1 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.

2 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:

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

```scheme
> (abs-list '(1 -1 2 -3 5))
(1 1 2 3 5)
```
3 Higher-Order Functions: map...

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

\[(\text{map } f \ L)\]

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

\[(\text{map } f \ '(e_1 \ e_2 \ e_3 \ e_4))\]
\[= ((f \ e_1) \ (f \ e_2) \ (f \ e_3) \ (f \ e_4))\]

4 Higher-Order Functions: map...

- \(\text{map}\) is a higher-order function, i.e. it takes another function as an argument.

\[(\text{define} \ (\text{addone} \ a) \ (+ \ 1 \ a))\]

\[> \ (\text{map} \ \text{addone} \ '(1 \ 2 \ 3))\]
\[(2 \ 3 \ 4)\]

\[> \ (\text{map} \ \text{abs} \ '(-1 \ 2 \ -3))\]
\[(1 \ 2 \ 3)\]

5 Higher-Order Functions: map...

- We can easily define \(\text{map}\) ourselves:

\[(\text{define} \ (\text{mymap} \ f \ L)\]
\(= \text{(cond}\)
\[[\text{null?} \ L] '()\]
[else
\[(\text{cons} \ (f \ \text{car} \ L) \ (\text{mymap} \ f \ \text{cdr} \ L)))\]]\]

\[> \ (\text{mymap} \ \text{abs} \ '(-1 \ 2 \ -3))\]
\[(1 \ 2 \ 3)\]

6 Higher-Order Functions: map...

- If the function takes \(n\) arguments, we give \(\text{map}\) \(n\) lists of arguments:

\[> \ (\text{map} \ \text{string-append} \'
\[\text{"A" "B" "C"} \ \text{"1" "2" "3"})\]
\[\text{"A1" "B2" "C3"})\]

\[> \ (\text{map} \ + \ '(1 \ 2 \ 3) \ '(1 \ 2 \ 3))\]
\[= (\text{list} \ 2 \ 4 \ 6)\]

\[> \ (\text{map} \ \text{cons} \ '(a \ b \ c) \ '((1) \ (2)) \ (3)))\]
\[= ((a \ 1) \ (b \ 2) \ (c \ 3))\]
7 Lambda Expressions

- A lambda-expression evaluates to a function:
  \[(\text{lambda } (x) (* x x))\]
  
  \(x\) is the function's formal parameter.

- Lambda-expressions don't give the function a name — they're anonymous functions.

- Evaluating the function:
  \[
  > ((\text{lambda } (x) (* x x)) 3)
  \]
  \[
  9
  \]

8 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:
  \[
  (\text{define } (\text{addone } a) (* 1 a))
  \]
  \[
  > (\text{map addone } '(1 2 3)
  \]
  \[
  (2 3 4)
  \]
  \[
  > (\text{map } (\text{lambda } (a) (* 1 a)) '(1 2 3))
  \]
  \[
  (2 3 4)
  \]

9 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.

  \[
  (\text{define } (\text{filter } p L)
  \]
  \[
  \text{(cond}
  \]
  \[
  [(\text{null? } L) '\()]
  \]
  \[
  [(p (\text{car } L))
  \]
  \[
  (\text{cons } (\text{car } L) (\text{filter } p (\text{cdr } L))))
  \]
  \[
  [\text{else } (\text{filter } p (\text{cdr } L))])]
  \]
  \[
  \text{)}
  \]
  \[
  > (\text{filter } (\text{lambda } (x) (> x 0)) '(1 -2 3 -4))
  \]
  \[
  (1 3)
  \]

10 Higher-Order Functions: fold

Consider the following two functions:
11 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:

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

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

12 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.

\[
\text{apply } f \ (e_1 \ e_2 \ e_3 \ e_4) \Rightarrow (f \ e_1 \ (f \ e_2 \ (f \ e_3 \ e_4)))
\]