05: Structures

Compound data

Sometimes data seems to always belong together. For example,

- A point on a plane always has both x and y values.
- A book is characterized by an ID, title, author, and genre.

Some other examples of compound data:

A complex number

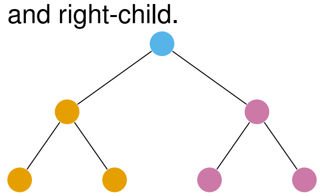
$$z = a + bi$$

is built of a real part a and an imaginary part b.

An employment record might include the name, ID number, and unit.

```
{
 name: "James Bond"
 ID: 007
 unit: "MI6"
}
```

A *labelled rooted binary tree* has a label, left-child and right-child.



Structures

Racket represents the concept of **compound data** (data that has several parts) with **structures**.

Think of a structure as a box representing one thing (such as the employee, James Bond) but with several parts (**fields**) inside the box: name, ID, organizational unit, etc. Each of those fields has a name.

We can have many structures, each representing a different employee – one structure for James Bond, another structure for Bond's boss, "M", etc.

Racket has a general mechanism, `define-struct`, that allows us to define custom structures for employees or whatever structures the program needs.

Example: Rectangle

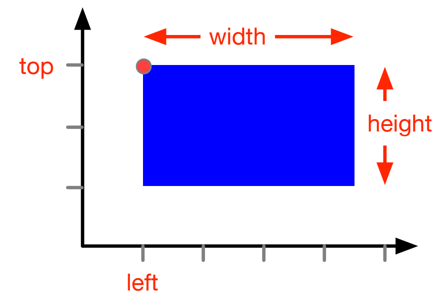
A positioned rectangle can be characterized by it's top-left corner, width, height, and colour.

Define a structure for a rectangle with:

```
(define-struct rect (top left width height colour))
```

`define-struct` is a special form. It consumes the name of the structure (`rect` in our case) and a list of field names.

`define-struct` creates a number of functions that operate on the structure. The first is `make-rect`. We can use it to make a rectangle corresponding to the picture.




```
;; Example rectangles:
(define blue-rect (make-rect 3 1 3.5 2 'blue))
(define unit-rect (make-rect 1 0 1 1 'red))
```


Example: Consuming a Rectangle

Now we can write a function that consumes an entire rectangle structure and produces, say, the area of the rectangle:

```
;; (rect-area r) produces the area of rectangle r.
(check-expect (rect-area blue-rect) 7)
(check-expect (rect-area unit-rect) 1)

;; rect-area: Rect -> Num
(define (rect-area r) ...)
```

Note that `rect-area` is consuming a *single* parameter, a structure, that includes all the information about the rectangle.

Example: Consuming a Rectangle

```
(define-struct rect (top left width height colour))
```

Makes a total of 7 functions for us:

- The **constructor** function, `make-rect`.
- Five **selector functions**, one for each field: `rect-top`, `rect-left`, `rect-width`, `rect-height`, and `rect-colour`.
- A type predicate, `rect?`, which produces `true` if and only if its argument is a `Rect`.

We can use the selector functions to finish `rect-area`:

```
;; rect-area: Rect -> Num
(define (rect-area r)
  (* (rect-width r)
     (rect-height r)))
```

Example: Producing a Rectangle

Write `(rect-move r dx dy)` which moves rectangle `r` the distance given by `dx` and `dy`.

```
;; (rect-move r dx dy) produces a new rectangle that is the
;;    same as r but offset or moved by dx and dy.
;; Examples:
(check-expect (rect-move blue-rect 1 2)
              (make-rect 5 2 3.5 2 'blue))
(check-expect (rect-move unit-rect 1 1)
              (make-rect 2 1 1 1 'red))

;; rect-move: Rect Num Num -> Rect
(define (rect-move r dx dy)
  (make-rect (+ (rect-top r) dy)
             (+ (rect-left r) dx)
             (rect-width r)
             (rect-height r)
             (rect-colour r)))
```

Tracing `rect-move`


```
⇒ (rect-move (make-rect 1 2 3 (+ 2 2) 'red) 5 6)
⇒ (rect-move (make-rect 1 2 3 4 'red) 5 6)
⇒ (make-rect (+ (rect-top (make-rect 1 2 3 4 'red)) 6)
             (+ (rect-left (make-rect 1 2 3 4 'red)) 5)
             (rect-width (make-rect 1 2 3 4 'red))
             (rect-length (make-rect 1 2 3 4 'red))
             (rect-colour (make-rect 1 2 3 4 'red)))
⇒ (make-rect (+ 1 6)
             (+ (rect-left (make-rect 1 2 3 4 'red)) 5)
             (rect-width (make-rect 1 2 3 4 'red))
             (rect-length (make-rect 1 2 3 4 'red))
             (rect-colour (make-rect 1 2 3 4 'red)))
```

```
⇒ (make-rect 7
             (+ (rect-left (make-rect 1 2 3 4 'red)) 5)
             (rect-width (make-rect 1 2 3 4 'red))
             (rect-length (make-rect 1 2 3 4 'red))
             (rect-colour (make-rect 1 2 3 4 'red)))
⇒ (make-rect 7 (+ 2 5)
             (rect-width (make-rect 1 2 3 4 'red))
             (rect-length (make-rect 1 2 3 4 'red))
             (rect-colour (make-rect 1 2 3 4 'red)))
⇒ (make-rect 7 7
             (rect-width (make-rect 1 2 3 4 'red))
             (rect-length (make-rect 1 2 3 4 'red))
             (rect-colour (make-rect 1 2 3 4 'red)))
⇒ (make-rect 7 7 3
             (rect-length (make-rect 1 2 3 4 'red))
             (rect-colour (make-rect 1 2 3 4 'red)))
⇒* (make-rect 7 7 3 4 'red)
```

Syntax and semantics

The special form

`(define-struct sname (fname_1 ... fname_n))`

defines the structure type `sname` with **fields** `fname_1` to `fname_n`. It also automatically defines the following primitive functions:

- **Constructor:** `make-sname`
- **Selectors:** `sname-fname_1 ... sname-fname_n`
- **Predicate:** `sname?`

`Sname` (note the capitalization) may be used in contracts.

Substitution rules

`(make-sname v_1 ... v_n)` is a value.

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

`(sname-fname_i (make-sname v_1 ... v_i ... v_n))` ⇒ `v_i`.

Finally, the substitution rules for the new predicate are:

`(sname? (make-sname v_1 ... v_n))` ⇒ `true`
`(sname? V)` ⇒ `false` for `V` a value of any other type.

Design Recipe of custom structures

A **`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:

```
(define-struct rect (top left width height colour))
;; A Rect is a (make-rect Num Num Num Num Sym)
;; Requires: width and height are non-negative
```

The data definition tells us:

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

The **`define-struct`** and the data definition are distinct from each other (one is for Racket; one is for us) but belong together.

Ex. 1

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.

Ex. 2

Just after your `(define-struct book ...)` line, write a data definition for a `Book`.

Templates and data-directed design

One of the main ideas in CS135 is that the form of a program often mirrors the form of the data.

A template is a general framework within which we fill in details for a specific function.

We create a template once for each new form of data, and then apply it many times in writing functions that consume that type of data.

A template is derived from a data definition.

Structure templates

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 rect (top left width height colour))
;; A Rect is a (make-rect Num Num Num Num Sym)
;; Requires: width and height are non-negative


;; rect-template: Rect -> Any
(define (rect-template r)
  (... (rect-top r)
       (rect-left r)
       (rect-width r)
       (rect-height r)
       (rect-colour r)))
```

- The structure definition, data definition, and template function are only required *once per file*.
- The ... indicates an omission – a place where code will be added when the template is used for a specific function.
- Code may be needed at other places in the template as well.

Structure templates

Compare `rect-template` to `rect-move`:

```
;; rect-template: Rect -> Any
(define (rect-template r)
  (... (rect-top r)
       (rect-left r)
       (rect-width r)
       (rect-height r)
       (rect-colour r)))
```

```
;; rect-move: Rect Num Num -> Rect
(define (rect-move r dx dy)
  (make-rect (+ (rect-top r) dy)
             (+ (rect-left r) dx)
             (rect-width r)
             (rect-height r)
             (rect-colour r)))
```

Templates:

- Give us a starting point for writing functions.
- Remind us of the information we have available.
- Remind us of the structure of the data.

Another Example: Inventory (1/4)

A program is needed to manage the inventory for a bulk food store. Each item in the inventory has a description, a price, an a quantity available.

```
(define-struct inventory (desc price available))
;; An Inventory is a (make-inventory Str Num Nat)
;; Requires: price >= 0

;; inventory-template: Inventory -> Any
(define (inventory-template item)
  ( ... (inventory-desc item)
        (inventory-price item)
        (inventory-available item)))
```

Another Example: Inventory (2/4)


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

makes 5 functions:

- A constructor: `(make-inventory "dry lentils" 0.79 42)`
- Selector: `(inventory-desc lentils)` ⇒ `"dry lentils"`
- Selector: `(inventory-price lentils)` ⇒ `0.79`
- Selector: `(inventory-available lentils)` ⇒ `42`
- Predicate: `(inventory? lentils)` ⇒ `true`;
  `(inventory? blue-rect)` ⇒ `false`

Another Example: Inventory (3/4)

We can use the template to derive several functions:

```
;; inventory-template: Inventory -> Any
(define (inventory-template item)
  ( ... (inventory-desc item)
        (inventory-price item)
        (inventory-available item)))

;; (total-value item) produces the cost of all our items.
;; Example:
(check-expect (total-value (make-inventory "rice" 5.50 6)) 33.00)

;; total-value: Inventory → Num
(define (total-value item)
  (* (inventory-price item)
     (inventory-available item)))
```

Another Example: Inventory (4/4)

We can use the template to derive several functions:

```
;; inventory-template: Inventory -> Any
(define (inventory-template item)
  ( ... (inventory-desc item)
        (inventory-price item)
        (inventory-available item)))

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

;; raise-price: Num Inventory → Inventory
(define (raise-price dollars item)
  (make-inventory (inventory-desc item)
                  (+ dollars (inventory-price item))
                  (inventory-available item)))
```

Nested Structures

Consider the following structure and data definitions:

```
(define-struct point (x y))
;; A Point is a (make-point Num Num)

(define-struct rect (topleft w h))
;; A Rect is a (make-rect Point Num Num)
;; Requires: w, h >= 0
```

Some names have been shortened to make room on slides later.

How do you make a rectangle?

Templates

What are the templates?

```
(define-struct point (x y))
;; A Point is a (make-point Num Num)

(define-struct rect (topleft w h))
;; A Rect is a (make-rect Point Num Num)
;; Requires: w, h >= 0
```

```
;; point-template: Point -> Any
(define (point-template p)
  (... (point-x p)
       (point-y p)))

;; rect-template: Rect -> Any
(define (rect-template r)
  (... (rect-topleft r)     ;; a Point
       (rect-w r)
       (rect-h r)))
```

Is there more we can do?

Templates: Two options to complete


```
;; point-template: Point -> Any
(define (point-template p)
  (... (point-x p)
       (point-y p)))

;; rect-template-v1: Rect -> Any
(define (rect-template-v1 r)
  (... (point-template (rect-topleft r))
       (rect-w r)
       (rect-h r)))

;; rect-template-v2: Rect -> Any
(define (rect-template-v2 r)
  (... (point-x (rect-topleft r))
       (point-y (rect-topleft r))
       (rect-w r)
       (rect-h r)))
```

```
(define (point-mv p dx dy)
  (make-point (+ (point-x p) dx)
              (+ (point-y p) dy)))

(define (rect-mv-1 r dx dy)
  (make-rect (point-mv (rect-topleft r)
                       dx dy)
             (rect-w r)
             (rect-h r)))

(define (rect-mv-2 r dx dy)
  (make-rect
    (make-point
      (+ (point-x (rect-topleft r)) dx)
      (+ (point-y (rect-topleft r)) dy))
    (rect-w r)
    (rect-h r)))
```

Mixed Data


```
(define-struct point (x y))
;; A Point is a (make-point Num Num)

(define-struct rect (topleft w h))
;; A Rect is a (make-rect Point Num Num)
;; Requires: w, h >= 0

(define-struct circle (centre radius))
;; A Circle is a (make-circle Point Num)
;; Requires: radius >= 0

;; A Shape is one of:
;; * Rect
;; * Circle
```

Mixed Data: Shape template

```
;; shape-template: Shape -> Any
(define (shape-template s)
  (cond [(rect? s)   (... s)]
        [(circle? s) (... s)]))
```

```
;; shape-template: Shape -> Any
(define (shape-template s)
  (cond [(rect? s)   (rect-template s)]
        [(circle? s) (circle-template s)]))
```


Mixed Data: Moving a Shape

```
(define (point-mv p dx dy)
  (make-point (+ (point-x p) dx)
              (+ (point-y p) dy)))

(define (rect-mv r dx dy)
  (make-rect
   (point-mv (rect-topleft r) dx dy)
   (rect-w r)
   (rect-h r)))

(define (circle-mv c dx dy)
  (make-circle
   (point-mv (circle-centre c) dx dy)
   (circle-radius c)))
```

```
;; shape-mv: Shape Num Num -> Shape
(define (shape-mv s dx dy)
  (cond [(rect? s)
         (rect-mv s dx dy)]
        [(circle? s)
         (circle-mv s dx dy)]))

(define r
  (make-rect (make-point 1 2) 3 4))
(define c
  (make-circle (make-point 1 2) 3))
(check-expect
 (shape-mv r 1 2)
 (make-rect (make-point 2 4) 3 4))
(check-expect
 (shape-mv c 1 2)
 (make-circle (make-point 2 4) 3))
```

Ex. 3 Add two more shapes, squares and triangles, to our collection of shapes. Write structure definitions (**`define-struct`**), data definitions ("A ___ is a ___"), and templates. Modify the `Shape` data definition and template appropriately.

Ex. 4 Write a function, `shape-area`, that consumes a `Shape` and produces that shape's area.

anyof types

We had the data definition

```
;; A Shape is one of:
;; * Rect
;; * Circle
```

An alternative is

```
;; A Shape is (anyof Rect Circle)
```

Both of these allow the type name `Shape` to be used in contracts.

If only needed a very few times, one can skip the data definition:

```
;; shape-mv: (anyof Rect Circle) Num Num -> (anyof Rect Circle)
(define (shape-mv s dx dy) ... )
```

Violating contracts and data definitions

The data definition in the following is simply a comment.

```
(define-struct point (x y))
;; A Point is a (make-point Num Num)
```

There is nothing that prevents us from ignoring it:

```
(define misused (make-point "one" 'two))
(point-mv misused 1 2)
```

This causes a run-time error.

Violating contracts and data definitions

Racket does not enforce contracts and data definitions. They are simply comments and are ignored by the machine.

Each value created in the execution of a program has a type (`Int`, `Str`, `Bool`, `Rect`, etc). A function can be applied to value of any type (cool!) but will result in an error *at run time* if the function can't handle the value's type (not cool!). This is known as **dynamic typing**.

Violating contracts and data definitions

Languages with **static typing** (e.g. Scala, C, ...) assign types to data definitions, function results, parameters, etc. as well as values. For example, `point-mv` in Scala:

```
case class Point(x:Double, y:Double)

def pointMv(p:Point, dx:Double, dy:Double):Point =  {
  Point(p.x + dx, p.y + dy)
}
```

The fields in a `Point` are declared to be numbers of a particular type (`Double`). The first parameter for `pointMv` is declared to be a `Point`, etc.

The following program is not legal and would not be allowed to run because `"one"` and `'two` do not have types that match `Double`, as required by `Point`.

```
pointMv(Point("one", 'two), 3, 4)
```

Checked functions

It's possible to add type checking to Racket programs manually:

```
;; checked-make-point: Num Num -> Point
(define (checked-make-point x y)
  (cond [(and (number? x) (number? y))
         (make-point x y)]
        [else (error "checked-make-point: requires numbers for x and y")]))


(check-expect (checked-make-point 1.5 2) (make-point 1.5 2))
(check-error (checked-make-point "two" 2)
             "checked-make-point: requires numbers for x and y")
```

You are *always* welcome to add such checking to your code but are never required to do so unless explicitly specified in a problem statement.

Goals of this module

- 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 template for mixed data and be able to write functions based on it.

The following functions and special forms have been introduced in this module:

`...` **`define-struct`** `error`

You should complete all exercises and assignments using only these and the functions and special forms introduced in earlier modules. The complete list is:

`* + - ... / < <= = > >= abs` **`and`** `boolean? ceiling check-error check-expect check-within` **`cond`** `cos` **`define define-struct`** `e` **`else`** `error even? exp expt floor integer? log max min modulo negative? not number->string number? odd?` **`or`** `pi positive? quotient remainder round sgn sin sqr sqrt string-append string-downcase string-length string-lower-case? string-numeric? string-upcase string-upper-case? string<=? string<? string=? string>=? string>? string? substring symbol=? symbol? tan zero?`