Higher-order functions

A higher-order function is a function that takes at least one function as an argument or that returns a function as its result (or both).

map, filter, fold

Here are some common higher-order functions that come with Racket. These functions correspond to common patterns in programming. They let us fulfill the mission of functional programming, which is to “say more what; say less how”.

(map f L): given a transforming function f and a list L, map produces a new list L' where each element of L' is the result of applying f to the corresponding element of L. The function f takes a single element of L and transforms it to a new value.

(filter f L): given a predicate function f and a list L, filter produces a new list L' that contains only the elements of L for which the predicate is true. The function f takes a single element of L and returns either true or false.

(foldr f seed L): given a folding function f, an initial seed value, and a list L, foldr reduces L to a single value by repeatedly applying f to the list. The function f takes two arguments: an element of L and the accumulated value; it returns a new accumulated value. foldr starts by applying f to the last element of L and the seed, then moves its way towards the front of the list:

\[(\text{foldr } f \text{ seed } L) ≡ (f \, v_0 \, (... \, (f \, v_n \, \text{seed}) \, ...))\]

(foldl f seed L): given a folding function f, an initial seed value, and a list L, foldl reduces L to a single value by repeatedly applying f to the list. The function f takes two arguments: an element of L and the accumulated value; it returns a new accumulated value. foldl starts by applying f to the first element of L and the seed, then moves its way towards the back of the list:

\[(\text{foldl } f \text{ seed } L) ≡ (f \, v_n \, (... \, (f \, v_1 \, (f \, v_0 \, \text{seed})) \, ...))\]

Caution: when f is not commutative, foldr and foldl might return different results.

anonymous functions (i.e., lambdas)

An anonymous function is just a function without a name. They’re handy when we want call a function only once.

(lambda (parameter_1 ... parameter_n)
    body-expr)

Association lists

An association list is a list of pairs, e.g., '((1 "one") (2 "two")) . The first element of each pair is the key; the second element is the value.

We can look up a value in an association pair using assoc: (assoc key alist). The assoc function returns the appropriate pair (if key is one of the keys in the list), or it returns false (if key is not one of the keys in the list).
Racket reference (for when you’re working on the exercises)

s-expressions

\[(op \ \text{arg}_1 \ \text{arg}_2 \ \ldots \ \text{arg}_n)\]

Rules:

- the operation always comes first
- its arguments (if there are any) follow the operation
- no commas between arguments
- everything goes between parentheses

Common mistakes:

- forgetting parentheses
- rational vs. integer division (\(/\) vs. \(\text{quotient}\))
- equality (\(=\) vs. \(\text{equal?}\))

let*

\[(\text{let*} \ (\begin{array}{l}
  \text{var}_1 \ \text{expr}_1 \\
  \ldots \\
  \text{var}_n \ \text{expr}_n
\end{array})
\ \text{body-expr})\]

Bindings: an “assignment”. We say: “bind a value to a variable” or “bind a variable to a value”.

The body is the scope of the variables that are bound in the first part of the let*.

Conditionals

\[(\text{if} \ \text{conditional-expr} \\
  \text{true-expr} \\
  \text{false-expr})\]

\[(\text{cond} \ \begin{array}{l}
  \text{condition}_1 \ \text{expr}_1 \\
  \ldots \\
  \text{condition}_n \ \text{expr}_n
\end{array} \\
  \begin{array}{l}
  \text{else-expr}
\end{array})\]

If you have more than one condition, use cond; otherwise use if.

Functions

\[(\text{define} \ (\begin{array}{l}
  \text{function-name} \ \text{parameter}_1 \ \ldots \ \text{parameter}_n
\end{array})
\ \text{body-expr})\]

Next time: recursive problem-solving strategies