16: Functional Abstraction

Abstraction

Abstraction is the process of finding similarities or common aspects, and forgetting unimportant differences.

Example: writing a function.

- The differences in parameter values are forgotten, and the similarity is captured in the function body.
- We have seen many similarities between functions, and captured them in function templates.

Previously, we used functions as first class values to capture similarities that we couldn't capture before using the example of `filter`. We'll see four more examples of similar **higher order functions** (functions that either consume or produce a function, or both) in this module.

Higher order functions (HOFs)

Remember `eat-apples` and `keep-odds`? Those two functions had a very similar structure. Each selected items from a list to keep (or discard, depending on your viewpoint).

We abstracted those into a function called `filter` that consumed a predicate governing the items to keep. That could simplify the code on the left to the code on the right:

```
(define (keep-odds lst)
  (cond [(empty? lst) empty]
        [(odd? (first lst))
         (cons (first lst)
               (keep-odds (rest lst)))]
        [else (keep-odds (rest lst))]))
```

```
(define (keep-odds lst)
  (filter odd? lst))
```

We will now look for other patterns where we can perform similar abstractions.

Deriving map

Here are two early list functions we wrote.

```
(define (negate-list lst)
  (cond [(empty? lst) empty]
        [else (cons (-      (first lst))
                    (negate-list (rest lst)))]))

(define (compute-tax pay)
  (cond [(empty? pay) empty]
        [else (cons (sr->tr (first pay))
                    (compute-tax (rest pay)))]))
```

> Deriving map

To abstract the commonality, we look for a difference that can't be explained by renaming (it being what is applied to the first item of a list) and make that a parameter.

```
(define (negate-list lst)
  (cond [(empty? lst) empty]
        [else (cons (-      (first lst))
                    (negate-list (rest lst)))]))

(define (compute-tax pay)
  (cond [(empty? pay) empty]
        [else (cons (sr->tr (first pay))
                    (compute-tax (rest pay)))]))

(define (my-map f lst)
  (cond [(empty? lst) empty]
        [else (cons (f (first lst))
                    (my-map f (rest lst)))]))
```

> Tracing my-map


```
(define (my-map f lst)
  (cond [(empty? lst) empty]
        [else (cons (f (first lst))
                    (my-map f (rest lst)))]))
```

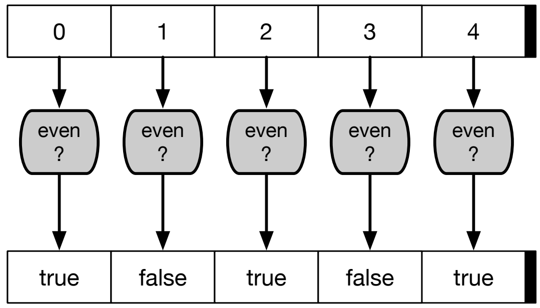
```
(my-map sqr (list 3 6 5))
⇒ (cons 9 (my-map sqr (list 6 5)))
⇒ (cons 9 (cons 36 (my-map sqr (list 5))))
⇒ (cons 9 (cons 36 (cons 25 (my-map sqr empty))))
⇒ (cons 9 (cons 36 (cons 25 empty)))
```

my-map performs the general operation of transforming a list element-by-element into another list of the same length.

> Effect of `my-map`

`(my-map f (list x_1 x_2 ... x_n))` has the same effect as evaluating `(list (f x_1) (f x_2) ... (f x_n))`.

```
(my-map even? (list 0 1 2 3 4))
```



> Using `my-map`

We can use `my-map` to give short definitions of a number of functions we have written to consume lists:

```
(define (negate-list lst)   (my-map - lst))
(define (compute-taxes lst) (my-map sr->tr lst))
```

> The contract for `my-map`

`my-map` consumes a function and a list, and produces a list.

How can we be more precise about its contract, using parametric type variables?

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

CS135 will discuss five of them: `filter`, `map`, `foldr`, `foldl`, and `build-list`.

The higher order 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.

Ex. 1

Digital signals are often recorded as values between 0 and 255, but we often prefer to work with numbers between 0 and 1.
Use `map` to write a function `(squash-range lst)` that consumes a `(listof Nat)`, and produces a `(listof Num)` so numbers on the interval $[0, 255]$ are scaled to the interval $[0, 1]$.

```
(check-expect (squash-range (list 0 204 255)) (list 0 0.8 1))
```

Ex. 2

Write a function that consumes a `(listof Str)`, where each `Str` is a person's name, and produces a list containing a greeting for each person.

```
(check-expect (greet-each (list "Ali" "Carlos" "Sai"))
                          (list "Hi Ali!" "Hi Carlos!" "Hi Sai!"))
```

Ex. 3

Using **`lambda`**, **`cond`** and `map`, write a function `neg-odd` that consumes a `(listof Nat)`. The function produces a `(listof Int)` where all odd numbers are made negative, and all even numbers are left positive.

```
(check-expect (neg-odd (list 2 5 8 11 14 17)) (list 2 -5 8 -11 14 -17))
```

Higher order functions that produce values

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 simple values?

Let's look at these, find common aspects, and then try to generalize from the template.

> Examples


```
(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 (all-true? lst)
  (cond [(empty? lst) true]
        [else (and (first lst)
                   (all-true? (rest lst)))]))
```

> Similarities and differences

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.

> Comparison to the list template

```
(define (list-template lst)
  (cond [(empty? lst) ...]
        [else (... (first lst)
                   (list-template (rest lst)))]))
```

We replace the first ellipsis by a base value.

We replace the other ellipsis by some function which combines `(first lst)` and the result of a recursive function application on `(rest lst)`.

This suggests passing the base value and the combining function as parameters to a higher order function.

> The higher order 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 `my-foldr`

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

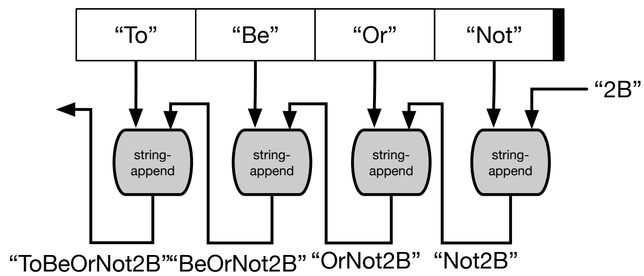
```
(my-foldr f 0 (list 3 6 5)) ⇒
(f 3 (my-foldr f 0 (list 6 5))) ⇒
(f 3 (f 6 (my-foldr f 0 (list 5))) ⇒
(f 3 (f 6 (f 5 (my-foldr f 0 empty))) ⇒
(f 3 (f 6 (f 5 0))) ⇒ ...
```

Intuitively, the effect of the application `(foldr f b (list x_1 x_2 ... x_n))` is to compute the value of the expression `(f x_1 (f x_2 (... (f x_n b))))`.

> Tracing `my-foldr`

```
(foldr f b (list x_1 x_2 ... x_n)) ⇒ (f x_1 (f x_2 (... (f x_n b))))

(foldr string-append "2B" (list "To" "Be" "Or" "Not")) => "ToBeOrNot2B"
```

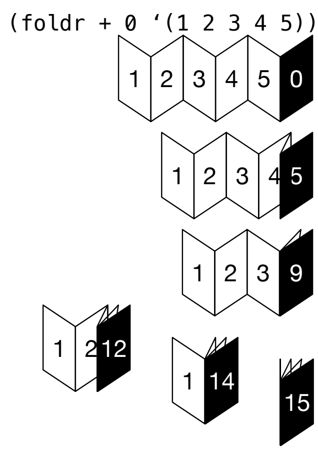


> `foldr`

`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 higher order functions.



> The contract for foldr

`foldr` consumes three arguments:

- a function which combines the first list item with the result of reducing the rest of the list;
- a base value;
- a list on which to operate.

What is the contract for `foldr`?

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

If `lst` is `(list x_1 x_2 ... x_n)`, then by our intuitive explanation of `foldr`, the expression `(foldr + 0 lst)` reduces to

```
(+ x_1 (+ x_2 (+ ... (+ x_n 0))))
```

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

The function provided to `foldr` consumes two parameters: one is an element in the list which is an argument to `foldr`, and 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`.

```
(define (count-symbols lst)
  (cond [(empty? lst) 0]
        [else (+ 1
                 (count-symbols (rest lst)))]))
```


The important thing about the first argument to the function provided to `foldr` is that it contributes 1 to the count; its actual value is irrelevant.

Thus the function provided to `foldr` in this case can ignore the value of the first parameter, and just add 1 to the reduction of the rest of the list.

```
(define (count-symbols lst) (foldr (lambda (x rror) (add1 rror)) 0 lst))
```

The function provided to `foldr`, namely
`(lambda (x rror) (add1 rror))`,
ignores its first argument.

Its second argument is the **R**esult of **R**ecursing **O**n the **R**est (`rror`) of the list (in this case the length of the rest of the list, to which 1 must be added).

What do these functions do?

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

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

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

(foo (list "one" "two" "three"))
```

Ex. 4
Use `foldr` to write a function `count-odd` that produces the number of odd numbers in a `(listof Nat)`.
There are several ways to do this. Can you do this using `map` and `foldr`? `foldr` with `filter`? Just using `foldr`?

Ex. 5
Use `foldr` to write a function `prod` that produces the product of a `(listof Num)`.
`(prod (list 2 2 3 5))` ⇒ 60

Ex. 6
Use `foldr` to write a function `total-length` that produces the number of elements in a list of lists. For your first version, use `length`. Once that is working, write a second version that replaces `length` with a second use of `foldr`.
`(total-length (list (list 1 2 3) (list 4 5) (list 1 1 1)))` ⇒ 8

Ex. 7
Use `foldr` to write a function that produces the average (mean) of a non-empty `(listof Num)`.

```
(check-expect (average (list 2 4 9)) 5)
(check-expect (average (list 4 5 6 6)) 5.25)
```

Hint: Consider using the `length` function. Can you replace `length` with a higher order function?

Ex. 8
Write a function `times-square` that consumes a `(listof Nat)` and produces the product of all the perfect squares $(1, 4, 9, 16, 25, \ldots)$ in the list.

```
(check-expect (times-square (list 1 25 5 4 1 17)) 100)
;; Since (times-square (list 1 25 5 4 1 17)) ⇒ (* 1 25 4 1) => 100
```

> 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 simple recursion on lists, so we should be able to use it to implement `negate-list` from module 06.

`negate-list` takes the first element from the list, negates it, and `cons`es it onto the result of the recursive function application.

We need to define a function `(lambda (x rror) ...)` that combines `x` and `rror` where `x` is the first element of the list and `rror` is the result of the recursive function application on the rest of the list.

> `negate-list` using `foldr`

The function we need is

```
(lambda (x rror) (cons (- x) rror))
```

Thus we can give a version of `negate-list` that is not explicitly recursive (that is, `foldr` does all the recursion).

```
(define (negate-list lst)
  (foldr (lambda (x rror) (cons (- x) rror)) empty lst))
```

Because we generalized `negate-list` to `map`, we should be able to use `foldr` to define `map`.

> `my-map` using `foldr`

Let's look at the code for `my-map`.

```
(define (my-map f lst)
  (cond [(empty? lst) empty]
        [else (cons (f (first lst))
                    (my-map f (rest lst)))]))
```

Clearly `empty` is the base value, and the combining function provided to `foldr` is something involving `cons` and `f`.

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).

```
(define (my-map f lst)
  (foldr (lambda (x rror) (cons (f x) rror)) empty lst))
```

We can also implement my-filter using foldr.

Ex. 9

The function double-each works. Rewrite it using foldr, without using map.

```
(define (double n) (* n 2))
(define (double-each lst) (map double lst))
```

Ex. 10

Using foldr and not using filter, rewrite (keep-evens lst) so it still produces the list containing all the even values in lst.

```
(define (keep-evens lst) (filter even? lst))
(check-expect (keep-evens (list 1 2 3 4 5 6)) (list 2 4 6))
```

Ex. 11

Write a function sum-even that consumes a (listof Int) and produces the sum of all the even values. (sum-evens (list 2 3 4 5)) ⇒ 6

- Use foldr, filter, and even?.
- Use foldr and filter, but use a **lambda** expression instead of even?.
- Use foldr, **lambda**, and even? but not filter.

Ex. 12

Write a function (multiply-each lst n). It consumes a (listof Num) and a Num, and produces the list containing all the values in lst, each multiplied by n.
(multiply-each (list 2 3 5) 4) ⇒ (list 8 12 20)

Ex. 13

Write a function (add-total lst) that consumes a (listof Num), and adds the total of the values in lst to each value in lst.
(add-total (list 2 3 5 10)) ⇒ (list 22 23 25 30)

Ex. 14 Write `(discard-bad lst lo hi)`. It consumes a `(listof Num)` and two `Num`. It produces the list of all values in `lst` that are between `lo` and `hi`, inclusive.
`(discard-bad (list 12 5 20 2 10 22) 10 20)` ⇒ `(list 12 20 10)`

Ex. 15 Write `(squash-bad lo hi lst)`. It consumes two `Num` and a `(listof Num)`. Values in `lst` that are greater that `hi` become `hi`; less that `lo` become `lo`.
`(squash-bad 10 20 (list 12 5 20 2 10 22))` ⇒ `(list 12 10 20 10 10 20)`

Ex. 16 Write a function `above-average` that consumes a `(listof Num)` and produces a list containing just the values which are greater than or equal to the average (mean) value in the list.

> Aside: comparison to imperative languages

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

Higher order 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.

> Summary: higher order functions vs. the list template

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.

Higher order functions should be used judiciously, to replace relatively simple uses of recursion.

Generalizing accumulative recursion

Let's look at several past functions that use recursion on a list with one accumulator.

```
;; code from lecture module 14

(define (sum-list lst0)
  (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 lst0 0)))

(check-expect (sum-list (list 1 2 3 4)) 10)
```

> Generalizing accumulative recursion

Let's look at several past functions that use recursion on a list with one accumulator.

```
;; code from lecture module 9 rewritten to use local

(define (rev-list lst0)
  (local [(define (rev-list/acc lst lst-so-far)
            (cond [(empty? lst) lst-so-far]
                  [else (rev-list/acc (rest lst)
                                      (cons (first lst) lst-so-far))]))]
    (rev-list/acc lst0 empty)))

(check-expect (rev-list (list 1 2 3 4 5)) (list 5 4 3 2 1))
```

> `foldl`

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.

```
(define (my-foldl combine base lst0)
  (local [(define (foldl/acc lst acc)
            (cond [(empty? lst) acc]
                  [else (foldl/acc (rest lst)
                                   (combine (first lst) acc))]))]
    (foldl/acc lst0 base)))
```

```
(define (sum-list lon) (my-foldl + 0 lon))
(define (my-reverse lst) (my-foldl cons empty lst))
(define (max-list lon) (my-foldl max (first lon) (rest lon)))
```

`foldl` is defined in the Intermediate Student language and above.

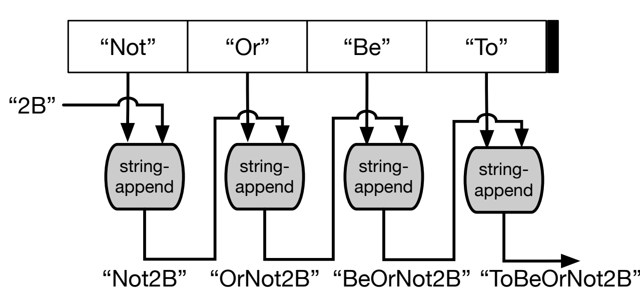
We noted earlier that intuitively, the effect of the application
`(foldr f b (list x_1 x_2 ... x_n))`
is to compute the value of the expression
`(f x_1 (f x_2 (... (f x_n b) ...)))`

What is the intuitive effect of the following application of `foldl`?

`(foldl f b (list x_1 ... x_n-1 x_n))`

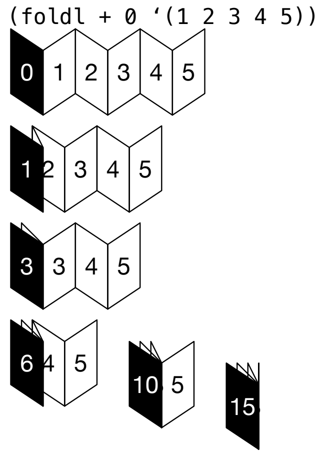
`(foldl f b (list x_1 x_2 ... x_n))` ⇒ `(f x_n (f x_n-1 (... (f x_1 b))))`

`(foldl string-append "2B" (list "Not" "Or" "Be" "To"))` ⇒ `"ToBeOrNot2B"`



foldl is short for “fold left”.

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



What is the contract of foldl?

Ex. 17

Manually evaluate the two expressions:

```
(foldl (lambda (x y) (+ x y y)) 1 (list 3 4 5))
(foldr (lambda (x y) (+ x y y)) 1 (list 3 4 5))
```

Are the values the same? Why or why not?
Then check your answer using DrRacket.

Deriving `build-list`

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

`(list (f 0) (f 1) ... (f (sub1 n)))`

Examples:

`(build-list 4 (lambda (x) x))` ⇒ `(list 0 1 2 3)`

`(build-list 4 (lambda (x) (* 2 x)))` ⇒ `(list 0 2 4 6)`

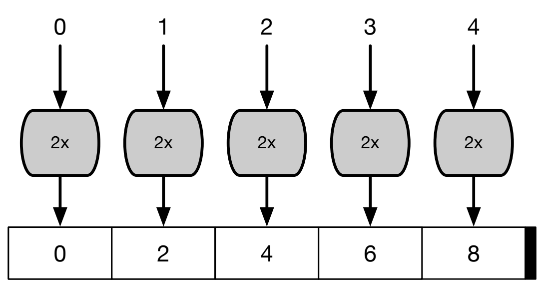
Clearly `build-list` abstracts the "count up" pattern, and it is easy to write our own version.

> `my-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)))
```

> Visualizing `build-list`

`(build-list 5 (lambda (x) (* 2 x)))`



$\sum_{i=0}^{n-1} f(i)$

```
(define (sum n f)
  (foldr + 0 (build-list n f)))

(sum 4 sqr)
⇒ (foldr + 0 (build-list 4 sqr))
⇒ (foldr + 0 (list 0 1 4 9))
⇒ 14
```

Note that two or more higher order functions can be used together to accomplish a task.

Ex. 18

The *n*th Triangular Number is given by $T_n = \frac{n(n+1)}{2}$
Write a function `(triangles k)` that uses `build-list` to produce a list containing the first `k` triangular numbers.
For example,
`(triangles 4)` ⇒ `(list 0 1 3 6)`

We can now simplify `mult-table` even further.

```
(define (mult-table nr nc)
  (build-list nr
              (lambda (r)
                (build-list nc
                            (lambda (c)
                              (* r c))))))
```

Ex. 19

Using the stepping rules as presented in this course, evaluate the following Racket expression:

```
(((lambda (m)
   (lambda (n)
     (local [(define sum (+ n m))]
       (build-list
        m
        (lambda (i)
          (+ n (cond
                 [(odd? i) (+ sum i)]
                 [else (- sum i)])))))))
  5) 3)
```

Goals of this module

- You should be familiar with the built-in higher order functions `filter`, `map`, `foldr`, `foldl`, and `build-list`. You should understand how they abstract common recursive patterns, and be able to use them to write code.
- You should be able to derive the contracts for higher order functions.
- You should be able to write your own higher order 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.

Summary: built-in functions

The following functions and special forms have been introduced in this module:

`build-list foldl foldr map`

You should complete all exercises and assignments using only these and the functions and special forms introduced in earlier modules. The complete list is:

```
* + - ... / < <= = > >= abs add1 and append boolean? build-list ceiling char-alphabetic?
    char-downcase char-lower-case? char-numeric? char-upcase char-upper-case?
    char-whitespace? char<=? char<? char=? char>=? char>? char? check-error check-expect
    check-within cond cons cons? cos define define-struct define/trace e eighth else
    empty? equal? error even? exp expt fifth filter first floor foldl foldr fourth
    integer? lambda length list list->string list? local log map max min modulo negative?
    not number->string number? odd? or pi positive? quotient remainder rest reverse round
    second seventh sgn sin sixth sqr sqrt string->list string-append string-downcase
    string-length string-lower-case? string-numeric? string-upcase string-upper-case?
    string<=? string<? string=? string>=? string>? string? sub1 substring symbol=? symbol?
    tan third zero?
```