Programming Abstractions

Lecture 12: Modules and data types

Stephen Checkoway

Modules in Racket

(Lack of) modules in Scheme

Traditional

Scheme has a `(load file-name)` that's like C's `#include`

- `(load "foo.scm")` simply reads in the content of the file `foo.scm` as scheme code

Upsides

- It's simple: It simply makes all of the definitions in the loaded file available in the current file

Downsides

- Only a single namespace so different files can't have procedures with the same name
- Allows no separation between an interface and an implementation
  - E.g., "private" helper functions aren't actually private

Modules in Racket

Modern

Each file that starts with `#lang` creates a module named after the file

`#lang` also specifies the language of the file

- Racket was designed to implement programming languages
- We're only going to use the Racket language itself
- All of our files start with `#lang racket`

Exposing definitions

`(provide …)`

By default, each definition you make in a Racket file is private to the file

To expose the definition, you use `(provide …)`

To expose all definitions, you use
`(provide (all-defined-out))`

E.g.,
```
#lang racket
(provide (all-defined-out))

(define (mul2 x)
  (* x 2))
```

This explains the line in the homework

```
; Export all of our top-level definitions so that tests.rkt
; can import them. See tests.rkt.
(provide (all-defined-out))
```

Exposing only some definitions

`(provide sym1 sym2…)`

You can specify exactly which definitions are exposed by specifying them via one or more `provides`

```
#lang racket
(provide foo-a foo-b)
(provide bar-a bar-b)

(define helper …)   ; Not exposed

(define foo-a …)
(define foo-b …)

(define bar-a …)
(define bar-b …)
```

Importing definitions from modules

`(require …)`

To get access to a module's definitions we need to `require` the module

E.g., the `test.rkt` files in the assignments require the homework file

- `(require "hw1.rkt")` imports the definitions from the file `hw1.rkt`

Imagine you're writing code to analyze DNA sequence data so you create a Racket file `analysis.rkt`. You write two procedures `dna-match` and `edit-distance`. The `edit-distance` procedure is just a helper procedure and is only used inside of `dna-match`. What code should you add to `analysis.rkt` to expose the appropriate procedures?

A. `(provide (all-defined-out))`

B. `(provide dna-match)`

C. `(provide edit-distance)`

D. `(hide edit-distance)`

E.
```
(provide dna-match)
(hide edit-distance)
```

Now, you're writing another module (i.e., another file) that wants to use the `dna-match` procedure in the `analysis.rkt` file. What code should you use to make that procedure available for use inside the current module?

A. `(import dna-match)`

B. `(require dna-match)`

C. `(import "analysis.rkt")`

D. `(require "analysis.rkt")`

E. `(load "analysis.rkt")`

Plus a whole lot more

Racket supports a dizzying array of module options

`#lang racket` is a shorthand for `(module module-id racket …)`

Submodules can be created using a variety of module forms: `module`, `module*`, `module+`

Requiring modules can

- Import specific symbols
- Exclude specific symbols
- Rename symbols (e.g., two modules can be imported with conflicting names)

We won't need any of this extra functionality in the course, because our programs are so short and we don't need it

Data types

Data types

We're going to construct data types out of lists (of course)

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?

A representation for the data type: a list with a particular structure

Procedures to work with an instance of the data type

- 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

Since we're working functionally, we don't need to *set* any fields, we would just create a new object with the appropriate field

Representation

We're going to use lists to represent instances of a data type

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

Our set will contain just a single field: the elements of the set

Empty set: let's use the list `'(set ())`

Nonempty set: let's take the list of elements (with no duplicates)

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

Note: We use the name of the data type as the first element

Recognizers

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

- These are 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

- These are 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

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 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)))))
; We could have used (set (cons x (set-members s))) too

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

; And so on. Note that these aren't super efficient
```

A set module

```
#lang racket

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

…
```