# Functional abstraction

**Readings:** HtDP, sections 21-24.

**Language level:** Intermediate Student With Lambda

**Topics:**

- Anonymous functions
- Syntax & semantics
- Example: Transforming strings
- Abstracting: Map
- Abstracting: Foldr
- Abstracting: Foldl
- Abstracting: Build-list

# 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.

In the previous module 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 **Abstract List Functions** in this module.

But first, we promised an easier way to produce functions.

# Anonymous functions

```
(define (make-adder n)
  (local [(define (f m) (+ n m))]
    f))
(make-adder 3)
```

The result of evaluating this expression is a function.

What is its name? It is **anonymous** (has no name).

This is sufficiently valuable that there is a special mechanism for it.

```
(define (not-symbol-apple? item) (not (symbol=? item 'apple)))
(define (eat-apples lst) (filter not-symbol-apple? lst))
```

This is a little unsatisfying, because `not-symbol-apple?` is such a small and relatively useless function.

It is unlikely to be needed elsewhere.

We can avoid cluttering the top level with such definitions by putting them in **`local`** expressions.

```
(define (eat-apples lst)
  (local [(define (not-symbol-apple? item)
            (not (symbol=? item 'apple)))]
    (filter not-symbol-apple? lst)))
```

This is as far as we would go based on our experience with **local**.

But now that we can use functions as values, the value produced by the local expression can be the function `not-symbol-apple?`.

We can then take that value and deliver it as an argument to `filter`.

# > Producing anonymous functions

```
(define (eat-apples lst)
  (filter (local [(define (not-symbol-apple? item)
                    (not (symbol=? item 'apple)))]
            not-symbol-apple?)
          lst))
```

But this is still unsatisfying. Why should we have to name `not-symbol-apple?` at all? In the expression `(* (+ 2 3) 4)`, we didn't have to name the intermediate value 5.

Racket provides a mechanism for constructing a nameless function which can then be used as an argument.

# > Introducing lambda

```
(local [(define (name-used-once x_1 ... x_n) exp)]
   name-used-once)
```

can also be written

```
(lambda (x_1 ... x_n) exp)
```

`lambda` can be thought of as “make-function”.

It can be used to create a function which we can then use as a value – for example, as the value of the first argument of `filter`.

# > Example: define `eat-apples` with `lambda`

We can use **`lambda`** to replace

```
(define (eat-apples lst)
  (filter (local [(define (not-symbol-apple? item)
                    (not (symbol=? item 'apple)))]
             not-symbol-apple?)
          lst)
```

with the following:

```
(define (eat-apples lst)
  (filter (lambda (item) (not (symbol=? item 'apple))) lst))
```

# > Introducing `lambda`

`lambda` is available in Intermediate Student with Lambda, and discussed in section 24 of the textbook.

The word `lambda` comes from the Greek letter, used as notation in the first formal model of computation.

We'll learn more about its central importance in the history of computation in the last lecture module.

# > Using `lambda`

We can use `lambda` to simplify `make-adder`. Instead of

```
(define (make-adder n)
  (local [(define (f m) (+ n m))]
    f))
```

we can write:

```
(define (make-adder n)
  (lambda (m) (+ n m)))
```

`lambda` underlies the definition of functions.

Until now, we have had two different types of definitions.

```
;; a definition of a numerical constant
(define interest-rate 3/100)
;; a definition of a function to compute interest
(define (interest-earned amount)
  (* interest-rate amount))
```

But there is really only one kind of `define`, which binds a name to a value.

# > `lambda` and function definitions

Internally,

```
(define (interest-earned amount)
  (* interest-rate amount))
```

is translated to

```
(define interest-earned
  (lambda (amount) (* interest-rate amount)))
```

which binds the name `interest-earned` to the value `(lambda (amount) (* interest-rate amount))`.

# > `lambda` and function definitions

We should change our semantics for function definition to represent this rewriting.

But doing so would make traces much harder to understand.

As long as the value of defined constants (now including functions) cannot be changed, we can leave their names unsubstituted in our traces for clarity.

In stepper questions, if a function is defined using function syntax, you can skip the lambda substitution step. If a function is defined as a constant using lambda, you must include the lambda substitution step.

For example, here's `make-adder` rewritten using **`lambda`**.

```
(define make-adder
  (lambda (x)
    (lambda (y)
      (+ x y))))
```

What is `((make-adder 3) 4)`?

# > Example: Tracing with lambda

```
(define make-adder
  (lambda (x)
    (lambda (y)
      (+ x y))))
```

```
(define make-adder (lambda (x) (lambda (y) (+ x y))))
((make-adder 3) 4) ⇒    ;; substitute the lambda expression
(((lambda (x) (lambda (y) (+ x y))) 3) 4) ⇒
((lambda (y) (+ 3 y)) 4) ⇒
(+ 3 4) ⇒ 7
```

`make-adder` is defined as a constant using lambda. Like any other constant, `make-adder` is replaced by its value (the **`lambda`** expression).

# Exercise 1

Using `lambda` and `filter` but no named helper functions, write a function that consumes a `(listof Str)` and returns a list containing all the strings that have a length of 4.

```
(keep4
'("There's" "no" "fate" "but" "what" "we" "make" "for" "ourselves"))
=> '("fate" "what" "make")
```

# Syntax and semantics of Intermed. Student w/ lambda

We need to revise our syntax and semantics to handle cases such as `((make-adder 3) 4)`. We noted the differences earlier:

| Before | Now |
|---|---|
| First position in an application must be a built-in or user-defined function. | First position can be an expression (computing the function to be applied). Evaluate it along with the other arguments. |
| A function name had to follow an open parenthesis. | A function application can have two or more open parentheses in a row: `((make-adder 3) 4)`. |

# > Substitution rule

We need a rule for evaluating applications where the function being applied is anonymous (a `lambda` expression).

```
((lambda (x_1 ... x_n) exp) v_1 ... v_n) => exp'
```

where `exp'` is `exp` with all occurrences of `x_1` replaced by `v_1`, all occurrences of `x_2` replaced by `v_2`, and so on.

As an example:

```
((lambda (x y) (* (+ y 4) x)) 5 6)
⇒ (* (+ 6 4) 5)
⇒ ... ⇒ 50
```

# Example: character transformation in strings

Suppose during a computation, we want to specify some action to be performed one or more times in the future.

Before knowing about `lambda`, we might build a data structure to hold a description of that action, and a helper function to consume that data structure and perform the action.

Now, we can just describe the computation clearly using `lambda`.

# > Example: character transformation in strings

We'd like a function, `transform`, that transforms one string into another according to a set of rules that are specified when it is applied.

In one application, we might want to change every instance of 'a' to a 'b'. In another, we might transform lowercase characters to the equivalent uppercase character and digits to '*'.

```
(check-expect (transform "abracadabra" ...) "bbrbcbdbbrb")
(check-expect (transform "Testing 1-2-3" ...) "TESTING *-*-*")
```

We use ... to indicate that we still need to supply some arguments.

# > Example: inspiration

We could imagine `transform` containing a **cond**:

```
(cond [(char=? ch #\a) #\b]
      [(char-lower-case? ch) (char-upcase ch)]
      [(char-numeric? ch) #\*]
      ...)
```

But this fails for a number of reasons:

- The rules are "hard-coded"; we want to supply them when `transform` is applied.
- A lower case 'a' would always be transformd to 'b'; never to 'B'

But the idea is inspiring...

Suppose we supplied `transform` with a list of question/answer pairs:

```
;; A TransformSpec is one of:
;; * empty
;; * (cons (list Question Answer) TransformSpec)
```

Like **`cond`**, we could work our way through the `TransformSpec` with each character. If the Question produces `true`, then apply the Answer to the character. If the Question produces `false`, go on to the next Question/Answer pair.

What are the types for `Question` and `Answer`?

Functions as first class values can help us. Both `Question` and `Answer` are functions that consume a `Char`.

`Question` produces a `Bool` and `Answer` produces a character. This completes our data definition, above:

```
;; A Question is a (Char → Bool)
;; An Answer is a (Char → Char)
```

And a completed example:

```
(check-expect (transform "Testing 1-2-3"
                         (list (list char-lower-case? char-upcase)
                               (list char-numeric? (lambda (ch) #\*))))
              "TESTING *-*-*")
```

# > Transform: developing the code (1/3)

`transform` consumes a string and produces a string but we need to operate on characters. This suggests a wrapper function:

```
;; A TransformSpec is one of:
;; * empty
;; * (cons (list Question Answer) TransformSpec)


;; (transform s spec) transforms the string s according to the given
;;    specification.
;; transform: Str TransformSpec → Str
(define (transform s spec)
  (list→string (trans-loc (string→list s) spec)))
```

# > Transform: developing the code (2/3)

```
;; trans-loc (listof Char) TransformSpec → (listof Char)
(check-expect (trans-loc (list #\a #\9)
                         (list (list char-lower-case? char-upcase)))
                            (list #\A #\9))
(define (trans-loc loc spec)
  (cond [(empty? loc) empty]
        [(cons? loc) (cons (trans-char (first loc) spec)
                           (trans-loc (rest loc) spec))]))


(define (trans-char ch spec)
  (cond [(empty? spec) ch]
        [((first (first spec)) ch) ((second (first spec)) ch)]
        [else (trans-char ch (rest spec))]))
```

# > Transform: developing the code (3/3)

```
(check-expect (transform "Testing 1-2-3"
                         (list (list char-lower-case? char-upcase)
                               (list char-numeric? (lambda (ch) #\*))))
              "TESTING *-*-*")
(check-expect (transform "abracadabra"
                         (list (list (lambda (ch) (char=? ch #\a))
                                     (lambda (ch) #\b))))
              "bbrbcbdbbrb")
```

The repeated **lambda** expressions suggest some utility functions:

```
(define (is-char? c1) (lambda (c2) (char=? c1 c2)))
(define (always c1)   (lambda (c2) c1))
```

# 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-taxes payroll)
  (cond [(empty? payroll) empty]
        [else (cons (sr→tr (first payroll))
                    (compute-taxes (rest payroll)))]))
```

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 (compute-taxes payroll)
  (cond [(empty? payroll) empty]
        [else (cons (sr→tr (first payroll))
                    (compute-taxes (rest payroll)))]))

(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)))]))
```

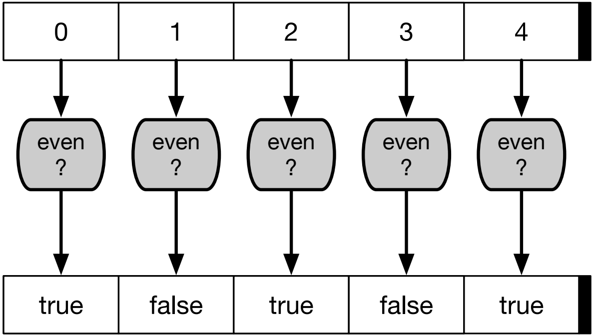
```
(my-map sqr (list 3 6 5))
⇒ (cons 9 (my-map sqr (list 6 5)))
⇒ (cons 9 (cons 36 (my-map sqr (list 5))))
⇒ (cons 9 (cons 36 (cons 25 (my-map sqr empty))))
⇒ (cons 9 (cons 36 (cons 25 empty)))
```

`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? '(0 1 2 3 4))
```



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))
```

How can we use my-map to rewrite trans-loc?

# > 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?

# > Built-in abstract list functions

In addition to `filter`, 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.

# Exercise 2

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 L)` that consumes a `(listof Nat)`, and returns a `(listof Num)` so numbers on the interval $[0, 255]$ are scaled to the interval $[0, 1]$.

```
(squash-range '(0 204 255)) => '(0 0.8 1)
```

# Exercise 3

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

`(greet-each '("Ali" "Carlos" "Sai"))` ⇒ `'("Hi Ali!" "Hi Carlos!" "Hi Sai!")`

# Exercise 4

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

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

# ALFs 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 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 rest of the ellipses 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 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 my-foldr

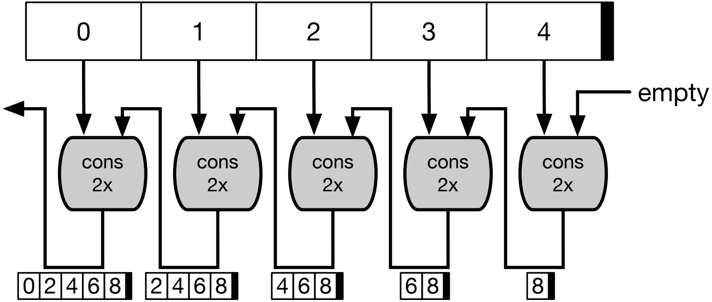
```
(my-foldr f 0 (list 3 6 5)) ⇒
(f 3 (my-foldr f 0 (list 6 5))) ⇒
(f 3 (f 6 (my-foldr f 0 (list 5)))) ⇒
(f 3 (f 6 (f 5 (my-foldr f 0 empty)))) ⇒
(f 3 (f 6 (f 5 0))) ⇒ ...
```

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))))
```

```
(define (cons2x x lst) (cons (* 2 x) lst))
(foldr cons2x empty '(0 1 2 3 4))
(cons2x 0 (cons2x 1 (cons2x 2 (cons2x 3
    (cons2x 4 empty)))))
```



`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:

- 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`.

# > Using foldr

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)))]))
```

# > Using `foldr`

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 '(1 5 23 3 99 2))

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

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

# Exercise 5

Use `foldr` to write a function `count-odd` that returns the number of odd numbers in a `(listof Nat)`.

Hint: read the documentation on `remainder`.

Can you do this using `map` and `foldr`? Just using `foldr`?

# Exercise 6

Use `foldr` to write a function `prod` that returns the product of a `(listof Num)`.

`(prod '(2 2 3 5))` $\Rightarrow$ `60`

# Exercise 7

Use `foldr` to write a function `total-length` that returns the total length of all the values in a `(listof Str)`.

`(total-length (list "hello" "how" "r" "u?"))` ⇒ 11

# Exercise 8

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

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

# Exercise 9

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

```
(check-expect (times-square '(1 25 5 4 1 17)) 100)
;; Since (times-square '(1 25 5 4 1 7)) => (* 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.

We need to define a function `(lambda (x rror) ...)` 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 `cons`es it onto the result of the recursive function application.

The function we need is

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

Thus we can give a nonrecursive version of `negate-list` (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`.

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.

# Exercise 10

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

```
(define (double n) (* n 2))


(define (double-each L) (map double L))
```

# Exercise 11

Using `foldr`, write a function `(keep-evens L)` that returns the list containing all the even values in `L`.

That is, rewrite this function, using `foldr` but not using `filter`:

```
(define (keep-evens L)
  (filter even? L))

(check-expect (keep-evens '(1 2 3 4 5 6)) '(2 4 6))
```

# Exercise 12

Using `lambda` but no named help functions, write a function that consumes a `(listof Int)` and returns the sum of all the even values.

`(sum-evens (list 2 3 4 5))` ⇒ `6`

Can you do it using `lambda` just once and `foldr` just once?

# Exercise 13

Write a function `(multiply-each L n)`. It consumes a `(listof Num)` and a `Num`, and returns the list containing all the values in `L`, each multiplied by `n`.

`(multiply-each (list 2 3 5) 4)` $\Rightarrow$ `(list 8 12 20)`

# Exercise 14

Write a function `(add-total L)` that consumes a `(listof Num)`, and adds the total of the values in `L` to each value in `L`.

`(add-total (list 2 3 5 10))` $\Rightarrow$ `(list 22 23 25 30)`

# Exercise 15

Write `(discard-bad L lo hi)`. It consumes a `(listof Num)` and two `Num`. It returns the list of all values in `L` that are between `lo` and `hi`, inclusive.

`(discard-bad '(12 5 20 2 10 22) 10 20)` ⇒ `'(12 20 10)`

# Exercise 16

Write `(squash-bad lo hi L)`. It consumes two `Num` and a `(listof Num)`. Values in `L` that are greater that `hi` become `hi`; less that `lo` become `lo`.

`(squash-bad 10 20 '(12 5 20 2 10 22)))` ⇒ `'(12 10 20 10 10 20)`

# Exercise 17

Write a function `above-average` that consumes a `(listof Num)` and returns the 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.

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.

# > Summary: ALFs 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.

Abstract list 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 12

(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)))
```

# > 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)))
```

# > `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.

# > foldl

```
(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))
```

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) ...)))
```

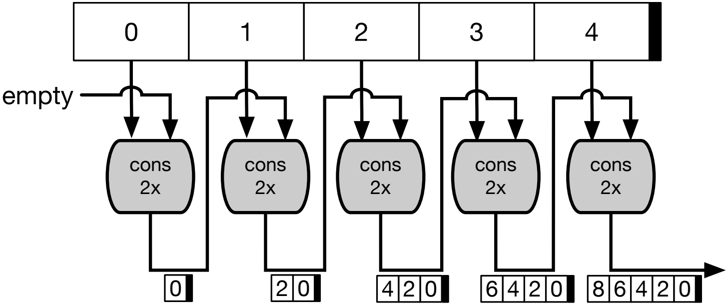
What is the intuitive effect of the following application of `foldl`?

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

# > Tracing `foldl`

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

```
(define (cons2x x lst) (cons (* 2 x) lst))
(foldl cons2x empty '(0 1 2 3 4))
(cons2x 4 (cons2x 3 (cons2x 2 (cons2x 1
    (cons2x 0 empty)))))
```



# > Contract for `foldl`

The function `foldl` is provided in Intermediate Student.

What is the contract of `foldl`?

# Deriving `build-list`

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

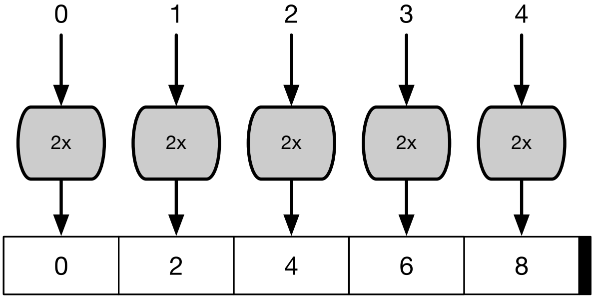
```
(list (f 0) (f 1) ... (f (sub1 n)))
```

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

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

```
(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 5 (lambda (x) (* 2 x)))
```



# > Build-list examples

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

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

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

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))))))
```

# Goals of this module

- You should be able to produce functions using `lambda`.
- You should understand how `lambda` underlies our usual definition of functions.
- You should be familiar with the built-in abstract list 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 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.