CS115 - Module 4 - Compound data: structures

Cameron Morland

Fall 2017

Reminder: if you have not already, ensure you:

- Read *How to Design Programs*, sections 6-7, omitting 6.2, 6.6, 6.7, and 7.4.

Compound data

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 produce such packages.

A few examples:

A complex number

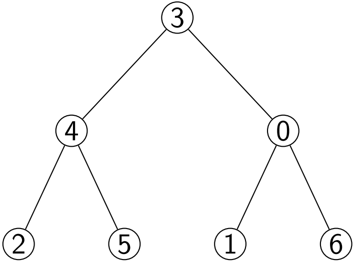
$$z = a + bi$$

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: 00000007
 program: 'pure-math
}
```

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



A Built-in structure: `Posn`

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

```
;; make-posn Num Num -> Posn
(define my-posn (make-posn 4 3))
```

Note here were are storing *two things*, namely the *x* and *y* coordinates, in *one* value, which is a `Posn`.

A Built-in structure: `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` or `Sym`:

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

(define my-sym 'oak)
my-sym ⇒ 'oak
```

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 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.

Type predicates

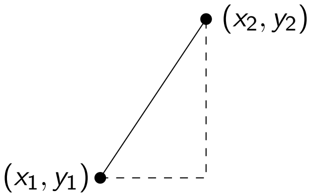
One last function: the **type predicate**.

```
(posn? 42) ⇒ false
(posn? 'oak) ⇒ false
(posn? my-posn) ⇒ true
```

The type predicate produces `true` if its argument is some object of that type.

Recall `(posn-x p)` produces the x coordinate, and `(posn-y p)` produces the y coordinate. So the difference in x coordinate between points `p1` and `p2` is

```
(- (posn-x p1) (posn-x p2))
```




```
;; (distance p1 p2) produce the distance between
;;   p1 and p2.
;; distance: Posn Posn -> Num
;; Example:
(check-expect (distance (make-posn 0 0)
                        (make-posn 3 4)) 5)

(define (distance p1 p2)

...)
```

Complete the function `distance`.

A function may produce a `Posn` just like any other value.

```
;; (offset-a-little x y) produce 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)))
```

Write a function **`+vector`** that consumes two `Posn` and does *vector addition*. (That is, it produces a new `Posn` where the *x* value is the sum of the *x* values, and *y* is the sum of the *y* values.)

Custom structures

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

```
(define-struct polarcoord (r theta))
```

- `polarcoord` is the name of the new structure type
- `(r theta)` are the names of the fields of the structure.

This automatically creates several functions:

Constructor `make-polarcoord` allows us to create values of this type

Predicate `polarcoord?` lets us determine if a value is of this type

Selectors are created, one for each field.
In this example, `polarcoord-r` and `polarcoord-theta`.

```
(define-struct polarcoord (r theta))
```

This `define-struct` 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 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.

Consuming custom structures

Write a function `polarcoord->posn` that consumes a `Polarcoord` and produces the `Posn` corresponding to the same point.
(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)))))
```

Producing custom structures

Write a function `rotate-polar` that consumes a `Polarcoord` and a `Num` and produces a `Polarcoord` modified by rotating it by the `Num`.

```
;; (rotate-polar p angle) produce p rotated by angle.
;; rotate-polar: Polarcoord Num -> Polarcoord
;; Example:
(check-expect (rotate-polar (make-polarcoord 3 0.4) 0.2)
              (make-polarcoord 3 0.6))


(define (rotate-polar p angle)
  (make-polarcoord (polarcoord-r p)
                   (+ angle (polarcoord-theta p))))
```

Misusing `Posn`

What is the result of evaluating the following expression?

```
(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.

Contract Errors

```
;; (scale pt factor) produce 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`.

Static Typing

Many languages have **static typing**. Here the type of every value and parameter is specified as part of the code. If you break the "contract" the code will not compile.

```c
typedef struct {
  float x;
  float y;
} Posn;

float distance (Posn p1, Posn p2);
```

```c
int main(void) {
  Posn p1 = {2.0, 5.0};
  Posn p2 = {3.0, 4.0};

  distance(p1, p2);
}
```

This works fine! But if I add `distance(p1, 3);` it gives this error:

```
distance.c: In function 'main':
distance.c:20:16: error: incompatible type for argument 2 of 'distance'
   distance(p1, 3);
                ^
```

This *can* make it easier to write and debug your code.

Other languages, including Racket, have **dynamic typing**.

This gives certain flexibility:

```
;; (classify n) determine if n is even or negative.
;;   Otherwise produce n.
;; classify: Int -> (anyof Sym Nat)

(define (classify n)
  (cond [(even? n) 'even]
        [(negative? n) 'negative]
        [else n]))

(define (explain n)
  (cond [(symbol? (classify n)) 'even-or-negative]
        [else 'positive-and-odd]))
```

Here a function can produce or consume values of different types in different contexts.

This *can* make it easier to write and debug your code.

Dealing with dynamic typing

While Racket does not enforce contracts, we will always assume that contracts are followed.

> Never call a function with arguments that violate the contract and requirements. If you desire to use one of your own helper functions in a way that violates its contract, that likely means you should modify its contract!

One of the ideas of the HtDP textbook is that the form of a program may mirror the form of the data.

A **template** is a general framework which we will complete with specifics. It is a starting point for our implementation.

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.

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

```
;; template for a function that
;;   consumes a Student.

;; my-student-fn: Student -> Any
(define (my-student-fn s)
  (...(student-name s)...
```

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

Suppose I have a complicated structure:

```
(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?!?

```
;; 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. Structures in Racket 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

```
house.cat = newcat
```

Additions to syntax for structures

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 `Num`, `Str`, `Sym`, 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.

```
(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 x-axis.
```

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

```
(define (my-polarcoord-fn 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.

Module summary

Become comfortable using structures: using the built-in `Posn` structure, and making your own structures using `define-struct`.

When working with your custom structures, understand how to use the constructor function, the type predicate, and the selector functions.

Before we begin the next module, please

- Read *How to Design Programs*, sections 9 and 10.