Chapter 27

Local definitions

For this chapter, switch languages in DrRacket to "Intermediate Student Language" or higher.

27.1 Using locals for efficiency

Suppose we wanted to write a function `smallest` to find the smallest of a list of numbers. A straightforward solution, based on what you've already seen, would be

```
; smallest:  non-empty-list-of-numbers -> number
(check-expect (smallest (list 4)) 4)
(check-expect (smallest (list 4 7)) 4)
(check-expect (smallest (list 7 4)) 4)
(check-expect (smallest (list 6 9 4 7 8 3 6 10 7)) 3)
(define (smallest nums)
  (cond [(empty?  (rest nums))     (first nums)]
        [(cons?  (rest nums))
         ; (first nums)            number
         ; (rest nums)             non-empty list of numbers
         ; (smallest (rest nums))  number
         (cond [(<= (first nums) (smallest (rest nums)))
                (first nums)]
               [else (smallest (rest nums))])]))
```

This definition works, and produces right answers, but consider the following two examples:

```
(check-expect
  (smallest (list 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18))
  1)
(check-expect
  (smallest (list 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1))
  1)
```

On my computer (as of 2009), the former takes about 3 milliseconds; the latter takes almost 9 seconds — 3000 times longer, even though both examples find the smallest of the same set of numbers! What's going on?

To figure this out, let's pick some simpler examples:

```
(check-expect (smallest (list 1 2 3 4)) 1)
(check-expect (smallest (list 4 3 2 1)) 1)
```

and use the Stepper to see what's happening. The former example calls

```
(smallest (list 1 2 3 4))
  (smallest (list 2 3 4))
    (smallest (list 3 4))
      (smallest (list 4))
      return 4
    return 3
  return 2
return 1
```

The latter example behaves differently:

```
(smallest (list 4 3 2 1))
  (smallest (list 3 2 1))
    (smallest (list 2 1))
      (smallest (list 1))
      return 1
      (smallest (list 1)) again!
      return 1
    (smallest (list 2 1)) again!
      (smallest (list 1)) a third time!
      return 1
      (smallest (list 1)) a fourth time!
      return 1
    return 1
  (smallest (list 3 2 1)) again!
    (smallest (list 2 1)) a third time!
      (smallest (list 1)) a fifth time!
      return 1
      (smallest (list 1)) a sixth time!
      return 1
    (smallest (list 2 1)) a fourth time!
      (smallest (list 1)) a seventh time!
      return 1
      (smallest (list 1)) an eighth time!
      return 1
    return 1
  return 1
return 1
```

In other words, the function is calling itself on the exact same question over and over, wasting a lot of time. Any time that `(first nums)` is larger than `(smallest (rest nums))`, it calls `(smallest (rest nums))` all over again.

How can we avoid this waste of time? One reasonable approach is to compute `(smallest (rest nums))`, *save* the result in a variable, then use that result twice without re-computing it. Unfortunately, the syntax rules we've seen so far don't allow us to define a variable inside a function definition.

There is a way to do it, however.

Syntax Rule 8

```
(local [definition definition ...]
  expression)
```

is an expression. Each definition *can be a variable definition (rule 4), a function definition (rule 5), or a struct definition (rule 7).*

The effect is to apply all the definitions *temporarily*, evaluate the inner expression, and then *forget* all the new definitions; the result is the value of the inner expression.

Here's a simple, unrealistic example:

```
(local [(define y 5)]
  (* y y)) ; returns 25
y ; produces an error message, because y is undefined
```

In fact, a "local" definition can *temporarily hide* a definition that's already in effect:

```
(define y 17)
y ; returns 17
(local [(define y 5)] (* y y)) ; returns 25
y ; returns 17 again
```

More realistically, the main reason people use `local` is to define a variable inside a function definition:

```
; smallest:  non-empty-list-of-numbers -> number
(check-expect (smallest (list 4)) 4)
(check-expect (smallest (list 4 7)) 4)
(check-expect (smallest (list 7 4)) 4)
(check-expect (smallest (list 6 9 4 7 8 3 6 10 7)) 3)
(define (smallest nums)
  (cond [(empty?  (rest nums))          (first nums)]
        [(cons?  (rest nums))
         ; (first nums)                 number
         ; (rest nums)                  non-empty list of numbers
         ; (smallest (rest nums))       number
         (local [(define winner (smallest (rest nums)))]
            (cond [(<= (first nums)  winner) (first nums)]
                  [else  winner]))]))
```

Now *both* of the examples

```
(check-expect
  (smallest (list 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18))
  1)
(check-expect
  (smallest (list 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1))
  1)
```

run in about 3 milliseconds on my computer; we've sped up the latter by a factor of 3000.

Exercise 27.1.1 ***Re-write*** *the* `spams` *function from Exercise 24.4.5 using* `local` *to call itself only once. Does it work correctly? Is it significantly faster?*

Exercise 27.1.2 ***Re-write*** *the* `copies` *function from Exercise 24.4.2 or 24.4.6 by using* `local` *to call itself only once. Does it work properly? Is it significantly faster?*

Exercise 27.1.3 ***Re-write*** *the* `dot-grid` *function from Exercise 24.4.8 using* `local` *to call itself only once. Does it work properly? Is it significantly faster?*

Exercise 27.1.4 ***Re-write*** *the* `randoms` *function from Exercise 24.4.9 using* `local` *to call itself only once. Does it work correctly? Is it significantly faster?*

Exercise 27.1.5 *Consider the functions* `binary-add1` *from Exercise 24.4.3,* `binary-add` *from Exercise 25.4.9,* `binary-mult` *from Exercise 25.4.10, and* `binary-raise` *from Exercise 25.4.11. Which of these (if any) would benefit from this treatment? Why?*

27.2 Using locals for clarity

Consider a `distance` function that takes in two `posn`s and computes the distance between them:

```
; distance :  posn posn -> number
(check-expect (distance (make-posn 3 5) (make-posn 3 5)) 0)
(check-expect (distance (make-posn 3 5) (make-posn 6 5)) 3)
(check-expect (distance (make-posn 3 5) (make-posn 3 -10)) 15)
(check-expect (distance (make-posn 3 5) (make-posn 6 9)) 5)
(check-within (distance (make-posn 3 5) (make-posn 4 4)) 1.41 .1)
(define (distance here there)
  (sqrt (+ (* (- (posn-x here) (posn-x there))
              (- (posn-x here) (posn-x there)))
           (* (- (posn-y here) (posn-y there))
              (- (posn-y here) (posn-y there))))))
```

This passes all its tests, and it's reasonably efficient, but the definition is long, complicated, and hard to read. The formula computes the difference of x coordinates, squares that, computes the difference of y coordinates, squares that, adds the squares, and square-roots the result.

It would arguably be easier to read if we had *names* for "the difference of x coordinates" and "the difference of y coordinates". We can do that with `local`:

```
(define (distance here there)
  (local [(define xdiff (- (posn-x here) (posn-x there)))
          (define ydiff (- (posn-y here) (posn-y there)))]
    (sqrt (+ (*  xdiff xdiff) (*  ydiff ydiff)))))
```

The expression on the last line is much shorter and clearer: one can easily see that it's the square root of the sum of two squares. (It may also be slightly more efficient, but not the dramatic improvement we saw for `smallest` above.)

Exercise 27.2.1 ***Develop a function*** `rotate-colors` *that takes in an image and (using* `map-image`*) creates a new image whose red component is the old green component, whose green component is the old blue component, and whose blue component is the old red component. Use* `local` *to give names to the old red, green, and blue components.*

Exercise 27.2.2 *What other functions have you written that would benefit from this technique? Try rewriting them and see whether they're shorter and clearer.*

27.3 Using locals for information-hiding

Another approach to making `smallest` more efficient would have been to write a helper function `smaller`:

```
; smaller :  number number -> number
; returns the smaller of two numbers
(check-expect (smaller 3 8) 3)
(check-expect (smaller 9 7) 7)
(check-expect (smaller 2 2) 2)

(define (smaller a b)
  (cond [(<= a b) a]
        [else b]))

(define (smallest nums)
  (cond [(empty?  (rest nums)) (first nums)]
        [(cons?  (rest nums))
         ( smaller (first nums) (smallest (rest nums)))]))
```

This definition, too, calls itself recursively only once, so it doesn't have the efficiency problems of the first version above. But it requires a helper function, which may be of no use in the rest of the program.

As mentioned above, you can also define a *function*, or even a *struct*, locally. So if we wanted, we could *hide* the definition of `smaller` inside that of `smallest`:

```
(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)))])))
```

I recommend moving `smaller` into a local definition only *after* you've tested and debugged it as usual.

Suppose you were hired to write a `sort` function like that of Exercise 23.6.1. It needed a helper function `insert` which inserts a number in order into an already-sorted list of numbers. But "insert" is a fairly common word, and your customer might want to write a function by the same name herself, which would be a problem because DrRacket won't allow you to define two functions with the same name. So again, one could hide the `insert` function inside the definition of `sort`:

```
(define (sort nums)
  (local [(define (insert num nums) ...)]
     (cond [(empty?  nums) empty]
           [(cons?  nums)
            (insert (first nums) (sort (rest nums)))])))
```

For another example, the `wn-prime?` function of Exercise 24.3.8 needed a helper function `not-divisible-up-to?`, which nobody would ever want to use unless they were writing a prime-testing function. So after you've tested and debugged both functions, you can *move* the definition of `not-divisible-up-to?` function inside the definition of `wn-prime?`:

```
(define (wn-prime?  num)
  (local [(define (not-divisible-up-to?  m n) ...)]
    (not-divisible-up-to?  num (- num 1))))
```

For one more example, recall exercise 11.5.1, a `road-trip-cost` function which depended on six other functions: `gas-cost`, `cost-of-gallons`, `gas-needed`, `motel-cost`, `nights-in-motel`, and `rental-cost`. Any of those other functions could conceivably be useful in its own right, but suppose we knew that they *wouldn't* be used on their own. It would still be useful to write and test the functions individually, but once they all work, they (and the constants) could be *hidden* inside the definition of `road-trip-cost`:

```
(define (road-trip-cost miles days)
  (local [ (define MPG 28)
           (define PRICE-PER-GALLON 2.459)
           (define MOTEL-PRICE-PER-NIGHT 40)
           (define CAR-RENTAL-FIXED-FEE 10)
           (define CAR-RENTAL-PER-DAY 29.95)
           (define CAR-RENTAL-PER-MILE 0.10)
           (define (gas-needed miles)
                   (/ miles MPG))
           (define (cost-of-gallons gallons)
                   (* PRICE-PER-GALLON gallons))
           (define (gas-cost miles)
                   (cost-of-gallons (gas-needed miles)))
           (define (nights-in-motel days)
                   (- days 1))
           (define (motel-cost days)
                   (* MOTEL-PRICE-PER-NIGHT (nights-in-motel days)))
           (define (rental-cost miles days)
                   (+ CAR-RENTAL-FIXED-FEE
                      (* days CAR-RENTAL-PER-DAY)
                      (* miles CAR-RENTAL-PER-MILE))) ]
    (+ (gas-cost miles)
       (motel-cost days)
       (rental-cost miles days))))
```

Of course, since each of the helper functions is called only once, there's not much point in defining them as functions at all. For this problem, it would be simpler and more realistic to define them as *variables* instead:

```
(define (road-trip-cost miles days)
  (local [(define MPG 28)
          (define PRICE-PER-GALLON 2.459)
          (define MOTEL-PRICE-PER-NIGHT 40)
          (define CAR-RENTAL-FIXED-FEE 10)
          (define CAR-RENTAL-PER-DAY 29.95)
          (define CAR-RENTAL-PER-MILE 0.10)
          (define  gas-needed (/ miles MPG))
          (define  gas-cost (* PRICE-PER-GALLON gas-needed))
          (define  motel-cost (* MOTEL-PRICE-PER-NIGHT (- days 1)))
          (define  rental-cost
              (+ CAR-RENTAL-FIXED-FEE
                 (* days CAR-RENTAL-PER-DAY)
                 (* miles CAR-RENTAL-PER-MILE)))]
    (+  gas-cost motel-cost rental-cost)))
```

We're now using `local` partly for information-hiding (it's a convenient place to put the constants `PRICE-PER-GALLON`, `MPG`, *etc.* without the rest of the program seeing those names) and partly for clarity (`gas-needed`, `gas-cost`, *etc.* are just intermediate steps in calculating the answer).

27.4 Using locals to insert parameters into functions

In all of the above examples, we've written helper functions as usual, tested and debugged them, then moved them into local definitions inside the main function. In this section, we'll see problems for which the helper function *must* be defined locally inside another function — it doesn't work by itself.

Worked Exercise 27.4.1 ***Modify*** *the solution to Exercise 7.8.12 so the amount of blue increases smoothly from top to bottom, regardless of the height of the image.*

Solution: The `apply-blue-gradient` function will use the `map-image` function, which requires a function with the contract

```
; new-pixel :  number(x) number(y) color -> color
```

Let's write some examples of this function. We'll want one at the top of the image, one at the bottom, and one in between. The one at the top is easy:

```
(check-expect (new-pixel 40 0 (make-color 30 60 90))
              (make-color 30 60 0))
```

But how can we write an example at the bottom or in between when we don't know how tall the given image is?

Let's pretend for a moment the image was 100 pixels tall. Then we would choose examples

```
(check-expect (new-pixel 36 100 (make-color 30 60 90))
              (make-color 30 60 255))
(check-expect (new-pixel 58 40 (make-color 30 60 90))
              (make-color 30 60 102))
```

because 40 is 40% of the way from top to bottom, and 102 is 40% of the way from 0 to 255. The function would then look like

```
(define (new-pixel x y old-color)
  ; x         a number
  ; y         a number
  ; old-color a color
  (make-color (color-red old-color)
              (color-green old-color)
              (real->int (* 255 (/ y 100)))))

(define (apply-blue-gradient pic)
  ; pic       an image
  (map-image new-pixel pic))
```

This works beautifully for images that happen to be 100 pixels high. To make it work in general, we'd like to replace the 100 in the definition of `new-pixel` with `(image-height pic)`, but this doesn't work because `new-pixel` has never heard of `pic`: `pic` won't even be defined until somebody calls `apply-blue-gradient`. As a step along the way, let's define a variable `pic` directly in the Definitions pane:

```
(define pic (ellipse 78 100 "solid" "green"))
(check-expect (new-pixel ...)  ...)
(define (new-pixel x y old-color)
  (make-color (color-red old-color)
              (color-green old-color)
              (real->int (* 255 (/ y  (image-height pic))))))

(map-image new-pixel pic)
```

Now we can run the `check-expect` test cases for `new-pixel`, and also look at the result of the `map-image` to see whether it looks the way it should. But it still doesn't work in general.

So we'll get rid of the `pic` variable, comment out the `check-expect` test cases, and *move the definition* of `new-pixel` *inside* the `apply-blue-gradient` function:

```
(define (apply-blue-gradient pic)
  ; pic      an image
  (local [(define (new-pixel x y old-color)
            (make-color (color-red old-color)
                        (color-green old-color)
                        (real->int (* 255 (/ y (image-height pic))))))]
    (map-image new-pixel pic)))
```

Note that when the variable name `pic` appears in `new-pixel`, it refers to the parameter of `apply-blue-gradient`.

Try this definition of `apply-blue-gradient` on a variety of pictures of various sizes. ∎

A disadvantage of writing a function inside another function is that you can't test an inner function directly, so I recommend the process above: define global variable(s) for the information from the outer function that the inner function needs, test the inner function in the presence of these variables, and once it passes all the tests, move the inner function inside a function with parameters with the same names as those variables (and now you can get rid of the global variables).

Exercise 27.4.2 ***Develop a function*** `add-red` *that takes in a number and an image, and adds that number to the red component of every pixel in the image. (Remember to keep the red component below 256.)*

Exercise 27.4.3 ***Develop a function*** `substitute-color` *that takes in two colors and an image, and replaces every pixel which is the first color with the second color.*

Exercise 27.4.4 ***Develop a function*** `horiz-stripes` *that takes in a width, a height, a stripe width, and two colors, and produces a rectangular image of the specified width and height with horizontal stripes of the specified width and colors.*

Exercise 27.4.5 ***Develop a function*** `smooth-image` *that takes in an image and replaces each color component of each pixel with the* average *value of that color component in the pixel and its four neighbors (up, down, left, and right).*

Hint: Use the `get-pixel-color` function to get the values of the neighboring pixels. Use another `local` to give names (*e.g.* `up`, `down`, `left`, and `right`) to these values, for clarity.

You'll need to decide what to do at the borders. The easiest answer is to just call `get-pixel-color` even though the position may be outside the borders; it will return black in that case, so the resulting picture will have darkened edges. A more proper, and more difficult, solution is to average together only the neighboring pixels that actually exist. This will probably require two or three helper functions and a *list* of neighboring colors. These helper functions will also help you with Exercise 27.4.8.

A "higher-order function" is a function that takes in functions as parameters, and/or returns a function as its value. (We'll learn how to write such functions in Chapter 28.) Some examples are `map3-image`, `build3-image`, `map-image`, and `build-image`. We've seen how to write a function that (locally) defines another function from its parameters, then passes that new function as an argument to a higher-order function.

What other higher-order functions have we seen? How about `on-tick`, `on-draw`, `on-key`, `on-mouse`, *etc.*? The same technique now allows us to write a function that uses its parameters to construct event handlers and run an animation on them.

Exercise 27.4.6 *Recall Exercise 6.5.2, which placed a stick-figure at a fixed location on a background scene and had the stick figure turn upside-down every second or so.*

Develop a function `flipping-figure` *that takes in a background scene and the x and y coordinates where you want the figure to be, and runs an animation with the figure flipping upside-down every second or so at that location on that background.*

Hint: Define a redraw handler *locally*, using the specified background scene, and then call `big-bang` inside the body of the `local`.

Exercise 27.4.7 ***Develop a function*** `add-dots-with-mouse` *that takes in a color and a number, and runs an animation that starts with a white screen, and every time the mouse is clicked, adds a circular dot of the specified color and radius at the mouse location.*

Exercise 27.4.8 *Look up John Conway's "Game of Life" on the Web (*e.g. *Wikipedia).*

Develop a function `life-gen` *that takes in an image representing a grid of cells (think of the color white as "dead" and any other color as "alive"), and produces the grid of cells one generation later. If a pixel has fewer than two or more than three "live" neighbors (from among its eight neighbors — above, below, left, right, and the four diagonals), it dies. If it has exactly two, it stays the same as it was (alive or dead). If it has exactly three, it becomes alive (or stays alive, if it already was). You may want to use another* `local` *to give names to the eight neighboring pixels, or to a list of their colors, or something.*

Define an animation *with* `life-gen` *as its tick handler.*

Start it with a random image (see Exercise 15.3.3) as the initial model.

If you want more control over the animation, recall that `big-bang` *returns its final model. Try starting this "life" animation with the result of* `add-dots-with-mouse` *(see Exercise 27.4.7).*

SIDEBAR:

Most programming languages allow you to define local variables. Some (like Java and Pascal) allow you to define local structs and functions. Some of these allow you to write a function whose parameters are then inserted into locally-defined functions, although they require more complicated syntax and put extra restrictions on what you can do with these parameters. Racket makes this stuff easier than any other language I know of.

27.5 Review of important words and concepts

Racket (like most programming languages) allows you to define variables *locally*: you introduce a new variable, work with it for a little while, and then *forget* it. This is used for four common reasons:

- for efficiency: suppose a function calls itself more than once on the same argument(s). We call it only once, store the result in a local variable, and use the variable more than once.
- for clarity: suppose a particular long expression, an intermediate step in computing a function, appears several times in the function definition. Define a local variable to hold the result of that expression, and the resulting definition may be significantly shorter and easier to understand.
- for information-hiding: suppose a constant, struct, or function is only needed within a particular function, especially if it has a common name that somebody might want to use somewhere else in a large program. Define it locally, use it in this function as many times as you want, and be confident it won't conflict with or be confused with things by the same name defined elsewhere.
- for defining functions that can be passed to a higher-order function like `build-image`, `map-image`, `on-draw`, *etc.* In particular, the ability to define a function inside another function, using the parameters of the outer one, enables you to do image manipulations and animations that you couldn't do before. We'll see another way to do this in Section 28.6.

27.6 Reference: New syntax for local definitions

This chapter introduced one new syntax rule, Rule 8 introducing **local** definitions of variables, functions, and even structs.