# Chapter 28

# Functions as objects

For this chapter, switch languages in DrRacket to "Intermediate Student Language with Lambda" or higher. (We'll discuss `lambda` in Section 28.6.)

## 28.1 Adding parameters

The original reason for defining functions was to avoid writing almost the same expression over and over. Recall the rule from Chapter 4, *if you write almost the exact same thing over and over, you're doing something wrong.* So we *parameterized* the expression: made it a function definition with one or more *parameters*. The part of those expressions that was *always the same* became the *body* of the function definition, and the part that was *different* became a *parameter* to the function.

Recall Exercise 22.5.3, a function `contains-doll?` which checked whether the string `"doll"` occurs in a given list. Now imagine `contains-baseball?`, which checks whether the string `"baseball"` occurs in a given list. The definitions of these two functions would be almost identical, except that the word `"doll"` in one is replaced with `"baseball"` in the other.

We respond, again, by *parameterizing* — in this case, *adding a parameter* to a function that already has one. In Exercise 22.5.4, we wrote a function `any-matches?` that took in an extra parameter and searches for *that object* in a given list. Once we've written, tested, and debugged that more *general* function, `contains-doll?` and `contains-baseball?` and anything else like them is trivial.

Now suppose we had written a function `any-over-10?` that took a list of numbers and tells whether any of the numbers are larger than 10. (This should be an easy exercise by now.) An obvious variant on this would be `any-over-5?`, whose definition would be exactly the same, only with the number 5 replacing the number 10. To avoid having to write each of these nearly-identical functions, we again *add a parameter*:

```
; any-over?  :    number list-of-numbers -> boolean
(check-expect (any-over?  5 empty) false)
(check-expect (any-over?  5 (list 3)) false)
(check-expect (any-over?  5 (list 5)) true)
(check-expect (any-over?  5 (list 6)) true)
(check-expect (any-over?  5 (list 7 2 4)) true)
(check-expect (any-over?  5 (list 3 2 4)) false)
(check-expect (any-over?  5 (list 3 2 5)) true)
```

```
(define (any-over?    threshold numbers)
  (cond [(empty?  numbers) false]
        [(cons?  numbers)
         (or (> (first numbers)  threshold)
             (any-over?    threshold (rest numbers)))]))
```

## 28.2 Functions as parameters

What if we wanted to find whether any of the elements of a list were *greater than or equal to* a particular threshold? Or if any of the elements of a list were *smaller* than a particular threshold? `any-over?` won't do it; the obvious analogues would be `any-at-least?` and `any-under?`, which would be just like `any-over?` but with a >= or < in place of the > in the above definition.

Again, we're writing *almost exactly the same code over and over*, which probably means we're *doing something wrong*. And as before, the answer is to add a parameter. But this time the "part that's different" is a *function* rather than a number or a string. Which means that our general function will have to *take a function as a parameter*.

You've already seen some functions — `on-tick`, `on-draw`, `check-with`, `map-image`, `build-image`, *etc.* — that take in a function as an argument. But you haven't been allowed to *write* such functions yourself. This is because, for absolute beginning programmers, confusing a function with a variable is a very common mistake. But now that you're past those beginning mistakes, **switch languages** to DrRacket's "Intermediate Student Language with lambda".

In Racket, *function* is a data type, just like *number* or *string* or *list*. If you can pass a number or a list as an argument, there's no reason not to pass a function as an argument. Functions that operate on other functions are called *higher-order functions*.

The following exercise generalizes `any-over?` to allow for `any-under?`, `any-at-least?`, `any-equal?`, *etc.*

**Worked Exercise 28.2.1** ***Develop*** *a function* `any-compares?` *that takes in a function, a number, and a list of numbers, and tells whether any of the numbers in the list has the specified relationship to the fixed number.*

**Solution:** The contract and examples look like

```
; any-compares?  :    function number list-of-numbers -> boolean
(check-expect (any-compares?  >= 5 (list 2 5 1)) true)
(check-expect (any-compares?  > 5 (list 2 5 1)) false)
(check-expect (any-compares?  = 5 (list 2 5 1)) true)
(check-expect (any-compares?  = 5 (list 2 6 1)) false)
(check-expect (any-compares?  < 5 (list 2 6 1)) true)
(check-expect (any-compares?  < 5 (list 7 6 8)) false)
```

Before we go on with developing the function, let's look again at the contract. The first argument to `any-compares?` is supposed to be a function, so let's try some other functions.

What would `(any-compares? + 5 (list 7 6 8))` do? Well, it compares each number in the list with 5, using the + operator, and if any of these questions returns `true` ... but wait! The + operator doesn't return a boolean at all! Calling `any-compares?` doesn't make sense unless the function you give it returns a boolean.

What would `(any-compares? positive? 5 (list 7 6 8))` do? It should compare each number in the list with 5, using `positive?` ... but wait! The `positive?` function only takes one parameter, so how can it possibly "compare" two numbers? In fact, calling `any-compares?` doesn't make sense unless the function you give it takes exactly two parameters.

To rule out such nonsensical uses of `any-compares?`, let's make the contract more precise: instead of just saying "function", we'll write down (in parentheses) the *contract* that the function must satisfy:

```
; any-compares? : (number number -> boolean)
                  number
                  list-of-numbers
                  -> boolean
```

The skeleton looks like

```
(define (any-compares? compare? num nums)
  ; compare?          number number -> boolean
  ; num               number
  ; nums              list of numbers
  (cond [(empty? nums) ...]
        [(cons? nums)
         ; (first nums) number
         ; (rest nums)  list of numbers
         ; (any-compares? compare? num (rest nums))  boolean
         ...]))
```

So what can we do with `compare?`? It's a function on two numbers, returning a boolean, so the obvious thing to do with it is to call it on two numbers. Conveniently, we have two numbers in the inventory: `num` and `(first nums)`. There are two ways we could call the function:

```
; (compare? (first nums) num) boolean
; (compare? num (first nums)) boolean
```

To see which one (or both) will actually help us, let's use an inventory with values. We'll pick the example `(any-compares? < 5 (list 2 6 7))` in which the first comparison should be true, and the others false.

```
(define (any-compares?  compare?  num nums)
  ; compare?        number number -> boolean          <
  ; num                                 number        5
  ; nums                                list of numbers (list 2 6 7)
  (cond [(empty?  nums) ...]
        [(cons?  nums)
         ; (first nums)                 number        2
         ; (rest nums)                  list of numbers (list 6 7)
         ; (compare?  (first nums) num) boolean       true
         ; (compare?  num (first nums)) boolean       false
         ; (any-compares?  compare?  num (rest nums))
                                        boolean       false
         ; right answer                 boolean       true
         ...]))
```

So of the two, we seem to need `(compare?  (first nums) num)`. The final definition (with the scratch work removed) is

```
(define (any-compares?  compare?  num nums)
  (cond [(empty?  nums) false ]
        [(cons?  nums)
         (or (compare?  (first nums) num)
             (any-compares?  compare?  num (rest nums)))]))
```



Now that we've written `any-compares?`, if we *want* `any-over?`, `any-greater?`, `any-less?`, `any-over-5?`, *etc.*, we can write them easily:

```
(define (any-over?  num nums)
  (any-compares?  > num nums))
(define (any-at-least?  num nums)
  (any-compares?  >= num nums))
(define (any-less?  num nums)
  (any-compares?  < num nums))
(define (any-over-5?  nums)
  (any-compares?  > 5 nums))
```

With the aid of a helper function, we can go even farther:

```
(define (divisible-by?  x y)
  (zero?  (remainder x y)))
(define (any-even?  nums)
  (any-compares?  divisible-by?  2 nums))
```

Or, if `divisible-by?` isn't likely to be used anywhere else, we could wrap it up in a `local`:

```
(define (any-even?  nums)
  (local [(define (divisible-by?  x y)
            (zero?  (remainder x y))) ]
    (any-compares?  divisible-by?  2 nums)))
```

What if we had defined `prime?`, as in Exercise 24.3.8, and wanted to know whether any of the numbers in the list are prime? There's no obvious way to use `any-compares?` to solve this problem, because `any-compares?` insists on comparing each element of the list with a particular fixed value. In fact, a more natural function than `any-compares?` would be

```
; any-satisfies?  (number -> boolean) list-of-numbers -> boolean
(check-expect (any-satisfies?  even?  (list 3 5 9)) false)
(check-expect (any-satisfies?  even?  (list 3 5 8)) true)
(define (over-5?  x) (> x 5))
(check-expect (any-satisfies?  over-5?  (list 2 3 4)) false)
(check-expect (any-satisfies?  over-5?  (list 2 6 4)) true)
(check-expect (any-satisfies?  prime?  (list 2 6 4)) true)
(check-expect (any-satisfies?  prime?  (list 8 6 4)) false)
```

The definition (after taking out the scratch work) is

```
(define (any-satisfies?  test?  nums)
  (cond [(empty?  nums) false]
        [(cons?  nums)
         (or (test?  (first nums))
             (any-satisfies?  test?  (rest nums)))]))
```

**Worked Exercise 28.2.2** *Suppose we had written* ***`any-satisfies?`*** first. ***Define*** *the function* ***`any-compares?`****, taking advantage of having already written the more general function.*

**Solution:** Since `any-satisfies?` takes in a function of only one parameter, we need a helper function that takes one parameter.

```
; ok?  :  num -> boolean
; applies the comparison function to
; the parameter and the fixed number
```

But what are "the comparison function" and "the fixed number"? They're both parameters to `any-compares?`, so we can't possibly know what they are until `any-compares?` is called. However, we can write and test it by storing sample values of both in global variables:

```
(define num 17)
(define compare?  >)
(check-expect (ok?  14) false)
(check-expect (ok?  17) false)
(check-expect (ok?  19) true)
(define (ok?  num-from-list)
  ; num-from-list number
  ; num           number
  ; compare?      (number number -> boolean)
  (compare?  num-from-list num))
```

Once this has passed its tests, *change* the definitions of `num` and `compare?`, *change* the test cases appropriately, and see if it still passes the tests. If so, we perhaps have enough confidence in it to *move it inside another function*:

```
(define (any-compares?  compare?  num nums)
  (local [(define (ok?  num-from-list)
            (compare?  num-from-list num)) ]
    (any-satisfies?  ok?  nums)))
```

Again, this helper function *could not* have been written to stand by itself at the top level, because it needs to know what `compare?` and `num` are. ∎

Looking closely at the definition of `any-satisfies?`, we notice that nothing in the code (except the variable names) actually mentions *numbers*. In fact, if we were to call `any-satisfies?` on a function from *string* to boolean and a list of *strings*, it would work just as well:

```
(define (contains-doll?  toys)
  (local [(define (is-doll?  toy) (string=?  toy "doll"))]
    (any-satisfies?  is-doll?  toys)))
```

So what is the contract really? It wouldn't work with a function from string to boolean and a list of numbers, or *vice versa*; the input type of the function has to be the same type as the elements of the list. We often write this using a capital letter like X to represent "any type":

```
; any-satisfies?  (X -> boolean) list-of-X -> boolean
```

SIDEBAR:

The Java and C++ languages also allow you to write a function whose contract has a "type variable" in it, like the X above. C++ calls these "templates", and Java calls them "generics". Needless to say, the syntax is more complicated.

**Exercise 28.2.3** *Use `any-satisfies?` to write a `any-over-100K?` function that takes in a list of `employee` structs (as in Exercise 21.2.1), and tells whether any of them earn over $100,000/year. Your `any-over-100K?` function should have no recursive calls.*

**Exercise 28.2.4** ***Develop*** *a function `count-if` that takes in a function (from X to boolean) and a list of X's, and returns* how many *of the elements of the list have the property.*

**Exercise 28.2.5** *Use `count-if` to write a `count-evens` function that takes a list of numbers and returns how many of them are even. Your `count-evens` function should have no recursive calls.*

**Exercise 28.2.6** *Use `count-if` to write a `count-dolls` function that takes a list of strings and returns how many of them are `"doll"`. Your function should have no recursive calls.*

**Exercise 28.2.7** *Use `count-if` to write a `count-multiples` function that takes a number and a list of numbers and returns how many of them are multiples of that number. Your function should have no recursive calls.*

**Exercise 28.2.8** *Use `count-if` to write a `count-earns-over-100K` function that takes a list of `employee` structs and returns how many of them earn over $100,000/year. Your function should have no recursive calls.*

**Exercise 28.2.9** *Use* `count-if` *to write a* `count-earns-over` *function that takes a list of* `employee` *structs and a number, and returns how many of the employees earn over that amount. Your function should have no recursive calls.*

**Exercise 28.2.10** ***Develop*** *a function* `remove-if` *that takes in a function (from X to boolean) and a list of X's, and returns a list of the elements of the list for which the function returns false (*i.e. *it "removes" the ones for which the function returns true).*

(A function similar to `remove-if` is actually built into DrRacket: now that you've written your own version, look up the `filter` function in the Help system.)

**Exercise 28.2.11** *Use* `remove-if` *to write a* `remove-over-5` *function that takes in a list of numbers and removes all the ones* $> 5$*. Your function should have no recursive calls.*

**Exercise 28.2.12** *Use* `remove-if` *to write a* `remove-over` *function that takes in a number and a list of numbers and removes all the ones over that number. Your function should have no recursive calls.*

**Exercise 28.2.13** *Use* `remove-if` *to write a* `fire-over` *function that takes in a number and a list of* `employee` *structs and removes all the ones that earn over that amount of money per year. Your function should have no recursive calls.*

**Exercise 28.2.14** ***Develop a function*** `nth-satisfying` *that takes in a whole number n and a Boolean-valued function, and returns the n-th whole number that has the desired property. For example,*

```
(check-expect (nth-satisfying 3 even?)  4)
  ; even natural numbers 0, 2, 4
(check-expect (nth-satisfying 5 prime?)  11)
  ; prime numbers 2, 3, 5, 7, 11
(check-expect (nth-satisfying 5 over-10?)  15)
  ; 11, 12, 13, 14, 15
(check-expect (nth-satisfying 4 integer?)  3)
  ; 0, 1, 2, 3
```

**Hint:** You'll probably need a helper function that takes in an extra parameter.

## 28.3 Functions returning lists

Consider the function `cube-each`:

```
; cube-each :  list-of-numbers -> list-of-numbers
; returns a list of the cubes of the numbers, in the same order
(check-expect (cube-each empty) empty)
(check-expect (cube-each (list 2 6 -3)) (list 8 216 -27))
```

Defining this function should be straightforward by now. But what if you were then asked to write `sqrt-each` or `negate-each`? Obviously, these are all defined in essentially the same way, differing only in what function is applied to each element of the list. To avoid having to write each of these separately, we'll *parameterize* them with that function:

```
; do-to-each :  (X -> X) list-of-X -> list-of-X
(check-expect (do-to-each sqrt (list 0 1 4)) (list 0 1 2))
(define (cube y)
  (* y y y))
(check-expect (do-to-each cube (list 2 6 -3)) (list 8 216 -27))
```

**Exercise 28.3.1** ***Develop*** *this* `do-to-each` *function.*

**Exercise 28.3.2** *Use* `do-to-each` *to write* `sqrt-each` *with no recursive calls.*

**Exercise 28.3.3** *There's a built-in function named* `identity` *which does nothing: it returns its argument unchanged. It's often useful as a simple test case for functions like* `do-to-each`*.*

*What should* `(do-to-each identity ...)` *do? Try it.*

**Exercise 28.3.4** *Use* `do-to-each` *to write* `add-3-to-each` *with no recursive calls.*

**Exercise 28.3.5** *Use* `do-to-each` *to write a* `add-to-each` *function that takes in a number and a list of numbers and adds the number to each element of the list. No recursive calls.*

**Exercise 28.3.6** *Use* `do-to-each` *to write a* `give-10%-raises` *function that takes in a list of* `employee` *structs and returns a list of the same employees, in the same order, but with each one earning 10% more than before. Your function should have no recursive calls.*

Now that you've written `do-to-each`, notice that not only is there nothing in the code that requires the elements of the list to be numbers; there is *also* nothing in the code that requires the input list to be the same type as the output type. For example,

```
(check-expect
  (do-to-each string-length (list "hello" "hi" "mazeltov"))
  (list 5 2 8))
```

works, even though it violates the contract as we stated it above. In fact, the contract should really be rewritten:

```
; do-to-each :  (X -> Y) list-of-X -> list-of-Y
```

which makes sense: if X and Y are *any* two types (possibly the same), it takes in a function from X to Y, applies it to each of a list of X's, and produces a list of Y's.

**Exercise 28.3.7** *Use* `do-to-each` *to write a* `names` *function that takes a list of* `employee` *structs and returns a list of their names. Your function should have no recursive calls.*

Actually, there's a function similar to `do-to-each` built into DrRacket; now that you've written your own version, look up the `map` function in the Help system. (Just as `map-image` does something to each pixel of an image and produces an image the same size, `map` does something to each element of a list and produces a list the same size.) The biggest difference between `do-to-each` and `map` is that `do-to-each` always applies a one-parameter function, whereas `map` takes in a function of *any number* of parameters, and takes that number of lists. For example,

```
(check-expect (map + (list 1 2 3 4) (list 50 40 30 20))
              (list 51 42 33 24))
```

**Exercise 28.3.8** ***Rewrite*** *exercise 23.4.1 using* `map` *or* `do-to-each` *wherever possible. Is the result significantly shorter or clearer?*

**Exercise 28.3.9** ***Develop a function*** `do-to-each-whole` *that takes in a whole number n and a function* `f: whole -> X` *and produces a list of X's:* `(f 0)`, `(f 1)`, ... `(f (- n 1))`. *For example,*

```
(check-expect (do-to-each-whole 5 sqr) (list 0 1 4 9 16))
```

Again, there's actually a built-in function that does exactly this:

```
; build-list :  whole (whole -> X) -> list-of-X
```

As `build-image` builds an image of a specified size and shape by calling a function on the coordinates of each pixel, `build-list` builds a list of length $N$ by calling a function on each of the numbers $0, 1, \ldots N-1$. Now that you've written `do-to-each-whole`, feel free to use the predefined `build-list` instead.

**Exercise 28.3.10** *The* `sort` *function from Section 23.6 sorts a list of numbers in increasing order. One could easily sort a list of numbers in* decreasing *order by replacing a* $<$ *with a* $>$ *in the function definition.* ***Generalize*** *the* `sort` *function so it takes in a function (call it* `precedes?`*) to tell whether one item in the list should come before another.*

*Your function definition should no longer depend on the data in the list being* numbers*;* `generalize the contract` *as far as you can.*

**Exercise 28.3.11** ***Develop a function*** `sort-by-salary` *that takes in a list of* `employee` *structures and sorts them from highest-paid to lowest-paid. You should be able to do this in a few lines of code (not counting contracts, inventories, and test cases), without recursion, by using* `general-sort`*.*

**Exercise 28.3.12** ***Rewrite*** `sort-candidates` *(from exercise 23.6.2) with the help of* `general-sort`*. It should take about four reasonably short lines, not counting contracts, inventories, and test cases.*

**Exercise 28.3.13** ***Develop a function*** `ranked-election` *that takes in a list of strings (representing the votes cast by individual voters) and returns an ordered list of candidates, from most votes to fewest votes. It should take two or three reasonably short lines, not counting contracts, inventories, and test cases.*

## 28.4 Choosing a winner

The `smallest` and `largest` functions, finding the smallest and largest number (respectively) in a non-empty list, are of course very similar: where one has a $<$, the other has a $>$. The `highest-paid` function, taking in a list of `employee` structs and returning the one with the highest salary, is a little more complicated, but still does basically the same thing: it compares two objects, picks one of them as the "winner", then compares the winner of this bout with another object, and so on until you've reduced the whole list to one "champion".

Obviously, the difference between one of these functions and another is the comparison function, which we might call `beats?`. We can generalize all of these functions to a `champion` function that takes in the `beats?` function and a non-empty list (it doesn't make sense on an empty list), runs a single-elimination tournament, and returns the list element that "beats" all the rest.

**Exercise 28.4.1** ***Develop*** *the function* `champion` *as described above.*

**Hint:** For efficiency, it's probably a good idea to adapt the technique from Section 27.1 so you don't call the function recursively twice.

**Exercise 28.4.2** *Use* `champion` *to rewrite* `smallest` *with no recursive calls.*

**Exercise 28.4.3** *Use* `champion` *to write* `highest-paid` *with no recursive calls.*

## 28.5 Accumulating over a list

Consider the functions `add-up` and `multiply-all` (Exercises 22.5.2 and 22.5.11). These functions are extremely similar: they both *combine* two objects to get a third object, which is then combined with another object to get a fourth, and so on. We should be able to generalize these. How do they differ, and how are they the same?

The `add-up` and `multiply-all` functions obviously differ in what function they apply (+ and ∗ respectively). But they also differ in the answer to the base case: the right answer to `(add-up empty)` is 0, while the right answer to `(multiply-all empty)` is 1. So we'll need to add *two* parameters:

```
; combine :  X (X X->X) list-of-X -> X
...
(define (add-up nums) (combine 0 + nums))
(define (multiply-all nums) (combine 1 * nums))
(check-expect (add-up (list 1 2 3 4)) 10)
(check-expect (multiply-all (list 1 2 3 4)) 24)
```

**Exercise 28.5.1** ***Develop*** *this* `combine` *function.*

You may notice that there's nothing in the function definition that requires the various types to be the same. On the other hand, not every possible combination of data types would make sense either.

**Exercise 28.5.2** *Correct the contract for* `combine` *to reflect which things* must *be the same type. Allow as much generality as you can.*

**Exercise 28.5.3** *Use* `combine` *to rewrite* `any-satisfies?` *with no recursive calls.*

**Exercise 28.5.4** *Use* `combine` *to rewrite* `count-if` *with no recursive calls.*

**Exercise 28.5.5** *Use* `combine` *to rewrite* `do-to-each` *with no recursive calls.*

**Exercise 28.5.6** *Use* `combine` *to rewrite* `champion` *with no recursive calls.*

There are two different functions built into DrRacket that act like `combine`. Now that you've written your own version, look up `foldr` and `foldl` in the Help system.

## 28.6 Anonymous functions

In many of the above exercises, we needed to write a little function (either `local`ly or standing on its own) for the sole purpose of passing it to a higher-order function like `any-satisfies?`, `count-if`, `remove-if`, `champion`, or `combine`. This feels silly and wasteful to me.

By way of analogy, suppose we wanted to compute $3+(4\cdot 5)$. We *could* do this in two steps:

```
(define temp (* 4 5))
(+ 3 temp)
```

but it's much shorter and simpler to just say

```
(+ 3 (* 4 5))
```

In other words, we don't need to give a *name* to the result of `(* 4 5)` if all we're going to do it pass it as an argument, once, to +.

If we're going to use something many times, it makes sense to give it a name and use that name each time. But if we're going to use it only once, it makes more sense to just use its value directly. Similarly, it seems silly to define a function with a name if we're only going to use it once, as an argument to another function.

**Syntax Rule 9** `(lambda (`*parameter parameter ...*`)` *expression*`)`

*is an expression whose value is a function that takes in the specified number of parameters. The* parameter *names may appear in the* expression*.*

`lambda` can be thought of as just like `define`, except that it doesn't bother giving a *name* to the function, but just returns it instead. Any place that a function can appear – a "function" argument to a higher-order function, or even just after a left parenthesis — a `lambda` expression can appear too. For example,

```
((lambda (y) (* y y y)) 1234567890)
```

is equivalent to

```
(* 1234567890 1234567890 1234567890)
```

More usefully,

```
(do-to-each (lambda (y) (* y y y)) (list 1 2 5 -3))
```

is equivalent to

```
(local [(define (cube y) (* y y y))]
    (do-to-each cube (list 1 2 5 -3)))
```

which of course returns `(list 1 8 125 -27)`.

**Worked Exercise 28.6.1** *Recall Exercise 28.2.2, in which we re-wrote* `any-compares?` *using* `any-satisfies?`. ***Re-do*** *this exercise using* `lambda` *in place of* `local`*.*

**Solution:** Instead of defining the `ok?` function `local`ly, we'll define it without a name using `lambda`:

```
(define (any-compares?  compare?  num nums)
  (any-satisfies?  (lambda (x) (compare?  x num))
                   nums))
```

∎

**Exercise 28.6.2** *For each function you've defined in this chapter using* `local` *to create a function to pass to a higher-order function,* ***rewrite*** *it using* `lambda` *instead. Is the result longer or shorter than the* `local` *version?*

A list of lists can be thought of as a two-dimensional table: the $i, j$ element of the table is simply the $j$'th element of the $i$'th list.

**Exercise 28.6.3** *Define a function* `multiplication-table` *that takes in a list of numbers and produces a two-dimensional table of multiplication results on those numbers: for example,*

```
(check-expect (multiplication-table (list 1 2 3 5))
              (list (list 1 2 3 5)
                    (list 2 4 6 10)
                    (list 3 6 9 15)
                    (list 5 10 15 25))
```

**Hint:** You can write this in two or three lines, with no `local`s or recursive calls, by using `lambda` and functions that you've already written.

**Exercise 28.6.4** *Explain how* any *expression using* `local` *definitions can be rewritten to use* `lambda` *instead.*

**Hint:** In a sense, this exercise asks you to write a function that takes in Racket expressions and produces Racket expressions. We haven't discussed how to do that in Racket yet, however, so for this exercise you may simply describe, in English, what you would do to an expression containing `local` to convert it into an equivalent expression containing `lambda`.

**Exercise 28.6.5** *Explain how* any *expression using* `lambda` *can be rewritten to use* `local` *instead.*

**Hint:** See previous problem.

SIDEBAR:

Some programming languages, like C++, have nothing corresponding to `lambda`: there's no way to define a function without giving it a name. In Java, it can be done using something called an "anonymous inner class". Needless to say, the syntax is more complicated.

## 28.7 Functions in variables

If *function* is a data type, along with *number*, *string*, and so on, we should be able to store functions in variables. For example, following Syntax Rule 4, we could write

```
(define cube (lambda (y) (* y y y)))
```

Then, since the variable `cube`'s value is a function, we should be able to use it anywhere that we could use a function, *e.g.*

```
(check-expect (cube 3) 27)
(check-expect (do-to-each cube (list 1 3 5)) (list 1 27 125))
```

In other words, such a variable would act exactly as though we had used Syntax Rule 5 to define a function by that name. In fact, deep down inside DrRacket,

```
(define (cube y) (* y y y))
```

is *just an abbreviation*[1] for

```
(define cube (lambda (y) (* y y y)))
```

Recall Section 27.3, in which we defined a `smallest` function

```
(define (smallest nums)
  (local [(define (smaller a b) (cond [(<= a b) a] [else b]))]
     (cond [(empty?  (rest nums)) (first nums)]
           [(cons?  (rest nums))
            (smaller (first nums) (smallest (rest nums)))])))
```

One might object to this definition that it re-defines the `smaller` function each time `smallest` is called, a small but annoying inefficiency. But now that we know that defining a function is just an abbreviation for defining a variable whose value is a `lambda` expression, we can rewrite this:

```
(define  smallest
  (local [(define (smaller a b) (cond [(<= a b) a] [else b]))]
     (lambda (nums)
         (cond [(empty?  (rest nums)) (first nums)]
               [(cons?  (rest nums))
                (smaller (first nums)
                         (smallest (rest nums)))]))))
```

Rather than defining `smaller` inside the `smallest` function, we've defined it once, locally, for just long enough to build an anonymous function, then (outside the `local`) give it the name `smallest`.

This isn't an essential, earth-shaking change to the function definition. Remember this technique, though, because it'll make a big difference in Chapter 30.

**Exercise 28.7.1** *Rewrite the `sort` function from Section 27.3 in this style, with the `local` outside the function body.*

## 28.8 Functions returning functions

Just as a function can *take in* functions as parameters, a function can also *return* a function as its result. For a simple (and unrealistic) example, suppose we needed functions `add-2-to-each` and `add-3-to-each`, which could be defined easily by

```
(define (add-2-to-each nums)
  (do-to-each (lambda (x) (+ x 2)) nums))
(define (add-3-to-each nums)
  (do-to-each (lambda (x) (+ x 3)) nums))
```

[1] A well-known programming textbook using Scheme (Racket's immediate ancestor), Abelson & Sussman's *Structure and Interpretation of Computer Programs* [ASS96], uses the latter notation from the beginning.

We've written two extremely similar `lambda`-expressions, which we could generalize into a function that takes in the 2 or the 3 and returns the function that will be passed to `do-to-each`.

**Worked Exercise 28.8.1** *Define a function* `make-adder` *that takes in a number, and returns a function that adds that number to its one argument.*

**Solution:** The contract looks like

```
; make-adder :  number -> (number -> number)
```

How would we test this? It turns out that `check-expect` doesn't work very well on functions — how do you test whether two functions are the same, short of calling them both on *every possible input*? So we'll have to *apply* the result of `make-adder` to a number and see what it produces.

```
(define add2 (make-adder 2))
(define add5 (make-adder 5))
(check-expect (add2 4) 6)
(check-expect (add2 27) 29)
(check-expect (add5 4) 9)
(check-expect (add5 27) 32)
(check-expect ((make-adder 3) 2) 5)
```

Defining the function, however, is easy:

```
(define (make-adder num)
  ; num      number
  (lambda (x) (+ x num)))
```

or, if you prefer `local`,

```
(define (make-adder num)
  ; num      number
  (local [(define (addnum x) (+ x num))]
    addnum))
```

**Exercise 28.8.2** *Define a function* `make-range` *that takes in two numbers and returns a function that takes in a number and tells whether it's between those two numbers (inclusive,* i.e. *it can equal either of them). For example,*

```
(define teen?  (make-range 13 19))
(define two-digit?  (make-range 10 99))
(check-expect (teen?  12) false)
(check-expect (teen?  13) true)
(check-expect (teen?  16) true)
(check-expect (teen?  19) true)
(check-expect (teen?  22) false)
(check-expect (two-digit?  8) false)
(check-expect (two-digit?  10) true)
(check-expect (two-digit?  73) true)
(check-expect (two-digit?  99) true)
(check-expect (two-digit?  100) false)
(check-expect ((make-range 39 45) 38) false)
(check-expect ((make-range 39 45) 42) true)
```

**Exercise 28.8.3** *Define a function* `twice` *that takes in a function with contract* $X \to X$*, and returns another function with the same contract formed by calling the given function on its own result. For example,*

```
(define add2 (twice add1))
(define fourth-root (twice sqrt))
(check-expect (add2 5) 7)
(check-expect (fourth-root 256) 4)
```

*If you did Exercise 9.2.8, recall the definition of the function* `digits`*, which tells how many digits long the decimal representation of an integer is. What does* `(twice digits)` *do?*

*What does* `(twice twice)` *do?*

**Exercise 28.8.4** *Define a function* `iterate` *that takes in a whole number and a function with contract* $X \to X$*, and returns a function that applies the specified function the specified number of times. For example,*

```
(define add5 (iterate 5 add1))
(check-expect (add5 17) 22)
(define eighth-root (iterate 3 sqrt))
(check-expect (eighth-root 256) 2)
```

*Note that the* `twice` *function above is a special case of* `iterate`*:*

```
(define (twice f) (iterate 2 f))
```

*What does* `(iterate 3 twice)` *do?*

In solving problems 28.8.3 and 28.8.4, you needed to define a new function as the *composition* of two existing functions (that is, one function applied to the result of another). This is such a common operation that Racket gives you a built-in function to do it:

```
; compose :  (Y -> Z) (X -> Y) -> (X -> Z)
```

For example,

```
(define f (compose sqr add1))
(check-expect (f 0) 1)
(check-expect (f 1) 4)
(check-expect (f 2) 9)
```

**Exercise 28.8.5** ***Rewrite*** *the* `twice` *function of Exercise 28.8.3 using* `compose`*.*

**Exercise 28.8.6** ***Rewrite*** *the* `iterate` *function of Exercise 28.8.4 using* `compose`*.*

SIDEBAR:

The Java language technically doesn't allow you to create, return, or pass functions as values, but you can create, return, or pass "objects" that have functions associated with them, which gives you the same power with more complicated syntax. However, although you can create functions while the program is running, you have to specify their *contracts* (which Java calls "interfaces") in advance, while writing the program; you can't decide at run-time what contract the new function should have.

## 28.9 Sequences and series

Mathematicians often work with *sequences* of numbers. A *sequence* can be defined as a function from whole numbers to numbers. For example, the sequence of even numbers can be written $0, 2, 4, 6, 8, \ldots$ or, thinking of it as a function, as `(lambda (n) (* 2 n))`. Thus one could write

```
(define evens (lambda (n) (* 2 n)))
```

**Exercise 28.9.1** ***Develop a function*** `take` *which, given a whole number n and a sequence, produces a list of the first n values of that sequence. For example,*

```
(check-expect (take 5 evens) (list 0 2 4 6 8))
```

*This function will make it much easier to write test cases for functions that return sequences!*

**Exercise 28.9.2** ***Develop a function*** `arithmetic-sequence` *which, given two numbers* `initial` *and* `difference`*, produces the "arithmetic sequence" starting at* `initial` *and increasing by* `difference` *at each step. For example,*

```
(define evens (arithmetic-sequence 0 2))
(define odds (arithmetic-sequence 1 2))
(define ends-in-3 (arithmetic-sequence 3 10))
(check-expect (take 5 evens) (list 0 2 4 6 8))
(check-expect (take 6 odds) (list 1 3 5 7 9 11))
(check-expect (take 5 ends-in-3) (list 3 13 23 33 43))
```

**Exercise 28.9.3** ***Develop a function*** `geometric-sequence` *which, given two numbers* `initial` *and* `ratio`*, produces the "geometric sequence" starting at* `initial` *and growing by a factor of* `ratio` *at each step.*

**Exercise 28.9.4** ***Develop a function*** `constant-sequence` *that takes in a number and produces a sequence that always has that value.*

*(You can write this from scratch with* `lambda`*, or by re-using a previously-written function.)*

**Exercise 28.9.5** ***Define a variable*** *whose value is the "harmonic sequence": 1, 1/2, 1/3, 1/4, 1/5, ....*

**Exercise 28.9.6** ***Define a variable*** `wholes` *whose value is the sequence of whole numbers.*

**Hint:** You can do this in three words, with no `local` and no `lambda`.

**Exercise 28.9.7** ***Develop a function*** `scale-sequence` *which, given a number and a sequence, returns a sequence whose elements are that number times the corresponding element of the original sequence.*

**Exercise 28.9.8** ***Develop a function*** `add-sequences` *which, given two sequences, returns a sequence whose n-th element is the n-th element of one sequence plus the n-th element of the other.*

**Exercise 28.9.9** ***Develop a function*** `subtract-sequences` *which, given two sequences, returns a sequence whose n-th element is the n-th element of the first sequence minus the n-th element of the second.*

**Exercise 28.9.10** ***Develop a function*** `mult-sequences` *which, given two sequences, returns a sequence whose n-th element is the product of the n-th elements of the two sequences.*

**Exercise 28.9.11** ***Develop a function*** `div-sequences` *which, given two sequences, returns a sequence whose n-th element is the n-th element of the first sequence divided by the n-th element of the second.*

**Note:** If the second sequence is ever 0, the resulting "sequence" won't be defined on all whole numbers.

**Exercise 28.9.12** ***Develop a function*** `shift-sequence` *which, given an integer number d and a sequence, returns a sequence whose n-th element is the $n + d$-th element of the given sequence. For example,*

```
(define positive-evens (shift-sequence 1 evens))
(check-expect (take 5 positive-evens) (list 2 4 6 8 10))
```

**Note:** If the integer is negative, the result may not technically qualify as a "sequence", because it may not be defined on all whole numbers.

**Exercise 28.9.13** ***Develop a function*** `patch` *which, given two numbers n and x and a sequence, produces a sequence exactly like the given sequence except that it returns x on input n. (In other words, you've "patched" the sequence by changing its value at one particular input.)*

**Exercise 28.9.14** ***Develop a function*** `differences` *which, given a sequence, returns its sequence of differences: element 1 minus element 0, element 2 minus element 1, element 3 minus element 2, ...*

*(You can write this from scratch, but try writing it by re-using some of the above functions instead.)*

**Exercise 28.9.15** ***Develop a function*** `partial-sum` *which, given a sequence, returns its sequence of partial sums. The n-th element of the sequence of partial sums is the sum of elements 0 through* $n-1$ *of the original sequence. For example,*

```
(define sum-of-wholes (partial-sum wholes))
(check-expect (take 5 sum-of-wholes) (list 0 1 3 6 10))
(define sum-of-odds (partial-sum odds))
```

*What does* `sum-of-odds` *do? Why?*

*Define a variable to hold the sequence of partial sums of* `(geometric-sequence 1 1/2)`*, and play with it. What can you say about its value?*

**Exercise 28.9.16** ***Define a variable*** `fact` *to hold the sequence* $1, 1, 2, 6, 24, 120, \ldots$ *of factorials.*

*(Obviously, you can do this from scratch using recursion. But try doing it by using operations on sequences, like the above.)*

Many important mathematical functions can't be computed exactly by a few additions or multiplications, but are instead *approximated* by adding up the first entries of a particular sequence of numbers; the more entries you add up, the closer the approximation. These are called *Taylor series*. For example, $e^x$ can be computed by the series

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!)}$$

where $n!$ is the usual mathematical notation for `(factorial n)` from exercise 24.1.8. In other words,

$$e^x = \frac{x^0}{1} + \frac{x^1}{1} + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \ldots$$

**Exercise 28.9.17** ***Develop a function*** `e-to-the` *that takes in a number* $x$ *and returns the sequence of partial sums of this series.*

*Theoretically,* $e^x \cdot e^{-x}$ *should be exactly 1 for all values of* $x$*. So you can test an approximation's accuracy by multiplying these two and comparing with* 1*.* ***Define*** *a function* `exp-error` *that takes in the value of* $x$ *and the number of terms of the series to use, multiplies the approximations of* $e^x$ *and* $e^{-x}$*, and tells how far this is from* 1*.*

*How many steps does it take to get within 0.1? Within 0.01? Within 0.001? Within 0.000001?*

*The built-in Racket function* `exp` *computes* $e^x$*. Compare* `(e-to-the x)` *with* `(exp x)` *for various positive and negative values of* $x$*, looking in particular at how many steps it takes to get to various levels of accuracy.*

**Exercise 28.9.18** *The trigonometric function* $\sin(x)$ *has the Taylor series*

$$\sin(x) = \sum_{n=0}^{\infty} (-1)^n \cdot \frac{x^{2n+1}}{(2n+1)!}$$

*That is,*

$$\sin(x) = \frac{x}{1} - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040} + \dots$$

***Develop a function*** `taylor-sine` *that takes in a number* $x$ *and returns the sequence of partial sums of this series. (Again, I did this by using previously-defined operations on sequences.)*

*Compare* `(taylor-sine x)` *with* `(sin x)` *for various positive and negative values of* $x$*, looking at how many steps it takes to get to various levels of accuracy.*

**Exercise 28.9.19** *Did you ever wonder how people discovered that* $\pi$ *was about 3.14159? There are various series that compute* $\pi$*, of which the simplest and best-known is*

$$\pi = \sum_{n=0}^{\infty} (-1)^n \cdot \frac{4}{2n+1}$$

*In other words,*

$$\pi = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \dots$$

***Define the variable*** `my-pi` *to be the sequence of partial sums of this series. (You can do this from scratch, of course, but try doing it by using previously-defined sequences and operations on sequences.)*

*Compare* `my-pi` *with the built-in Racket variable* `pi`*. How many steps does it take to get to various levels of accuracy?*

## 28.10 Review of important words and concepts

Racket treats *function* as a data type, just like *number* or *string* or *list*. One can write a function that takes in, or returns, a function just as easily as one can write functions working on other data types (although it's a bit harder to write test cases). This technique allows a programmer to write, test, and debug a single *general* function that covers the functionality of many others, thus saving enormous amounts of programming time on re-inventing the wheel. Some other languages allow you to do this too — for example, some of this can be done in Java and C++ — but the syntax is usually more complicated and confusing.

Racket also allows a programmer to construct functions "on the fly", just in time to pass them as arguments to other functions, without bothering to name them. Again, some other languages allow this — including Java, but not C++ — but the syntax is more complicated and confusing.

## 28.11 Reference: higher-order and anonymous functions

This chapter introduced Syntax Rule 9, which constructs an anonymous function using **lambda** and returns it.

It also introduced several predefined (mostly) higher-order functions:

- `identity`
- `filter`
- `map`
- `build-list`
- `foldl`
- `foldr`