CSCI 275: Programming Abstractions

Lecture 12: Structs & Data Types

Spring 2024

Stephen Checkoway, Oberlin College
Slides gratefully borrowed from Molly Q Feldman

Questions? Concerns?

What are some of the benefits of defining our own data types? What is an example of a data type you defined recently (in a class, internship, personal project, etc.)?

Constructing data types!

What else! ☺

- We're going to construct data types out of lists
- The first element in the list is going to be a symbol that's the name of the data type
- The other elements in the list will be the fields of the data type

What do we need to implement a data type?

***Representation for the Data Type:* a list with a particular structure**

Recognizers	A way to test whether a thing is an object of type *X*?
Constructors	A way to create an object of type *X*
Accessors	A way to get field *Y* from an object of type *X*

Running Example: set

A `set` data type which will hold a (mathematical) set of values for us

Some example sets:

`{}`

`{1,2,3}`

`{a,b,y,z}`

Important attribute of mathematical sets:
no duplicates!

Representation

- We're going to use lists to represent instances of a data type
- Our set will contain just a single field: the elements the set contains

Empty Set

```
'(set ())
```

Name of the data type is the first entry

Non-Empty Sets

```
`(set (1 3 5 7 9))
`(set (a))
`(set (x z y))
```

Recognizers

Recognizers are procedures that return `#t` or `#f` corresponding to whether or not the passed in object *is of the appropriate type*

Analogous to `number?` and `list?`

There are also recognizers that return `#t` or `#f` corresponding to whether or not the passed in object *has a particular value of the type*

Analogous to `zero?` and `empty?`

Recognizers for our set data type

We want to know if a particular object is a set, so we'll write a procedure `set?`

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

This is analogous to `list?` except it returns `#t` if the object is a set

Just as `(empty? x)` returns `#t` if `x` is an empty list, let's write `(empty-set? x)` which returns `#t` if `x` is an empty set.

Remember, we're representing a set as a 2-element list where the first is `'set` and the second is the list of elements. How do we do this?

```
A. (define (empty-set? obj)
     (empty? obj))
B. (define (empty-set? obj)
     (and (= (first obj) 'set)
          (empty? (second obj))))
C. (define (empty-set? obj)
     (and (set? obj)
          (empty? (second obj))))
D. Any of A, B, or C
E. Either B or C
```

Constructors

Now that we know how to recognize if something is an instance of our data type, we need procedures to create them

Typically, we use the name of the data type itself

Example:

- To create a set, we need a list of elements
- The list might have duplicates, so we should remove those

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

Special value for our set data type

Just as list has a special value, empty, it might be nice to have an empty-set

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

Accessors

We need a way to access the fields of an instance of our data type

For our set example, we have only a single field: a list of elements

- Therefore, we only need a single accessor: `set-elements`

If we had more fields, we'd need more accessors

- `(point x y)` needs two accessors: `point-x` and `point-y`
- `(student name t-number year)` needs 3

Note, pay attention to the naming here: hyphens are very Racket style, they'll also appear in a related idea later

Set accessor

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

There are multiple forms of the (error …) procedure, this one is
`(error procedure-name format-string arguments)`

The `~v` means to substitute a string representation of the object for the `~v`

```
> (set-elements '(1 2 3))
set-elements: '(1 2 3) is not a set
```

Complete data type example: 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))
```

Additional procedures

```
(define (set-contains? x s)
  (member x (set-elements s)))

(define (set-insert x s)
  (if (set-contains? x s)
    s
    (list 'set (cons x (set-elements s)))))

(define (set-union s1 s2)
  (foldl set-insert s1 (set-elements s2)))
```

A set module

```
#lang racket

(provide set set? empty-set? set-elements)
(provide set-contains? set-insert set-union)
(provide empty-set)

...
```

Make the definitions available to use by others!

Imagine you have a point data type with this constructor.

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

Why is this constructor for a point data type not great?

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)
  (equal? (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. `equal?` should be =
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 always give an error
D. More than one of A, B, or C
E. Nothing is wrong with it

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 [(point? p) (second p)]
        [else (error 'point-x "~v is not a point" p)]))

(define (point-y p)
  (cond [(point? 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 data structures and their 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

Next Up

Homework 2 is due **tonight**

Homework 3 will be posted **later today**