Programming Abstractions

Lecture 14: Structs and keyword arguments

Stephen Checkoway

Data types

We need a way to store some fields and procedures to create and work with instances of the type

Procedures

- Recognizers: Is this thing an object of type X?
- Constructors: Create an object of type X
- Accessors: Get field Y from an object of type X

Imagine you have a point data type with this constructor.

```
(define (point x y)
  (list x y))
```

What is wrong with this constructor, if anything?

A. The result cannot be distinguished from a normal list

B. (point x y) should return a closure (a lambda), not a list

C. `(list x y)` should be `'(x y)`

D. A and C

E. The constructor is correct

Imagine you have a point data type with this constructor and recognizer.

```
(define (point x y)
  (list 'point x y))

(define (point? obj)
  (eq? (first obj) 'point))
```

What is wrong with this recognizer?

A. It doesn't always return `#t` when passed a point

B. It doesn't always return `#f` when passed something other than a point

C. `eq?` should be `equal?`

D. A and B

E. B and C

Imagine you have a point data type with this constructor and accessor.

```
(define (point x y)
  (list 'point x y))

(define (point-x p)
  (second p))
```

What is wrong with this accessor, if anything?

A. It doesn't return the x field of a point

B. When called with something that's not a point, it gives an error rather than returning #f

C. When called with something that's not a point, it doesn't give an error

D. More than one of A, B, or C

E. Nothing is wrong with it

Example from last time: set

```
(define (set elements)
  (list 'set (remove-duplicates elements)))

(define (set? obj)
  (and (list? obj)
       (not (empty? obj))
       (eq? (first obj) 'set)))

(define (empty-set? obj)
  (and (set? obj)
       (empty? (second obj))))

(define (set-elements s)
  (if (set? s)
      (second s)
      (error 'set-elements "~v is not a set" s)))

(define empty-set (set empty))
```

Example: point

```
(define (point x y)
  (list 'point x y))
(define (point? obj)
  (and (list? obj)
       (not (empty? obj))
       (eq? (first obj) 'point)))
(define (point-x p)
  (cond [(set? p) (second p)]
        [else (error 'point-x "~v is not a point" p)]))
(define (point-y p)
  (cond [(set? p) (third p)]
        [else (error 'point-y "~v is not a point" p)]))
```

Too much repetitive code to write by hand

`(struct name (field-a field-b) …)`

Racket has a very general mechanism for creating structures and the associated procedures

To create our point data type, we can instead use

```
(struct point (x y))
```

This will create a new type named point and the following procedures:

- `(point x y)` produces a new point with the given coordinates
- `(point? obj)` returns `#t` if `obj` is a `point`
- `(point-x p)` returns the `x` field
- `(point-y p)` returns the `y` field

Example point

```
(struct point (x y))

(define p (point 3 4))

(point? p) ; returns #t
(point? '(point 3 4)) ; returns #f
(point-x p) ; returns 3
(point-y p) ; returns 4
p ; DrRacket prints this as #<point>
```

1st problem: Hard to debug

```
(define (thing p)
  (cond [(negative? (point-x p))
         (error 'thing "Invalid point: ~s" p)]
        [else '...]))

(thing (point -3 2)) => thing: Invalid point: #<point>
```

2nd problem: Equality isn't structural equality

```
; With lists, equal? performs structural comparison
(equal? '(point 3 4) '(point 3 4)) => #t

; eq? asks if the arguments are the same object
(eq? '(point 3 4) '(point 3 4)) => #f

; With structs, equal? acts like eq? by default!
(equal? (point 3 4) (point 3 4)) => #f
```

Solve both by making the struct transparent

```
(struct point (x y) #:transparent)

(point 3 4) => (point 3 4) rather than #<point>

(equal? (point 3 4) (point 3 4)) => #t
```

`#:transparent` is a **keyword argument**

Aside: Keyword arguments

Procedures can take keyword arguments

Keyword argument are specified as `#:name value`

For example, sort has 2 required positional arguments and 2 optional keyword arguments

```
(sort  lst                                                    procedure
       less-than?
      [#:key extract-key
       #:cache-keys? cache-keys?]) → list?
  lst : list?
  less-than? : (any/c any/c . -> . any/c)
  extract-key : (any/c . -> . any/c) = (lambda (x) x)
  cache-keys? : boolean? = #f
```

Keyword arguments

Keyword arguments can be given in any order

- `(foo 4 #:thing 8 10)` and `(foo #:thing 8 4 10)` are the same
  - Positional arguments are 4 and 10
  - Keyword argument `#:thing` with value 8

Keyword arguments can have default values

- Keyword arguments almost always have default values
- For sort, the `#:key` keyword has a default value of `(lambda (x) x)` and `#:cache-keys?` has default value `#f`

Sort example

```
(sort '(1 5 3 4) <) => '(1 3 4 5)

(sort (list (point 1 2) (point 0 5) (point 1 -1))
      <
      #:key point-x)
=> (list (point 0 5) (point 1 2) (point 1 -1))
```

This is equivalent to

```
(sort (list (point 1 2) (point 0 5) (point 1 -1))
      (λ (a b) (< (point-x a) (point-x b))))
```

Special forms can have keyword arguments

`struct` supports a variety of keyword arguments, including `#:transparent`

In some cases, the keyword arguments don't need values, they are aliases for other keywords with specific values

For `struct`, `#:transparent` is the same as `#:inspector #f`

tree.rkt

tree.rkt

```
#lang racket

; Provide the procedures for working with trees.
(provide tree make-tree empty-tree
         tree? empty-tree? leaf?
         tree-value tree-children)

; Provide 8 example trees.
(provide empty-tree T1 T2 T3 T4 T5 T6 T7 T8)
```

Tree definition and a special value

```
; Definition of tree datatype
(struct tree (value children) #:transparent)

; An empty tree is represented by null
(define empty-tree null)

; (empty-tree? empty-tree) returns #t
(define empty-tree? null?)

; Convenience constructor
; (make-tree v c1 c2 ... cn) is equivalent to
; (tree v (list c1 c2 ... cn))
(define (make-tree value . children)
  (tree value children))
```

Utility procedure

```
; Returns #t if the tree is a leaf.
(define (leaf? t)
  (cond [(empty-tree? t) #f]
        [(not (tree? t)) (error 'leaf? "~s is not a tree" t)]
        [else (empty? (tree-children t))]))
```

Example trees

```
(define T1 (make-tree 50))
(define T2 (make-tree 22))
(define T3 (make-tree 10))
(define T4 (make-tree 5))
(define T5 (make-tree 17))
(define T6 (make-tree 73 T1 T2 T3))
(define T7 (make-tree 100 T4 T5))
(define T8 (make-tree 16 T6 T7))
```

A tree is represented as a struct `(tree value children)`.

If you want to count how many children a particular (nonempty) tree `t` has, what's the best way to do it?

A. `(length (tree-children t))`

B. `(length (third t))`

C. `(length (rest t))`

D. `(length (rest (rest t)))`

E. `(length (caddr t))`

Example: leaves

Let's write `(leaves t)` that takes a tree as input and returns a list of the values of its leaves