We have used short, fixed-length, lists for data that seems to always belong together. For example, in M08 we had an association list with keys and values:

\[
\text{(list (list 8 "Asha") (list 2 "Joseph") (list 5 "Sami"))}
\]

Some other examples of compound data:

A complex number

\[
z = a + bi
\]

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

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

\{
  name: "James Bond"
  ID: 007
  program: "CS"
\}

A *labelled rooted binary tree* has a label, left-child and right-child.
We could represent a student with a short list containing their name, program, and a list of courses.

But this is annoying: we need to remember that the first item is the name, the second is the program, and the third is the courses.

Or, like we did earlier, we need to define trivial functions to do this for us.

This sounds like work, so we’re likely doing something wrong.

Remember: “Computers should work. People should think.”

Racket already has this feature built in: structures.
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"
```

**Exercise**

Create a constant somewhere that stores a Posn where the coordinates are (7, 2).

```
somewhere ⇒ (make-posn 7 2)
```
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 produces 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.

**Exercise**

Use `posn-x` and `posn-y` on your constant `somewhere`. Ensure you understand the result.
One last function: the **type predicate**.

\[(\text{posn? 42}) \Rightarrow \text{false}\]

\[(\text{posn? "oak"}) \Rightarrow \text{false}\]

\[(\text{posn? my-posn}) \Rightarrow \text{true}\]

The type predicate produces **true** if its argument is a value of that type, and **false** otherwise.

**Exercise**

Find at least two values for which `posn?` produces **true**, and two for which it produces **false**.
Example: distance between two Posn

\[(\text{posn-x p2}, \text{posn-y p2})\]

\[(\text{posn-x p1}, \text{posn-y p1})\]

%; (distance p1 p2) produce the distance
%; between p1 and p2.
%; Posn Posn -> Num
%; Examples:
(check-expect (distance (make-posn 0 7) (make-posn 0 0)) 7)
(check-expect (distance (make-posn 5 6) (make-posn 2 2)) 5)

Exercise

The template for the built-in Posn structure is as follows:

;; my-posn-template: Posn -> Any
(define (my-posn-template p)
  (...(posn-x p)...
  (...(posn-y p)...))

Using the template, complete the function distance.
A function may produce a `Posn` just like any other value. It needs to create it using `make-posn`. Here's an example:

```scheme
;; (offset-a-little x y) produce the point over 3 and up 3 from (x, y).
;; Example:
(check-expect (offset-a-little 5 7) (make-posn 8 10))
```

```scheme
;; offset-a-little: Num Num -> Posn
(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.)

```scheme
;; (vector2D+ v1 v2) produce 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))
```
We can define a custom structure using the `define-struct` special form:

```
(define-struct struct-name (field0 field1 ... fieldn))
```

For example, suppose we are building a store inventory system, and for each item we need to store its description, price, and number available.

```
(define-struct inventory (desc price available))
```

This one line of code automatically creates several functions:

- **Constructor** `make-inventory` allows us to create values of this type
- **Predicate** `inventory?` lets us determine if a value is of this type
- **Selectors** are created, one for each field.

In this example, `inventory-desc`, `inventory-price` and `inventory-available`. 
Once we have created the new data type:

```lisp
(define-struct inventory (desc price available))
```

...we can create a new value using the constructor, and store it in a constant:

```lisp
(define lentils (make-inventory "dry lentils" 2.49 42))
```

...and we can extract the values using the selector functions:

```lisp
(inventory-desc lentils)  ⇒  "dry lentils"
(inventory-price lentils)  ⇒  2.49
(inventory-available lentils)  ⇒  42
```

**Exercise**

1. Create a structure data type called `book`, with fields `title`, `author`, and `year`.
2. Use the constructor to create a constant of this type.
3. Use the selector functions to extract the individual values from the constant.
(define-struct inventory (desc price available))

This `define-struct` determines the names of the fields, but it does not tell us what the fields are for. So we need to document these, by writing a data definition:

```scheme
;; an Inventory is a (make-inventory Str Num Nat)
```

The data definition tells us:

- the **type** of each field, in a line resembling a contract;
- as needed, any **requirements** for the field values.

---

Just after your `(define-struct book ...) line, write a data definition for a Book.
(define-struct inventory (desc price available))
;; an Inventory is a (make-inventory Str Num Nat)

Reminder: from this structure, if thing is an Inventory, you can access the fields using (inventory-desc thing), (inventory-price thing) and (inventory-available thing).

Exercise

Complete the function total-value that consumes an Inventory and produces the amount of money we would get if we sell out of item.
;; (total-value item) produce cost of all our item.
;; Example:
(check-expect (total-value (make-inventory "rice" 5.50 6)) 33.00)

;; total-value: Inventory -> Num
(define-struct inventory (desc price available))
;; an Inventory is a (make-inventory Str Num Nat)

To create an Inventory, use the make-inventory function.

Exercise

Write a function (raise-price dollars item) that consumes a Num and a Inventory and produces the Inventory that results from increasing the price of item by dollars.

;; (raise-price dollars item) produce item with price increased by dollars.
;; Example:
(check-expect (raise-price 0.49 (make-inventory "rice" 5.50 6))
  (make-inventory "rice" 5.99 6))

;; raise-price: Num Inventory -> Inventory
The special form

\((\text{define-struct} \ sname \ (\text{fname}_1 \ \ldots \ \text{fname}_n))\)

defines the structure type \(\text{sname}\) with \textbf{fields} \(\text{fname}_1\) to \(\text{fname}_n\). It also automatically defines the following primitive functions:

- **Constructor:** \texttt{make-sname}
- **Selectors:** \texttt{sname-fname}_1 \ldots \texttt{sname-fname}_n
- **Predicate:** \texttt{sname}?

\(\text{sname}\) (note the capitalization) may be used in contracts.
(make-sname v_1 ... v_n) is a value.

The substitution rule for the $i$th selector is:

$$(\text{sname-fname}_i (\text{make-sname } v_1 ... v_i ... v_n)) \Rightarrow v_i.$$ 

Finally, the substitution rules for the new predicate are:

$$(\text{sname? } (\text{make-sname } v_1 ... v_n)) \Rightarrow \text{true}$$

$$(\text{sname? } V) \Rightarrow \text{false} \text{ for } V \text{ a value of any other type.}$$
The template function for a structure simply selects all its fields, in the same order as listed in the `define-struct`. For example,

```
(define-struct inventory (desc price available))
;; an Inventory is a (make-inventory Str Num Nat)

(define (inventory-template item)
  ( ... (inventory-desc item)
    ... (inventory-price item)
    ... (inventory-available item) ... ))
```

The above (structure definition, data definition, and template function) are only required *once per file*.
Racket provides predicates such as `number?` and `symbol?` to identify data types.

`define-struct` also defines a predicate that tests whether its argument is that type of structure (e.g. `std?`).

We can use these to check aspects of contracts and to write functions that consume mixed data – data of several (probably related) types.

**Example**: A university has (undergraduate) students as well as graduate students. Graduate students are like other students except that they also have a supervisor. Some courses may have both kinds of students.
Data definitions

(define-struct ustd (name prog classes))
;; A UStd (undergraduate student) is a (make-ustd Str Str (listof Str))

(define-struct gstd (name prog supervisor classes))
;; A GStd (graduate student) is a (make-gstd Str Str Str (listof Str))

;; A Student is one of:
;; * a UStd
;; * a GStd

;; A Classlist is a (listof Student)

There is no structure definition for mixed data. There is, however, a data definition that describes the data and gives a name that can be used in contracts.
The template function for mixed data will determine the type of the data and then include a template for that type.

;; student-template:  Student -> Any
(define (student-template s)
  (cond [(ustd? s) (... (ustd-name s)...
      (ustd-prog s) ...
      (ustd-classes s)...)]
[[(gstd? s) (... (gstd-name s)...
      (gstd-prog s)...
      (gstd-supervisor s)...
      (gstd-classes s)...)])})
We don’t have to use structures. We could construct a class list with simple lists:

```
(define cs135/s (list
    (make-ustd "AJ" "CS" (list "CS 488" "CS 449"))
    (make-gstd "Jo" "CS" "Ian" (list "CS 688" "CS 749"))
    (make-ustd "Di" "Math" (list "CS 488" "PMATH 330"))))
```

```
(define cs135/l (list
    (list "AJ" "CS" (list "CS 488" "CS 449"))
    (list "Jo" "CS" "Ian" (list "CS 688" "CS 749"))
    (list "Di" "Math" (list "CS 488" "PMATH 330"))))
```

What are the advantages and disadvantages of each approach?
Structures vs. lists

Structures:
- help avoid some programming errors (e.g. extracting the wrong field)
- provide meaningful names that are easier to read and understand.
- automatically generate significant code (including a predicate).

Lists:
- make it possible to write “generic” functions that operate on several types of data (e.g. \( \text{first} \ s \) will extract the name for both undergraduates and graduates; with structures you need to use \text{cond} first).
- can be expressed more compactly than structures.
Do we need to extend the language?

Mathematicians and computer scientists like to imagine the simplest system they can. Adding structures seems to make our language more complex. So let’s do a though experiment:

If Racket did not already have structures built-in, could we simulate this feature?

We can already store as much information as we want in a list. So for each structure, instead of using the \texttt{define-struct} special form, we need only create the \( n + 2 \) functions:

1. a constructor function,
2. a type predicate,
3. \( n \) selector functions.
Imaging we want to “simulate” the code:

```scheme
(define-struct std (name prog classes))
```

We need to create:

1. `make-std`, that consumes 3 values (one per field)
2. `std?`, a predicate that can determine if a value is the right type
3. `std-name`, `std-prog`, and `std-classes`, to select the value of each field.
;; A Std (student) is a (make-std Str Str (listof Str))

;; A large "random" value to check for legit student values
(define STD-TAG "std_391249569284455218")

;; (make-std name prog classes) makes a new student structure
;; containing the name, program and classes for the student.
;; make-std: Str Str (listof Str) -> Std
(define (make-std name prog classes)
  (list STD-TAG name prog classes))

;; A sample student for testing
(define Juan (make-std "Juan" "CS" (list "CS 135" "MATH 137")))
;; (std? v) produces true if v is a Std and false otherwise.
(check-expect (std? Juan) true)
(check-expect (std? (list "Juan" "CS" (list "CS 135" "MATH 137"))) false)
(check-expect (std? "Juan") false)

;; std?: Any -> Bool
(define (std? s)
  (and (cons? s)
       (= (length s) 4)
       (string=? (first s) STD-TAG)))
;; (std-name s) extracts the name field from student s; error
;; if s is not a student
(check-expect (std-name Juan) "Juan")

;; std-name: Std -> Str
(define (std-name s)
  (cond [(std? s) (second s)]
        [else (error "std-name: expects a std, given " s)]))

std-prog and std-classes are nearly identical to std-name.

The built-in structures raise errors when called with invalid values. Try (posn-x 42); what happens? So it seems appropriate for us to do the same. But we could have omitted that.

So yes: we could simulate structures, if we wanted too.
You should be able to write code to define a structure and to use the functions that are defined when you do so.

You should understand the data definitions we have used and be able to write your own.

You should be able to write a structure definition’s template 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 use them to work with mixed data.

You should understand the similar uses of structures and fixed-size lists and be able to write functions that consume either type of data.
In this module we added the following to our toolbox:

```
define-struct
```

These are the functions and special forms currently in our toolbox:

```
```