CS450

Structure of Higher Level Languages

Lecture 5: Modules, structs, updating lists, exercises

Tiago Cogumbreiro

Modules

Modules

- Modules encapsulate a unit of functionality
- A module groups a set of constants and functions
- A module encapsulates (hides) auxiliary top-level functions
- Each file represents a module

Modules in Racket

> Each file represents a module. A bindings becomes visible through the `provide` construct. Function `(require "filename")` loads a module

- `(provide (all-defined-out))` makes all bindings visible
- `(provide a c)` makes binding `a` and `c` visible
- `(require "foo.rkt")` makes all bindings of the module in file `foo.rkt` visible in the current module. Both files have to be in the same directory.

File: `foo.rkt`

```
#lang racket
; Make variables a and c visible
(provide a c)
(define a 10)
(define b (+ a 30))
(define (c x) b)
```

File: `main.rkt`

```
(require "foo.rkt")
(c a)
; b is not visible
```

Revisiting user data structures

User data structures

Recall the 3D point from Lecture 3

```
; Constructor
(define (point x y z) (list x y z))
; Accessors
(define (point-x pt) (first pt))
(define (point-y pt) (second pt))
(define (point-z pt) (third pt))
```

And the `name` data structure

```
; Constructor
(define (name f m l) (list f m l))
; Accessor
(define (name-first n) (first n))
(define (name-middle n) (second n))
(define (name-last n) (third n))
```

How do we prevent such errors?

```
(define p (point 1 2 3))
(name-first p) ; This should be an error, and instead it happily prints 1
```

Introducing `struct`

```
#lang racket
(require rackunit)
(struct point (x y z) #:transparent)
(define pt (point 1 2 3))
(check-equal? 1 (point-x pt))  ; the accessor point-x is automatically defined
(check-equal? 2 (point-y pt))  ; the accessor point-y is automatically defined
;
(struct name (first middle last))
(define n (name "John" "M" "Smith"))
(check-equal? "John" (name-first n))
(check-true (name? n))    ; We have predicates that test the type of the value
(check-false (point? n)) ; A name is not a point
(check-false (list? n))  ; A name is not a list
; (point-x n) ;; Throws an exception
; point-x: contract violation
;   expected: point?
;   given: #<name>)
```

Benefits of using structs

- Reduce boilerplate code
- Ensure type-safety

Implementing Racket's AST

Implementing Racket's AST

Grammar

```
expression = value | variable | apply | define
value = number | void | lambda
apply = ( expression+ )
lambda = ( lambda ( variable* ) term+)
```

Implementing values

```
value = number | void | lambda
lambda = ( lambda ( variable* ) term+)
```

Implementing values

value = *number* | *void* | *lambda*
lambda = `( lambda` `(` *variable** `)` *term+*`)`

```
(define (r:value? v)
  (or (r:number? v)
      (r:void? v)
      (r:lambda? v)))
(struct r:void () #:transparent)
(struct r:number (value) #:transparent)
(struct r:lambda (params body) #:transparent)
```

- We are using a prefix `r:` because we do not want to redefined standard-library definitions.

Implementing expressions

```
expression = value | variable | apply
apply = ( expression+ )
```

Implementing expressions

expression = *value* | *variable* | *apply*
apply = `(` *expression+* `)`

```
(define (r:expression? e)
  (or (r:value? e)
      (r:variable? e)
      (r:apply? e)))
(struct r:variable (name) #:transparent)
(struct r:apply (func args) #:transparent)
```

> In `r:apply` we distinguish between the expression that represents the function `func`, and the (possibly empty) list of arguments `args`.

Implementing terms

```
term = define | expression
define = ( define identifier expression ) | ( define ( variable+ ) term+)
```

Implementing terms

term = *define* | *expression*
define = `( define` *identifier* *expression* `)` | `( define (` *variable+* `)` *term+*`)`

```
(define (r:term? t)
  (or (r:define? t)
      (r:expression? t)))
(struct r:define (var body) #:transparent)
```

> For our purposes of defining the semantics in terms of implementing an interpreter, we do not want to distinguish between a basic definition and a function definition, as this would unnecessarily complicate our code. We, therefore, represent a definition with a single structure, which pairs a variable and an expression (eg, a lambda). In our setting, the distinction between a basic and a function definition is syntactic (not semantic).

Summary of `struct`

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

Simplifies the definition of data structures:

- Creates selectors automatically, eg, `point-x`
- Creates type query, eg, `point?`
- Ensures that functions of a given struct can only be used on values of that struct.
  Because, not everything is a list.

> What is `#:transparent`? A transparent struct prints its contents when rendered as a string.

UMass Boston

Functional pattern:

Updating elements

Convert a list from floats to integers

Spec

```
(require rackunit)
; Supplied by the stdlib
(check-equal? 3 (exact-floor 3.14))
(check-equal?
  (list 1 2 3)
  (list-exact-floor (list 1.1 2.6 3.0)))
```

Convert a list from floats to integers

Spec

```
(require rackunit)
; Supplied by the stdlib
(check-equal? 3 (exact-floor 3.14))
(check-equal?
  (list 1 2 3)
  (list-exact-floor (list 1.1 2.6 3.0)))
```

Solution

```
(define (list-exact-floor l)
  (cond [(empty? l) l]
        [else
          (cons
            (exact-floor (first l))
            (list-exact-floor (rest l)))]))
```

> Can we generalize this for any operation on lists?

```
(check-equal?
  (list-exact-floor (list 1.1 2.6 3.0))
  (list (exact-floor 1.1) (exact-floor 2.6) (exact-floor 3.0)))
```

Function map

Generic solution

```
(define (map f l)
  (cond [(empty? l) l]
        [else (cons (f (first l)) (map f (rest l)))]))
```

Using map

```
(define (list-exact-floor l)
  (map exact-floor l))
```

map

Overview of our solution

> Recursive code mirrors the structure your data!

Think of how many constructors your data has, those will be your recursive cases.

- Case `(list)`: the `empty` list constructor
- Case `(list h l ...)`: add one element to the list with the `(cons x l)` constructor
- Recursive call must handle "smaller" data
  - with lists: `(rest l)`
  - with numbers: `(+ n 1)` if you approach an upper bound
  - with numbers: `(- n 1)` if you approach a lower bound

A general recursion pattern for handling lists

1. Case `(list)` (handle-base)
2. Case `(list h l ...)` (handle-step)
3. Recursive call handles "smaller"

```
(define (rec l)
  (match l
    [(list)           handle-base]
    [(list h l ...) (handle-step h (rec l))]))
```

A general recursion pattern for handling lists

1. Case `(list)` (handle-base)
2. Case `(list h l ...)` (handle-step)
3. Recursive call handles "smaller"

```
(define (rec l)
  (match l
    [(list)          handle-base]
    [(list h l ...) (handle-step h (rec l))]))
```

Example for `member`

```
(define (member x l)
  (cond
    [(list) #f]      ; <------------------- handle-base = #f
    [(list h l ...) ; <------------------- (handle-step h (rec l))
      (cons (f h)    ; <------------------- use: h
      (map f l))     ; <------------------- use: (rec l)
```

In this version, we make the base and handle-steps explicit. Previous solution coalesces nested conds into one.