If you have not already, make sure you

- Read *How to Design Programs* Sections 6, 7.
Suppose I want to keep track of students by ID number, to store name and program. For each student I could make a list containing the data I want to store. I could make a list, and put item \( n \) in the \( n \)th position in my list, using our \( n \)-th-item function to get the \( n \)th student.

\[
\text{(define students}
\begin{array}{l}
\text{(list (list "Al Gore" "government")}
\text{(list "Barack Obama" "law")}
\text{(list "Bill Gates" "appliedmath")}
\text{(list "Conan O'Brien" "history")})
\end{array}
\text{)}
\]

But there is no student 0, no student 1, no student 2, \ldots. I could fill these in with the empty list, '()', but then I have millions of empty elements just to store thousands of student records!
Dictionaries

A better way: make a list, where each item in the list is itself a list, containing two items: a key (ID number) and a value (information about students)

;; A Student is a (list Str Str)
;; A LEntry is a (list Nat Student)
;; A LDict is a (listof LEntry)

(define student-dict
  (list (list 6938 (list "Al Gore" "government"))
        (list 7334 (list "Bill Gates" "appliedmath"))
        (list 8535 (list "Conan O'Brien" "history"))
        (list 8838 (list "Barack Obama" "law"))))

Exercise

Write a function (find-ldict key dict) that consumes a Nat and a LDict. The function returns the value in dict associated with the key. You may assume key appears exactly once in dict.

(check-expect (find-ldict 6938 student-dict) (list "Al Gore" "government"))

You could solve this using recursion, but it might be easier using filter.
Consider the data definition:

>; A Student is a (list Str Str)
>; A LEntry is a (list Nat Student)

Now we need to remember that if \( v \) is a LEntry, \((\text{first} \ v)\) returns the key, and \((\text{second} \ v)\) returns the associated value.

We also need to remember that if \( s \) is a Student, \((\text{first} \ s)\) returns the name, and \((\text{second} \ s)\) returns the program.

This is true, but inconvenient.

To help keep track of our data, we will now introduce \textbf{structures}, a new data type where the different parts have names.
It often comes up that we wish to join several pieces of data to form a single “package”. We can then write function that consume and return such packages.

A few examples:

A complex number

\[ z = a + bi \]

is built of a real part \( a \) and an imaginary part \( bi \).

A record in a student database might include the student’s name, ID number, and program.

```
{  
  name: "James Bond"  
  ID: 007  
  program: "pure-math"  
}
```

A labelled rooted binary tree has a label, left-child and right-child.
A **Posn** (short for Position) is a built-in structure that has two **fields** containing numbers intended to represent $x$ and $y$ coordinates. The computer knows these are called $x$ and $y$. You can create a **Posn** using the **constructor** function, `make-posn`. Its contract is

```scheme
;; make-posn Num Num -> Posn
```

For example,

```scheme
(define my-posn (make-posn 4 3))
```

Note here we are storing *two things*, namely the $x$ and $y$ coordinates, in *one* value. This one value is a **Posn**.
If you ask for the value of a Posn, it appears to just copy whatever you said.

```
(define my-posn (make-posn 4 3))
my-posn ⇒ (make-posn 4 3)
```

This is just like the quotation marks on a Str:

```
(define my-str "foo")
my-str ⇒ "foo"
```
Selectors

With a Str, we have special functions which get a part of the value:

(substring "foobar" 0 3) ⇒ "foo"

In a somewhat similar way, with a Posn, we have two selector functions. Each selector returns the field which has the name of the selector:

(posn-x (make-posn 4 3)) ⇒ 4
(posn-y (make-posn 4 3)) ⇒ 3

Note: these selectors are called posn-x and posn-y because the value is a Posn, and the fields are named x and y. Every structure has only the fields which are defined on it.
One last function: the **type predicate**.

- (posn? 42) ⇒ #false
- (posn? "oak") ⇒ #false
- (posn? my-posn) ⇒ #true

The type predicate returns #true if its argument is some object of that type.
Templates and Structures

Any time we create a structure we should create a data definition, and a template that goes along with it.

A template is derived from a **data definition**. When we create a new form of data, create the template. Use the template in writing functions to consume that type of data.

```scheme
(define-struct student
    (name id program))

;; a Student is a
;; (make-student Str Nat Str)
;; Requires:
;; name is the student's name
;; id is 8 digits long
;; program is sometimes fun.

;; my-student-template: Student -> Any
(define (my-student-template s)
  (.... (student-name s)...
  ....(student-id s)...
  ....(student-program s)....))
```

The template lists all the selectors, but does nothing. To write a function, replace the dots with code, and remove unused selectors.
Distance between two Posn

Recall \((\text{posn-x } p)\) returns the \(x\) coordinate, and \((\text{posn-y } p)\) returns the \(y\) coordinate. So the difference in \(x\) coordinate between \(p_1\) and \(p_2\) is: \((- (\text{posn-x } p_1) (\text{posn-x } p_2))\)

\[
((\text{posn-x } p_2), (\text{posn-y } p_2)) \\
((- (\text{posn-x } p_1), (\text{posn-y } p_1))
\]

;; \((\text{distance } x \ y)\) return the distance between \((x,y)\) and the origin.
;; Num Num -> Num
;; Examples:
(check-expect (distance 7 0) 7)
(check-expect (distance 3 4) 5)

Ex. Complete the function distance.
A function may return a Posn just like any other value.

;; (offset-a-little x y) return the point which is
;; moved over 3 and up 3 from (x, y).
;; offset-a-little: Num Num -> Posn
;; Example:
(check-expect (offset-a-little 5 7) (make-posn 8 10))

(define (offset-a-little x y)
  (make-posn (+ x 3) (+ y 3)))

Exercise Write a function vector2D+ that consumes two Posn and does vector addition. (That is, the new x is the sum of the x values, and the new y is the sum of the y values.)

;; (vector2D+ v1 v2) return the vector sum of v1 and v2.
;; vector2D+: Posn Posn -> Posn
;; Example:
(check-expect (vector2D+ (make-posn 2 3) (make-posn 5 8)) (make-posn 7 11))
Custom structures

We can define a custom structure using the `define-struct` special form:

\[
\text{\texttt{(define-struct \textit{struct-name} \textit{field0} \textit{field1} \ldots \textit{fieldn})}}
\]

For example, to define a structure to store polar coordinates \((r, \theta)\):

\[
\text{\texttt{(define-struct \textit{polarcoord} \textit{r} \textit{theta})}}
\]

- \textit{polarcoord} is the name of the new structure type
- \((r \ \textit{theta})\) are the names of the fields of the structure.

This automatically creates several functions:

\begin{itemize}
  \item **Constructor** \texttt{make-polarcoord} allows us to create values of this type
  \item **Predicate** \texttt{polarcoord?} lets us determine if a value is of this type
  \item **Selectors** are created, one for each field.
    \begin{itemize}
      \item In this example, \texttt{polarcoord-r} and \texttt{polarcoord-theta}.
    \end{itemize}
\end{itemize}
Creating and reading from custom structures

(define-struct polarcoord (r theta))
(define mypolar (make-polar 4 0.3)) ; constructor to create a polar
(polar-r mypolar) => 4 ; selector to read one field
(polar-theta mypolar) => 0.3 ; selector to read another field
(define-struct polarcoord (r theta))

This `define-struct` determines the names of the fields, but it does not tell us what the fields mean. So we need to document these; this is done in a comment called a data definition:

`; a Polarcoord is a (make-polarcoord Num Num)`
`; Requires:`
`;   r is the distance to the point. r >= 0.`
`;   theta is angle from the positive x-axis.`

The data definition tells us:

- the type of each field, in a line resembling a contract.
- the meaning of each field, in a Requires section.
(define-struct polarcoord (r theta))

;; a Polarcoord is a (make-polarcoord Num Num)
;; Requires:
;;   r is the distance to the point. r >= 0.
;;   theta is angle from the positive x-axis.

Reminder: from this structure, if thing is a Polarcoord, you can access the fields using
(polarcoord-r thing) and (polarcoord-theta thing).

Exercise

Complete the function polarcoord->posn that consumes a Polarcoord and returns the Posn

(Mathematically, \( x = r \cos \theta \) and \( y = r \sin \theta \).

;; (polarcoord->posn p) convert p to rectangular coordinates
;; polarcoord->posn: Polarcoord -> Posn
;; Example:
(check-within (polarcoord->posn (make-polarcoord 2 (/ pi 4)))
(make-posn (sqrt 2) (sqrt 2)) 0.0001)

(define (polarcoord->posn p)
  (make-posn (* (polarcoord-r p) (cos (polarcoord-theta p)))
             (* (polarcoord-r p) (sin (polarcoord-theta p))))
Exercise

Write a function \((\text{rotate-polar } p \ \text{angle})\) that consumes a \text{Polarcoord} and a \text{Num} and returns the \text{Polarcoord} that results from rotating \(p\) by \(\text{angle}\).

\[
\text{;; (rotate-polar } p \ \text{angle}) \ \text{return } p \ \text{rotated by } \text{angle}.
\]

\[
\text{;; rotate-polar: Polarcoord Num } \to \ \text{Polarcoord}
\]

\[
\text{;; Example:}
\]

\[
\text{(check-expect (rotate-polar (make-polarcoord 3 0.4) 0.2)}
\]

\[
\text{ (make-polarcoord 3 0.6))}
\]

\[
\text{(define (rotate-polar } p \ \text{angle)}
\]

\[
\text{(make-polarcoord (polarcoord-r } p)
\]

\[
\text{ (+ } \text{angle (polarcoord-theta } p))\text{))}
\]
What is the result of evaluating the following expression?

```
(define (distance a b)
  (sqrt (+ (- (posn-x a) (posn-x b))
           (- (posn-y a) (posn-y b))))
)
```

```
(define pt1 (make-posn "Math135" "CS115"))
(define pt2 (make-posn "Red" #true))
```

```
(distance pt1 pt2)
```

This causes a run-time error, but not at `make-posn`.

Inside `distance`, there it attempts to compute `(- "Math 135" "Red")`, which is nonsense. The system does not enforce contracts. If your contract says `Int`, but you give it a `Str`, problems will probably occur. Carefully watch your contracts, and be sure your code follows them!
Contract Errors

;; (scale pt factor) return pt
;; scaled by factor.
;; scale: Posn Num -> Posn

(define (scale pt factor)
  (make-posn (* factor (posn-x pt))
              (* factor (posn-y pt))))

(scale 2 "George")
⇒
(make-posn (* "George" (posn-x 2))
             (* "George" (posn-y 2)))

Contract errors will often manifest when we can’t simplify an expression.
In this case, we can’t use posn-x on 2.
Suppose I have a complicated structure:

```scheme
(define-struct household (me sally fish cat thing1 thing2))
;; a Household is is a (make-household Nat Nat Nat Nat Nat Nat)
;; Requires:
;; me is my age
;; sally is Sally's age
;; cat is the cat's age, etc.
```

and I want to change just one field. Do I really have to do all this work?!?

```scheme
;; update-cat: Household -> Household
(define (update-cat house newcat)
  (make-household
   (household-me house)
   (household-sally house)
   (household-fish house)
   newcat
   (household-thing1 house)
   (household-thing2 house)))
```

...Yes. Racket structures, as defined it this course, are clumsy. But don’t get put off structures! They are very useful, and much easier to use in every other language I know. In many languages you would just say

```scheme
house.cat = newcat
```
There are two new things in our syntax.

1. The special form `(define-struct sname (field1 ... fieldn))` defines the structure type and creates:
   - a **constructor** function `make-sname`
   - a **predicate** function `sname?`
   - *n selectors*, one for each field, named `sname-field1`...

2. A **value** has additional possibilities. In addition to begin a `Nat`, `Int`, `Num`, `Str`, or `Bool`, it may be of the form:
   `(make-sname v1...vn)`
Additions to the Design Recipe for Structures

1. Place your **structure definitions** and **data definitions** right at the top of the file, just after the file header.

   ```scheme
   (define-struct polarcoord (r theta))
   ;; a Polarcoord is a (make-polarcoord Num Num)
   ;; Requires:
   ;; r is the distance to the point. r >= 0.
   ;; theta is angle from the positive x-axis.
   ```

2. Write a **template**, with a generic name and generic contract.

   ```scheme
   (define (my-polarcoord-template p)
   (...(polarcoord-r p)...)
   (...(polarcoord-theta p)...))
   ```

The rest of the design recipe is essentially unchanged, except now you have the custom type (e.g. Polarcoord) which you added.
Back on slide 3 we used the following data definitions:

```scheme
;; A Student is a (list Str Str)
;; A LEntry is a (list Nat Student)
;; A LDict is a (listof LEntry)
```

```scheme
(define student-dict
  (list (list 6938 (list "Al Gore" "government"))
        (list 7334 (list "Bill Gates" "appliedmath"))
        (list 8535 (list "Conan O'Brien" "history"))
        (list 8838 (list "Barack Obama" "law"))))
```

This made it hard to keep track of which bits of data were which.

For example, I can get the first student from `student-dict` using `(first student-dict)`. But why should I get that student’s name with `(first (second (first student-dict)))`?

This is a perfect place to use structures.
We can use a structure to store each student. Then to store an **association list** of students, we have:

```scheme
(define-struct student (name program))
;; a Student is a (make-student Str Str)
;; A LDict is a (listof (list Nat Any))
(define student-ldict
  (list (list 6938 (make-student "Al Gore" "government")))
  (list 7334 (make-student "Bill Gates" "appliedmath"))
  (list 8535 (make-student "Conan O'Brien" "history"))
  (list 8838 (make-student "Barack Obama" "law"))))
```

Suppose we have some student:

```scheme
(define test-student (first student-dict))
```

- (first test-student) => 6938
- (second test-student) => (make-student "Al Gore" "government")

...It works, but we still need to use **first** and **second** to get the key and value.
We can take it another level. Use another structure to keep track of keys and values.

```scheme
(define-struct association (key val)) ;; An Association is a (make-association Nat Any) ;; a Dict (dictionary) is a (listof Association)

(define student-dict
  (list (make-association 6938 (make-student "Al Gore" "government"))
        (make-association 7334 (make-student "Bill Gates" "appliedmath"))
        (make-association 8838 (make-student "Barack Obama" "law"))))

Extracting some test data from this list:
(define test-student (first student-dict))

- (association-key test-student) => 6938
- (association-val test-student) => (make-student "Al Gore" "government")
```
(define-struct association (key val))
;; An Association is a (make-association Nat Any)
;; a Dict (dictionary) is a (listof Association)

(define student-dict
  (list (make-association 6938 (make-student "Al Gore" "government"))
        (make-association 7334 (make-student "Bill Gates" "appliedmath"))
        (make-association 8838 (make-student "Barack Obama" "law")))))

Reminder: if a is an Association: (association-key a) returns the key
(association-value a) returns the associated value.

Exercise

Complete dict-find. You may assume key appears at most once in dict.

;; (dict-find d key) return value associated with key in d.
;; If key is not in d, return #false.
;; dict-find: Dict Nat -> Any
;; Examples:
(check-expect (dict-find student-dict 7334)
  (make-student "Bill Gates" "appliedmath"))
(check-expect (dict-find student-dict 9999) #false)
## Complete dict-add.

;;; (dict-add d k v) return a new dictionary containing all values in d,  
;;; and new value (make-association k v). Keep data sorted by key.  
;;; If key is already in d, replace its value.  
;;; dict-add: Dict Nat Any -> Dict  
;;; Example:

(check-expect
  (dict-add student-dict
    7587
    (make-student "George W Bush" "business"))
  (list (make-association 6938 (make-student "Al Gore" "government"))
        (make-association 7334 (make-student "Bill Gates" "appliedmath"))
        (make-association 7587 (make-student "George W Bush" "business"))
        (make-association 8838 (make-student "Barack Obama" "law"))))
Module Summary

- Be comfortable with the following terms: structure, field, constructor, selector, type predicate, structure definition.
- Be able to write functions that consume and return structures, include \texttt{Posn} and custom data structures.
- Be able to create structure and data definitions for a new structure, determining an appropriate type for each field.
- Understand what functions are created by \texttt{define-struct}, and be able to use them.
- Be able to write write the template associated with a structure definition, and to expand it into the body of a particular function that consumes that type of structure.
- You should understand the use of type predicates and be able to write code that handles mixed data.

Before we begin the next module, please

- Read \textit{How to Design Programs} Sections 14, 15, 16.