CS450

Structure of Higher Level Languages

Lecture 11: Currying, exists, update many

Tiago Cogumbreiro

Today we will learn...

- Currying (recap)
- Test if element exists (`member`, `exists`)
- Pointwise update (`map`)
- Tail-call optimized code

Currying

Applying a function with fewer arguments than required

yields a function that expects the remaining arguments

Uncurried functions

UMASS BOSTON

All arguments must be provided at call-time, otherwise error.

Python example

```
def add(l, r):
  return l + y

add(10)
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# TypeError: add() missing 1 required positional argument: 'r'
```

Curried functions

> If we provide one argument to a 2-parameters function, the result is a 1-parameter function that expects the second argument.

Haskell example

```haskell
-- Define addition
add x y = x + y
-- Define adding 10 to some number
add10 = add 10
-- 10 + 30
add10 30
-- 40
```

Currying in Racket

Function curry **converts** an uncurried function into a curried function.

```
#lang racket
(define curried-add (curry +))
(define add10 (curried-add 10))
(require rackunit)
(check-equal? (+ 10 30) (add10 30))
```

HW2

- In HW2 you will need to implement the reverse, function uncurry.
- You are now ready to solve exercises 1, 4, and 5.

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 $\mathrm{curry}_{f,n,a}(x)$ 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:

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

Exercise 6

What is the output of this program?

Program

```
(define curried-add
  (lambda (arg1)
    (lambda (arg2)
      (+ arg1 arg2))))

(define a (curried-add 10))
(define b (curried-add 20))
a
b
(a 30)
(b 40)
```

Exercise 6

What is the output of this program?

Program

```
(define curried-add
  (lambda (arg1)
    (lambda (arg2)
      (+ arg1 arg2))))

(define a (curried-add 10))
(define b (curried-add 20))
a
b
(a 30)
(b 40)
```

Output

```
(lambda (arg2) (+ 10 arg2))
(lambda (arg2) (+ 20 arg2))
40
60
```

Functional patterns:

Does it exist?

Element in the 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)))
```

Element in the 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?

Element in the 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!**

Element in the list?

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 `empty`: the `empty` list constructor, same as `(list)`
- Case `cons`: 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 `empty` (handle-base)
2. Case `cons` (handle-step)
3. Recursive call handles "smaller"

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

A general recursion pattern for handling lists

1. Case `empty` (handle-base)
2. Case `cons` (handle-step)
3. Recursive call handles "smaller"

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

Example for `member`

```
(define (member x l)
  (cond
    [(empty? l) #f]  ; <------------------------- handle-base: #f
    [else            ; <------------------------- handle-step
      (cond [(equal? (first l) x) #t] ;
            [else (member x (rest l))])])) ; <--- (decrement v) = (rest l)
```

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

Common mistake 1

Forgetting the base case

- **Symptom:** `first contract violation`

Example

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

Base case missing

```
(define (member x l)
  (cond
    [(equal? (first l) x) #t]
    [else (member x (rest l))]))
; first: contract violation
;   expected: (and/c list? (not/c empty?))
;   given: '()
; [,bt for context]
```

Common mistake 2

Forgetting to make the list smaller

- **Symptom:** program hangs (runs forever) for some inputs

Correct

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

Incorrect

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

Generalizing `member`

Exists prefix in list?

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")))
```

Exists prefix in list?

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?

```
; Example 1
(define (member x l)
  (cond
    [(empty? l) #f]
    [(equal? (first l) x) #t]
    [else (member x (rest l))]))
```

```
; Example 2
(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?

```
; Example 1
(define (member x l)
  (cond
    [(empty? l) #f]
    [(equal? (first l) x) #t]
    [else (member x (rest l))]))
```

```
; Example 2
(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))]))
```

```
; Example 1
(define (member x l)
  (exists
    (lambda (y) (equal? x y)) l))
; Example 2
(define (match-prefix? x l)
  (exists
    (lambda (y) (string-prefix? y 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.

The tail-recursive optimization pattern

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)
```