Higher-Order Functions

A function is \textit{higher-order} if
\begin{itemize}
\item it takes another function as an argument, or
\item it returns a function as its result.
\end{itemize}

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: \texttt{map}

- Map a list of numbers to a new list of their absolute values.
- Here’s the definition of \texttt{abs-list} from a previous lecture:

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

\begin{verbatim}
> (abs-list '(1 -1 2 -3 5))
(1 1 2 3 5)
\end{verbatim}

Higher-Order Functions: \texttt{map}...

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

\begin{verbatim}
(map f L)
\end{verbatim}

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

\begin{verbatim}
(map f '(e1 e2 e3 e4))
\end{verbatim}

\begin{verbatim}
((f e1) (f e2) (f e3) (f e4))
\end{verbatim}
Higher-Order Functions: map...

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

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

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

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

We can easily define map ourselves:

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

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

Lambda Expressions

- A lambda-expression evaluates to a function:

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

\[
x\text{ is the function's formal parameter.}
\]

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

- Evaluating the function:

\[
> ((\text{lambda (x) (* x x)}) 3)
\]
\[
9
\]

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

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

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

\[
> \text{(map cons '(a b c) '(1) (2) (3))}
\]
\[
((a 1) (b 2) (c 3))
\]
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 \( \texttt{#t} \).

```
(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 (highlightbox+ 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"
```

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

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