CS450

Structure of Higher Level Languages

Lecture 08: Functions as values

Tiago Cogumbreiro

Today we will learn about...

- Modules in Racket
- Functions as values
- Creating functions dynamically
- Using functions as data-structures

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
```

Functions as values

What is functional programming

Functional programming has different meanings to different people

- Avoid mutation
- **Using functions as values**
- A programming style that encourages recursion and recursive data structures
- A programming model that uses ***lazy*** evaluation (discussed later)

First-class functions

- **Functions are values:** can be passed as arguments, stored in data structures, bound to variables, ...
- **Functions for extension points:** A powerful way to factor out a common functionality

Functions as parameters

Functions as parameters

Monotonic increasing function (for one input)

> Function `monotonic?` takes a function `f` as a parameter and a value `x`, and then checks if `f` increases monotonically for a given `x`.

Example

```
#lang racket
(define (double n) (* 2 n))
(define (monotonic? f x)
  (≥ (f x) x))
;; Tests
(require rackunit)
(check-true (monotonic? double 3))
(check-false (monotonic? (lambda (x) (- x 1)) 3))
```

How do we evaluate?

```
(monotonic? double 3)
```

Functions as parameters

Monotonic increasing function (for one input)

> Function `monotonic?` takes a function `f` as a parameter and a value `x`, and then checks if `f` increases monotonically for a given `x`.

Example

```
#lang racket
(define (double n) (* 2 n))
(define (monotonic? f x)
  (≥ (f x) x))
;; Tests
(require rackunit)
(check-true (monotonic? double 3))
(check-false (monotonic? (lambda (x) (- x 1)) 3))
```

How do we evaluate?

```
  (monotonic? double 3)

= (≥ (double 3) 3)
= (≥ ((lambda (n) (* 2 n)) 3) 3)
= (≥ (* 2 3) 3)
= (≥ 6 3)
= #t
```

Functions as parameters

Recursively apply a function n-times

> Function `apply-n` takes a function `f` as parameter, a number of times `n`, and some argument `x`, and then recursively calls `(f (f (... (f x))))` an `n`-number of times.

```
#lang racket
(define (apply-n f n x)
  (cond [(≤ n 0) x]
        [else (apply-n f (- n 1) (f x))]))
;; Tests
(require rackunit)
(define double (lambda (x) (* 2 x)))
(check-equal? (* 2 (* 2 (* 2 1))) (apply-n double 3 1))
(check-equal? (+ 3 (+ 3 (+ 3 1))) (apply-n (lambda (x) (+ 3 x)) 3 1))
```

Example apply-n

Let us unfold the following...

```
(apply-n double 3 1)                    ; (≤ 3 0) = #f
```

Example apply-n

Let us unfold the following...

```
  (apply-n double 3 1)                        ; (≤ 3 0) = #f

= (apply-n double (- 3 1) (double 1))
= (apply-n double 2 2)                        ; (≤ 2 0) = #f
= (apply-n double (- 2 1) (double 2))
= (apply-n double 1 4)                        ; (≤ 1 0) = #f
= (apply-n double (- 1 1) (double 4))
= (apply-n double 0 8)                        ; (≤ 0 0) = #t
= 8
```

Functions as data-structures

Functions as data-structures

The following is a function that returns a constant value (returns 3 always):

```
(define three:2 (lambda () 3))

(three:2) ; ← We need to call the function to obtain its contents
          ;    Note that we are passing 0 parameters.
```

Note the difference...

The following is a variable binding (not a function!):

```
(define three:1 3)
```

Variable `three:1` ***evaluates*** to the number 3.

A factory of constant-return functions

We can generalize the procedure by creating a function that returns a new function declaration that returns a given parameter n.

```
(define (factory n)
  (lambda () n)) ; ⟵ calling 'factory' creates a new function dynamically
                 ;     each new function captures the given 'n',
                 ;     which changes according to how it was called

(define three:4 (factory 3)) ; ← same as: (define three:4 (lambda () 3))
(three:4) ; Returns: 3

(define four:1 (factory 4)) ; ← same as: (define four:1 (lambda () 4))
(four:1) ; Returns 4
```

Implementing a pair with functions alone

If we can capture one parameter, then we can also capture two parameter. **Let us implement a pair-data structure with only functions!**

```
(define (cons x y)
  (lambda (b)   ; ← we use a parameter to choose which stored data to return
    (cond [b x] [else y]))) ; ← passing #t returns x
                            ;     passing #f returns y
; We now define our own 'car' and 'cdr'
(define (car f) (f #t))   ; Returns the first element of the pair
(define (cdr f) (f #f))   ; Returns the second element of the pair

(define p (cons 10 20))   ; Same as: (define (p b) (cond [b 10] [else 20])))
(car p)                   ; Returns 10 because (car p) → (p #f) → 10
(cdr p)                   ; Returns 20 because (cdr p) → (p #t) → 20
```