CS 135 Exam Review Package

This package includes the packages from the first and second midterm sessions.

CS 135 Midterm 1 Review Package

Introduction to Scheme

Writing Arithmetic Expressions

- 4 - 5 * (3 + 2) => (- 4 (* 5 (+ 3 2)))
- 3 * 5 / (2 + 7) + 3 => (+ 3 / (* 3 5) (+ 2 7))

Things to Remember:

- Extra parentheses, while okay in mathematics, affect expressions in Scheme
- Operations are written before the associated arguments
- Operations need two arguments
- Substitutions are done left to right

Functions in Scheme

(define (f x y) (+ (* x x) y))

- The word define is used to bind a name to an expression
- f is the name of the function
- x and y are the parameters
- Values that are passed as x and y are the arguments of the function
- (+ (* x x) y) is the body expression, and tells us what the function does
- x and y have no meaning in any other part of the program

(define b “constant”)

- b is a constant
- If b is used in any body expressions, “constant” will be used
- Constants can make programs easier to understand

Design Recipe

Contract

- Describes the arguments that your function takes as well as the type that your function will produce
- Some possibilities are Int, Nat, Symbol, String, Posn, etc.
- For the output of your function, None is also a possibility

Examples:

Symbol -> None
String Int -> Boolean
Posn Symbol Int[>= 7] -> Posn

Purpose

- Explains what your function does using the names of your parameters
- Purpose comes from the question

Example:

The unit of speed most often used in physics is meters per second (m/s). A common Imperial unit of speed is miles per hour (mph). Write a function mph->m/s which consumes a speed in the units of mph and produces the same speed in units of m/s.

Purpose: converts the given speed, s, from mph to m/s

Examples

- Used to make sure you understand how the function is supposed to work
- Written below the purpose and above the function
- Don't necessarily test all cases; mostly try and get a general idea about what your function should do

Example:

Write a function final-cs135-grade that consumes four numbers:
(a) the first midterm grade (10%),
(b) the second midterm grade (20%),
(c) the final exam grade (45%), and
(d) the assignments grade (20%).
This function should produce the final grade in the course. You may need to review the mark allocation in the course. You can assume that all input marks are percentages and are given as integers between 0 and 100, inclusive. Also, assume a grade of 100 for participation (5%).

Examples:

```
(check-expect (final-cs135-grade 50 50 50 50) 52.5)
(check-expect (final-cs135-grade 75 87 72 83) 78.9)
```

Definition

- This is where you write your function
- Must include the header and the body
- The header is the name of your function as well as the parameters
- The body is the expression: what your function actually does with the parameters
- The header should be written before your purpose is written so that you can refer to parameters in your purpose
- Examples should be written after your header is complete so that you know all necessary parameters

Example:

```
(define (final-cs135-grade mid1 mid2 final assign)
        (+ (* mid1 0.10) (* mid2 0.20) (* final 0.45) (* assign 0.20) 5))
```

- Note: using constants for the weight of each section would make your program more readable

Tests

- Tests are written at the very end of the design recipe
- Used to find any errors in code or logic
- Use check-expect to test your function
- Form of check-expect is:
  - (check-expect (*function-name argument1 argument2 …) answer*)
  - If the function evaluates to the given answer, it will produce true
  - Otherwise check-expect evaluates to false, and you then need to check your calculations and/or your code
- Make sure to test helper functions as well to better locate any errors
- Test one or two general cases in addition to all special cases, or those cases with the potential to cause problems

Example:

For the previous function:

```
(check-expect (final-cs135-grade 0 0 0 0) 5)
(check-expect(final-cs135-grade 100 100 100 100) 100)
(check-expect(final-cs135-grade 100 0 0 0) 15)
(check-expect(final-cs135-grade 0 100 0 0) 25)
(check-expect(final-cs135-grade 0 0 100 0) 50)
(check-expect(final-cs135-grade 0 0 0 100) 25)
```

Helper Functions

- Helper functions are smaller functions in your program that perform specific tasks
- Make code easier to read and understand, especially for larger programs
- Breaking down a problem into smaller functions can make it easier to think about
- Facilitate the debugging of specific sections of code and help reduce errors for computations done multiple times
- Need to use the design recipe when creating helper functions as well

Example:

For Assignment 2, question four you could use a helper function that takes in a note and converts it to a natural number. This would then make it easier to check the number of tones between the three notes given.

Data Types

Symbols and Strings

- Symbols are indicated with a ‘ (‘blue, ‘green, ‘omg are all examples of symbols)
- The word used after the symbol mark only has meaning to the user, not to Scheme
- Strings are similar to symbols on the surface
  - Strings are actually a sequence of characters (written with “” around the character sequence, such as “blue”, “grey”, “elephant”)
  - Strings can use characters that symbols cannot, such as spaces
  - Strings have more built-in functions (string-append, string-length, etc)
- Symbols are used for fixed labels; strings are used when the data may change or when specific parts of the data are needed

Boolean Values and Predicates

- There are two Boolean values: true and false
- All relational operators evaluate to either true or false
- Some relational operators are =, >, <, <=, >=
- There are functions called predicates which produce true or false depending on the argument
  - Some are built into Scheme (even?, negative?, zero?, equal?, symbol=?)
  - Predicates can also be written by the programmer
  - Basically, predicates answer a question with true or false

Examples:

```
(= x 0)
(<= 3 19) => evaluates to true
(even? 5) => evaluates to false
(symbol=? ‘blue ‘green) => evaluates to false
(define (multiple-of-4? num)
        (= (% num 4) 0))
```

Conditional Expressions

Compound Conditions

- and, or, and not allow for a broader range of comparisons (for example, to determine whether a value x is between 0 and 100, you can write (and (> x 0) (< x 100))
- and
    - Can have more than 2 arguments
    - Evaluates to true when all arguments are true
    - If any argument is false, Scheme will stop evaluating the arguments and produce false
- or
    - Can have more than 2 arguments
    - Evaluates to true when at least one argument evaluates to true
    - If all arguments are false, then evaluates to false
    - Once Scheme evaluates a true argument, it stops and produces true
- not
    - Takes one argument
    - Evaluates to true when the argument is false
    - Evaluates to false when the argument is true

Examples:

(and (symbol=? ‘blue ‘blue) (<= 4 8)) => evaluates to true, since both arguments are true
(not (negative? -4)) => evaluates to false
(or (> (+7 4) (*7 4)) (< (+7 4) (*7 4))) => evaluates to true, since the second argument is true

Conditional Expressions

- For these, we use cond
- These expressions are a group of arguments, each one consisting of a question and answer
    - The question is always a Boolean expression
    - The answer is only evaluated if the Boolean expression is true
    - The questions are evaluated from top to bottom
- There must always be one argument that evaluates to true
- Many times, the last argument in a conditional expression contains the question else, which automatically evaluates to true
- Once a question evaluates to true, the answer associated with that question becomes the value of the entire expression
- To write tests for conditional expressions:
    - Write a test for each boundary condition (for example, if one question is (<= x 6), you should test for values less than 6, equal to 6 and greater than 6)
    - Write a test for a value that satisfies each question
    - Make sure all code has been tested (DrRacket will highlight any code that hasn't been tested)

Examples:

```
(cond [(> x 500) ‘largest]
      [(>= x 100) ‘medium]
      [else ‘smallest])

(cond
      [(cond
            [(p1? x) (p2? x)]
            [else (p1? x)])
               (cond
                     [(p3? x) ’Three]
                     [else ’Four])]
      [else ’Five])
```

Note: The third cond expression is the answer to the first question-answer pair of the main cond expression.

Syntax and Semantics

- Identifiers (names of constants, parameters, functions) cannot contain a space or ( ) , . ; { } [ ] ' "
- The semantic rules of Scheme allow for only one interpretation of a program
- Using the rules of the semantic model, we can step through a program to get to its most basic level

Examples:

```
1)  (+ 4 6) => 10
2)  (define (multiply x y) (* x y))
    (multiply 3 6)
    ⇨  (* 3 6)
    ⇨  18
3)  (define (my-fn x y)
            (cond [(< x y) x]
                  [(= x y) 'equal]
                  [(> x y) y]))
    (my-fn 10 10)
    ⇨  (cond [(< 10 10) x]
               [(= x y) 'equal]
               [(> x y) y])
    ⇨  (cond [false x]
               [(= x y) 'equal]
               [(> x y) y])
     ⇨  (cond [(= x y) 'equal]
               [(> x y) y])
     ⇨  (cond [(= 10 10) 'equal]
               [(> x y) y])
     ⇨  (cond [true 'equal]
               [(> x y) y])
     ⇨  'equal
```

Stepping Through Compound Conditions

- And
    - (and true expr2 . . .) => (and expr2 . . .)
    - (and false expr2 . . .) => false
    - (and) => true, since there are no false arguments in the expression
- Or
    - (or true expr2 . . .) => true
    - (or false expr2 . . .) => (or expr2 . . .)
    - (or) => false, since there are no true arguments in the expression

Structures

- Allow you to store several values in one
- A predefined structure in Scheme is the posn

Posn

- A posn is a structure containing two numbers, given the names x and y
- There are built-in functions, a constructor function and selector functions
- The constructor is make-posn and has the contract:
    - make-posn: Number Number -> Posn
- The selector functions allow you to access either number using posn-x and posn-y
    - posn-x: Posn -> Number
    - posn-y: Posn -> Number
- Just like with a number or a symbol, (make-posn 4 5) is a value and cannot be simplified further

General Structures

- To create a structure, we use define-struct instead of define
    - (define-struct posn (x y))
- When defining a structure, you need a name, and the names of the fields in parentheses
- The functions associated with any structure are the constructor, selectors, and the predicate
    - Constructor: make-*(the name of your structure)*
    - Selectors: (for posn) posn-x, posn-y
    - Predicate: (*the name of your structure)*? (for example, posn?)
        - The predicate returns true if the argument is a structure of that type, and false otherwise

Structures and the Design Recipe

```
(define-struct student (first last id))
;; a student = (make-student String String Number)
;; my-student-fn: Student -> Any
(define (my-student-fn data)
        … (student-first data) …
        … (student-last data) …
        … (student-id data) …)
```

- The above is what is needed in the design recipe for each new structure
- You must define the structure
- Write the data definition (communicate what type each field is required to be)
- Create a template for a function that consumes the structure
    - Your actual function may not use all the fields, but it is good practice to write the template to avoid forgetting a field

Practice Problems

HTDP, Section 2, Question 1: Write the contract, purpose, and header for a function that computes the area of a rectangle given its length and width. Write three examples for the behaviour of this program.

Solution:

```
;; area: Number Number -> Number
;; Purpose: calculates the area of a rectangle given the length, l, and the width, w
;; Examples:
(check-expect (area 5 3) 15)
(check-expect (area 1 1) 1)
(check-expect (area 10 15) 150)

(define (area l w))
```

HTDP, Section 2, Question 11: The nation of Progressiva has a simple tax code. The tax you pay is your salary times the tax rate, and the tax rate is 1/2% per thousand dollars of salary. For example, if you make $40,000, your tax rate is 1/2% times 40, which is 20%, so you pay 20% of $40,000, which is $8,000. Develop a function to compute the net pay (i.e. pay after taxes) of a person with a given salary. HINT: develop one or two auxiliary functions as well as net pay.

Solution:

To start this problem, break it into smaller, workable pieces. The simplest place to start is by computing the tax rate for a given salary.

```
(define tax-rate-percent 0.5) ;;0.5 is a constant provided in the question
```

Start with the contract, the only piece of information that needs to be passed to the function is the salary.

```
;; tax-rate: Number -> Number
;; Purpose: compute the tax rate for the given salary, s, as a decimal
```

For the examples, test the case given in the question as well as cases with numbers that are relatively simple to work out so you can get a feel of how the function works.

```
;; Examples:
(check-expect (tax-rate 40000) .20)
(check-expect (tax-rate 10000) .05)
(check-expect (tax-rate 100000) .50)
```

*From the question, we know that the tax rate = [0.5 * (s/1000)]/100. The division by 100 gives the decimal representation of the percentage. Now convert this to a Scheme expression:*

```
(define (tax-rate s)
  (/ (* tax-rate-percent (/ s 1000)) 100))
```

For tests, find special cases that may show errors in your code. Most general cases will be covered by examples.

```
;; Tests:
(check-expect (tax-rate 0) 0)
```

```
(check-expect (tax-rate 1000) 0.005)
(check-expect (tax-rate 200000) 1.00)
```

The second part of this problem is to calculate how much of the salary is taken as taxes. Start with the contract; again, the only piece of information passed is the salary. The function tax-rate is part of the body of the function.

```
;; taxes: Number -> Number
;; Purpose: compute the amount of tax paid for the given salary, s
```

Again, for examples, test simple cases as well as the case given in the question. Here I chose to use the same salaries as the examples in the above function.

```
;; Examples:
(check-expect (taxes 40000) 8000)
(check-expect (taxes 10000) 500)
(check-expect (taxes 100000) 50000)
```

To calculate the amount of money taken for taxes, simply multiply the salary by the tax rate. Notice that in the body of taxes, the function tax-rate is called.

```
(define (taxes s)
  (* s (tax-rate s)))
```

Again, I chose the same salaries to test as I did in tax-rate. This helps to follow the problem through and makes any errors easier to spot.

```
;; Tests:
(check-expect (taxes 0) 0)
(check-expect (taxes 1000) 5)
(check-expect (taxes 200000) 200000)
```

Finally, put everything together. From the question, we know that the only piece of information supplied to net-pay is the salary.

```
;; net-pay: Number -> Number
;; Purpose: given a salary, s, produce the amount the person is paid after taxes

;; Examples:
(check-expect (net-pay 40000) 32000)
(check-expect (net-pay 10000) 9500)
(check-expect (net-pay 100000) 50000)
```

Because of the helper functions we created, net-pay is simply the result of taxes applied to the salary, subtracted from the initial salary.

```
(define (net-pay s)
  (- s (taxes s)))

;; Tests:
(check-expect (net-pay 0) 0)
(check-expect (net-pay 1000) 995)
(check-expect (net-pay 200000) 0)
```

HTDP, Section 4, Question 1: Develop the function *within?*, which consumes three numbers representing the *x* and *y* coordinates of a point and the radius *r* of a circle centred around the origin. It returns true if the point is within or on the circle. It returns false otherwise. The distance of the point to the origin is the square root of $x^2 + y^2$.

Solution:

Start with the contract of the function. From the question, we know that it takes in three numbers and will produce a Boolean based on the relationship between the three arguments

```
;; within?: Number Number Number -> Boolean
;; Purpose: produce true if the point represented by x and y lies within or on a circle centered about the origin with a radius r
```

Choose simple examples to illustrate the different outputs that are expected.

```
;; Examples:
(check-expect (within? 3 4 5) true)
(check-expect (within? 5 6 3) false)
(check-expect  (within? 2 2 8) true)
```

To simplify the problem, we will create a helper function called sqr-distance to compute the square of the distance from the origin to the point (x, y). By comparing this value to the square of the radius, inexact numbers no longer need be considered.

```
(define (within? x y r)
  (<= (sqr-distance x y) (sqr r)))
```

Be sure to test different cases that were not covered by the examples.

```
;; Tests:
(check-expect (within? 0 0 5) true)
(check-expect (within? 7 4 9) true)
```

Our helper function takes in the coordinates of the point and computes the square of the distance from it to the origin. If we found the distance, we would end up with inexact numbers that would cause problems for close comparisons.

```
;; sqr-distance: Number Number -> Number
;; Purpose: computes the square of the distance of the point, represented by x and y, to the origin
```

Again, I chose simple numbers to illustrate the point of this function.

```
;; Examples:
(check-expect (sqr-distance 3 4) 25)
(check-expect (sqr-distance 2 2) 8)
```

This function is fairly simple: just use the formula given in the question and add the squares of the coordinates.

```
(define (sqr-distance x y)
  (+ (sqr x) (sqr y)))

;; Tests:
(check-expect (sqr-distance 0 0) 0)
(check-expect (sqr-distance 7 4) 65)
```

HTDP, Section 4, Question 2: A local discount store has a policy of putting labels with dates on all of its new merchandise. If an item has not sold within two weeks, the store discounts the item by 25% for the third week, 50% for the fourth week, and 75% for the fifth week. After that, no additional discounts are given. Develop the function *new-price*, which takes the initial price of an item and the number of weeks since the item was dated and produces the selling price of the item.

As always, start with the contract. From the question, we know the function will take in a number for the initial price, and a number for the number of weeks the item has been on sale.

```
;; new-price: Number Number -> Number
;; Purpose: using the initial price, iprice, and the number of weeks the item has gone unsold, week,
;;          produce the new selling price of the item
```

Work through some simple examples to get a feel for what the function does.

```
;; Examples:
(check-expect (new-price 10.00 4) 5)
(check-expect (new-price 16.60 5) 4.15)
```

To make the program more readable, I've included constants for the discount for weeks 3-5.

```
(define week3 0.25)
(define week4 0.50)
(define week5 0.75)
```

Depending on how many weeks the item has been on sale, the new price of the item will vary. Therefore, we need to use a conditional expression. If the item has been on sale for less than 3 weeks, no discount is applied. For three weeks, we apply the week 3 discount. At four weeks, we apply the week 4 discount, and after that, the week 5 discount is applied. To further improve readability, and to reduce the chance for error, a helper function could be used to calculate the new price.

```
(define (new-price iprice week)
  (cond [(< week 3) iprice]
        [(= week 3) (- iprice (* week3 iprice))]
        [(= week 4) (- iprice (* week4 iprice))]
        [else (- iprice (* week5 iprice))]))
```

For the tests, be sure to check when week is smaller than 3, equal to 3, equal to 4, equal to 5 and greater than 5. This ensures that no value of week will cause an error. Testing the case where week is equal to 0 is an extra precaution.

```
;; Tests:
(check-expect (new-price 133 0) 133)
(check-expect (new-price 8.40 3) 6.30)(check-expect (new-price 16.60 7) 4.15)
```

HTDP, Section 6, Question 4: Provide a structure definition and a data definition for an online auction entry, which is characterized by four pieces of information: the item number, the highest bidder, the current bid, and 'Open or 'Closed. Develop a function that consumes a bidder, a bid amount, and an auction entry and then returns a new entry. If the bid amount is less than or equal to the current bid or if the auction is closed, then the original entry is returned.

This is part of the design recipe for structures. You must define your structure, explain the argument types, and create a template for a function that uses the structure.

```
(define-struct auction-entry (item-num high-bid curr-bid status))
;; a auction-entry = (make-auction-entry Number Symbol Number Symbol)
;; my-auction-entry-fn: Auction-entry -> Any
;(define (my-auction-entry-fn data)
;  ... (auction-entry-item-num data) ...
;  ... (auction-entry-high-bid data) ...
;  ... (auction-entry-curr-bid data) ...
;  ... (auction-entry-status data) ...)
```

Start with the contract. From the question, we know that it takes in a name (in this case, let's assume it's a symbol since it wasn't specified in the question), a number corresponding to the new bid, and an auction entry. It then returns an updated auction entry.

```
;; update-entry: Symbol Number Auction-entry -> Auction-entry
;; Purpose: update and return the given auction-entry, ae, with the new bidder, name, and new bid, bid,
;;          unless the new bid is less than or equal to the current bid, or the auction is closed
```

Create a couple of examples to ensure you understand what the question is asking before writing the body of the function.

```
;; Examples:
(check-expect (update-entry 'Jack 1800 (make-auction-entry 20332 'Jane 1600 'Open))
        (make-auction-entry 20332 'Jack 1800 'Open))
(check-expect (update-entry 'Alice 12 (make-auction-entry 20332 'Jane 1600 'Open))
        (make-auction-entry 20332 'Jane 1600 'Open))
```

From the question, we know that if the auction is closed, or the new bid is less than or equal to the current bid, we produce the original auction-entry. The first two questions could be combined into one 'or' statement. If neither of these conditions are met, the auction-entry is changed to reflect the new bidder and bid. To access a specific field from your structure, hyphenate the structure name with the field you need to access.

```
(define (update-entry name bid ae)
  (cond [(equal? (auction-entry-status ae) 'Closed) ae]
        [(<= bid (auction-entry-curr-bid ae)) ae]
        [else (make-auction-entry (auction-entry-item-num ae) name bid (auction-entry-status ae))]))
```

Use the tests to check that all conditions from the question are met. Have one test where the auction is closed, one where the new bid is less than the current bid, one where the bids are equal, and one where the new bid is larger than the current bid.

```
;; Tests:
(check-expect (update-entry 'Jack 1800 (make-auction-entry 20332 'Jane 1600 'Closed))
          (make-auction-entry 20332 'Jane 1600 'Closed))
(check-expect (update-entry 'Pete 300 (make-auction-entry 20332 'Mary 1800 'Open))
          (make-auction-entry 20332 'Mary 1800 'Open))
(check-expect (update-entry 'Jack 1800 (make-auction-entry 20332 'Jane 1800 'Open))
          (make-auction-entry 20332 'Jane 1800 'Open))
(check-expect (update-entry 'Jack 1800 (make-auction-entry 20332 'Jane 0 'Open))
          (make-auction-entry 20332 'Jack 1800 'Open))
```

HTDP, Section 7, Question 2: Develop data and structure definitions for a collection of 3D shapes. The collection includes

- **cubes,** with relevant properties being the length of an edge;
- **prisms,** which are rectangular solids and with relevant properties being length, width, and height;
- **spheres,** with relevant property being the radius.

Develop the function *volume*. The function consumes a 3D shape and produces the volume of the object. The volume of a cube is the cube of the length of one of its edges. The volume of a prism is the product of its length, width, and height. The volume of a sphere is 4/3 * PI * r^3.

Solution:

Data definition and function template for a cube.

```
(define-struct cube (edge))
;; a Cube = (make-cube Number)
;; my-cube-fn: Cube -> Any
;(define (my-cube-fn data)
;  (...(cube-edge data)...))
```

Data definition and function template for a prism.

```
(define-struct prism (length width height))
;; a Prism = (make-prism Number Number Number)
;; my-prism-fn: Prism -> Any
;(define (my-prism-fn data)
;  (...(prism-length data)...
;   ...(prism-width data)...
;   ...(prism-height data)...))
```

Data definition and function template for a sphere.

```
(define-struct sphere (radius))
;; a Sphere = (make-sphere Number)
;; my-sphere-fn: Sphere -> Any
;(define (my-sphere-fn data)
;  (...(sphere-radius data)...))
```

A 3D-shape also needs a data definition and function template. The data definition is slightly different from a structure.

```
;; A 3D-shape is either:
;; * a cube
;; * a prism
;; * a sphere
;;my-3D-shape-fn: 3D-shape -> Any
;(define (my-3D-shape-fn data)
;  (cond [(cube? data) (...(cube-edge data)...)]
;           [(prism? data) (...(prism-length data)...
;                                ...(prism-width data)...
;                                ...(prism-height data)...)]
;           [(sphere? data) (...(sphere-radius data)...)]))
```

Define the constant PI for calculations involving the sphere.

```
(define PI 3.14159)
```

Start with the contract as usual. Use the data definition of a 3D-shape in the contract.

```
;; volume: 3D-shape -> Number
;; Purpose: consume a 3D-shape and produce the volume of the shape
```

Your examples should show what happens with each different 3D-shape.

```
;; Examples:
(check-expect (volume (make-cube 3)) 27)
(check-expect (volume (make-prism 3 4 5)) 60)
(check-expect (volume (make-sphere 3)) (* (/ 4 3) PI (expt 3 3)))
```

The function itself is fairly short. I used the function template for a 3D-shape and filled in the necessary details. Remember, each structure has a predicate you can use to test whether you have that structure or not (example: cube?).

```
(define (volume shape)
  (cond
     [(cube? shape) (expt (cube-edge shape) 3)]
     [(prism? shape) (* (prism-length shape) (prism-width shape) (prism-height shape))]
     [(sphere? shape) (* (/ 4 3) PI (expt (sphere-radius shape) 3))]))
```

Make sure to thoroughly test each 3D-shape. Check-within is not usually used in most programs in CS 135 as these programs rarely deal with comparing inexact numbers. To use it, the answer you expect is followed by the error you allow in the answer. In this case, my last test allows the answer to be between 523.597 to 523.599.

```
;; Tests:
(check-expect (volume (make-cube 0)) 0)
(check-expect (volume (make-prism 0 0 0)) 0)
(check-expect (volume (make-sphere 0)) 0)
(check-expect (volume (make-cube 5)) 125)
```

```
(check-expect (volume (make-prism 2 2 2)) 8)
(check-within (volume (make-sphere 5)) 523.598 0.001)
```

CS 135 Midterm 2 Review Package

Structures

- Allow you to store several values in one entity
- A predefined structure in Scheme is the posn

General Structures

- To create a structure, we use define-struct instead of define
  - (define-struct posn (x y))
- When defining a structure, you need a name, and the names of the fields in parentheses
- The functions associated with any structure are the constructor, selectors, and the predicate
  - Constructor: make-*(the name of your structure)*
  - Selectors: (for posn) posn-x, posn-y
  - Predicate: (*the name of your structure)*? (for example, posn?)
    - The predicate returns true if the argument is a structure of that type, and false otherwise

Structures and the Design Recipe

```
(define-struct student (first last id))
;; a student = (make-student String String Number)
;; my-student-fn: Student -> Any
(define (my-student-fn data)
        … (student-first data) …
        … (student-last data) …
        … (student-id data) …)
```

- The above is what is needed in the design recipe for each new structure
- You must define the structure
- Write the data definition (communicate what type each field is required to be)
- Create a template for a function that consumes the structure
  - Your actual function may not use all the fields, but it is good practice to write the template to avoid forgetting a field

Lists

- More flexible than structures
- Used for varying amounts of data
- It is possible to hold different data types in a general list, although it is usually specified in comments what type(s) all the values should be

What is a List?

- empty is a value in Scheme that represents an empty list
- The built-in predicate empty? returns true for an empty list and false otherwise
- The built-in predicate cons? returns true for a list and false otherwise
- The keyword cons is used to build lists
  - Takes two arguments which are a Scheme value and a list (the list could be empty)
  - Example: (cons ‘blue empty)
- The data definition is:
  ;; A List is one of:
  ;; * empty
  ;; * (cons Any List)
- This definition is recursive
  - At least one part of the definition (in this case empty) must not refer back to the list; this is the base case
  - The other part of the definition (in this case (cons Any List)) is the recursive part of the definition
- The first item, Any, cannot be empty
- When stepping through code, a list, like structures, cannot be simplified any further

Example: (cons 1 (cons 2 (cons 3 (cons 4 (cons 5 empty)))))

The first cons contains 1 and a list which is a cons that contains 2 and a list … and the last cons contains 5 and an empty list.

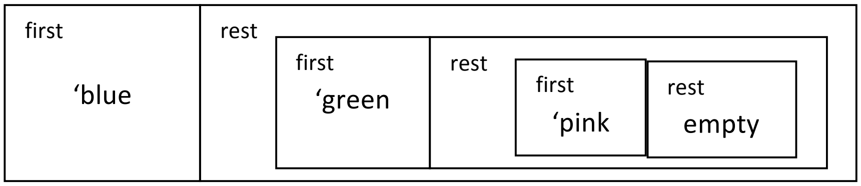
Selectors

- First and rest are selectors for lists
- First returns the first item in the list
  - For the list in the above example, first would return 1
- Rest returns everything after the first item in the list
  - For the list in the above example, rest would return (cons 2 (cons 3 (cons 4 (cons 5 empty))))
- For stepping through a program with first or rest, the following rules hold for a list (cons a b) where a is Any and b is a list:
  - (first (cons a b)) => a
  - (rest (cons a b)) => b

Visualizing Lists

- One method is to use nested boxes

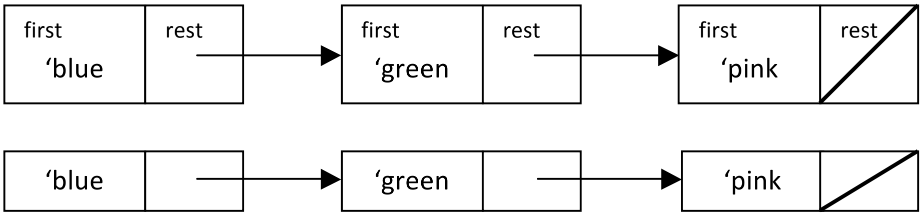
Example: (cons ‘blue (cons ‘green (cons ‘pink empty)))



Nested boxes

- Another method is box-and-pointer

Example: (cons ‘blue (cons ‘green (cons ‘pink empty)))



Coding with Lists

- The data definition above can be tailored to suit any type of list by replacing List with any of the following and Any with the corresponding data type:
    - (listof Symbol) for a list containing only symbols
    - (listof Number) for a list containing only numbers
    - (listof (union …)) for a list containing multiple types, replace the ellipsis with the types in the list
    - (listof Any) for a general case
- There is a template for functions using lists:

```
;; my-lst-fn: (listof Any)->Any
(define (my-lst-fn lst)
(cond
        [(empty? lst) . . . ]
        [else . . . (first lst) . . .
        . . . (my-lst-fn (rest lst)) . . . ]))
```

- This is for a general list; it can be made more specific depending on what type(s) your list contains

Stepping with Lists

- Stepping through a program as before requires too much writing to actually do for programs involving lists
- Instead we use a condensed trace for these recursive functions
- Steps included in a condensed trace for a function, my-recurs-fn:
    - Any time my-recurs-fn is applied to a value
    - The first line that does not include my-recurs-fn
    - The final, simplified value

```
Example: (define (scale lon num)

            (cond [(empty? lon) empty]

                 [else (cons (* (first lon) num) (scale (rest lon) num))]))

           (scale (cons 1 (cons 2 (cons 3 (cons 4 (cons 5 empty))))) 2)

            ⇨  (cons 2 (scale (cons 2 (cons 3 (cons 4 (cons 5 empty)))) 2))
            ⇨  (cons 2 (cons 4 (scale (cons 3 (cons 4 (cons 5 empty))) 2)))
            ⇨  (cons 2 (cons 4 (cons 6 (scale (cons 4 (cons 5 empty)) 2))))
            ⇨  (cons 2 (cons 4 (cons 6 (cons 8 (scale (cons 5 empty) 2)))))
            ⇨  (cons 2 (cons 4 (cons 6 (cons 8 (cons 10 (scale empty 2))))))
            ⇨  (cons 2 (cons 4 (cons 6 (cons 8 (cons 10 empty)))))
```

Design Recipe

- Just as with structures, a data definition is required for each different type of list
    - At least one part of the definition must not refer back to the definition; these are base cases
- You must also include a template for a function that takes the list as an argument
    - These functions will use a cond expression with each question dealing with one part of the data definition
- Use the function templates when creating your function
    - Fill in base cases first, as these are usually easiest to deal with
    - The recursive parts involve thinking about how you need to combine values after each step

Built-in List Functions

- empty?
  - Consumes a list
  - Checks whether the given list is empty
  - Returns true if the list is empty, or false if it is not
  - (empty? lst)
- cons?
  - Consumes anything
  - Checks whether the argument is a list or not
  - Returns true if the argument is a list, or false otherwise
  - (cons? data)
- first
  - Consumes a list
  - Produces the first element of the list
  - (first lst)
- rest
  - Consumes a list
  - Produces the list without the first element
  - (rest lst)
- length
  - Consumes a list
  - Produces the length of the given list (excluding empty)
  - (length lst)
- member?
  - Consumes an element of any type and a list
  - Checks whether the element is in the list
  - Returns true if the element is in the list, or false otherwise
  - (member? elem lst)
- reverse
  - consumes a list
  - produces the list in reverse
  - (reverse lst)

More Complex Lists

- While lists may contain simple data types, lists can also be more complicated
- Some lists can contain structures
    - Example: (define-struct posn (x y))
      (define structlist (cons (make-posn 2 3) (cons (make-posn 1 4) empty)))

- A list can also be a list of lists
    - Example: (define lst-of-lst (cons (cons 3 (cons 4 empty)) (cons (cons ‘blue (cons ‘green empty)) empty)))
        - (cons 3 (cons 4 empty)) is the first list in lst-of-lst
        - (cons (cons ‘blue (cons ‘green empty)) empty) is the rest of the list in lst-of-lst

List Abbreviations

- Instead of writing a long list using cons (example (cons 1 (cons 2 (cons 3 empty)))) we can use list to make a list (example is now (list 1 2 3))
- You can also write a list as ‘(exp1 exp2 … expn) for lists containing only symbols, strings and numbers
    - Example: ‘(red blue green)
- The keyword list creates a list of fixed length while cons is used to add a new element to the front of a list
- A list of lists is now much easier to visualize
    - Example: (cons (cons 3 (cons 4 empty)) (cons (cons ‘blue (cons ‘green empty)) empty)) is equivalent to (list (list 3 4) (list ‘blue ‘green)) and ‘((3 4) (blue green))

Types of Lists

- There are flat lists
    - These contain elements that are not lists
    - Example: (list 1 2 3 4 5)
- There are lists of lists
    - These contain elements that are lists
    - Example: (list (list 1 2) (list 3 4))
- There are nested lists
    - These contain elements that are lists containing elements (can be lists) to any depth
    - Example: (list (list (list ‘colour ‘blue) (list ‘colour ‘pink)) (list (list ‘sport ‘soccer)))

Dictionaries

- These are keys that are associated with a value
- Three operations that can be used with dictionaries are
    - lookup
        - Consumes a key
        - Produces the value associated with the given key
    - add
        - Consumes a key and value
        - Adds the pair to the dictionary
    - remove
        - Consumes a key
        - Removes the key and associated value from the dictionary

Association Lists

- In this way, a dictionary is a stored list of pairs
    - A pair is a two element list
    - The first element, for this course, is a number for the key
    - The second, for this course, is a string for the value
- The data definition for an association list is:

```
;; An association list (AL) is one of:
;; * empty
;; * (cons (list Number String) AL)
```

- Every key is unique although some keys might have the same values
- In a contract, since there is a data definition, we could use AL instead of (listof (list Number String))
- The template for an association list is

```
;; my-al-fn: AL -> Any
(define (my-al-fn alst)
(cond
[(empty? alst) . . . ]
[else . . . (first (first alst)) . . . ;; first key
        . . . (second (first alst)) . . . ;; first value
        . . . (my-al-fn (rest alst))]))
```

Recursion

Structural Recursion

- This is what is used for the data definition and template for lists
- For this type of recursion, each recursive call leaves all parameters unchanged or moves the recursion one step closer to a base case

Natural Numbers

- A natural number has a recursive definition

```
;; A Nat is one of:
;; * 0
;; * (add1 Nat)
```

- A template can also be made for this definition, just like with lists
- The template can be changed to make the base case any integer larger than 0

Accumulative Recursion

- This is quite close to structural recursion except that at least one parameter is an accumulator and does not follow the rules of structural recursion
- A wrapper function sets the initial value of the accumulator and then sends it, along with the other parameters, to a helper function

Generative Recursion

- Parameters are calculated at each step
- Much harder to code and debug
- Also harder to know that the function will always terminate

Processing Two Lists

- Four cases for the cond expression
  - list1 is empty and list2 is empty
  - list1 is empty and list2 is not empty
  - list1 is not empty and list2 is empty
  - list1 is not empty and list2 is not empty

Processing Just One List

- Only one list is processed by the function
- Just two possibilities to look at, either list1 is empty or not empty
- Example: appending one list to another

Processing in Lockstep

- Both lists are consumed by the function at the same rate
- This means that the lists must be the same length
- Only the same two possibilities need to be examined, since list2 will be empty (or not empty) at the same time that list1 is empty (or not empty)
- Example: calculating the dot product of two vectors inputted as lists
  - The first element of each list is multiplied together and added to the product of the second element from each list and so on until the last element of each list

Processing at Different Rates

- All four possibilities must be considered since either list could be empty at any time
- When both lists are empty, it is a base case
- The possibilities where either list1 or list2 are empty may be base cases as well depending on the function you are writing
- The fourth possibility can have many different templates depending on the program
- Example: merging two lists into one sorted list

Practice Problems

HTDP, Section 6, Question 4: Provide a structure definition and a data definition for an online auction entry, which is characterized by four pieces of information: the item number, the highest bidder, the current bid, and 'Open or 'Closed. Develop a function that consumes a bidder, a bid amount, and an auction entry and then returns a new entry. If the bid amount is less than or equal to the current bid or if the auction is closed, then the original entry is returned.

This is part of the design recipe for structures. You must define your structure, explain the argument types, and create a template for a function that uses the structure.

```
(define-struct auction-entry (item-num high-bid curr-bid status))
;; a auction-entry = (make-auction-entry Number Symbol Number Symbol)
;; my-auction-entry-fn: Auction-entry -> Any
;(define (my-auction-entry-fn data)
;  ... (auction-entry-item-num data) ...
;  ... (auction-entry-high-bid data) ...
;  ... (auction-entry-curr-bid data) ...
;  ... (auction-entry-status data) ...)
```

Start with the contract. From the question, we know that it takes in a name (in this case, let's assume it's a symbol since it wasn't specified in the question), a number corresponding to the new bid, and an auction entry. It then returns an updated auction entry.

```
;; update-entry: Symbol Number Auction-entry -> Auction-entry
;; Purpose: update and return the given auction-entry, ae, with the new bidder, name, and new bid, bid,
;;              unless the new bid is less than or equal to the current bid, or the auction is closed
```

Create a couple of examples to ensure you understand what the question is asking before writing the body of the function.

```
;; Examples:
(check-expect (update-entry 'Jack 1800 (make-auction-entry 20332 'Jane 1600 'Open))
           (make-auction-entry 20332 'Jack 1800 'Open))
(check-expect (update-entry 'Alice 12 (make-auction-entry 20332 'Jane 1600 'Open))
            (make-auction-entry 20332 'Jane 1600 'Open))
```

From the question, we know that if the auction is closed, or the new bid is less than or equal to the current bid, we produce the original auction-entry. The first two questions could be combined into one 'or' statement. If neither of these conditions are met, the auction-entry is changed to reflect the new bidder and bid. To access a specific field from your structure, hyphenate the structure name with the field you need to access.

```
(define (update-entry name bid ae)
  (cond [(equal? (auction-entry-status ae) 'Closed) ae]
        [(<= bid (auction-entry-curr-bid ae)) ae]
        [else (make-auction-entry (auction-entry-item-num ae) name bid (auction-entry-status ae))]))
```

Use the tests to check that all conditions from the question are met. Have one test where the auction is closed, one where the new bid is less than the current bid, one where the bids are equal, and one where the new bid is larger than the current bid.

```
;; Tests:
(check-expect (update-entry 'Jack 1800 (make-auction-entry 20332 'Jane 1600 'Closed))
        (make-auction-entry 20332 'Jane 1600 'Closed))
(check-expect (update-entry 'Pete 300 (make-auction-entry 20332 'Mary 1800 'Open))
        (make-auction-entry 20332 'Mary 1800 'Open))
(check-expect (update-entry 'Jack 1800 (make-auction-entry 20332 'Jane 1800 'Open))
        (make-auction-entry 20332 'Jane 1800 'Open))
(check-expect (update-entry 'Jack 1800 (make-auction-entry 20332 'Jane 0 'Open))
        (make-auction-entry 20332 'Jack 1800 'Open))
```

HTDP, Section 7, Question 2: Develop data and structure definitions for a collection of 3D shapes. The collection includes

- **cubes,** with relevant properties being the length of an edge;
- **prisms,** which are rectangular solids and with relevant properties being length, width, and height;
- **spheres,** with relevant property being the radius.

Develop the function *volume*. The function consumes a 3D shape and produces the volume of the object. The volume of a cube is the cube of the length of one of its edges. The volume of a prism is the product of its length, width, and height. The volume of a sphere is 4/3 * PI * r^3.

Solution:

Data definition and function template for a cube.

```
(define-struct cube (edge))
;; a Cube = (make-cube Number)
;; my-cube-fn: Cube -> Any
;(define (my-cube-fn data)
;  (...(cube-edge data)...))
```

Data definition and function template for a prism.

```
(define-struct prism (length width height))
;; a Prism = (make-prism Number Number Number)
;; my-prism-fn: Prism -> Any
;(define (my-prism-fn data)
;  (...(prism-length data)...
;   ...(prism-width data)...
;   ...(prism-height data)...))
```

Data definition and function template for a sphere.

```
(define-struct sphere (radius))
;; a Sphere = (make-sphere Number)
;; my-sphere-fn: Sphere -> Any
;(define (my-sphere-fn data)
;  (...(sphere-radius data)...))
```

A 3D-shape also needs a data definition and function template. The data definition is slightly different from a structure.

```
;; A 3D-shape is either:
;; * a cube
;; * a prism
;; * a sphere
;;my-3D-shape-fn: 3D-shape -> Any
;(define (my-3D-shape-fn data)
;  (cond [(cube? data) (...(cube-edge data)...)]
;        [(prism? data) (...(prism-length data)...
;                        ...(prism-width data)...
;                        ...(prism-height data)...)]
;        [(sphere? data) (...(sphere-radius data)...)]))
```

Define the constant PI for calculations involving the sphere.

```
(define PI 3.14159)
```

Start with the contract as usual. Use the data definition of a 3D-shape in the contract.

```
;; volume: 3D-shape -> Number
;; Purpose: consume a 3D-shape and produce the volume of the shape
```

Your examples should show what happens with each different 3D-shape.

```
;; Examples:
(check-expect (volume (make-cube 3)) 27)
(check-expect (volume (make-prism 3 4 5)) 60)
(check-expect (volume (make-sphere 3)) (* (/ 4 3) PI (expt 3 3)))
```

The function itself is fairly short. I used the function template for a 3D-shape and filled in the necessary details. Remember, each structure has a predicate you can use to test whether you have that structure or not (example: cube?).

```
(define (volume shape)
 (cond
     [(cube? shape) (expt (cube-edge shape) 3)]
     [(prism? shape) (* (prism-length shape) (prism-width shape) (prism-height shape))]
     [(sphere? shape) (* (/ 4 3) PI (expt (sphere-radius shape) 3))]))
```

Make sure to thoroughly test each 3D-shape. Check-within is not usually used in most programs in CS 135 as these programs rarely deal with comparing inexact numbers. To use it, the answer you expect is followed by the error you allow in the answer. In this case, my last test allows the answer to be between 523.597 to 523.599.

```
;; Tests:
(check-expect (volume (make-cube 0)) 0)
(check-expect (volume (make-prism 0 0 0)) 0)
(check-expect (volume (make-sphere 0)) 0)
(check-expect (volume (make-cube 5)) 125)
(check-expect (volume (make-prism 2 2 2)) 8)
(check-within (volume (make-sphere 5)) 523.598 0.001)
```

HTDP, Section 9, Question 4: Develop the function *string-append-n**, which consumes a non-empty list of strings and produces a single string resulting from appending all of the strings together. Use the built-in function *string-append*: String String -> String, which appends two strings.

Now develop the more general function *string-append**, which consumes any list of strings and produces a single string as described above. This new function must handle empty in a reasonable way.

Be sure to include the template for list functions in your code unless otherwise stated.

```
;; my-lst-fn: (listof String)-> Any
;(define (my-lst-fn lst)
;  (cond [(empty? lst) . . . ]
;         [else  . . . (first lst) . . .
;                  . . . (my-lst-fn (rest lst)) . . . ]))
```

The contract, purpose and examples are the same as always, we just have a new type to use in the contract, (listof String).

```
;; string-append-n*: (listof String) -> String
;; Purpose: append all the strings in the non-empty list, strlst, to form one single string
```

This program is fairly simple with regards to examples, as there are really only two things to check. One is that your program can handle a general number of strings with different characters and spaces, and the other is if your list is of length one.

```
;; Examples:
(check-expect (string-append-n* (list "wow " "it's " "a " "sentence"))
            "wow it's a sentence")
```

This function is recursive. The function does not take an empty list, so your base case should look at the result just before a list is empty. In this case, return the last string remaining. There are no other cases to be considered except the general case. For the else part of your cond expression, you append the first string in the list to the recursive calls. The recursive calls will produce more strings, and once the base case is reached, the function terminates.

```
(define (string-append-n* strlst)
  (cond [(empty? (rest strlst)) (first strlst)]
        [else (string-append (first strlst)
                    (string-append-n* (rest strlst)))]))

;; Tests:
(check-expect (string-append-n* (list "short")) "short")
(check-expect (string-append-n* (list " " "twospaces" " "))
            " twospaces ")
```

I did not do a contract, purpose or example for this function since it is quite similar to the previous function. DO NOT DO THIS unless specifically told you only need certain parts of the design recipe. This *function can have an empty list as its argument, but is otherwise the same as the previous function. The else statement stays the same as string-append-n*, but now we need to check if the list is empty for our base case. To keep the function simple, return "". This will be accepted by string-append and will not change the string output. Additionally, if the list is empty to begin with, this outputs only an empty string.*

```
(define (string-append* strlst)
  (cond [(empty? strlst) ""]
       [else (string-append (first strlst)
                         (string-append* (rest strlst)))]))
```

I tested this function with a general list of strings, as well as the empty case to ensure it can handle any list of strings, no matter the length.

```
;; Tests:
 (check-expect (string-append* (list "whoa " "another " "sentence"))
             "whoa another sentence")
(check-expect (string-append* (list " ")) " ")
(check-expect (string-append* empty) "")
```

Practice Problem 4: Create the function remove for an association list. The function remove consumes an association list and a key, and produces the association list without the key/value pair for the given key. If the key is not in the association list, return the original association list.

I included the data definition and template for an association list. This way I have a precise definition, I have an idea of how my function should look, and I can use AL in my contract.

```
;; An association list (AL) is one of:
;; * empty
;; * (cons (list Number String) AL)

;; my-al-fn: AL -> Any
;(define (my-al-fn alst)
;  (cond
;    [(empty? alst) . . . ]
;    [else . . . (first (first alst)) . . .
;          . . . (second (first alst)) . . .
;          . . . (my-al-fn (rest alst))]))
```

I defined an association list to use in examples and tests to make it easier to read.

```
(define alist (list (list 1 "a") (list 7 "b") (list 3 "c")))

;; remove-al: AL Number -> AL
;; Purpose: remove a key/value pair from an AL, alst, given the key
```

Check general cases to understand how the function works.

```
;; Examples:
(check-expect (remove-al alist 7) (list (list 1 "a") (list 3 "c")))
(check-expect (remove-al alist 2) alist)
```

If alst is empty, then there is nothing left to do but return empty. If we have found the key, then return the rest of the list after the key/value pair containing the key. If alst is not empty and the key still has not been found, cons the first of alst to the recursive call on the rest of alst.

```
(define (remove-al alst key)
  (cond [(empty? alst) empty]
        [(= key (first (first alst))) (rest alst)]
        [else (cons (first alst) (remove-al (rest alst) key))]))
```

The last test ensures that the function can handle an empty alst, as well as that it returns the proper AL.

```
;; Tests:
(check-expect (remove-al alist 1) (list (list 7 "b") (list 3 "c")))
(check-expect (remove-al (remove-al (remove-al (remove-al alist 3) 7) 1) 3) empty)
```

Practice Problem 5: Write the function extract which takes in an arbitrary list and a list of indices. Extract should produce a list containing all the elements of the list of anything at the given indices. The first position in the list is at index 0. You can assume that the list of indices is sorted in increasing order.

This is the data definition for a list of numbers. I included this so that data definitions become more familiar. You do not need to do this unless asked to.

```
;; A (listof Number) is either:
;; * empty, or
;; * (cons Number (listof Number))
```

A general template for a function consuming two lists. This problem requires accumulative recursion, so modifications are necessary, but this is a good place to start.

```
;; my-lst-fn: (listof Any) (listof Any) -> Any
;(define (my-lst-fn lst1 lst2)
;  (cond [(and (empty? lst1) (empty? lst2)) ...]
;        [(empty? lst1) ...]
;        [(empty? lst2) ...]
;        [else ... (first lst1) ...
;              ... (first lst2) ...
;              ... (my-lst-fn ...lst1... ...lst2...) ...]))
```

This is a wrapper function, it allows us use an extra parameter in a helper function. I use this extra parameter to keep track of the current position.

```
;; extract: (listof Any) (listof Number) -> (listof Any)
;; Purpose: output a list of the items in the specified positions from the arbitary list
;; Examples:
(check-expect (extract '(a g q f toast 1) '(0 3 4)) '(a f toast))
(check-expect (extract '(1 2 3 4 5) empty) empty)
```

The wrapper function sends all the given parameters to a helper function as well as sets an extra parameter to 0 to keep track of the current index.

```
(define (extract content indices)
  (find-items content indices 0))
```

This is where all of the magic happens. Note that from our template, we can combine the first three cases into one using an or statement. The second case takes care of when we have reached the index that we are currently looking for, and builds the list that will be produced. The last case is a recursive call to find one of the indices that we are looking for.

```
;; find-items: (listof Any) (listof Number) Nat -> (listof Any)
(define (find-items content indices position)
 (cond [(or (empty? indices)
        (empty? content))
     empty]
     [(= position (first indices))
     (cons (first content) (find-items (rest content)
                         (rest indices)
                         (add1 position)))]
     [else (find-items (rest content) indices (add1 position))]))
```

Test the case where the arbitrary list is an empty list, the case where the index is larger than the last position, and an arbitrary case.

```
;; Tests:
(check-expect (extract empty (list 0)) empty)
(check-expect (extract (list 1 2 3 4) (list 4)) empty)
(check-expect (extract (list 0) (list 0)) (list 0))
```

HTDP, Exercise 17.3.1: Develop list-pick0, which consumes a list and an index. The function produces the item at the given index, and an error if there is no nth item. The first item in the list is at index 0.

Be sure to include the template for list functions in your code unless otherwise stated. Notice that in the template I've included the natural number. Natural numbers also have a recursive definition, so it's useful to consider it in your template.

```
;; my-lst-fn: (listof Any) Nat -> Any
;(define (my-lst-fn lst n)
;  (cond [(empty? lst) . . . ]
;           [(= n 0) . . .]
;           [else  . . . (first lst) . . .
;                    . . . (my-lst-fn (rest lst) (sub1 n)) . . . ]))
```

For the contract, note that the 'Any' produced must match the type of element from the list.

```
;; list-pick0: (listof Any) Nat -> Any
;; Purpose: consumes a list, lst, and produces the element at index n of lst, or an error if the list is too short
```

This function uses check-error to test that the function will produce the error message when necessary.

```
;; Examples:
(check-expect (list-pick0 (list 0 1 2 3 4 5) 4) 4)
(check-error (list-pick0 (list 0 1 2 3 4 5) 6)
          (error 'list-pick0 "The list is too short."))
```

If the list is empty, then we have not yet found the nth index, or the nth index is empty, both of which should produce an error message. If this evaluates to false, then if n=0 we have found the correct index. Return the first part of the remaining list. For the recursive case, we call list-pick0 with the rest of the list, and subtract one from n, moving both closer to their respective base cases.

```
(define (list-pick0 lst n)
  (cond [(empty? lst) (error 'list-pick0 "The list is too short.")]
      [(= n 0) (first lst)]
      [else (list-pick0 (rest lst) (sub1 n))]))
```

For the tests, I did another general case, as well as another way to produce the error message.

```
;; Tests:
(check-error (list-pick0 empty 3)
          (error 'list-pick0 "The list is too short."))
(check-expect (list-pick0 '(z y x w v u) 3) 'w)
```

Post Midterm Review Package

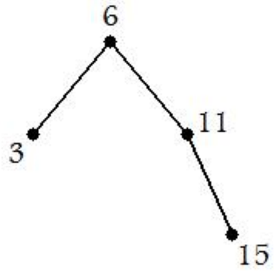
Trees

Tree Terminology

- Parent
    - The parent of a node is the connected node in the level above
- Child
    - The lower level of a connection
- Siblings
    - Nodes that share a common parent
- Root
    - The top of the tree
    - Has no parents
- Internal node
    - Has at least one child and one parent
- Leaves
    - Has no children
    - Has a parent
- Subtree
    - A tree that contains a child of the node you are considering

Example:

- 11 is the parent of 15
    - 6 is the parent of both 3 and 11
- 3 is the child of 6
    - 11 is the child of 6
    - 15 is the child of 11
- 3 and 11 are siblings
- 6 is the root
- 11 is an internal node
- 3 and 15 are leaves
- 11 – 15 is a subtree



Characteristics

- There is no set way of making a tree: each tree structure has its own rules for creating it
- Internal nodes can have exactly one child, two children, at most two children or any number
- Labels may be placed on all the nodes or just the leaves
- The children may have to follow an ordering property

Example from the notes: Evolution Tree

- Internal nodes have exactly 2 children
- Leaves are labelled with names and populations of modern species
- Internal nodes are labelled with names and dates of evolutionary events
- The structure follows evolution hypotheses

Binary Search Trees

- Referred to as BST for the rest of the package
- Another way to implement dictionaries
- Each key-value pair is stored as the nodes in the tree
- Many different BSTs are possible from the same dictionary even though they all stick to the BST properties

Properties

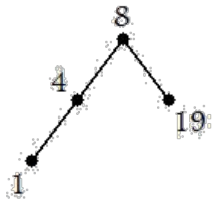
- Each node consists of a key (k), a value (v), a left subtree (l) and a right subtree (r)
- There is an ordering property for BSTs
  - Every key in l is less than k
  - Every key in r is greater than k
- No two nodes have the same key

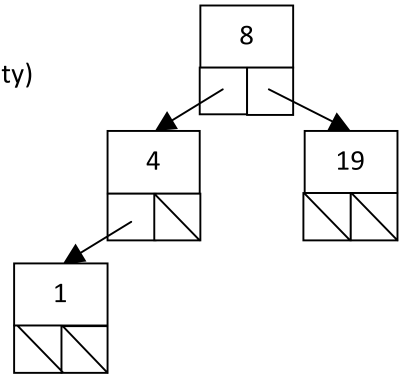
Coding BSTs in DrRacket

- For this class, a node is written as (make-node k v l r), and node is defined in the beginning of the program (define-struct node key val left right)
- Data definition:
  ```
  (define-struct node (key val left right))
  ;; A binary search tree (Bst) is one of:
  ;; * empty
  ;; * (make-node Number String Bst Bst)
  ```
- For a subtree with nothing in it, i.e. a base case, use the keyword empty
- val can be anything, it does not have to be a string

Example: A BST containing the (key, value) pairs (1, ‘leaf), (4, ‘internal), (8, ‘root), (19, ‘leaf)

```
(make-node 8 'root
      (make-node 4 'internal
            (make-node 1 'leaf empty empty)
            empty)
      (make-node 19 'leaf empty empty))
```





- Template for using BSTs
  ```
  ;; my-bst-fn: Bst →Any
  (define (my-bst-fn t)
     (cond [(empty? t) . . . ]
           [else . . . (node-key t) . . .
                 . . . (node-val t) . . .
                 . . . (my-bst-fn (node-left t)) . . .
                 . . . (my-bst-fn (node-right t)) . . .]))
  ```

Searching a BST

- The ordering property of a BST allows for more efficient searching of a dictionary than a list
- Suppose you are searching for a key x in a BST
- Base case
    - If the BST is empty, n is not in the BST
- Otherwise the BST has the form (make-node k v l r)
    - If x = k, the node is found
    - If x < k, the node will be in l if in the BST, no need to check r
    - If x > k, the node will be in r if in the BST, no need to check l
        - Because of the ordering property, one recursive call is saved at each step

```
;; search-bst: Number Bst→(union String false)
;; Produces value associated with n or false if n is not in t.
(define (search-bst n t)
   (cond [(empty? t) false]
          [(= n (node-key t)) (node-value t)]
          [(< n (node-key t)) (search-bst n (node-left t))]
          [(> n (node-key t)) (search-bst n (node-right t))]))
```

Creating a BST

- A BST can be formed from a list of key-value pairs
- If the list is empty, then the BST is empty and the BST is done
- If the list is (cons (list k v) lst) then the pair (k v) is added to the BST

```
;; create-bst: (listof (list Number Any))→Bst
;; creates a BST from the given list, lst.
(define (create-bst lst)
   (cond [(empty? lst) empty]
          [else (add-bst (first lst) (create-bst (rest lst)))]))
```

Adding to a BST

- Suppose you wish to add a key-value pair, (k, v) to a BST
- If the BST is empty, then the new BST is one with only one node
- Otherwise the BST has the form (make-node j u l r)
    - If k = j, replace j with k and u with v
    - If k < j, the pair is to be added somewhere in l
    - If k > j, the pair is to be added somewhere in r
        - Because of the ordering property of BSTs, only one recursive call, rather than 2 is necessary

```
;; add-bst: (list Number Any) Bst→Bst
;; adds lst to t and produces the new BST.
(define (add-bst lst t)
    (cond [(empty? t)
              (make-node (first lst) (second lst) empty empty)]
             [(= (first lst) (node-key t))
              (make-node (node-key t)
                              (second lst)
                              (node-left t)
                              (node-right t))]
               [(< (first lst) (node-key t))
                (make-node (node-key t)
                              (node-val t)
                              (add-bst lst (node-left t))
                              (node-right t))]
               [(> (first lst) (node-key t))
                (make-node (node-key t)
                              (node-val t)
                              (node-left t)
                              (add-bst lst (node-right t)))]))
```

General Trees

- Any number of children are allowed for a node
- Instead of having a left and right subtree, the children can be put in a list

Leaf-labelled Trees

- Leaves store data
- Internal nodes are only used to show structure
- These trees can be represented using a nested list
- Template for leaf-labelled trees

```
(define (my-llt-fn l)
    (cond [(empty? l) . . . ]
             [(cons? (first l))
               . . . (my-llt-fn (first l)) . . .
                . . .(my-llt-fn (rest l)) . . .]
             [else . . . (first l) . . . (my-llt-fn (rest l)) . . . ]))
```

Mutual Recursion

- Occurs when data definitions reference each other
- The number of data definitions can be two or greater
- In order to write a program using mutual recursion:
    - Create a template for each data definition
    - Create a function for each template
- An example of mutual recursion is working with the general tree for an arithmetic expression (found in course notes)

Local

- Allows definitions inside functions
- Can make programs easier to write and understand, as well as more efficient
- Previously used names can be redefined inside local
- A convention is to use square brackets around the body of the local expression

Semantics

- Definitions are "promoted" to the top level
- Once they are promoted, the definitions are evaluated right away, and then the program continues
- Look at a general expression (local [(define x1 exp1) … (define xn expn)] bodyexp)
  - x1 is replaced with a new identifier, such as x1_0
  - This is repeated with each definition from x2 to xn
  - The definitions are now brought to the top level
  - What is left is (local [] bodyexp') where bodyexp' has the new names of the definitions
  - All the renaming is a single step

Encapsulation

- Local expressions have no value outside the local expression
- Local allows for the binding of names to functions and values
- This creates unique, new names

- Allows for helper functions to be available to only one function instead of out in the open

Design Recipe

- Requires contract and purpose
- Does not need examples or tests

Terminology

- Binding occurrence of a name
  - The use of a name in a definition of formal parameter
- Bound occurrences
  - Uses of that name corresponding to that binding
- Lexical scope
  - All places where a binding occurrence has effect
  - Holes may be created because of the reuse of names
- Global scope
  - The scope of top-level definitions
  - Generally they are accessible everywhere in the code

Functional Abstraction

- Abstraction is the process of finding similarities and disregarding differences between items
- In the intermediate language, functions are themselves values
  - Called first-class values
- This means that anything you can do with normal values, you can do with functions
  - Include them as an element in a list or structure
  - Consume them as arguments
  - Produce them as the result of another function

Filter

- Takes a function and a list
  - The function must be a predicate (i.e. produces true or false when applied to the elements of the list)
- Filter applies the function to each element of the list
- If an element produces true when the function is applied to it, it is added to a new list
- Filter produces this list of all the elements that produce true when the given function is applied

Using Local to Produce Functions

- Local can produce a function if not all its parameters have been given arguments

- These functions can be used in another expression or in a data structure

Contracts

- For functions that consume functions, we need a way to reflect this in the contract
- To represent a function in the contract, write out the contract for the consumed function as one part of the main function
- Example: If mystery-fn consumes a predicate (that requires a number as the input) and a list, the contract could look like:
    - mystery-fn: (Number -> Boolean) (listof Number) -> Any
- Parametric contracts are necessary for abstract list functions
- The predicate taken in by filter needs to apply to the elements of the list consumed
- Use a variable such as X to indicate that, while X can be any type, the list and predicate must use the same type
- Example: filter: (X -> Boolean) (listof X) -> (listof X)
- If a variable is only used once, it is unnecessary and the type "Any" should be used

Lambda

- Lambda creates a nameless function that can be used as a value
- Quite useful for abstract functions that take in functions
    - No need to name functions that are only used for this purpose
- Lambda can be used to do the same thing that local does
    - Consider (local [(define x exp1)] exp2)
    - This is equivalent to ((lambda (x) exp2) exp1)

Map

- The abstract function map consumes a function and a list
- Map applies a function to each element in the given list and produces the new list
- map: (X -> Y) (listof X) -> (listof Y)

Foldr

- Short for "fold right"
- This function uses the provided combining function, starting from the right-hand end of the list, to produce a value
- Using foldr, it is possible to recreate map and filter
- foldr consumes a function that takes in 2 parameters, a base value, and a list
- (X Y -> Y) Y (listof X) -> Y
- Can be used to create lists; the provided function must build a list

Foldl

- foldl is short for "fold left"
- It is similar to foldr except the the function starts from the left-hand end of the list
- It consumes a combining function, a base value and a list
- foldl: (X Y -> Y) Y (listof X) -> Y

Higher-order Functions

- Refers to functions that consume or produce functions
- All the previous abstract functions are higher-order functions
- Another example is build-list
  - Consumes a natural number and a function and produces the list containing the numbers from 0 to n-1 after the function has been applied
  - (build-list 5 sub1) => (list -1 0 1 2 3)

Generative Recursion

- Recursive cases involve calculations for the next step
- Hard to know if the function will always terminate
- Difficult to code

Quicksort

- Select a pivot point
- Move items so that all items to the left of the pivot are smaller than it, and on the right side all the items are greater
- Choose a new pivot to each side of the current pivot and continue until the list is sorted
- The function always terminates because each sub-problem contains fewer elements than the one before until it consists of lists of length 1

BSTs Again

- One way to sort a list is to use inorder traversal
- Take a list and turn it into a BST
- To get the list back, but in sorted order, flatten the list in order from the left side to the right side of the tree
- This is called treesort
- Quicksort is like treesort without the tree
  - The pivot is the root of the tree, and then each subtree

-
-
-
-

Practice Problems – Trees

Practice Problem 1: Create a function common-parent that consumes a BST, bst, and two keys and produces the smallest subtree from a given bst such that both keys are in the subtree. Assume that both keys are in bst.

Be sure to define a node if needed. It will probably be specified in the question as to whether you must define it.

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

These are 2 trees I built to help make testing easier.

```
(define bst-1 (make-node 5 "a"
                (make-node 3 "b"
                    (make-node 1 "c" empty empty)
                    (make-node 4 "d" empty empty))
                (make-node 8 "e"
                    (make-node 6 "f"
                        empty
                        (make-node 7 "g" empty empty))
                    (make-node 9 "h" empty empty))))
```

Here is the contract. A BST is a type, so it should be included in the contract.

```
;; common-parent: Bst Number Number -> (listof Number)
;; produces the smallest subtree containing both keys
```

This example is one where the root is neither of the keys. It is just a general example where both keys are close to the node with key 5.

```
;; Example:
(check-expect (common-parent bst-1 1 8)
     (make-node 5 "a"
         (make-node 3 "b"
             (make-node 1 "c" empty empty)
             (make-node 4 "d" empty empty))
         (make-node 8 "e"
             (make-node 6 "f"
                 empty
                 (make-node 7 "g" empty empty))
             (make-node 9 "h" empty empty))))
```

To find the root of the subtree that is the smallest one to contain both keys, check if key1 and key2 are both in the left subtree of bst or both in the right subtree of bst. If one of these situations is the case, then a smaller subtree can be found and you should recursively call the function on the appropriate subtree (i.e. left or right of the current node) Eventually the current bst will result in either 1) one of the keys being the root of the subtree or 2) one key being in the left subtree and one key being in the right subtree. Once one of these two options occurs, there is no node that will make a smaller subtree and still contain both keys. At this point, return the current BST.

```
(define (common-parent bst key1 key2)
  (cond [(and (< key1 (node-key bst))
           (< key2 (node-key bst)))
        (common-parent (node-left bst) key1 key2)]
       [(and (> key1 (node-key bst))
           (> key2 (node-key bst)))
        (common-parent (node-right bst) key1 key2)]
       [else bst]))
```

The example checks that the function works when a key is in the left subtree and the other key is in the right subtree. The tests should check the function works if both keys are initially in the left subtree, both keys are initially in the right subtree, and also when a key is the root of the correct subtree.

```
;; Tests:
(check-expect (common-parent bst-1 6 9)
          (make-node 8 "e"
                 (make-node 6 "f"
                         empty
                         (make-node 7 "g" empty empty))
                 (make-node 9 "h" empty empty)))
(check-expect (common-parent bst-1 1 4)
          (make-node 3 "b"
                 (make-node 1 "c" empty empty)
                 (make-node 4 "d" empty empty)))
(check-expect (common-parent bst-1 3 1)
          (make-node 3 "b"
                 (make-node 1 "c" empty empty)
                 (make-node 4 "d" empty empty)))
```

Practice Problem 2: Create a function values-inorder that consumes a BST and produces the string resulting from appending the values of the BST in order. (Note that this is also an example of generative recursion)

Be sure to know how to define a node.

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

```
;; values-inorder: Bst -> String
;; produce the appended values of the BST in-order
```

This example does not have any tricks to it. The output is just a normal string.

```
;; Examples:
(check-expect (values-inorder (make-node 5 "I’m"
                                  (make-node 3 "know"
                                        (make-node 2 "I"
                                              empty
                                              empty)
                                        empty)
                                  (make-node 8 "ready."
                                        empty
                                        empty)))
        "IknowI’mready.")
```

To get the string in the proper order, you must do an in-order traversal of the tree, and append the values accordingly. For the base case, if bst is empty, I chose to put the empty string. It makes sense as if the tree is empty, as there are no string values in it, not even whitespace.

```
(define (values-inorder bst)
  (cond [(empty? bst) ""]
        [else (string-append
              (values-inorder (node-left bst))
              (string-append (node-val bst))
              (values-inorder (node-right bst)))]))
```

The first test is repetition of what was tested in the example. It is good to vary where the empty parts of the tree occur to ensure that your function can handle when a node has a left subtree and an empty right subtree and vice-versa. The second example tests explicitly that an empty bst is a valid input for the function.

```
;; Tests:
(check-expect (values-inorder (make-node 5 "purple"
                                  (make-node 3 "pink"
                              (make-node 2 "blue"
                                    empty
                                    empty)
                              empty)
                        (make-node 8 " " empty empty)))
        "bluepinkpurple ")
(check-expect (values-inorder empty) "")
```

Practice Problem 3: Create a function quadtree-search that, given a posn and a quadtree, produces the value associated with the posn or false if it is not in the tree. A quad-tree is like a BST, except there are up to four children from each node. In this problem, I decided to call them nw (northwest), ne (northeast), sw (southwest) and se (southeast). For a node, the key is a posn, and the value is a string. Each posn in nw has a smaller x value and larger y value than the node. In ne, both the x and y values of each posn are larger than the node. In sw, both the x and y values of each posn are smaller than the node. And in se, each posn has a larger x value and smaller y value that the node.

First, define the structure with which we're working.

```
(define-struct quadnode (key val nw ne sw se))
```

Here are four quadtrees to use with the examples and tests.

```
(define qtree-1 (make-quadnode (make-posn 0 0) "a"
                  (make-quadnode (make-posn -6 6) "b"
                           (make-quadnode
                            (make-posn -12 6) "c"
                            empty empty empty empty)
                           (make-quadnode
                            (make-posn -5 7) "d"
                            empty empty empty empty)
                           (make-quadnode
                            (make-posn -9 4) "e"
                            empty empty empty empty)
                           empty)
                  empty empty empty))

(define qtree-2 (make-quadnode (make-posn 0 0) "a"
                  empty
                  (make-quadnode (make-posn 6 6) "f"
                           (make-quadnode
                            (make-posn 4 8) "g"
                            empty empty empty empty)
                           empty
                           (make-quadnode
                            (make-posn 2 2) "h"
                            empty empty empty empty)
                           (make-quadnode
                            (make-posn 12 3) "i"
                            empty empty empty empty))
```

```
                 empty empty))




(define qtree-3 (make-quadnode (make-posn 0 0) "a"
                 empty
                 empty
                 (make-quadnode (make-posn -6 -6) "j"
                         (make-quadnode
                          (make-posn -9 -13) "k"
                          empty empty empty empty)
                         (make-quadnode
                          (make-posn -6 -2) "l"
                          empty empty empty empty)
                         empty
                         (make-quadnode
                          (make-posn -6 -55) "m"
                          empty empty empty empty))
                 empty))

(define qtree-4 (make-quadnode (make-posn 0 0) "a"
                 empty
                 empty
                 empty
                 (make-quadnode (make-posn 6 -6) "n"
                         empty
                         (make-quadnode
                          (make-posn 44 -6) "o"
                          empty empty empty empty)
                         (make-quadnode
                          (make-posn 4 -8) "p"
                          empty empty empty empty)
                         (make-quadnode
                          (make-posn 7 -9) "q"
                          empty empty empty empty))))

;; quadtree-search: Posn Quadtree -> (union String Boolean)
;; outputs either the value associated with the quadnode at the
;;  posn or false if there is no quadnode at the specified posn
```

This example looks at a quadnode where the posn is in the se child.

```
;; Example:
(check-expect (quadtree-search (make-posn 7 -9) qtree-4) "q")
```

To find the posn, first check if qtree is empty. If so, then the posn was not in the quadtree and the function produces false. Next, check if the current quadnode is the one we're looking for. If so, produce the value associated with this posn. If neither of these situations is the one we are currently in, check how the x and y value of posn compares with the key of the qtree. Recursively call the function using the appropriate child.

```
(define (quadtree-search posn qtree)
 (cond [(empty? qtree) false]
      [(equal? posn (quadnode-key qtree))
       (quadnode-val qtree)]
      [(and (< (posn-x posn) (posn-x (quadnode-key qtree)))
           (>= (posn-y posn) (posn-y (quadnode-key qtree))))
       (quadtree-search posn (quadnode-nw qtree))]
      [(and (>= (posn-x posn) (posn-x (quadnode-key qtree)))
           (>= (posn-y posn) (posn-y (quadnode-key qtree))))
       (quadtree-search posn (quadnode-ne qtree))]
      [(and (< (posn-x posn) (posn-x (quadnode-key qtree)))
           (< (posn-y posn) (posn-y (quadnode-key qtree))))
       (quadtree-search posn (quadnode-sw qtree))]
      [(and (>= (posn-x posn) (posn-x (quadnode-key qtree)))
           (< (posn-y posn) (posn-y (quadnode-key qtree))))
       (quadtree-search posn (quadnode-se qtree))]))
```

The tests should look at when the posn is in the ne child, the nw child, the sw child, and when the posn is not in the quadtree. The examples already tested the se child.

```
;; Tests:
(check-expect (quadtree-search (make-posn 12 3) qtree-2) "i")
(check-expect (quadtree-search (make-posn -12 6) qtree-1) "c")
(check-expect (quadtree-search (make-posn -6 -2) qtree-3) "l")
(check-expect (quadtree-search (make-posn 32 64) qtree-4) false)
```

Practice Problems – Local

Practice Problem 1: Step through the following code:

```
(define (list-max2 alon)
 (cond
  [(empty? (rest alon)) (first alon)]
  [else
   (local [(define max-rest (list-max2 (rest alon)))]
    (cond
     [(> (first alon) max-rest) (first alon)]
```

```
        [else max-rest])))]))

(list-max2 (list 1 13 8))
```

Some steps are omitted, such as stepping through cond expressions. Refer back to other parts of the package if you have trouble filling in the steps. On the exam, questions will probably ask for specific steps, such as when an expression is brought to the top level. Most of the steps included are those that deal specifically with the new challenges of local expressions.

```
=> ...
=> (local [(define max-rest (list-max2 (rest (list 1 13 8))))]
  (cond [(> (first alon) max-rest) (first alon)]
       [else max-rest]))
=> (define max-rest_0 (list-max2 (rest (list 1 13 8))))
(cond [(> (first (list 1 13 8)) max-rest_0) (first alon)]
     [else max-rest_0])
=> (define max-rest_0 (list-max2 (list 13 8)))
(cond [(> (first (list 1 13 8)) max-rest_0) (first alon)]
     [else max-rest_0])
=> ...
=> (define max-rest_0
  (local [(define max-rest (list-max2 (rest (list 13 8))))]
    (cond [(> (first (list 13 8)) max-rest) (first (list 13 8))]
         [else max-rest])))
(cond [(> (first (list 1 13 8)) max-rest_0) (first (list 1 13 8))]
     [else max-rest_0])
=> (define max-rest_1 (list-max2 (rest (list 13 8))))
(define max-rest_0
  (cond [(> (first (list 13 8)) max-rest_1) (first (list 13 8))]
       [else max-rest_1]))
(cond [(> (first (list 1 13 8)) max-rest_0) (first (list 1 13 8))]
     [else max-rest_0])
=> …
=> (define max-rest_1 first (list 8)))
(define max-rest_0
  (cond [(> (first (list 13 8)) max-rest_1) (first (list 13 8))]
       [else max-rest_1]))
(cond [(> (first (list 1 13 8)) max-rest_0) (first (list 1 13 8))]
     [else max-rest_0])
=> (define max-rest_1 8)
(define max-rest_0
  (cond [(> (first (list 13 8)) max-rest_1) (first (list 13 8))]
       [else max-rest_1]))
(cond [(> (first (list 1 13 8)) max-rest_0) (first (list 1 13 8))]
     [else max-rest_0])
=> ...
=> (define max-rest_1 8)
(define max-rest_0 13)
(cond [(> (first (list 1 13 8)) max-rest_0) (first (list 1 13 8))]
      [else max-rest_0])
=> ...
=> (cond [(> 1 13) (first (list 1 13 8))]
      [else max-rest_0])
=> ...
=> (cond [else max-rest_0])
=> max-rest_0
=> 13
```

HTDP, Section 2, Question 11: The nation of Progressiva has a simple tax code. The tax you pay is your salary times the tax rate, and the tax rate is 1/2% per thousand dollars of salary. For example, if you make $40,000, your tax rate is 1/2% times 40, which is 20%, so you pay 20% of $40,000, which is $8,000. Develop a function to compute the net pay (i.e. pay after taxes) of a person with a given salary. HINT: develop one or two auxiliary functions as well as net pay.

For this question, I used helper functions in the first midterm package. Here, we will use local. Refer back to the original solution for explanations with tests and examples.

```
;; net-pay: Number -> Number
;; Purpose: Given a salary, s, produce the amount the person is paid after taxes
;; Examples:
(check-expect (net-pay 40000) 32000)
(check-expect (net-pay 10000) 9500)
(check-expect (net-pay 100000) 50000)
```

Here, our functions are now constants inside a local expression. Since all the helper functions use the given salary, it makes more sense to put them in a local expression than to be spread out all over the program. The definitions still refer to one another, but since they are all within net-pay this works. All the definitions have their scope limited to inside net-pay.

```
(define (net-pay s)
  (local [(define tax-rate-percent 0.5)  ;from question, tax rate is 1/2% per thousand dollars of salary
          (define tax-rate (/ (* tax-rate-percent (/ s 1000)) 100))  ;total percentage of salary owed for taxes
          (define taxes (* s tax-rate))]  ;amount of salary taken as tax
    (- s taxes)))

;; Tests:
(check-expect (net-pay 0) 0)
(check-expect (net-pay 1000) 995)
(check-expect (net-pay 200000) 0)
```

Practice Problems – Functional Abstraction

This function produces the value of π; run it in Scheme to see.

```
;; foldr: (X Y -> Y) Y (listof X) -> Y
(foldr (lambda (x y) (+ x (/ y 10))) 0 '(3 1 4 1 5 9))
```

Practice Problem 1: Create a function fps that takes in a list of numbers and returns only the numbers in the list that are perfect squares.

Filter is a natural choice for this question, as we are looking to pick elements of the list that satisfy a specific property. To use filter, we need to create a predicate that determines whether a number is a perfect square. This can be done either in a local expression or using lambda. Then, once you have written your function, use it as the predicate for filter.

```
(define (fps lon)
  (local [(define (perfect? n)
          (integer? (sqrt n)))]
  (filter perfect? lon)))

(check-expect (fps (list 1 2 3 4 5 6 7 8 9)) (list 1 4 9))
(check-expect (fps (list 394 54 565 4354)) empty)
```

Practice Problem 2: Create a function do-nothing that consumes a function and produces the function.

This function takes in a function and uses lambda to output the same function.

```
;; do-nothing
;; do-nothing: (X -> Y) -> (X -> Y)
;; consumes a function, outputs a function
(define (do-nothing f)
  (lambda (x) (f x)))

(check-expect ((do-nothing sqr) 4) 16)
```

Practice Problem 3: Create a function my-compose that consumes two functions - the first takes the output of the second and the second one takes in a natural number – and produces the composition of them.

This function consumes two functions, f and g, and produces the composition f(g(x))

```
;; mycompose: (Y -> Z) (Nat -> Y) -> (Nat -> Z)
;; takes f and g and produces f(g(x))
```

The x is not specified until another argument is given to the resulting function.

```
(define (mycompose f g)
  (lambda (x) (f (g x))))

(check-expect ((mycompose sqr sqrt) 4) 4)
```

Practice Problem 4: Create a function even-strings which consumes a list of strings and produces the list containing the length of the even strings.

```
;; even-strings: (listof Strings) -> (listof Number)
;; produces a list of the strings in los that have an even length
```

This function first builds a list of the string lengths by using map. It then filters out all the lengths that are not even. For map, the function is string-length and the list is the one given. For filter, the function is even? and the list is the one produced by map.

```
(define (even-strings los)
  (filter even? (map string-length los)))

(check-expect (even-strings (list “a” “ab” “abc” “abcd” “abcde”)) (list 2 4))
```

Practice Problem 5: Write the function cross, which consumes two lists, A and B. It produces the nested cross product, where each item from A is paired with each item from B. All of the elements sharing the same element of A are grouped together into a list. You may assume that the elements of A are all different, and that the elements of B are all different. Keep the elements of the produced list in the same order as the corresponding inputs. For example:

```
;; (cross '(1 2 3) '(a b))
; => '(((1 a) (1 b)) ((2 a) (2 b)) ((3 a) (3 b)))
```

This method uses explicit recursion to solve the problem.

```
;(define (cross A B)
;  (cond
;    [(empty? A) empty]
;    [else (cons (prepend (first A) B)
;               (cross (rest A) B))]))
;
;(define (prepend e B)
;  (cond
;    [(empty? B) empty]
;    [else (cons (list e (first B))
;               (prepend e (rest B)))]))
```

This method involves the use of map and a helper function.

```
;(define (cross A B)
;  (map (lambda (x) (prepend x B)) A))
;
;(define (prepend e B)
;  (map (lambda (y) (list e y)) B))
```

This is the most concise version. This takes each element of A and applies (map (lambda (y) (list x y)) B)) *to it. This part takes each element of B as the value y. The value of x becomes the next element in A after looking at every element in B. This creates a list of pairs for each element of B with A.*

```
(define (cross A B)
  (map (lambda (x)
       (map (lambda (y) (list x y)) B)) A))
```

Practice Problem 6: Write the function poly-eval which consumes a polynomial given as a list of pairs, where the first element of the pair is the coefficient in front of x and the second element is the exponent of x, and an integer and produces the value of the polynomial for x equal to the integer.

```
;; poly-eval: (listof (list Integer Integer) Integer -> Integer
;; evaluates a polynomial given an int for the value of x
;; Example:
(check-expect (poly-eval '((8 5) (2 3) (4 2)) 6) 62784)
```

This function uses foldr to make the same calculation for each part of the polynomial, and then sum them all together. For foldr, the function is the lambda function, the base value is 0 (because it is addition) and the list is the list provided when the function is called.

```
(define (poly-eval poly x)
  (foldr (lambda (y z) (+ (* (first y) (expt x (second y))) z)) 0 poly))
```

Check when the polynomial is 0. Check how the function works with negative exponents. Check that the function works for an x value of 0.

```
;; Tests:
(check-expect (poly-eval '((0 0)) 7) 0)
(check-expect (poly-eval '((6 3) (2 -1) (8 -4)) 2) 49.5)
(check-expect (poly-eval '((8 5) (2 3) (4 2)) 0) 0)
```

Practice Problems – Generative Recursion

Practice Problem 1: Write a function matching-brackets? that consumes a list of characters and determines if a string has all matching brackets.

```
;;matching-brackets?: (listof Characters) -> Boolean
;; Write a function matching-brackets? which determines if a string has matching brackets.
;; Examples:
(check-expect (matching-brackets? (string->list "[{}]")) true)
(check-expect (matching-brackets? (string->list "){})){)}(")) false)
(check-expect (matching-brackets? (string->list "({[(])})")) false)
```

Use an accumulator to keep track of the open brackets that still need to be closed.

```
(define (matching-brackets? lst)
  (matching-helper lst empty))
```

If lst is empty, then the function should produce true or false depending on if the stack is also empty. If the first item in lst is an open bracket, recursively call with the rest of lst and add the open bracket to the top of the accumulator. If the first item in lst is a closed bracket that matches the first item in stack, recursively call with the rest of lst and the rest of stack. If none of these are the case, then we've found brackets that do not match up properly, and the function returns false.

```
(define (matching-helper lst stack)
  (cond [(empty? lst) (empty? stack)]
        [(open? (first lst))
         (matching-helper (rest lst) (cons (first lst) stack))]
        [(and (not (empty? stack))
              (match? (first stack) (first lst)))
         (matching-helper (rest lst) (rest stack))]
        [else false]))
```

Use this helper function to check if the bracket is open.

```
(define (open? b)
  (or (equal? b #\()
      (equal? b #\[)
      (equal? b #\{)))
```

Use this helper function to check if the brackets make an open-closed pair.

```
(define (match? a b)
  (or (and (equal? a #\() (equal? b #\)))
      (and (equal? a #\[) (equal? b #\]))
      (and (equal? a #\{) (equal? b #\}))))
```

Practice Problem 2: Create a function my-rev that takes in a list and uses foldl to produce the reversed list.

```
;; my-rev: (listof X) -> (listof X)
;; Use my-foldl to write my-rev to reverse a list.
;; For example:
;; (my-rev '(1 2 3 4)) => '(4 3 2 1)
;; (my-rev '(0 0 1 2)) => '(2 1 0 0)
```

The base value is empty. Foldl will cons the first of the list and empty, and will then cons the second of the list with the previous step, and so on until the list is reversed. The function is cons, the base value is empty, and the list is the one given as input to the function.

```
(define (my-rev list)
  (foldl cons empty list))

(check-expect (my-rev '(1 2 3 4)) '(4 3 2 1))
```