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:

```scheme
(define (abs-list L)
  (cond
    [(null? L) '()] ; null list returns an empty list
    [else (cons (abs (car L)) ;; abs of the first element
                (abs-list (cdr L))] ; and the rest of the list
  )
)

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

> (abs-list '(1 -1 2 -3 5))
(1 1 2 3 5)
Higher-Order Functions: \textit{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 \(L\).

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

\[
\Downarrow
\]

\[
((f \ e_1) \ (f \ e_2) \ (f \ e_3) \ (f \ e_4))
\]
map is a higher-order function, i.e. it takes another function as an argument.

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

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

> (map abs '(-1 2 -3))
(1 2 3)
```
Higher-Order Functions: \texttt{map}...

- We can easily define \texttt{map} ourselves:

\begin{verbatim}
(define (mymap f L)
  (cond
    [(null? L) '()]
    [else
     (cons (f (car L)) (mymap f (cdr L)))]
)
\end{verbatim}

> (mymap abs '(-1 2 -3))
(1 2 3)
Higher-Order Functions: \texttt{map}...

If the function takes $n$ arguments, we give \texttt{map} $n$ lists of arguments:

\begin{verbatim}
> (map string-append
       ('("A" "B" "C") '("1" "2" "3")))
("A1" "B2" "C3")
\end{verbatim}

\begin{verbatim}
> (map + '(1 2 3) '(1 2 3))
(list 2 4 6)
\end{verbatim}

\begin{verbatim}
> (map cons '(a b c) '((1) (2) (3)))
((a 1) (b 2) (c 3))
\end{verbatim}
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:

  \[> \ ((\text{lambda} \ (x) \ (* \ x \ x)) \ 3) \]

  \[9\]
We can use \texttt{lambda}-expressions to construct anonymous functions to pass to \texttt{map}. This saves us from having to define auxiliary functions:

\begin{verbatim}
(define (addone a) (+ 1 a))
\end{verbatim}

\begin{verbatim}
> (map addone '(1 2 3))
(2 3 4)
\end{verbatim}

\begin{verbatim}
> (map (lambda (a) (+ 1 a)) '(1 2 3))
(2 3 4)
\end{verbatim}
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:

\[
\text{(define (sum L)} \\
\quad \text{(cond}} \\
\quad \quad \text{[(null? L) 0]} \\
\quad \quad \text{[else (+ (car L) (sum (cdr L)))]})
\]

\[
\text{(define (concat L)} \\
\quad \text{(cond}} \\
\quad \quad \text{[(null? L) ""]} \\
\quad \quad \text{[else (string-append (car L) (concat (cdr L)))]})
\]

> (sum '(1 2 3))
6
> (concat '("1" "2" "3"))
"123"
The two functions only differ in what operations they apply (\(+\) vs. \texttt{string-append} \textsuperscript{,} and in the value returned for the base case (\texttt{0} vs. \texttt{""})).

The \texttt{fold} function abstracts this computation:

\begin{verbatim}
(define (fold L f n)
  (cond
   [(null? L) n]
   [else (f (car L) (fold (cdr L) f n))]))
\end{verbatim}

\begin{verbatim}
> (fold '(1 2 3) + 0)
6
> (fold '("A" "B" "C") string-append "")
"ABC"
\end{verbatim}
Higher-Order Functions: \texttt{fold}

In other words, \texttt{fold} folds a list together by successively applying the function \texttt{f} to the elements of the list \texttt{L}.

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