects

Improving the Human Interface

- ```
  ;; 3Dposn → number  Purpose: Return the x of the given 3Dposn
  (define (3Dposn-x a-3dposn) (a-3dposn 'getx))
  ;; Sample expressions for 3Dposn-x
  (define ORIGINX  (ORIGIN  'getx))    (define A3DPOSNX (A3DPOSN 'getx))
  ;; Tests using sample computations and values for 3Dposn-x
  (check-within (3Dposn-x ORIGIN)  ORIGINX  0.01)
  (check-within (3Dposn-x A3DPOSN) A3DPOSNX 0.01)
  (check-within (3Dposn-x (make-3Dposn 10 20 30)) 10 0.01)
  ```
  ```

Interfaces and Objects

Improving the Human Interface

- ```
  ;; 3Dposn → number  Purpose: Return the x of the given 3Dposn
  (define (3Dposn-x a-3dposn) (a-3dposn 'getx))
  ;; Sample expressions for 3Dposn-x
  (define ORIGINX  (ORIGIN  'getx))    (define A3DPOSNX (A3DPOSN 'getx))
  ;; Tests using sample computations and values for 3Dposn-x
  (check-within (3Dposn-x ORIGIN)  ORIGINX  0.01)
  (check-within (3Dposn-x A3DPOSN) A3DPOSNX 0.01)
  (check-within (3Dposn-x (make-3Dposn 10 20 30)) 10 0.01)
  ```
- ```
  ;; 3Dposn → number  Purpose: Return the y of the given 3Dposn
  (define (3Dposn-y a-3dposn) (a-3dposn 'gety))
  ;; Sample expressions for 3Dposn-y
  (define ORIGINY  (ORIGIN  'gety))    (define A3DPOSNY (A3DPOSN 'gety))
  ;; Tests using sample computations and values for 3Dposn-y
  (check-within (3Dposn-y ORIGIN)  ORIGINY  0.01)
  (check-within (3Dposn-y A3DPOSN) A3DPOSNY 0.01)
  (check-within (3Dposn-y (make-3Dposn 10 20 30)) 20 0.01)
  ```

Interfaces and Objects

Improving the Human Interface

- ```
  ;; 3Dposn → number  Purpose: Return the x of the given 3Dposn
  (define (3Dposn-x a-3dposn) (a-3dposn 'getx))
  ;; Sample expressions for 3Dposn-x
  (define ORIGINX  (ORIGIN  'getx))    (define A3DPOSNX (A3DPOSN 'getx))
  ;; Tests using sample computations and values for 3Dposn-x
  (check-within (3Dposn-x ORIGIN)  ORIGINX  0.01)
  (check-within (3Dposn-x A3DPOSN) A3DPOSNX 0.01)
  (check-within (3Dposn-x (make-3Dposn 10 20 30)) 10 0.01)
  ```
- ```
  ;; 3Dposn → number  Purpose: Return the y of the given 3Dposn
  (define (3Dposn-y a-3dposn) (a-3dposn 'gety))
  ;; Sample expressions for 3Dposn-y
  (define ORIGINY  (ORIGIN  'gety))    (define A3DPOSNY (A3DPOSN 'gety))
  ;; Tests using sample computations and values for 3Dposn-y
  (check-within (3Dposn-y ORIGIN)  ORIGINY  0.01)
  (check-within (3Dposn-y A3DPOSN) A3DPOSNY 0.01)
  (check-within (3Dposn-y (make-3Dposn 10 20 30)) 20 0.01)
  ```
- ```
  ;; 3Dposn → number  Purpose: Return the z of the given 3Dposn
  (define (3Dposn-z a-3dposn) (a-3dposn 'getz))
  ;; Sample expressions for 3Dposn-z
  (define ORIGINZ  (ORIGIN  'getz))    (define A3DPOSNZ (A3DPOSN 'getz))
  ;; Tests using sample computations and values for 3Dposn-z
  (check-within (3Dposn-z ORIGIN)  ORIGINZ  0.01)
  (check-within (3Dposn-z A3DPOSN) A3DPOSNZ 0.01)
  (check-within (3Dposn-z (make-3Dposn 10 20 30)) 30 0.01)
  ```
  ```

Interfaces and Objects

Improving the Human Interface

- ```
  ;; 3Dposn → number  Purpose: Return the x of the given 3Dposn
  (define (3Dposn-x a-3dposn) (a-3dposn 'getx))
  ;; Sample expressions for 3Dposn-x
  (define ORIGINX  (ORIGIN  'getx))    (define A3DPOSNX (A3DPOSN 'getx))
  ;; Tests using sample computations and values for 3Dposn-x
  (check-within (3Dposn-x ORIGIN)  ORIGINX  0.01)
  (check-within (3Dposn-x A3DPOSN) A3DPOSNX 0.01)
  (check-within (3Dposn-x (make-3Dposn 10 20 30)) 10 0.01)
  ```
- ```
  ;; 3Dposn → number  Purpose: Return the y of the given 3Dposn
  (define (3Dposn-y a-3dposn) (a-3dposn 'gety))
  ;; Sample expressions for 3Dposn-y
  (define ORIGINY  (ORIGIN  'gety))    (define A3DPOSNY (A3DPOSN 'gety))
  ;; Tests using sample computations and values for 3Dposn-y
  (check-within (3Dposn-y ORIGIN)  ORIGINY  0.01)
  (check-within (3Dposn-y A3DPOSN) A3DPOSNY 0.01)
  (check-within (3Dposn-y (make-3Dposn 10 20 30)) 20 0.01)
  ```
- ```
  ;; 3Dposn → number  Purpose: Return the z of the given 3Dposn
  (define (3Dposn-z a-3dposn) (a-3dposn 'getz))
  ;; Sample expressions for 3Dposn-z
  (define ORIGINZ  (ORIGIN  'getz))    (define A3DPOSNZ (A3DPOSN 'getz))
  ;; Tests using sample computations and values for 3Dposn-z
  (check-within (3Dposn-z ORIGIN)  ORIGINZ  0.01)
  (check-within (3Dposn-z A3DPOSN) A3DPOSNZ 0.01)
  (check-within (3Dposn-z (make-3Dposn 10 20 30)) 30 0.01)
  ```
- ```
  ;; 3Dposn → number  Purpose: Return distance to origin of given 3Dposn
  (define (dist-origin a-3dposn) (a-3dposn 'd20))
  ;; Sample expressions for dist-origin
  (define ORIGIND  (ORIGIN  'd20))    (define A3DPOSND (A3DPOSN 'd20))
  ;; Tests using sample computations and values for dist-origin
  (check-within (dist-origin ORIGIN)  ORIGIND  0.01)
  (check-within (dist-origin A3DPOSN) A3DPOSND 0.01)
  (check-within (dist-origin (make-3Dposn 10 20 30)) 37.42 0.01)
  ```

Interfaces and Objects

Services that Require More Input

- It may be necessary to add more services
- This means expanding the message-processing function
- If a value may be computed using only the information stored in an object then adding a service is no different that what was done for `dist-origin`
- If a service requires further input the answer cannot be computed using only the values stored in an object

Interfaces and Objects

Services that Require More Input

- It may be necessary to add more services
- This means expanding the message-processing function
- If a value may be computed using only the information stored in an object then adding a service is no different that what was done for `dist-origin`
- If a service requires further input the answer cannot be computed using only the values stored in an object
- The object providing the service (usually referred to as `this`) needs information beyond that which it stores

Interfaces and Objects

Services that Require More Input

- It may be necessary to add more services
- This means expanding the message-processing function
- If a value may be computed using only the information stored in an object then adding a service is no different that what was done for `dist-origin`
- If a service requires further input the answer cannot be computed using only the values stored in an object
- The object providing the service (usually referred to as `this`) needs information beyond that which it stores
- Consider adding a service that computes the distance to a given `3Dposn` object
- In addition to `this` object another `3Dposn` object is needed
- This other object is unknown when the interface is implemented and, therefore, cannot be provided as input (similar to receiving a message)
- The solution for messages is to make a curried function that consumes the extra input (i.e., a message)
- The same design tactic may be employed to add services that require extra input
- The interface must return a function that consumes the extra input

Interfaces and Objects

Services that Require More Input

- Add a service to compute the distance of `this 3Dposn` to a given `3Dposn`

Interfaces and Objects

Services that Require More Input

- Add a service to compute the distance of `this 3Dposn` to a given `3Dposn`
- The first step is to update the interface as follows:

```
'getx number
'gety number
'getz number
 'd2o number
 'd2p 3Dposn → number
```

- The data definition of a message is expanded to include `'d2p` for the new distance-computing service
- Given that extra input is needed the interface returns a function that consumes the extra input and that returns a number

Interfaces and Objects

Services that Require More Input

- The next step is to refine the `manager`:

```
;; message → 3Dposn service throws error
;; Purpose: To manage messages for a 3Dposn
(define (manager m)
  (match m
    ['getx x]
    ['gety y]
    ['getz z]
    ['d2o  (dist-origin x y z)]
    ['d2p  distance]
    [else (error (string-append "Unknown message to 3Dposn: "
                                (symbol->string m)))]))
```

- The new stanza matches the new message and returns the `distance` function

Interfaces and Objects

Services that Require More Input

- The next step is to refine the `manager`:

```
;; message → 3Dposn service throws error
;; Purpose: To manage messages for a 3Dposn
(define (manager m)
  (match m
    ['getx x]
    ['gety y]
    ['getz z]
    ['d2o  (dist-origin x y z)]
    ['d2p  distance]
    [else (error (string-append "Unknown message to 3Dposn: "
                                (symbol->string m)))]))
```

- The new stanza matches the new message and returns the `distance` function
- New local function:

```
;; 3Dpson arrow number
;; Purpose: Compute the distance from this to the given 3Dposn
(define (distance a-3dposn)
  (sqrt (+ (sqr (- x (3Dposn-x a-3dposn)))
           (sqr (- y (3Dposn-y a-3dposn)))
           (sqr (- z (3Dposn-z a-3dposn))))))
```

Services that Require More Input

Interfaces

- The final step is to develop a wrapper function for the new service:

```
(define B3DPOSN (make-3Dposn 1 1 1))

;; 3Dposn 3Dposn → number
;; Purpose: Return the distance between the given 3Dposns
(define (3Dposn-distance p1 p2) ((p1 'd2p) p2))

;; Sample expressions for 3Dposn-distance
(define ORIGINDP  ((ORIGIN  'd2p) ORIGIN))
(define A3DPOSNDP ((A3DPOSN 'd2p) B3DPOSN))

;; Tests using sample computations 3Dposn-distance
(check-within (3Dposn-distance ORIGIN ORIGIN) ORIGINDP 0.01)
(check-within (3Dposn-distance A3DPOSN B3DPOSN) A3DPOSNDP 0.01)

;; Tests using sample values for 3Dposn-distance
(check-within (3Dposn-distance (make-3Dposn 10 20 30)
                               (make-3Dposn 2 3 4))
              32.07
              0.01)
```

Interfaces and Objects

A Design Recipe for Interfaces

- 1. Identify the values that must be stored and the services that must be provided.
2. Develop an interface data definition and a data definition for messages.
3. Develop a function template for the class that consumes the values that must be stored and whose body is a `local`-expression returning the message-processing function.
4. Specialize the signature, purpose, class header and message-processing function.
5. Write and make local the auxiliary functions needed by the message-passing function.
6. Write and test a wrapper function for each service.

Interfaces and Objects

Homework

- Problems: 316–318, 320

Interfaces and Objects

Interfaces and Union Types

- Designing interfaces for data with variety requires individually reasoning about each variety
- Reasoning about each variety is how functions to process a union type are designed

Interfaces and Objects

Interfaces and Union Types

- Designing interfaces for data with variety requires individually reasoning about each variety
- Reasoning about each variety is how functions to process a union type are designed
- For each subtype there must be a class that encapsulates the code for that subtype
- Each subtype must offer the same services: all have a common interface
- Each class only implements the services for the subtype it is written for

Interfaces and Objects

Interfaces and Union Types

- Consider implementing an abbreviated interface for a `(listof X)`

Interfaces and Objects

Interfaces and Union Types

- Consider implementing an abbreviated interface for a `(listof X)`
- Include:
  - `empty?`
  - `first`
  - `rest`
  - `cons`
  - `map`
  - Transform into an `ISL+` list

Interfaces and Objects

Interfaces and Union Types

- Two classes are required: one for the empty `(listof X)` and one for the nonempty `(listof X)`.

Interfaces and Objects

Interfaces and Union Types

- Two classes are required: one for the empty `(listof X)` and one for the nonempty `(listof X)`.
- Empty `(listof X)` needs to store no values

Interfaces and Objects

Interfaces and Union Types

- Two classes are required: one for the empty `(listof X)` and one for the nonempty `(listof X)`.
- Empty `(listof X)` needs to store no values
- Nonempty `(listof X)` needs to store two values: an `X` and a `(listof X)`

Interfaces and Objects

Interfaces and Union Types

- Two classes are required: one for the empty `(listof X)` and one for the nonempty `(listof X)`.
- Empty `(listof X)` needs to store no values
- Nonempty `(listof X)` needs to store two values: an `X` and a `(listof X)`
- The following services are offered by a `(listof X)`:
  - Determine if the list is the empty list
  - Access the first element of the list
  - Access the rest of the list
  - Add a new element to the front of `this` list
  - Apply a function to every element of the list and return a list of the results.
  - Transform `this` list into an ISL+ list.

Interfaces and Objects

Interfaces and Union Types

- 6 return types and 6 message varieties

Interfaces and Objects

Interfaces and Union Types

- 6 return types and 6 message varieties
- `empty?` and the transformation into an `ISL+` list return a value with no need for further input

Interfaces and Objects

Interfaces and Union Types

- 6 return types and 6 message varieties
- `empty?` and the transformation into an `ISL+` list return a value with no need for further input
- The services `first` and `rest` either return a value or throw an error with no need for further input

Interfaces and Objects

Interfaces and Union Types

- 6 return types and 6 message varieties
- `empty?` and the transformation into an `ISL+` list return a value with no need for further input
- The services `first` and `rest` either return a value or throw an error with no need for further input
- Adding an element to the front of the list and mapping a function require further input

Interfaces and Objects

Interfaces and Union Types

- 6 return types and 6 message varieties
- `empty?` and the transformation into an ISL+ list return a value with no need for further input
- The services `first` and `rest` either return a value or throw an error with no need for further input
- Adding an element to the front of the list and mapping a function require further input
- The interface for a `listofx` is: Different from (listof X)

```
;; A listofx is an interface offering
;;  'empty:  Boolean
;;  'first:  X throws error
;;   'rest:  listofx throws error
;;   'cons:  X → listofx
;;    'map:  (X → Y) → listofy
;;  '2Rlst:  (listof X)
```

Interfaces and Objects

Interfaces and Union Types

```
;; ... → listofx
;; Purpose: Return a ... listofx object
(define (class-for-listofx ...)
 (local
      .
      .
  [   .
   ;; X → listofx
   ;; Purpose: Add given X to the front of this list
   (define (add2front an-x) ...)
   ;; (X → Y) → (listof Y)
   ;; Purpose: Map the given function to this list
   (define (map f) ...)
   ;; message → service throws error
   ;; Purpose: Provide service for the given message
   (define (manager m)
    (match m
     ['empty? ...]
     ['first  ...]
     ['rest   ...]
     ['cons   ...]
     ['map    ...]
     ['2Rlst  ...]
     [else
      (error
       (format "Unknown list service requested: ~s" m))]))]
  manager))

;; Sample listofx objects
(define L0 ...)
(define L1 ...)

     .
     .
```

Interfaces and Objects

Interfaces and Union Types

- ```
  ;; listofx → Boolean   Purpose: Determine if listofx is empty
  (define (listofx-empty? lox-o) ...)
  ;; Sample expressions for listofx-empty?
  (define L0E ...) (define L1E ...)...
  ;; Tests using sample computations and sample values for listofx-empty?
  (check-expect (listofx-empty? L0) L0E) (check-expect (listofx-empty? L1) L1E)...
  (check-expect (listofx-empty? ...) ...)...
  ```
  ```

Interfaces and Objects

Interfaces and Union Types

- ```
  ;; listofx → Boolean   Purpose: Determine if listofx is empty
  (define (listofx-empty? lox-o) ...)
  ;; Sample expressions for listofx-empty?
  (define L0E ...) (define L1E ...)...
  ;; Tests using sample computations and sample values for listofx-empty?
  (check-expect (listofx-empty? L0) L0E) (check-expect (listofx-empty? L1) L1E)...
  (check-expect (listofx-empty? ...) ...)...
  ```
- ```
  ;; listofx → X  throws error   Purpose: Return first element of listofx
  (define (listofx-first lox-o) ...)
  ;; Sample expressions for listofx-first
  (define L1F ...)...
  ;; Tests using sample computations and sample values for listofx-first
  (check-expect (listofx-first L1) L1F)...
  (check-error  (listofx-first ...) ...) (check-expect (listofx-first ...) ...)...
  ```

Interfaces and Objects

Interfaces and Union Types

- ```
  ;; listofx → Boolean   Purpose: Determine if listofx is empty
  (define (listofx-empty? lox-o) ...)
  ;; Sample expressions for listofx-empty?
  (define L0E ...) (define L1E ...)...
  ;; Tests using sample computations and sample values for listofx-empty?
  (check-expect (listofx-empty? L0) L0E) (check-expect (listofx-empty? L1) L1E)...
  (check-expect (listofx-empty? ...) ...)...
  ```
- ```
  ;; listofx → X  throws error   Purpose: Return first element of listofx
  (define (listofx-first lox-o) ...)
  ;; Sample expressions for listofx-first
  (define L1F ...)...
  ;; Tests using sample computations and sample values for listofx-first
  (check-expect (listofx-first L1) L1F)...
  (check-error  (listofx-first ...) ...) (check-expect (listofx-first ...) ...)...
  ```
- ```
  ;; listofx → listofx throws error    Purpose: Return rest of listofx
  (define (listofx-rest lox-o) ...)
  ;; Sample expressions for listofx-rest
  (define L1R ...)
  ;; Tests using sample computations and sample values for listofx-rest
  (check-expect (listofx-rest ...)) ...)...
  (check-error  (listofx-rest L0) ...) (check-expect (listofx-rest ...) ...)...
  ```
  ```

Interfaces and Objects

Interfaces and Union Types

- ```
  ;; listofx → Boolean   Purpose: Determine if listofx is empty
  (define (listofx-empty? lox-o) ...)
  ;; Sample expressions for listofx-empty?
  (define L0E ...) (define L1E ...)...
  ;; Tests using sample computations and sample values for listofx-empty?
  (check-expect (listofx-empty? L0) L0E) (check-expect (listofx-empty? L1) L1E)...
  (check-expect (listofx-empty? ...) ...)...
  ```
- ```
  ;; listofx → X  throws error   Purpose: Return first element of listofx
  (define (listofx-first lox-o) ...)
  ;; Sample expressions for listofx-first
  (define L1F ...)...
  ;; Tests using sample computations and sample values for listofx-first
  (check-expect (listofx-first L1) L1F)...
  (check-error  (listofx-first ...) ...) (check-expect (listofx-first ...) ...)...
  ```
- ```
  ;; listofx → listofx throws error    Purpose: Return rest of listofx
  (define (listofx-rest lox-o) ...)
  ;; Sample expressions for listofx-rest
  (define L1R ...)
  ;; Tests using sample computations and sample values for listofx-rest
  (check-expect (listofx-rest ...)) ...)...
  (check-error  (listofx-rest L0) ...) (check-expect (listofx-rest ...) ...)...
  ```
- ```
  ;; <X Y> listofx (X → Y) → listofx   Purpose: Map function onto listofx
  (define (listofx-map lox-o f) ...)
  ;; Sample expressions for listofx-map
  (define L0M ...)...(define L1M ...)
  ;; Tests using sample computations and sample values for listofx-map
  (check-expect (listofx-map L0 ...) ...) (check-expect (listofx-map L1 ...) ...)...
  (check-expect (listofx-map ... ...) ...)...
  ```

Interfaces and Objects

Interfaces and Union Types

- ```
  ;; listofx → Boolean   Purpose: Determine if listofx is empty
  (define (listofx-empty? lox-o) ...)
  ;; Sample expressions for listofx-empty?
  (define L0E ...) (define L1E ...)...
  ;; Tests using sample computations and sample values for listofx-empty?
  (check-expect (listofx-empty? L0) L0E) (check-expect (listofx-empty? L1) L1E)...
  (check-expect (listofx-empty? ...) ...)...
  ```
- ```
  ;; listofx → X  throws error   Purpose: Return first element of listofx
  (define (listofx-first lox-o) ...)
  ;; Sample expressions for listofx-first
  (define L1F ...)...
  ;; Tests using sample computations and sample values for listofx-first
  (check-expect (listofx-first L1) L1F)...
  (check-error  (listofx-first ...) ...) (check-expect (listofx-first ...) ...)...
  ```
- ```
  ;; listofx → listofx throws error    Purpose: Return rest of listofx
  (define (listofx-rest lox-o) ...)
  ;; Sample expressions for listofx-rest
  (define L1R ...)
  ;; Tests using sample computations and sample values for listofx-rest
  (check-expect (listofx-rest ...)) ...)...
  (check-error  (listofx-rest L0) ...) (check-expect (listofx-rest ...) ...)...
  ```
- ```
  ;; <X Y> listofx (X → Y) → listofx   Purpose: Map function onto listofx
  (define (listofx-map lox-o f) ...)
  ;; Sample expressions for listofx-map
  (define L0M ...)...(define L1M ...)
  ;; Tests using sample computations and sample values for listofx-map
  (check-expect (listofx-map L0 ...) ...) (check-expect (listofx-map L1 ...) ...)...
  (check-expect (listofx-map ... ...) ...)...
  ```
- ```
  ;; listofx → Boolean   Purpose: Convert listofx to (listof X)
  (define (listofx-2Rlst lox-o) ...)
  ;; Sample expressions for listofx-2Rlst
  (define L0RL ...)...(define L1RL ...)
  ```
  ```

Interfaces and Objects

Empty Class

- ```
  ;; Z → listofx
  ;; Purpose: Return an empty listofx object
  (define (mtList dont-care-param)
   (local [
  ```
  ```

Interfaces and Objects

Empty Class

- ```
  ;; Z → listofx
  ;; Purpose: Return an empty listofx object
  (define (mtList dont-care-param)
   (local [
  ```

- ```
    ;; message → service throws error
    ;; Purpose: Provide service for the given message
    (define (manager m)
     (match m
       ['empty? #true]
       ['first  (error "first requested from the empty list")]
       ['rest   (error "rest requested from the empty list")]
       ['cons   add2front]
       ['map    map]
       ['2Rlst  '()]
       [else
        (error (format "Unknown list service requested: ~s" m))]))]
  manager))
  ```

Interfaces and Objects

Empty Class

```
;; Z → listofx
;; Purpose: Return an empty listofx object
(define (mtList dont-care-param)
 (local [
```

```
   ;; X → listofx
   ;; Purpose: Add given X to the front of this list
   (define (add2front an-x) (consList an-x manager))
```

```
   ;; message → service throws error
   ;; Purpose: Provide service for the given message
   (define (manager m)
    (match m
      ['empty? #true]
      ['first  (error "first requested from the empty list")]
      ['rest   (error "rest requested from the empty list")]
      ['cons   add2front]
      ['map    map]
      ['2Rlst  '()]
      [else
       (error (format "Unknown list service requested: ~s" m))]))]
 manager))
```

Interfaces and Objects

Empty Class

```
;; Z → listofx
;; Purpose: Return an empty listofx object
(define (mtList dont-care-param)
 (local [
   ;; X → listofx
   ;; Purpose: Add given X to the front of this list
   (define (add2front an-x) (consList an-x manager))
   ;; (X → Y) → (listof Y)
   ;; Purpose: Map the given function to this list
   (define (map f) (mtList 'D))
   ;; message → service throws error
   ;; Purpose: Provide service for the given message
   (define (manager m)
    (match m
      ['empty? #true]
      ['first  (error "first requested from the empty list")]
      ['rest   (error "rest requested from the empty list")]
      ['cons   add2front]
      ['map    map]
      ['2Rlst  '()]
      [else
       (error (format "Unknown list service requested: ~s" m))]))]
  manager))
```

Interfaces and Objects

Interfaces and Union Types

- ```
  ;; X listofx → listofx
  ;; Purpose: Return a nonempty listofx object
  (define (consList first rest)
   (local [
  ```
  ```

Interfaces and Objects

Interfaces and Union Types

- ```
  ;; X listofx → listofx
  ;; Purpose: Return a nonempty listofx object
  (define (consList first rest)
   (local [
  ```

- ```
    ;; message → service throws error
    ;; Purpose: Provide service for the given message
    (define (manager m)
     (match m
       ['empty? #false]
       ['first  first]
       ['rest   rest]
       ['cons   add2front]
       ['map    map]
       ['2Rlst  (cons first (rest '2Rlst))]
       [else
        (error (format "Unknown list service requested: ~s" m))]))]
  manager))
  ```

Interfaces and Objects

Interfaces and Union Types

- ```
  ;; X listofx → listofx
  ;; Purpose: Return a nonempty listofx object
  (define (consList first rest)
   (local [
  ```
- ```
    ;; X → listofx
    ;; Purpose: Add given X to the front of this list
    (define (add2front an-x) (consList an-x manager))
  ```

- ```
    ;; message → service throws error
    ;; Purpose: Provide service for the given message
    (define (manager m)
     (match m
       ['empty? #false]
       ['first  first]
       ['rest   rest]
       ['cons   add2front]
       ['map    map]
       ['2Rlst  (cons first (rest '2Rlst))]
       [else
        (error (format "Unknown list service requested: ~s" m))]))]
  manager))
  ```
  ```

Interfaces and Objects

Interfaces and Union Types

```
;; X listofx → listofx
;; Purpose: Return a nonempty listofx object
(define (consList first rest)
 (local [
   ;; X → listofx
   ;; Purpose: Add given X to the front of this list
   (define (add2front an-x) (consList an-x manager))
    ;; (X → Y) → (listof Y)
    ;; Purpose: Map the given function to this list
    (define (map f) (consList (f first) ((rest 'map) f)))
   ;; message → service throws error
   ;; Purpose: Provide service for the given message
   (define (manager m)
    (match m
      ['empty? #false]
      ['first  first]
      ['rest   rest]
      ['cons   add2front]
      ['map    map]
      ['2Rlst  (cons first (rest '2Rlst))]
      [else
       (error (format "Unknown list service requested: ~s" m))]))]
 manager))
```

Interfaces and Objects

Interfaces and Union Types

- ```
  ;; Sample listofx objects
  (define L0 (mtList 'dummyval))
  (define L1 (consList
              1
              (consList 2
                        (consList 3 (mtList 'dummyval)))))
  ```
  ```

Interfaces and Objects

Interfaces and Union Types

```
;; listofx → Boolean
;; Purpose: Determine if the given listofx is empty
(define (listofx-empty? lox-o) (lox-o 'empty?))

;; Sample expressions for listofx-empty?
(define L0E (L0 'empty?))
(define L1E (L1 'empty?))

;; Tests using sample computations for listofx-empty?
(check-expect (listofx-empty? L0) L0E)
(check-expect (listofx-empty? L1) L1E)

;; Tests using sample values for listofx-empty?
(check-expect (listofx-empty? (mtList 'dummyval)) #true)
(check-expect
  (listofx-empty? (consList 'hi (mtList 'dummyval)))
  #false)
```

Interfaces and Objects

Interfaces and Union Types

- ```
  ;; listofx → X throws error
  ;; Purpose: Return first element of given list
  (define (listofx-first lox-o) (lox-o 'first))

  ;; Sample expressions for listofx-first
  (define L1F (L1 'first))

  ;; Tests using sample computations for listofx-first
  (check-expect (listofx-first L1) L1F)

  ;; Tests using sample values for listofx-first
  (check-error  (listofx-first L0)
                "first requested from the empty list")
  (check-error  (listofx-first (mtList 'dummyval))
                "first requested from the empty list")
  (check-expect
    (listofx-first (consList 'hi (mtList 'dummyval)))
    'hi)
  ```
  ```

Interfaces and Objects

Interfaces and Union Types

```
;; listofx → listofx throws error
;; Purpose: Return rest of given list
(define (listofx-rest lox-o) (lox-o 'rest))

;; Sample expressions for listofx-rest
(define L1R (L1 'rest))

;; Tests using sample computations for listofx-rest
(check-expect (listofx-2Rlst (listofx-rest L1)) '(2 3))

;; Tests using sample values for listofx-rest
(check-error  (listofx-rest L0)
              "rest requested from the empty list")
(check-error  (listofx-rest (mtList 'dummyval))
              "rest requested from the empty list")
(check-expect
  (listofx-2Rlst
    (listofx-rest
      (consList 'hi
                (mtList 'dummyval))))
  '())
```

Interfaces and Objects

Interfaces and Union Types

```
;; listofx X → listofx
;; Purpose: Add the given value to the front of the
;;          given listofx
(define (listofx-cons lox-o an-x) ((lox-o 'cons) an-x))

;; Sample expressions for listofx-cons
(define L0C ((L0 'cons) 31))
(define L1C ((L1 'cons) 0))

;; Tests using sample computations for listofx-cons
(check-expect (listofx-2Rlst (listofx-cons L0 31))
              (listofx-2Rlst L0C))
(check-expect (listofx-2Rlst (listofx-cons L1 0))
              (listofx-2Rlst L1C))

;; Tests using sample values for listofx-cons
(check-expect
  (listofx-2Rlst
    (listofx-cons (consList "hi" (mtList 'dummyval))
                  "hey"))
  '("hey" "hi"))
```

Interfaces and Objects

Interfaces and Union Types

```
;; <X Y> listofx (X → Y) → listofy
;; Purpose: Map given function onto given list
(define (listofx-map lox-o f) ((lox-o 'map) f))

;; Sample expressions for listofx-map
(define L0M ((L0 'map) sub1))
(define L1M ((L1 'map) (λ (n) (* 2 n))))

;; Tests using sample computations for listofx-map
(check-expect (listofx-2Rlst (listofx-map L0 sub1))
              (listofx-2Rlst L0M))
(check-expect
  (listofx-2Rlst (listofx-map L1 (λ (n) (* 2 n))))
  (listofx-2Rlst L1M))

;; Tests using sample values for listofx-map
(check-expect
  (listofx-2Rlst
    (listofx-map
      (consList "hi" (mtList 'dummyval))
      string-length))
  '(2))
```

Interfaces and Objects

Interfaces and Union Types

```
;; listofx → Boolean
;; Purpose: Convert given listofx to a ISL+ list
(define (listofx-2Rlst lox-o) (lox-o '2Rlst))

;; Sample expressions for listofx-2Rlst
(define L0RL (L0 '2Rlst))
(define L1RL (L1 '2Rlst))

;; Tests using sample computations for listofx-2Rlst
(check-expect (listofx-2Rlst L0) L0RL)
(check-expect (listofx-2Rlst L1) L1RL)

;; Tests using sample values for listofx-2Rlst
(check-expect (listofx-2Rlst (mtList 'dummyval)) '())
(check-expect (listofx-2Rlst
                (consList 'hi (mtList 'dummyval)))
              '(hi))
```

Interfaces and Objects

Homework

- Problems: 321–325