Programming Abstractions

Week 6-1: 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. `(import-all "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

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

Depending on the specific data type, the other elements in the list will either be the elements that comprise an instance or 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

Special values (analogous to empty for lists) (not applicable to all data types)

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

Empty set: let's use the list `'(set)` (equivalently `(list 'set)`)
Nonempty set: let's take the list of elements (with no duplicates) and cons `'set` on the front

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

Using the name of the data type as the first element of the list is traditional

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 list starting with 'set where the rest of the elements of the list are elements of the set. How do we do this?

A.
```
(define (empty-set? obj)
  (= (length obj) 1))
```

B.
```
(define (empty-set? obj)
  (and (= (first obj) 'set)
       (empty? (rest obj)))))
```

C.
```
(define (empty-set? obj)
  (and (set? obj)
       (empty? (rest 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)
  (cons '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 sub-objects of an instance of our data type

If we think about Python or Java objects, they have fields. We can do something similar in Racket (and will in just a few slides) and we'll need procedures to access them

For our set example, we have a list of elements and we would like to return that list

Set accessor

```
(define (set-members s)
  (if (set? s)
      (rest s)
      (error 'set-members "~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-members '(1 2 3))
set-members: '(1 2 3) is not a set
```

Example: set

```
(define (set elements)
  (cons '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? (rest obj))))

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

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

Additional procedures

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

(define (set-insert x s)
  (if (set-contains? x s)
    s
    (cons 'set (cons x (set-members s)))))
; We could have used (set (cons x (set-members s))) too

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

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

A set module

```
#lang racket

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

…
```

Before we continue…

`(struct name field-a field-b…)`

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

We're not going to use it in this course because it's worth learning how we can do this all ourselves

If you were writing production Racket code, you would definitely want to use this rather than doing it yourself!

Data type with fields

Let's create a type to represent a course

```
(define (course dept num name)
  (list 'course dept num name))

(define (course? c)
  (and (list? c) (not (empty? c)) (eq? (first c) 'course)))

(define (course-dept c)
  (if (course? c)
      (second c)
      (error 'course-dept "~v is not a course" c)))
```

```
(define (course-num c)
  (if (course? c)
      (third c)
      (error 'course-num "~v is not a course" c)))

(define (course-name c)
  (if (course? c)
      (fourth c)
      (error 'course-name "~v is not a course" c)))
```

TreeDatatype.rkt

```
#lang racket
; Definition of tree datatype
(provide tree make-tree empty-tree
         tree? empty-tree? leaf?
         tree-value tree-children)
(provide empty-tree T1 T2 T3 T4 T5 T6 T7 T8)
```

Tree constructors and a special value

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

; Create a tree with the given value and children.
(define (tree value children)
  (list 'tree value children))

; 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))
```

Recognizers

```
; Returns #t if t is an empty tree.
(define (empty-tree? t)
  (null? t))

; Returns #t if t is a tree (either empty or not).
(define (tree? t)
  (cond [(empty-tree? t) #t]
        [(list? t) (eq? (first t) 'tree)]
        [else #f]))

; 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))]))
```

Accessors

```
; Returns the tree's value. t must be a nonempty tree.
(define (tree-value t)
  (cond [(empty-tree? t)
         (error 'tree-value "argument is an empty tree")]
        [(tree? t) (second t)]
        [else (error 'tree-value "~s is not a tree" t)]))

; Returns the tree's children. t must be a nonempty tree.
(define (tree-children t)
  (cond [(empty-tree? t)
         (error 'tree-children "argument is an empty tree")]
        [(tree? t) (third t)]
        [else (error 'tree-children "~s is not a tree" 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 list ('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