CSc 520

Principles of Programming Languages

39 : Scheme — Higher-Order Functions

Christian Collberg

collberg+520@gmail.com

Department of Computer Science

University of Arizona

Higher-Order Functions

- A function is higher-order if
  1. it takes another function as an argument, or
  2. it returns a function as its result.
- Functional programs make extensive use of higher-order functions to make programs smaller and more elegant.
- We use higher-order functions to encapsulate common patterns of computation.

Higher-Order Functions: `map`

- Map a list of numbers to a new list of their absolute values.
- Here's the definition of `abs-list` from a previous lecture:

```
(define (abs-list L)
   (cond
      [(null? L) '()]
      [else (cons (abs (car L))
                  (abs-list (cdr L)))]
   )
)

> (abs-list '(1 -1 2 -3 5))
(1 1 2 3 5)
```


Higher-Order Functions: `map`...

- This type of computation is very common.
- Scheme therefore has a built-in function

  `(map f L)`

  which constructs a new list by applying the function `f` to every element of the list `L`.

  (map f '(e_1 e_2 e_3 e_4))

  $\Downarrow$

  ((f e_1) (f e_2) (f e_3) (f e_4)))

Higher-Order Functions: map...

- map is a higher-order function, i.e. it takes another function as an argument.

```
(define (addone a) (+ 1 a))

> (map addone '(1 2 3)
(2 3 4)

> (map abs '(-1 2 -3))
(1 2 3)
```


Higher-Order Functions: map...

- We can easily define map ourselves:

```
(define (mymap f L)
   (cond
      [(null?  L) '()]
      [else
         (cons (f (car L)) (mymap f (cdr L)))]))

> (mymap abs '(-1 2 -3))
(1 2 3)
```


Higher-Order Functions: map...

- If the function takes n arguments, we give map n lists of arguments:

```
> (map string-append
          '("A" "B" "C") '("1" "2" "3"))
("A1" "B2" "C3")

> (map + '(1 2 3) '(1 2 3))
(list 2 4 6)

> (map cons '(a b c) '((1) (2) (3)))
((a 1) (b 2) (c 3))
```


Lambda Expressions

- A lambda-expression evaluates to a function:

  `(lambda (x) (* x x))`

  x is the function's formal parameter.
- Lambda-expressions don't give the function a name — they're anonymous functions.
- Evaluating the function:

```
> ((lambda (x) (* x x)) 3)
9
```

Higher-Order Functions: map...

- We can use lambda-expressions to construct anonymous functions to pass to map. This saves us from having to define auxiliary functions:

```
(define (addone a) (+ 1 a))

> (map addone '(1 2 3))
(2 3 4)

> (map (lambda (a) (+ 1 a)) '(1 2 3))
(2 3 4)
```


Higher-Order Functions: filter

- The filter-function applies a predicate (boolean-valued function) p to all the elements of a list.
- A new list is returned consisting of those elements for which p returns #t.

```
(define (filter p L)
   (cond
      [(null?  L) '()]
      [(p (car L))
            (cons (car L) (filter p (cdr L)))]
      [else (filter p (cdr L))]))

> (filter (lambda (x) (> x 0)) '(1 -2 3 -4))
(1 3)
```


Higher-Order Functions: fold

Consider the following two functions:

```
(define (sum L)
   (cond
      [(null?  L) 0]
      [else (+ (car L) (sum (cdr L)))]))
(define (concat L)
   (cond
      [(null?  L) "" ]
      [else (string-append (car L) (concat (cdr L)))]))

> (sum '(1 2 3))
6
> (concat '("1" "2" "3"))
"123"
```


Higher-Order Functions: fold...

- The two functions only differ in what operations they apply (+ vs. string-append, and in the value returned for the base case (0 vs. "").
- The fold function abstracts this computation:

```
(define (fold L f n)
   (cond
      [(null?  L) n]
      [else (f (car L) (fold (cdr L) f n))]))

> (fold '(1 2 3) + 0)
6
> (fold '("A" "B" "C") string-append "")
"ABC"
```

Higher-Order Functions: `fold`

- In other words, `fold` folds a list together by successively applying the function `f` to the elements of the list `L`.

  (apply f '(e_1 e_2 e_3 e_4)) $\Rightarrow$
  (f e_1 (f e_2 (f e_3 e_4)))