Built-in abstract list functions

Intermediate Student also provides map as a built-in function, as well as many other abstract list functions. Check out the Help Desk (in DrRacket, Help → Help Desk → How to Design Programs Languages → 4.17 Higher-Order Functions)

The abstract list functions map and filter allow us to quickly describe functions to do something to all elements of a list, and to pick out selected elements of a list, respectively.
Abstracting another set of examples

The functions we have worked with so far consume and produce lists.

What about abstracting from functions such as `count-symbols` and `sum-of-numbers`, which consume lists and produce values?

Let’s look at these, find common aspects, and then try to generalize from the template.
(define (sum-of-numbers lst)
  (cond [(empty? lst) 0]
        [else (+ (first lst) (sum-of-numbers (rest lst)))]))

(define (prod-of-numbers lst)
  (cond [(empty? lst) 1]
        [else (* (first lst) (prod-of-numbers (rest lst)))]))

(define (count-symbols lst)
  (cond [(empty? lst) 0]
        [else (+ 1 (count-symbols (rest lst)))]))
Abstracting another set of examples

Note that each of these examples has a base case which is a value to be returned when the argument list is empty.

Each example is applying some function to combine (first lst) and the result of a recursive function application with argument (rest lst).

This continues to be true when we look at the list template and generalize from that.
(define (list-template lst)
  (cond [(empty? lst) . . .]
        [else . . . (first lst) . . .
         (list-template (rest lst)) . . .]]))

We replace the first ellipsis by a \textit{base value}.

We replace the rest of the ellipses by some \textit{function which combines} (first lst) and the result of a recursive function application on (rest lst).

\textit{Key Point:} This suggests passing the \textit{base value} and the \textit{combining function} as parameters to an abstract list function.
The abstract list function `foldr`

```
(define (my-foldr combine base lst)
  (cond [(empty? lst) base]
        [else (combine (first lst)
                        (my-foldr combine base (rest lst)))]
        ))
```

`foldr` is also a built-in function in Intermediate Student With Lambda.
Tracing \textit{my-foldr}

\[
\text{(my-foldr f 0 (list 3 6 5)) } \Rightarrow \\
\text{(f 3 (my-foldr f 0 (list 6 5))) } \Rightarrow \\
\text{(f 3 (f 6 (my-foldr f 0 (list 5))) } \Rightarrow \\
\text{(f 3 (f 6 (f 5 (my-foldr f 0 empty))) } \Rightarrow \\
\text{(f 3 (f 6 (f 5 0))) } \Rightarrow \ldots
\]

Intuitively, \textit{the effect} of the application
\[(\text{foldr f b (list x1 x2 \ldots xn)})\] is to compute the value of the expression
\[(f x1 (f x2 (\ldots (f xn b) \ldots))).\]
More on my-folddr

foldr is short for “fold right”.

The reason for the name is that it can be viewed as “folding” a list using the provided combine function, starting from the right-hand end of the list.

foldr can be used to implement map, filter, and other abstract list functions.
The contract for foldr

foldr consumes three arguments:

1. a function which combines the first list item with the result of reducing the rest of the list;

2. a base value;

3. a list on which to operate.

What is the contract for foldr?

;; foldr: (X Y → Y) Y (listof X) → Y
Using `foldr` to sum a list

```scheme
(define (sum-of-numbers lst) (foldr + 0 lst))
```

If `lst` is `(list x1 x2 ... xn)`, then by our intuitive explanation of `foldr`, the expression `(foldr + 0 lst)` reduces to

```scheme
(+ x1 (+ x2 (+ ... (+ xn 0) ...)))
```

Thus `foldr` does all the work of the template for processing lists, in the case of `sum-of-numbers`. 
Using `foldr`

The function provided to `foldr` consumes two parameters:

1. one is an element on the list which is an argument to `foldr`, and
2. one is the result of reducing the rest of the list.

Sometimes one of those arguments should be ignored, as in the case of using `foldr` to compute `count-symbols`.
Using \texttt{foldr} to count symbols

The important thing about the \textit{first argument} to the function provided to \texttt{foldr} is that it \textit{contributes 1 to the count}; its actual value is irrelevant.

Thus the function provided to \texttt{foldr} in this case can ignore the value of the first parameter, and \textit{just add 1 to the reduction of the rest of the list}. 
Using \texttt{foldr} to count symbols

\begin{verbatim}
(define (count-symbols lst) (foldr (lambda (x rror) (add1 rror)) 0 lst))
\end{verbatim}

The function provided to \texttt{foldr}, namely

\begin{verbatim}
(lambda (x rror) (add1 rror))
\end{verbatim}

\emph{ignores its first argument,} \(x\).

Its second argument is the result of recursing on the rest \(ror\) of the list (in this case the length of the rest of the list, to which 1 must be added).
More examples

What do these functions do?

(define (bar lon)
  (foldr max (first lon) (rest lon)))

(bar '(1 5 23 3 99 2))

(define (foo los)
  (foldr (lambda (s rror) (+ (string-length s) rror)) 0 los))

(foo '("one" "two" "three"))
Using **foldr** to produce lists

So far, the functions we have been providing to **foldr** have produced numerical results, but they can also produce **cons** expressions. **foldr** is an abstraction of structural recursion on lists, so we should be able to use it to implement **negate-list** from module 05.

**Key Idea:** define a function \((\text{lambda} \ (x \ rror) \ldots)\) where \(x\) is the first element of the list and \(rror\) is the result of the recursive function application.

**negate-list** takes this element, negates it, and **conses** it onto the result of the recursive function application.
Using **foldr** to implement **map**

The function we need is

\[(\text{lambda } (x \text{ rror}) (\text{cons } (\text{\textendash } x) \text{ rror}))\]

Thus we can give a nonrecursive version of **negate-list** (that is, **foldr** does all the recursion).

\[(\text{define } (\text{negate-list } \text{lst})
\quad (\text{foldr } (\text{lambda } (x \text{ rror}) (\text{cons } (\text{\textendash } x) \text{ rror})) \text{ empty } \text{lst}))\]

**Key Observation:** Because we generalized **negate-list** to **map**, we should be able to use **foldr** to define **map**.
Recall: the implemention of \texttt{map}

Let’s look at the code for \texttt{my-map}.

\begin{verbatim}
(define (my-map f lst)
  (cond [(empty? lst) empty]
       [else (cons (f (first lst))
                 (my-map f (rest lst)))]))
\end{verbatim}

Clearly \texttt{empty} is the base value, and the function provided to \texttt{foldr} is something involving \texttt{cons} and \texttt{f}.
Recall: the implementation of map using foldr

In particular, the function provided to foldr must apply f to its first argument, then cons the result onto its second argument (the reduced rest of the list).

\[
(\text{define } (\text{my-map } f \ \text{lst})
  (\text{foldr } (\lambda (x \ \text{rror}) (\text{cons } (f \ x) \ \text{rror})) \ \text{empty } \text{lst}))
\]

We can also implement my-filter using foldr.
Abstract list functions

Imperative languages, which tend to provide inadequate support for some aspects of recursion (such as mutual recursion), usually provide looping constructs such as “while” and “for” to perform repetitive actions on data.

**Key Point:** Abstract list functions cover many of the common uses of such looping constructs.

Our implementation of these functions is not difficult to understand, and we can write more if needed, but the set of looping constructs in a conventional language is fixed.
Abstract list functions

*Key Point:* Anything that can be done with the list template can be done using `foldr`, without explicit recursion (unless it ends the recursion early, like `insert`).

Does that mean that the list template is obsolete?

No. Experienced Racket programmers still use the list template, for reasons of readability and maintainability.

Abstract list functions should be used judiciously, to replace relatively simple uses of recursion.

Shorter code is not always better!
Generalizing accumulative recursion: ex 1

This function (from Mod 12-31) uses recursion (with an accumulator) on a list.

;; code from module 12

(define (sum-list lon)
  (local [(define (sum-list/acc lst sum-so-far)
              (cond [(empty? lst) sum-so-far]
                    [else (sum-list/acc (rest lst)
                                         (+ (first lst) sum-so-far))])]]
          (sum-list/acc lon 0)))
Generalizing accumulative recursion: ex 2

This function (from Mod 13-88) uses recursion (with an accumulator) on a list.

;; code from lecture module 9-14 rewritten to use local

(define (my-reverse lst0)
    (local [(define (my-rev/acc lst list-so-far)
                (cond [(empty? lst) list-so-far]
                      [else (my-rev/acc (rest lst) (cons (first lst) list-so-far))]))]
        (my-rev/acc lst0 empty)))
Contrasting: ex 1 and ex 2

The differences between these two functions are:

• the *initial value* of the accumulator;

• the computation of the *new value* of the accumulator, given the *old value* of the accumulator and the *first element of the list*. 
Introducing *foldl*

\[
\text{(define (my-foldl combine base lst0)}
\]
\[
\text{(local [(define (foldl/acc lst acc)
\text{\hspace{1em}}(\text{cond} [(\text{empty?} \ lst) \ acc]
\text{\hspace{1em}}\text{\hspace{1em}}[\text{else} \ (\text{foldl/acc} \ (\text{rest} \ lst)
\text{\hspace{1em}}\text{\hspace{1em}}\text{\hspace{1em}}(\text{combine} \ (\text{first} \ lst) \ acc)))]
\text{\hspace{1em}}(\text{foldl/acc} \ lst0 \ base))])}
\]

\[
\text{(define (sum-list lon) \ (my-foldl + 0 lon))}
\]
\[
\text{(define (my-reverse lst) \ (my-foldl cons empty lst))}
\]
We noted earlier that intuitively, the effect of the application
\[(\text{foldr } f \ b \ (\text{list } x_1 \ x_2 \ldots \ x_n))\]
is to compute the value of the expression
\[(f \ x_1 \ (f \ x_2 \ (\ldots \ (f \ x_n \ b) \ldots )))\]

What is the intuitive effect of the following application of \text{foldl}?
\[(\text{foldl } f \ b \ (\text{list } x_1 \ldots \ x_{n-1} \ x_n))\]

The function \text{foldl} is provided in Intermediate Student.

What is the contract of \text{foldl}?
foldr vs. foldl

- \((\text{foldr } f \ b \ (\text{list } x_1 \ x_2 \ \ldots \ x_n))\) computes \((f \ x_1 \ (f \ x_2 \ (\ldots \ (f \ x_n \ b)\ldots)))).\)

- \text{foldr} starts calculating from the right side of the list.

- \((\text{foldl } f \ b \ (\text{list } x_1 \ x_2 \ \ldots \ x_n))\) computes \((f \ x_n \ (f \ x_{n-1} \ (\ldots \ (f \ x_1 \ b)\ldots)))).\)

- \text{foldl} starts calculating from the left side of the list.

- The contract is the same for both \((X \ Y \rightarrow Y)\ Y \ (\text{listof } X) \rightarrow Y.\)
Higher-order functions

Functions that consume or produce functions like `filter`, `map`, and `foldr` are sometimes called higher-order functions.

Another example is the built-in `build-list`. This consumes a natural number `n` and a function `f`, and produces the list

$$(\text{list } (f \ 0) \ (f \ 1) \ldots \ (f \ (\text{sub1} \ n)))$$

$$\left(\text{build-list} \ 4 \ (\lambda (x) \ x)\right) \Rightarrow \left(\text{list} \ 0 \ 1 \ 2 \ 3\right).$$

Clearly `build-list` abstracts the “count up” pattern, and it is easy to write our own version.
An implementation of **build-list**

```
(define (my-build-list n f)
  (local
    [(define (list-from i)
        (cond [(>= i n) empty]
          [else (cons (f i) (list-from (add1 i)))]
        )])
  (list-from 0))
```

Build-list example: sum

\[ \sum_{i=0}^{n-1} x_i : \]

\[
(define (sum n f) \\
(foldr + 0 (build-list n f)))
\]

(sum 4 sqr) \(\Rightarrow\) 14
Build-list example: \texttt{mult-table}

We can now simplify \texttt{mult-table} even further.

\begin{verbatim}
(define (mult-table nr nc)
  (build-list nr
    (lambda (r)
      (build-list nc
        (lambda (c)
          (∗ r c))))))
\end{verbatim}
Goals of this module

You should understand the idea of functions as first-class values: how they can be supplied as arguments, produced as values using \texttt{lambda}, bound to identifiers, and placed in lists.

You should be familiar with the built-in abstract list functions provided by Racket, understand how they abstract common recursive patterns, and be able to use them to write code.
You should be able to write your own abstract list functions that implement other recursive patterns.

You should understand how to do step-by-step evaluation of programs written in the Intermediate language that make use of functions as values.
Module 13 Summary
Functions a First-class Values in Racket

1. **Abstraction** is the process of finding similarities or common aspects, and forgetting unimportant differences. [2]

2. In particular consider the general structure of two similar functions. [5]

3. In Racket functions are first-class values. [6]

4. **First-class** values are values that can be consumed as arguments [8-11], produced as results [13-16], bound to identifiers [18-19], and put in lists and structures [20-25].
Module 13 Summary

Functional Abstraction and their Contracts

5. **Functional abstraction** is the process of combining related functions into a single definition (such as filter). [12]
6. The body of `local` can be used to create functions [13-16] and `define` can be used to bind the results to an identifier. [18-19]
7. Use the contract for a function as its type. [26-27]
8. To capture the dependency between the type of the predicate and the type of list, use a type variable such as X. [30]
9. A function is **polymorphic** or **generic** if it works on many types of data. [31-33]
Module 13 Summary

Lambda

10. We skipped slides 34-38.

11. **Anonymous functions** are functions that do not have a name. [39]

12. Use `lambda` to produce anonymous functions (which can then be used as an argument). [43-51]

13. `lambda` replaces `local`, `define` and the identifier of an anonymous function but keeps the arguments and body. [46]

14. `define` binds a name to a value (where the value may be a function or a constant). [47-48]
Module 13 Summary
Syntax and Semantics of Lambda

15. During a trace, the value of a function does not get rewritten as a lambda expression. [49]

16. The first position in an application can be an expression computing a function, e.g. ((make-adder 3) 4), or a named function, e.g. (sum1 4). [52]

17. The rule for evaluating ((lambda(x1 ... xn) exp) v1 ... vn) ⇒ exp’ is that exp’ is exp with all occurrences of x1 replaced by v1, all occurrences of x2 replaced by v2, and so on. [53]

18. Note that exp can be another lambda expression. [50]
Module 13 Summary

Abstract List Functions filter, map and foldr

19. **filter** is a built-in function that consumes a predicate \((X \rightarrow \text{Bool})\) and a \((\text{listof } X)\) and produces a \((\text{listof } X)\) by selecting elements in the list where the predicate produces true. [60]

20. The built-in abstract list function \((\text{map } f \ \text{lst})\) applies \(f\) to each element of \(\text{lst}\) to produce a new list. [62-65]

21. Its contract is \(\text{map}: (X \rightarrow Y) \ (\text{listof } X) \rightarrow (\text{listof } Y)\) [66]

22. The built-in abstract list function \((\text{foldr } \text{combine } \text{base } \text{lst})\) has a base value, \(\text{base}\), a combining function, \(\text{combine}\) and a \(\text{lst}\) to create a new value. [68-80]
Module 13 Summary

Abstract List Functions: foldr and foldl

23. Its contract is foldr: \((X \rightarrow Y) \rightarrow Y (listof X) \rightarrow Y\) [75]

24. foldr can be used with cons to produce lists. [81-84]

25. The higher-order function (foldl combine base list) generalizes structural recursion on a list with one accumulator. [90]

26. foldl consumes

   (a) combine: a function that computes of the new value of the accumulator,

   (b) base: the initial value of the accumulator,

   (c) list: a list. [90]
Module 13 Summary

Abstract List Functions: foldl and build-list

27. Its contract is foldl: (X Y → Y) Y (listof X) → Y [91.1]

28. Functions that consume or produce functions are called higher-order functions. [92]

29. build-list is a built-in higher-order function that takes a function f and a natural number n and produces the list (list (f 0) (f 1) (f 2) ... (f (sub1 n))). [93]