Functional abstraction


Language level: Intermediate Student With Lambda

Topics:

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

\[(\text{define} \ (\text{make-adder} \ n) \ \\
\quad (\text{local} \ [(\text{define} \ (f \ m) \ (+ \ n \ m))] \ \\
\quad \ f))
\]
\[(\text{make-adder} \ 3)\]

The result of evaluating this expression is a function.

What is its name? It is \texttt{anonymous} (has no name).

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

```
(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.
(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 \texttt{lambda}

\begin{verbatim}
(local [(define (name-used-once x_1 ... x_n) exp)]
  name-used-once)
\end{verbatim}

can also be written

\begin{verbatim}
(lambda (x_1 ... x_n) exp)
\end{verbatim}

\texttt{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 \texttt{filter}.
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

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

we can write:

```scheme
(define (make-adder n)
  (lambda (m) (+ n m)))
```
lambda and function definitions

**lambda** underlies the definition of functions.

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

```scheme
;; 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.
Internally,

\[
(\text{define} \ (\text{interest-earned} \ \text{amount}) \\
(\ast \ \text{interest-rate} \ \text{amount}))
\]

is translated to

\[
(\text{define} \ \text{interest-earned} \\
(\lambda \ \text{amount} \ (\ast \ \text{interest-rate} \ \text{amount})))
\]

which binds the name \text{interest-earned} to the value

\[
(\lambda \ \text{amount} \ (\ast \ \text{interest-rate} \ \text{amount})).
\]
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.
Example: Tracing with lambda

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

**Before**
First position in an application must be a built-in or user-defined function.

A function name had to follow an open parenthesis.

**Now**
First position can be an expression (computing the function to be applied). Evaluate it along with the other arguments.

A function application can have two or more open parentheses in a row:
((make-adder 3) 4).
We need a rule for evaluating applications where the function being applied is anonymous (a \texttt{lambda} expression).

\[
((\texttt{lambda} \ (x_1 \ldots \ x_n) \ \text{exp}) \ v_1 \ldots \ v_n) \Rightarrow \text{exp}'
\]

where \text{exp}' is \text{exp} with all occurrences of \texttt{x\_1} replaced by \texttt{v\_1}, all occurrences of \texttt{x\_2} replaced by \texttt{v\_2}, and so on.

As an example:

\[
((\texttt{lambda} \ (x \ y) \ (* \ (+ \ y \ 4) \ x)) \ 5 \ 6)
\Rightarrow (* \ (+ \ 6 \ 4) \ 5)
\Rightarrow \ldots \Rightarrow 50
\]
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" ...) "bbrbcdbbbrb")
(check-expect (transform "Testing 1-2-3" ...) "TESTING *-*-*")

We use ... to indicate that we still need to supply some arguments.
We could imagine `transform` containing a `cond`:

```lisp
(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 transformed to ‘b’; never to ‘B’

But the idea is inspiring...
Suppose we supplied `transform` with a list of question/answer pairs:

```plaintext
;; 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:

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

And a completed example:

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

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.

```scheme
(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)))]))
```
> Abstracting another set of examples

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))))])
```

Anonymous functions
Syntax
Example
Map
Foldr
Foldl
Build-list
27/64
14: Functional abstraction
CS 135 Winter 2020
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))

```
0 | 1 | 2 | 3 | 4
---|---|---|---|---
even? | even? | even? | even? | even?
true | false | true | false | true
```

Anonymous functions

Syntax

Example

Map

Foldr

Foldl

Build-list

29/64 14: Functional abstraction

CS 135 Winter 2020
Using **my-map**

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

\[
\text{(define (negate-list lst) (my-map - lst))}
\]

\[
\text{(define (compute-taxes lst) (my-map sr→tr lst))}
\]

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?
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.
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

\[
\textbf{(define} \ (\text{sum-of-numbers}\ \text{lst})
\text{\textbf{(cond} } [(\text{empty?}\ \text{lst})\ 0]
\text{\textbf{else} } (+ \ (\text{first}\ \text{lst})
\text{\textbf{(sum-of-numbers} \ (\text{rest}\ \text{lst})))})])
\]

\[
\textbf{(define} \ (\text{prod-of-numbers}\ \text{lst})
\text{\textbf{(cond} } [(\text{empty?}\ \text{lst})\ 1]
\text{\textbf{else} } (* \ (\text{first}\ \text{lst})
\text{\textbf{(prod-of-numbers} \ (\text{rest}\ \text{lst})))})])
\]

\[
\textbf{(define} \ (\text{all-true?}\ \text{lst})
\text{\textbf{(cond} } [(\text{empty?}\ \text{lst})\ true]
\text{\textbf{else} } (\text{and} \ (\text{first}\ \text{lst})
\text{\textbf{(all-true?} \ (\text{rest}\ \text{lst})))})])
\]
> 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.

<table>
<thead>
<tr>
<th>Anonymous functions</th>
<th>Syntax</th>
<th>Example</th>
<th>Map</th>
<th>Foldr</th>
<th>Foldl</th>
<th>Build-list</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>oo</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35/64</td>
<td></td>
<td>14: Functional abstraction</td>
<td></td>
<td></td>
<td></td>
<td>CS 135 Winter 2020</td>
</tr>
</tbody>
</table>
> Comparison to the list template

\[
\text{(define \text{list-template} \text{lst})} \\
\quad \text{(cond \)} \\
\qquad \text{([empty? \text{lst}) ...]} \\
\qquad \qquad \text{[else ( ... (first \text{lst}) ...) \text{else \text{template} (rest \text{lst}) ...}]})
\]

We replace the first ellipsis by a base value.

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

This suggests passing the base value and the combining function as parameters to an abstract list function.
The abstract list function \texttt{foldr}

\begin{verbatim}
(define (my-foldr combine base lst)
  (cond [(empty? lst) base]
        [else (combine (first lst)
                      (my-foldr combine base (rest lst)))]))
\end{verbatim}

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

\[
\begin{align*}
\text{(my-foldr } f \ 0 \ (\text{list } 3 \ 6 \ 5)) \Rightarrow \\
(f \ 3 \ (\text{my-foldr } f \ 0 \ (\text{list } 6 \ 5))) \Rightarrow \\
(f \ 3 \ (f \ 6 \ (\text{my-foldr } f \ 0 \ (\text{list } 5)))) \Rightarrow \\
(f \ 3 \ (f \ 6 \ (f \ 5 \ (\text{my-foldr } f \ 0 \ \text{empty})))) \Rightarrow \\
(f \ 3 \ (f \ 6 \ (f \ 5 \ 0))) \Rightarrow ...
\end{align*}
\]

Intuitively, the effect of the application
\[
(foldr \ f \ b \ (\text{list } x_1 \ x_2 \ ... \ x_n))
\]
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?
Using foldr

\[
\text{(define (sum-of-numbers lst) (foldr + 0 lst))}
\]

If \( \text{lst} \) is \((\text{list } x_1 \ x_2 \ \ldots \ x_n)\), then by our intuitive explanation of \text{foldr}, the expression \((\text{foldr } + 0 \ \text{lst})\) reduces to

\[
(+ \ x_1 (+ x_2 (+ \ldots (+ x_n 0))))
\]

Thus \text{foldr} does all the work of the template for processing lists, in the case of \text{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)))])
```

Anonymous functions
Syntax
Example
Map
Foldr
Foldl
Build-list
14: Functional abstraction
CS 135 Winter 2020
> 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 result of recursing on the rest (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")
```
> 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 \( \text{lambda} (x \text{ rror}) \ldots \) where \( x \) is the first element of the list and \( \text{rror} \) is the result of the recursive function application.

negate-list takes this element, negates it, and \text{cons}es it onto the result of the recursive function application.
The function we need is

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

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

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

Because we generalized \text{negate-list} to \text{map}, we should be able to use \text{foldr} to define \text{map}. 

---

Anonymous functions  
Syntax  
Example  
Map  
Foldr  
Foldl  
Build-list  

47/64  
14: Functional abstraction  
CS 135 Winter 2020
> 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 \texttt{foldr} must apply \texttt{f} to its first argument, then \texttt{cons} the result onto its second argument (the reduced rest of the list).

\begin{verbatim}
(define (my-map \textit{f} \textit{lst})
    (foldr (lambda (x \texttt{rror}) (cons (\textit{f} x) \texttt{rror})) empty \textit{lst}))
\end{verbatim}

We can also implement \texttt{my-filter} using \texttt{foldr}.
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)))
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)))
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))
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

\[ (\text{foldl } f \ b \ (\text{list } x_1 \ x_2 \ \ldots \ x_n)) = (f \ x_n \ (f \ x_{n-1} \ (\ldots \ (f \ x_1 \ b)))) \]

\[ (\text{define } (\text{cons}2x \ x \ \text{lst}) = (\text{cons } (* 2 \ x) \ \text{lst})) \]

\[ (\text{foldl } \text{cons}2x \ \text{empty} \ '(0 1 2 3 4)) \]

\[ (\text{cons}2x \ 4 \ (\text{cons}2x \ 3 \ (\text{cons}2x \ 2 \ (\text{cons}2x \ 1 \ (\text{cons}2x \ 0 \ \text{empty})))))) \]
The function `foldl` is provided in Intermediate Student.

What is the contract of `foldl`?
Another useful built-in ALF is \texttt{build-list}. This consumes a natural number \(n\) and a function \(f\), and produces the list
\[
\text{\texttt{(list } (f \ 0) \ (f \ 1) \ldots \ (f \ \texttt{(sub1 } n)))}
\]

\[
\text{\texttt{(build-list 4 } \texttt{(lambda (x) x)) \Rightarrow (list } 0 \ 1 \ 2 \ 3).
\]

Clearly \texttt{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)))
Visualizing build-list

(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 \texttt{mult-table} even further.

\[
\begin{align*}
\text{(define} & \text{ (mult-table nr nc)} \\
\text{(build-list} & \text{ nr)} \\
\text{(lambda} & \text{ (r)} \\
\text{(build-list} & \text{ nc)} \\
\text{(lambda} & \text{ (c)} \\
\text{(*} & \text{ r c)})\\
\end{align*}
\]
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.