CS450

Structure of Higher Level Languages

Lecture 09: Evaluating expressions

Tiago Cogumbreiro

Today we will learn...

- academic honesty policy
- storing functions in data-structures
- creating functions dynamically
- currying functions
- finding elements in a list
- updating elements in a list

> Section 2.2.1 in SICP. Try out the interactive version of section 2.2 of the SICP book.

Academic dishonesty

Plagiarism in University

Copying code from others is wrong because:

- you do not learn
- you risk being expelled
- you are risking the other person being expelled
- you risk not completing your degree
- you risk being put on a list of cheaters (other universities may reject your application)

Plagiarism in the Industry

Is wrong, because:

- it is illegal
- you risk being dismissed from employment
- you risk being sued

Copying code (when it is right)

- software licenses define clear rules on
  how you can copy, use, and change other people's code
- open source promotes sharing of code
  - attribution is important (unless public domain)
  - good way to land on a job

Plagiarism in CS 450

UMASS BOSTON

Zero Tolerance

- student's responsibility to learn the Student's code of conduct
- we use plagiarism detection (renaming functions is not enough)
- we compare against solutions from past years (and instructor)
- be careful when working with others, any sharing code may trigger
- the plagiarism detection tool can detect code sharing among students

Plagiarism in CS 450

Last call

- statistically, there will be plagiarism this semester
- if I contact you regarding plagiarism, there will be zero tolerance **(no second chances)**
- you ***may*** void a submission **before** I contact you without repercussions
- I give you enough time to complete assignments, there is no excuse

Functions in data structures

Functions stored in data structures

"Freeze" one parameter of a function

> In this example, a `frozen` data-structure stores a binary-function and the first argument. Function `apply1` takes a frozen data structure and the second argument, and applies the stored function to the two arguments.

```
(struct frozen (func arg1) #:transparent)

(define (apply1 fr arg)
  (define func (frozen-func fr))    ; Bind a function to a local variable
  (define arg1 (frozen-arg1 fr))
  (func arg1 arg))                  ; Call a function bound to a local variable

(define frozen-double (frozen * 2)) ; Store function '*' in a data structure
(define (double x) (apply1 frozen-double x))
(check-equal? (* 2 3) (double 3))
```

Unfolding (double 3)

```
  (double 3)
= (apply1 frozen-double 3)
= (apply1 (frozen * 2) 3)
= (define fr (frozen * 2))
  ((frozen-func fr) (frozen-arg1 fr) 3)
= (* 2 3)
= 6
```

Functions stored in data structures

Apply a list of functions to a value

```
#lang racket
(define (double n) (* 2 n))
; A list with two functions:
; * doubles a number
; * increments a number
(define p (list double (lambda (x) (+ x 1))))
; Applies each function to a value
(define (pipeline funcs value)
  (cond [(empty? funcs) value]
        [else (pipeline (rest funcs) ((first funcs) value))]))
; Run the pipeline
(check-equal? (+ 1 (double 3)) (pipeline p 3))
```

Creating functions dynamically

Returning functions

- Functions in Racket automatically capture the value of any variable referred in its body.

Example

```
#lang racket
(define (frozen-* arg1)
  (define (get-arg2 arg2)
    (* arg1 arg2))
  ; Returns a new function
  ; every time you call frozen-*
  get-arg2)
(require rackunit)
(define double (frozen-* 2))
(check-equal? (* 2 3) (double 3))
```

Evaluating `(frozen-* 2)`

```
  (frozen-* 2)
= (define (get-arg2 arg2) (* 2 arg2)) get-arg2
= (lambda (arg2) (* 2 arg))
```

Evaluating `(double 3)`

```
  (double 3)
= ((frozen-* 2) 3)
= ((lambda (arg2) (* 2 arg2)) 3)
= (* 2 3)
= 6
```

Currying functions

Revisiting "freeze" function

Freezing binary-function

```
(struct frozen (func arg1) #:transparent)

(define (apply1 fr arg)
  (define func (frozen-func fr))
  (define arg1 (frozen-arg1 fr))
  (func arg1 arg))

(define frozen-double (frozen * 2))
(define (double x) (apply1 frozen-double x)
(check-equal? (* 2 3) (double 3))
```

Attempt #1

```
(define (freeze f arg1)
  (define (get-arg2 arg2)
    (f arg1 arg2))
  get-arg2)

(define double (freeze * 2))
(check-equal? (* 2 3) (double 3))
```

> Our `freeze` function is more general than `freeze-*` and simpler than `frozen-double`. We abstain from using a data-structure and use Racket's variable capture capabilities.

Generalizing "frozen" binary functions

Attempt #2

```
(define (freeze f)
  (define (expect-1 arg1)
    (define (expect-2 arg2)
      (f arg1 arg2))
    expect-2)
  expect-1)

(define frozen-* (freeze *))
(define double (frozen-* 2))
(check-equal? (* 2 3) (double 3))
```

Evaluation

```
  (define frozen-* (freeze *))
= (define frozen-*
    (define (expect-1 arg1)
      (define (expect-2 arg2)
        (* arg1 arg2))
      expect-2)
    expect-1)

  (define double (frozen-* 2))
= (define double
    (define (expect-2 arg2) (* 2 arg2))
    expect-2)

  (double 3)
= (* 2 3)
```

Currying functions

Currying is the general technique of "freezing" functions with multiple parameters. It provides a way of delaying (and caching) the passage of multiple arguments by means of new functions.

> A curried function $\text{curry}_{f,n,a}(x)$ is a unary function annotated with an uncurried function f arguments a and a number of expected arguments n that can be recursively defined as:

$$\text{curry}_{f,n+1,[a_1,\ldots,a_n]}(x) = \text{curry}_{f,n,[a_1,\ldots,a_n,x]}$$
$$\text{curry}_{f,0,[a_1,\ldots,a_n]}(x) = f(a_1,\ldots,a_n,x)$$

```
#lang racket
(define frozen-* (curry *))
(define double (frozen-* 2))
(require rackunit)
(check-equal? (* 2 3) (double 3))
```

Haskell Curry

Did you know?

- In some programming languages functions are curried by default. Examples include Haskell and ML.
- The term currying is named after Haskell Curry, a notable logician who developed combinatory logic and the Curry-Howard correspondence (practical applications include proof assistants).

  Haskell was born in Millis, MA (1 hour drive from UMB).

Source: public domain

How do we implement Currying?

How do we implement Currying?

We need two components:

1. A function that accepts each argument in curried form. (Lecture 6)
2. When function (1) receives its last argument, it must apply the curried-function to the stored curried arguments. (Lecture 9)

Functional pattern:
Finding elements

Find a value in a list

Let us implement a function `member` that tests whether or not a list contains a value.

Specification

```
; Unit test that tests
(require rackunit)
(check-true (member 1 (list 3 6 1)))
(check-true (member #t (list 3 #t (list))))
(check-false (member 1 (list 3 #t (list 1))))
(check-false (member #f (list)))
```

Find a value in a list

- Let us implement a function `member` that tests whether or not a list contains a value.

Specification

```
; Unit test that tests
(require rackunit)
(check-true (member 1 (list 3 6 1)))
(check-true (member #t (list 3 #t (list))))
(check-false (member 1 (list 3 #t (list 1))))
(check-false (member #f (list)))
```

Solution

```
(define (member x l)
  (cond
    [(empty? l) #f]
    [(equal? (first l) x) #t]
    [else (member x (rest l))]))
```

- Is the solution tail-recursive?

Find a value in a list

- Let us implement a function `member` that tests whether or not a list contains a value.

Specification

```
; Unit test that tests
(require rackunit)
(check-true (member 1 (list 3 6 1)))
(check-true (member #t (list 3 #t (list))))
(check-false (member 1 (list 3 #t (list 1))))
(check-false (member #f (list)))
```

Solution

```
(define (member x l)
  (cond
    [(empty? l) #f]
    [(equal? (first l) x) #t]
    [else (member x (rest l))]))
```

- Is the solution tail-recursive? **Yes!**

Common mistakes

Example

```
(define (member x l)
  (cond
    [(empty? l) #f]
    [(equal? (first l) x) #t]
    [else (member x (rest l))]))
```

- Forgetting the base case, results in calling `(first empty)`, which throws an error.
- Forgetting to make the list smaller, results in a non-terminating program.

Base case missing

```
(define (member x l)
  (cond
    [(equal? (first l) x) #t]
    [else (member x (rest l))]))
```

Value doesn't get smaller

```
(define (member x l)
  (cond
    [(empty? l) #f]
    [(equal? (first l) x) #t]
    [else (member x l)]))
```

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))
```

Is `map` function tail-recursive?

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))
```

> Is `map` function tail-recursive? **No.**

`map` passes the return value of the recursive call to `cons`. The order of applying `cons` is important, so we can't just apply it to an accumulator parameter (as that would reverse the order of application).

> **Idea: *delay adding to the right with a `lambda`***. First, run all recursive calls at tail-call, while creating a function that processes the result and appends the element to the left (`cons`). Second, run the accumulator function.

Tail-recursive `map`, using the generalized tail-recursion optimization pattern

```
(define (map f l)
  (define (map-iter accum l)
    (cond [(empty? l) (accum l)]
          [else (map-iter (lambda (x) (accum (cons (f (first l)) x))) (rest l))]))
  (map-iter (lambda (x) x) l))
```

The accumulator delays the application of `(cons (f (first l)) ?)`.

1. The initial accumulator is `(lambda (x) x)`, which simply returns whatever list is passed to it.
2. The base case triggers the computation of the accumulator, by passing it an empty list.
3. In the inductive case, we just augment the accumulator to take a list `x`, and return `(cons (f (first l)) x)` to the next accumulator.

> The accumulator works like a pipeline: each inductive step adds a new stage to the pipeline, and the base case runs the pipeline: `(stage3 (stage2 (stage1 ((lambda (x) x) nil))))`

Tail-recursive `map` run

```
(map f (list 1 2 3)) =
; First, build the pipeline accumulator
(define (accum0 x) x) (map-iter accum0 (list 1 2 3)) =
(define (accum1 x) (accum0 (cons (f 1) x))) (map-iter accum1 (list 2 3)) =
(define (accum2 x) (accum1 (cons (f 2) x))) (map-iter accum2 (list 3)) =
(define (accum3 x) (accum2 (cons (f 3) x))) (map-iter accum3 (list)) =
; Second, run the pipeline accumulator
(accum3 (list)) =
(accum2 (list (f 3))) =
(accum1 (list (f 2) (f 3))) =
(accum0 (list (f 1) (f 2) (f 3))) =
(list (f 1) (f 2) (f 3)))
```

Tail-recursive optimization pattern

> To summarize, when a value has base case and an inductive case, we identified the following pattern for a tail-recursive optimization:

Unoptimized

```
(define (rec v)
  (cond
    [(base-case? v) (base v)]
    [else (step v (rec (dec v)))]))
```

Optimized

```
(define (rec v)
  (define (rec-aux accum v)
    (cond
      [(base-case? v) (accum (base v))]
      [else
        (rec-aux
          (lambda (x) (accum (step v x)))
          (dec v))]))
  (rec-aux (lambda (x) x) v)
```

Functional patterns:

Queries

Any string in a list matches a prefix

Spec

```
(require rackunit)
(check-true (string-prefix? "Racket" "R")) ; available in standard library
(check-true (match-prefix? "R" (list "foo" "Racket")))
(check-false (match-prefix? "R" (list "foo" "bar")))
```

Any string in a list matches a prefix

Spec

```
(require rackunit)
(check-true (string-prefix? "Racket" "R")) ; available in standard library
(check-true (match-prefix? "R" (list "foo" "Racket")))
(check-false (match-prefix? "R" (list "foo" "bar")))
```

Solution

```
(define (match-prefix? prefix l)
  (cond
    [(empty? l) #f]
    [(string-prefix? (first l) prefix) #t]
    [else (match-prefix? prefix (rest l))]))
```

Can we generalize the search algorithm?

```
(define (member x l)
  (cond
    [(empty? l) #f]
    [(equal? (first l) x) #t]
    [else (member x (rest l))]))
```

```
(define (match-prefix? x l)
  (cond
    [(empty? l) #f]
    [(string-prefix? (first l) x) #t]
    [else (match-prefix? x (rest l))]))
```

Can we generalize the search algorithm?

```
(define (member x l)
  (cond
    [(empty? l) #f]
    [(equal? (first l) x) #t]
    [else (member x (rest l))]))
```

```
(define (match-prefix? x l)
  (cond
    [(empty? l) #f]
    [(string-prefix? (first l) x) #t]
    [else (match-prefix? x (rest l))]))
```

Solution

```
(define (exists predicate l)
  (cond
    [(empty? l) #f]
    [(predicate (first l)) #t]
    [else (exists predicate (rest l))]))
```

```
(define (member x l)
  (exists
    (lambda (y) (equal? x y)) l)
(define (match-prefix? x l)
  (exists
    (lambda (y) (string-prefix? y x))) l)
```