Chapter 11

Reduce, re-use, recycle

11.1 Planning for modification and extension

Professional programmers learn very quickly that *program requirements change*. You may be given an assignment, be halfway through writing it, and suddenly be told that the program needs to do something different (usually something additional) from what you were told originally. Even after you've turned in an assignment, or released a piece of software to the public, the requirements can continue to change: either somebody will complain about the way a particular feature works, and ask you to change it, or somebody will think of a neat new feature that they want you to add for the next version. In some of the courses I teach, I warn students in advance "I reserve the right to change the assignment slightly on the day that it's due; you'll have a few hours to accommodate the change."

How can anybody work in such conditions? One thing you can do is, when you first get an assignment, start thinking about likely ways it *might* change. Then you can plan your program in such a way that, if it *does* change in those ways, you can handle the change quickly and easily.

To be more specific, try to design things so that *each likely change affects only one variable or function*. Or, as Parnas writes in "On the Criteria to be Used in Decomposing Systems into Modules" [Par72],

> ...one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the other [modules].

The rest of this chapter will discuss various ways to do this.

11.2 Re-using variables

Worked Exercise 11.2.1 ***Design a function*** *named* `gas-cost` *that estimates how much you'll spend on gasoline for a trip. It should take in the number of miles you're driving, and return how much you expect to spend, in dollars. Your car gets approximately 28 miles per gallon (*i.e. *this is an* inexact *number), and gasoline costs* $2.459 *per gallon. (This example was written in 2006, when that was a reasonable price for gasoline!)*

Solution:
Contract: The function takes in one number and returns another. (The 28 and 2.459

are important parts of the problem, but they're *fixed* numbers: they'll be the same every time you call the function, so they shouldn't be specified as parameters.) Thus

```
; gas-cost :  number(miles) -> number
```

Examples: As usual, we'll start with easy examples and gradually work up to more complicated ones: 0 miles requires 0 gas, hence costs $0.00. 28 miles requires 1 gallon of gas, so it costs $2.459. And so on.

```
"Examples of gas-cost:"
(check-within (gas-cost 0) 0 .01)
(check-within (gas-cost 28) 2.459 .01) ; i.e.  one gallon
(check-within (gas-cost 56) 4.918 .01) ; i.e.  two gallons
(check-within (gas-cost 77) 6.76 .01) ; 2-3/4 gal; use calculator
(check-within (gas-cost 358) 31.44 .01) ; yecch; use calculator
```

Skeleton:

```
(define (gas-cost miles)
  ...)
```

Inventory:

```
(define (gas-cost miles)
  ; miles      a number
  ; #i28       a fixed number I know I'll need
  ; 2.459      ditto
  ...)
```

Body: If you already see how to write the expression, great. If not, let's try the "inventory with values" trick. Pick a not-too-simple example, *e.g.*

```
(check-within (gas-cost 56) 4.918 .01) ; i.e.  two gallons
```

and fill in values:

```
(define (gas-cost miles)
  ; miles        a number          56
  ; 28           a fixed number    28
  ; 2.459        ditto             2.459
  ; should be    a number          4.918

  ...)
```

The number 4.918 doesn't look much like any of the previous numbers, but the one it resembles most closely is 2.459. In fact, it is exactly twice 2.459. So where did the 2 come from? Well, the number of miles in this example is exactly twice the miles-per-gallon figure, so one might reasonably guess that the formula is

```
(* (/ miles 28) 2.459)
```

Of course, this formula works for *this example*; we still need to **test** it on the remaining examples to be convinced that it works in general. ∎

The arithmetic expression in the body could be simplified somewhat: multiplying by 2.459 and dividing by 28 is equivalent to multiplying by approximately 0.08782142857, so we could have written

```
(define (gas-cost miles)
  ; miles              a number
  ; #i0.08782142857    a fixed number I know I'll need
  (* #i0.08782142857 miles)
  )
```

However, this program is *much harder to understand*. If one of your classmates (or yourself, three months from now) were to look at it, they'd have no idea where the 0.08782142857 came from, whereas in the previous version the algebraic expression "explains itself."

Why is this important? Because *program requirements change*. Imagine that you've worked out this program, and are just about to turn it in, when you learn that the price of gasoline has gone up to $3.899 per gallon. In the original version of the program, you simply replace 2.459 with 3.899 wherever it appears (and change the "right answers" accordingly), and it should work. In the "simplified" version, however, it's not obvious how the number 0.08782142857 needs to be changed, unless you remember that you got it by dividing the gasoline price by the fuel efficiency.

Now suppose you've written not just one but *several* programs that involve the current price of gasoline: say, there's also one that estimates how much money is wasted through spills, and one that estimates how much profit oil companies are making, *etc.* When the price of gasoline rises again, you'll need to change *all* the programs. This is a pain, and it violates the principle that "each change to requirements should affect only one variable or function." So to make our lives easier, let's *define a variable* to represent this number, and rewrite *all* the functions that use the price of gasoline to use the variable name, rather than the number, *e.g.*

```
(define PRICE-PER-GALLON 2.459)

; gas-cost :  number (miles) -> number
(define (gas-cost miles)
  ; miles               a number
  ; #i28                a fixed number I know I'll need
  ; PRICE-PER-GALLON    ditto
  (* PRICE-PER-GALLON (/ miles #i28))
  )
"Examples of gas-cost:"
(check-within (gas-cost 0) 0 .01)
(check-within (gas-cost 28) 2.459 .01) ; i.e.  one gallon
(check-within (gas-cost 56) 4.918 .01) ; i.e.  two gallons
(check-within (gas-cost 77) 6.76 .01) ; 2-3/4 gal; use calculator
(check-within (gas-cost 358) 31.44 .01) ; yecch; use calculator
```

```
; spillage-cost :  number (gallons spilled) -> number
(define (spillage-cost gallons)
  ; gallons      a number
  (* PRICE-PER-GALLON gallons)
  )
"Examples of spillage-cost:"
(check-within (spillage-cost 0) 0 .01)
(check-within (spillage-cost 1) 2.459 .01)
(check-within (spillage-cost 20000) 49180 1)

; etc.
```

SIDEBAR:

The use of ALL-CAPITALS in the variable name is a convention among Racket programmers (as well as C, C++, and Java programmers) to indicate a variable that represents a "fixed" or "constant" value. Of course, it isn't *really* constant, but it changes much less frequently than the number of miles driven or the number of gallons spilled. In this book, we'll often use ALL-CAPITALS for variables defined in their own right, to distinguish them from function parameters (which are also a kind of variable).

Now, the next time you hear that the price of gasoline has changed, you only need to change the value of `PRICE-PER-GALLON` in one place, and all the functions should now work with the new price. (You may also need to recalculate the "right answers" to your examples, but if your program worked before, and the only thing that's changed is the price of gasoline, you can be reasonably confident that your program will still work.)

Obviously, there's another "fixed" value in this problem that could change: the 28 miles per gallon.

Exercise 11.2.2 ***Replace 28*** *everywhere it appears in the program with a variable named* `MILES-PER-GALLON`*,* ***define*** *that variable appropriately, and make sure the program still works.*
Change the values *of the variable and the "right answers", and test that the program produces the new correct answers.*

As a general rule, if the same number appears more than once in your program, it deserves a name. Even if it appears only once, it's often a good idea to give it a name; a complex expression with a meaningful name in it is often easier to understand than the same expression with a "magic number" in it.

Of course, by "give it a name" I don't mean something silly like

```
(define TWELVE 12)
```

But if the number 12 appears several times in your program, figure out what each one *means*, and define a variable that makes the *meaning* clear. You may even discover that the same number currently appears in your program for two different reasons: for example, a program dealing with annual orders for a grocery store that sells both eggs and soda pop might include

```
(define MONTHS-PER-YEAR 12)
(define EGGS-PER-CARTON 12)
(define OZ-PER-CAN 12)
```

If the store suddenly started selling eggs in 18-egg cartons, or 16-oz cans of soda pop, you would need to change only one variable definition rather than going through the whole program line by line, looking for twelves and deciding which ones were twelve for which reason.

11.3 Composing functions

Recall Exercise 7.7.17, the `fahrenheit->kelvin` function, which could be written by simply calling one previously-written function (`celsius->kelvin`) on the result of another (`fahrenheit->celsius`). This sort of *re-use* has several benefits:

- `fahrenheit->kelvin` is easy to write, without needing to look up the formulæ or numbers for the other two functions.
- If you make an improvement to the accuracy or efficiency of one of the other functions, `fahrenheit->kelvin` will automatically become more accurate or efficient too. (Remember using `pi` to make `area-of-circle` and `area-of-ring` more accurate?)
- Each of the three functions can be tested and debugged separately (although, since `fahrenheit->kelvin` depends on the other two, there's not much point testing it until you have confidence in the other two).
- If you have confidence in the correctness of the other two functions, but get wrong answers from `fahrenheit->kelvin`, you don't need to look at the other two functions; you can confine your attention to *how they're combined* (*e.g.* perhaps you called them in the wrong order).
- Each of the three functions can be useful in its own right.
- Each of the three functions is shorter, simpler, and easier to understand than if they were all combined into one big function.[1]

Now let's think about `gas-cost` again. Intuitively, it first computes how many gallons of gas we need (from the mileage and the fuel efficiency), and then computes how much money that many gallons of gas cost (from the price of gas). Each of these questions ("how much gas does it take to drive a specified distance?" and "how much does a specified amount of gas cost?") could be useful in its own right. So let's *break the program up* into three separate functions:

[1]The human mind seems to have a hard time thinking about more than seven "things" at a time, according to George Miller's famous paper "The Magical Number Seven, Plus or Minus Two" [Mil56]. If your function definition has much more than seven variables and operators in it, it might be a good idea to break it into smaller, simpler pieces so you can hold the whole thing in your mind at once.

```
; gas-needed :  number (miles) -> number
"Examples of gas-needed"
(check-within (gas-needed 0) 0 .01)
(check-within (gas-needed 28) 1 .01)
(check-within (gas-needed 56) 2 .01)
(check-within (gas-needed 77) 2.75 .01)
(check-within (gas-needed 358) 12.8 .01)

; cost-of-gallons :  number (gallons) -> number
"Examples of cost-of-gallons:"
(check-within (cost-of-gallons 0) 0 .01)
(check-within (cost-of-gallons 1) 2.459 .01)
(check-within (cost-of-gallons 2) 4.918 .01)
(check-within (cost-of-gallons 2.75) 6.76225 .01)

; gas-cost :  number (miles) -> number
"Examples of gas-cost:"
(check-within (gas-cost 0) 0 .01)
(check-within (gas-cost 28) 2.459 .01) ; i.e.  one gallon
(check-within (gas-cost 56) 4.918 .01) ; i.e.  two gallons
(check-within (gas-cost 77) 6.76 .01) ; 2-3/4 gal; use calculator
(check-within (gas-cost 358) 31.44 .01) ; yecch; use calculator
```

Each of these functions is easy to write, particularly now that we've given names to the price of gasoline and the fuel efficiency of the car. Note that `gas-cost` shouldn't need to use any numbers *or* those two variables; it should simply use the other two functions.

Exercise 11.3.1 ***Write, test, and debug*** *the `gas-needed`, `cost-of-gallons`, and (new, improved) `gas-cost` functions.*

In general, there are several ways a new function can use an old function:

- rearranging or adding arguments, and passing these to the old function (*e.g.* the `convert-3-reversed` function of Exercise 7.7.19, or the draw handler of Exercise 10.2.1).
- calling one old function on the result of another (such as `fahrenheit->kelvin` and the new `gas-cost`)
- using the same old function several times (*e.g.* the `counterchange` function, which used `beside` twice).

Exercise 11.3.2 ***Develop a function*** *`cylinder-volume` that takes in the radius and height of a cylinder, and computes its volume.*

Hint: Look for a previously-written function you can re-use to do part of the job.

Exercise 11.3.3 ***Develop a function*** *`cylinder-area` that takes in the radius and height of a cylinder, and computes its area.*

Hint: The area includes the vertical sides and both ends.

Exercise 11.3.4 ***Develop a function*** `pipe-area` *that takes in the inner radius of a pipe, the length of the pipe, and the thickness of the walls, and computes its area.*

Hint: The area includes the inner surface, the outer surface, and the narrow top and bottom.

Exercise 11.3.5 *The nation of Progressiva has a simple tax code. The tax you pay is your salary times the tax rate, and the tax rate is 0.5% per thousand dollars of salary. For example, if you make $40,000, your tax rate is 0.5% times 40, which is 20%, so you pay 20% of $40,000, which is $8,000.*

Develop a function *to compute the net pay (i.e. pay after taxes) of a person with a given salary.*

Hint: You'll probably need two auxiliary functions as well as `net-pay`.

Exercise 11.3.6 *This tax system has the peculiar feature that, beyond a certain income level, if you earn more, you actually get less take-home pay.* ***Use*** *your* `net-pay` *function to find this income level by experimentation.*

Now imagine the tax rate rises to 0.6% per thousand dollars of salary. What would you need to modify in the program to handle this change? What would be the new income level beyond which you get less take-home pay?

11.4 Designing for re-use

When you're writing a program, sometimes you'll realize that there's an existing program that does most of the work for you. Take advantage of this opportunity (unless your instructor has specifically told you *not* to in a particular case); a good programmer is lazy, and refuses to re-invent the wheel. Recognizing and using such opportunities will save you a lot of time in programming.

But if the previous program was written to solve only one very narrow, specific problem, you may not be able to re-use it for your new problem. So when you're writing a new function, even if you don't immediately see any other application for it, *design it to be easily re-used*; you never know when some future problem will need it. What does this mean?

- Don't make unnecessary assumptions about your parameters.

  Suppose you're writing a function that takes in a string and displays it in an animation window. In the particular animation we're working on, we know that the string in question will always be just a single letter. But unless it's considerably easier or more efficient to write the function for single-letter strings, *don't assume the parameter is a single letter.* In fact, *test* it on different-length strings, even though you'll only need it for single letters in the current project, because a future project might need it for other strings.

  Here's a situation I see often. I've assigned an animation, which will require writing two or three event-handling functions, with (let's say) a number as the model. In the current animation, the model will never be bigger than 100, so one student writes the functions so they only work on numbers less than 100, while another writes them to work for any number. Later in the course, I give another assignment that requires some of the same functions, but no longer guarantees that the model

is never bigger than 100. The student who wrote a general, re-usable function in the first place can simply re-use the old function; the one who wrote a "narrower" function has to write a new one from scratch.

- Write each function to do one clear, simple task, not several.

  Suppose the current project requires computing how much I'm going to spend at gasoline stations for a trip, considering that every time I stop for gas I also buy a soda pop. You could write a single function that solves this whole problem, but it would be fairly complicated, and it would "tie up" the solutions to all the sub-problems so you can't solve one without the others. In particular, if a future project needed to compute the amount of gasoline used on a trip or the cost of a specified amount of gasoline, you would have to write those functions (and figure out the right formulæ, and test and debug) then anyway. A better approach is to write several simple functions: how much gas do I need, how much will the gas cost, how much will I spend on soda pop, *etc.* This way I can re-use whichever *parts* of the current project are needed in the future project.

There are whole books written, and whole college courses taught, about "designing programs for re-use", but the above two rules will get you started.

11.5 Designing multi-function programs: a case study

Let's take the gasoline-cost problem a step farther.

Worked Exercise 11.5.1 ***Design a function*** `road-trip-cost` *which determines the cost of a road trip, given the number of miles you're driving and how many days you'll be away. The car gets roughly 28 miles to the gallon, gasoline costs* $2.459 *per gallon, and motels cost* $40 *per night. Furthermore, you don't actually* own *a car, so you have to rent one. The car rental agency charges a fixed processing fee of* $10*, plus* $29.95 *per day, plus* $0.10 *per mile . Assume that you're bringing all your own food and drinks, so you don't need to worry about the cost of food on the road. Also assume that the "number of days you'll be away" includes both the day you leave and the day you return.*

Solution: This is a more complicated problem than we've yet seen. If you try to solve the whole thing at once, you'll be overwhelmed. Even if you manage to write a single function that solves the whole problem, it'll be long, complicated, and confusing. We need to take a more careful, methodical approach.

We can write a contract fairly easily:

```
; road-trip-cost :  number (miles) number (days) -> number
```

The examples will be a pain, since they require that *we* solve the whole problem at once, at least in specific cases. So before we jump into that, let's think about how to *break the problem into smaller pieces.*

What are the important values and quantities in the problem?

- the total cost of the road trip
- the number of miles we're driving
- the number of days we'll be away

- the fuel efficiency of the car
- the price of gasoline
- the amount of gasoline we need
- the amount we spend on gasoline
- the cost per night of a motel
- the number of nights we need to stay in a motel
- the amount we spend on motels
- the fixed processing fee for car rental
- the daily charge for car rental
- the per-mile charge for car rental
- the amount we spend on car rental

These quantities fall into several categories: some are *fixed numbers* (for which we probably want to use variables — see Section 11.2) — some are *inputs* (*i.e.* parameters) to the function, some are *output* from the function, and some are *intermediate results* that we need along the way.

Fixed numbers:

- the fuel efficiency of the car
- the price of gasoline
- the cost per night of a motel
- the fixed processing fee for car rental
- the daily charge for car rental
- the per-mile charge for car rental

Inputs to the `road-trip-cost` function:

- the number of miles we're driving
- the number of days we'll be away

Output from the `road-trip-cost` function: the total cost of the road trip
Intermediate results

- the amount of gasoline we need
- the amount we spend on gasoline
- the number of nights we need to stay in a motel
- the amount we spend on motels
- the amount we spend on car rental

The "fixed numbers", at least, should be easy.

```
(define MILES-PER-GALLON #i28)
(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)
```

Note that I've given all the variables names that a casual reader could understand. This makes them a bit long, but experience shows that the time saved in figuring out what a variable name means far exceeds the time spent typing long names.

At this point it is useful to figure out *which quantities depend on which others*; see Figure 11.1. Each of these intermediate results is a good candidate for a separate function. Because they aren't the function you originally intended to write, but will help you write it, they're called *auxiliary functions* or *helper functions*. Conveniently enough, we've already written `gas-cost`, `gas-needed`, and `cost-of-gallons`, but if we hadn't, we could write them now in the same way we'll write the rest. [2]

We still have four functions to write: "the total cost of the road trip", "the number of nights we need to stay in a motel", "the amount we spend on motels", and "the amount we spend on car rental". At this point, we know enough about them that we could write contracts and, perhaps, examples for all of them. This is often a good idea, because some of the functions will need to call one another, so it's best to decide *how* they'll be called as early as possible.

We already have a contract for `road-trip-cost`. Next let's try "the amount we spend on motels". Since `road-trip-cost` will depend on this function (among others), let's insert it in the Definitions pane *ahead* of what we've written so far about `road-trip-cost`. Anyway, a good name for this function might be `motel-cost`, and it obviously returns a number in dollars. It depends on the number of nights we stay in motels, and the cost per night of a motel. The latter is a fixed number, and the former is another intermediate value which in turn depends on the number of days of the trip. So

```
; motel-cost :  number (days) -> number
```

Since this depends on another function we haven't dealt with yet, let's postpone its examples until we've handled that other function.

Next: "The number of nights we spend in motels". A good name for the function could be `nights-in-motel`; it returns a integer, like 0, 1, 2, etc. And since `motel-cost` depends on this one, let's insert this one *ahead* of what we've just written about `motel-cost` in the Definitions pane.

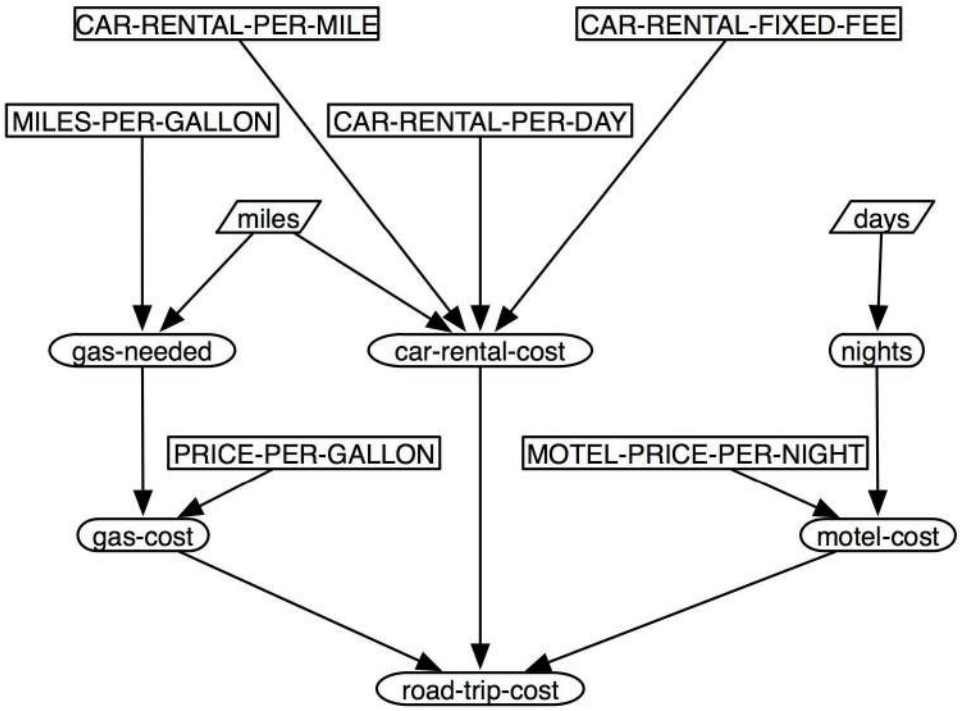
This function obviously depends on the number of days of the trip, so

```
; nights-in-motel :  number (days) -> number
```

As usual, we'll pick examples from the easiest to more complicated, and often the easiest number is 0. If the trip involves 0 days, then there *is no* "day that you leave" or "day that you return"; this example doesn't make sense. And a negative number or a fraction certainly wouldn't make sense. We've learned something about the limits of the problem; let's add this as an assumption.

[2] For some reason, many of my students seem to think that helper functions don't need contracts or test cases. This is analogous to building a house of cheap, low-quality bricks. If the bricks dissolve in the first rainstorm, the house will fall apart no matter how well designed it is. Similarly, if you're not clear on your helper functions' contracts, or you haven't tested them adequately, your whole program is unlikely to work.

Figure 11.1: Which quantities depend on which



- The fixed numbers don't depend on anything else.
- The inputs (`miles` and `days`) don't depend on anything else.
- "the total cost of the road trip" depends on three other quantities: how much we spend on gas, how much we spend on motels, and how much we spend on car rental.
- "the amount we spend on gasoline" depends on the amount of gasoline and the price of gasoline.
- "the amount of gasoline we need" depends on the number of miles and the fuel efficiency.
- "the number of nights we need to stay in a motel" depends on the number of days (but it's one less: if you leave on Monday and come back Tuesday, you've only stayed over one night).
- "the amount we spend on motels" depends on the number of nights and the cost per night of a motel.
- "the amount we spend on car rental" depends on the fixed fee, the daily charge, the number of days, the per-mile charge, and the number of miles we drive.

```
; nights-in-motel :  number (days) -> number
; Assumes the number of days is a positive integer.
```

The next simplest integer is 1, which would mean leaving today and coming back today, thus spending *zero* nights in motels. Similarly, if we took a 2-day trip, leaving today and coming back tomorrow, it would mean spending 1 night in a motel. In general, the number of nights is one less than the number of days.

```
"Examples of nights-in-motel:"
(check-expect (nights-in-motel 1) 0)
(check-expect (nights-in-motel 2) 1)
(check-expect (nights-in-motel 38) 37)
```

Now we can come back to the examples of `motel-cost`:

```
"Examples of motel-cost:"
(check-expect (motel-cost 1) 0)
(check-expect (motel-cost 2) 40)
(check-expect (motel-cost 38) 1480)
```

(Again, I used a calculator for the last one.)

Next is "the amount we spend on car rental." Let's name it `rental-cost`. It depends on five different quantities, but three of them are fixed numbers, and the other two are the number of miles and the number of days, which we have available as parameters to the main function. So

```
; rental-cost :  number (miles) number (days) -> number
```

The examples will take some arithmetic, but we can pick the numbers to make the arithmetic reasonably easy. Remember that we've already agreed 0 days doesn't make any sense. (One could make a good case that 0 miles doesn't make sense either; however, it's theoretically possible that we might get in the car, turn on the radio, chat for a while, and get out without ever going anywhere.)

As usual, we'll start from the easiest cases:

```
"Examples of rental-cost:"
(check-within (rental-cost 0 1) 39.95 .01)
(check-within (rental-cost 0 2) 69.90 .01)
(check-within (rental-cost 100 1) 49.95 .01)
(check-within (rental-cost 100 2) 79.90 .01)
(check-within (rental-cost 28 1) 42.75 .01)
(check-within (rental-cost 77 2) 77.60 .01)
(check-within (rental-cost 358 3) 135.65 .01)
```

The only function for which we don't have examples yet is `road-trip-cost` itself. So let's write some examples for it, using some of the numbers we've already worked out. The cost of the whole road-trip is found by adding up three other things: the cost of gasoline, the cost of motels, and the cost of car rental.

```
"Examples of road-trip-cost:"
(check-within (road-trip-cost 0 1) 39.95 .01)
; the gas and motels cost 0
(check-within (road-trip-cost 0 2) 109.90 .01)
; gas still 0, motel $40
(check-within (road-trip-cost 28 1) 45.209 .01)
; $42.75 for car, $0 for motel, $2.459 for gas
(check-within (road-trip-cost 77 2) 124.36 .01)
; $77.60 for car, c.  $6.76 for gas, $40 for motel
(check-within (road-trip-cost 358 3) 247.09 .01)
; $135.65 for car, c.  $31.44 for gas, $80 for motel
```

At this point, we've completed the "figure out what you want to do" for all four functions. This will be useful as we go on, because the definitions of some of them will depend on understanding what the others do. The Definitions pane should now look something like Figures 11.2 and 11.3.

We still have to move each of the four functions through the "skeleton", "inventory", "body", and "testing" stages ... but what to do first?

For the skeletons, inventories, and bodies, it doesn't really matter which function you work on first. Testing and debugging are another story. The `motel-cost` function depends on the `nights-in-motel` function, so we *can't* test the former until we've written the latter, and we certainly can't test the `road-trip-cost` function until everything else works. In other words, we have to build the program from the bottom up, like a brick building: finish the foundation before starting on the walls, and finish the walls before starting on the roof. Don't try to test and debug a function that depends on another function that hasn't been tested and debugged yet.

For clarity, I'll do one function (skeleton, inventory, body, testing) at a time; you could equally well do all the skeletons, then all the inventories, then all the bodies, then test them in order.

We have to start with a function that doesn't rely on any other functions (only fixed numbers and parameters to the main function). According to Figure 11.1, we have two choices: `nights-in-motel`, and `rental-cost`. Let's try `nights-in-motel`.

The skeleton and inventory should be straightforward and routine by now:

```
; nights-in-motel :  number (days) -> number
; Assumes the number of days is a positive integer.
(define (nights-in-motel days)
  ; days      a number
  ...)
```

The formula is obvious:

```
; nights-in-motel :  number (days) -> number
; Assumes the number of days is a positive integer.
(define (nights-in-motel days)
  ; days      a number
  (- days 1)
  )
```

Test this on the already-written examples. (To avoid getting error messages on examples of functions you haven't written yet, use the "Comment Out with Semicolons" menu

Figure 11.2: Constants and old functions

```
 ; Constants for the road-trip-cost problem:
(define MILES-PER-GALLON #i28)
(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)

; gas-needed :  number (miles) -> number
(define (gas-needed miles)
  ; miles               a number
  ; MILES-PER-GALLON    a number
  (/ miles MILES-PER-GALLON)
  )
"Examples of gas-needed:"
(check-within (gas-needed 0) 0 .01)
(check-within (gas-needed 28) 1 .01)
(check-within (gas-needed 56) 2 .01)
(check-within (gas-needed 77) 2.75 .01)
(check-within (gas-needed 358) 12.8 .01)

; cost-of-gallons :  number (gallons) -> number
(define (cost-of-gallons gallons)
  ; gallons             number
  ; PRICE-PER-GALLON    number
  (* gallons PRICE-PER-GALLON)
  )
"Examples of cost-of-gallons:"
(check-within (cost-of-gallons 0) 0 .01)
(check-within (cost-of-gallons 1) 2.459 .01)
(check-within (cost-of-gallons 2) 4.918 .01)
(check-within (cost-of-gallons 2.75) 6.76225 .01)

; gas-cost :  number (miles) -> number
(define (gas-cost miles)
  ; miles      number
  (cost-of-gallons (gas-needed miles))
  )
"Examples of gas-cost:"
(check-within (gas-cost 0) 0 .01)
(check-within (gas-cost 28) 2.459 .01) ; i.e.  one gallon
(check-within (gas-cost 56) 4.918 .01) ; i.e.  two gallons
(check-within (gas-cost 77) 6.76 .01) ; 2-3/4 gal; use calculator
(check-within (gas-cost 358) 31.44 .01) ; yecch; use calculator
```

Figure 11.3: Contracts and examples for new functions

```
 ; nights-in-motel :  number (days) -> number
; Assumes the number of days is a positive integer.
"Examples of nights-in-motel:"
(check-expect (nights-in-motel 1) 0)
(check-expect (nights-in-motel 2) 1)
(check-expect (nights-in-motel 38) 37)

; motel-cost :  number (days) -> number
"Examples of motel-cost:"
(check-expect (motel-cost 1) 0)
(check-expect (motel-cost 2) 40)
(check-expect (motel-cost 38) 1480)

; rental-cost :  number (miles) number (days) -> number
"Examples of rental-cost:"
(check-expect (rental-cost 0 1) 39.95)
(check-expect (rental-cost 0 2) 69.90)
(check-expect (rental-cost 100 1) 49.95)
(check-expect (rental-cost 100 2) 79.90)
(check-expect (rental-cost 28 1) 42.75)
(check-expect (rental-cost 77 2) 77.60)
(check-expect (rental-cost 358 3) 135.65)

; road-trip-cost :  number (miles) number (days) -> number
"Examples of road-trip-cost:"
(check-within (road-trip-cost 0 1) 39.95 .01) ; the gas and motels are 0
(check-within (road-trip-cost 0 2) 109.90 .01) ; gas still 0, motel $40
(check-within (road-trip-cost 28 1) 45.209 .01)
; $42.75 for car, $0 for motel, $2.459 for gas
(check-within (road-trip-cost 77 2) 124.36 .01)
; $77.60 for car, c.  $6.76 for gas, $40 for motel
(check-within (road-trip-cost 358 3) 247.09 .01)
; $135.65 for car, c.  $31.44 for gas, $80 for motel
```

command to comment out everything not related to `nights-in-motel`.) If it produces correct answers in every case, go on to the next function.

Staying on the same subject, let's do `motel-cost`. Uncomment the lines related to this function, and write the skeleton and inventory:

```
; motel-cost :  number (days) -> number
; Assumes the number of days is a positive integer.
(define (motel-cost days)
  ; days       a number
  ...)
```

In addition, we know from Figure 11.1 that the answer depends on the cost per night, `MOTEL-PRICE-PER-NIGHT`, so let's add that to the inventory. Furthermore, we don't actually care about the number of *days* so much as the number of *nights*, which we can get by calling `nights-in-motel`, so we'll add that to the inventory too:

```
; motel-cost :  number (days) -> number
; Assumes the number of days is a positive integer.
(define (motel-cost days)
  ; days                      a number
  ; MOTEL-PRICE-PER-NIGHT     a number
  ; (nights-in-motel days)    a number
  ...)
```

Now, since `(nights-in-motel days)` represents the number of nights, the formula is straightforward:

```
; motel-cost :  number (days) -> number
; Assumes the number of days is a positive integer.
(define (motel-cost days)
  ; days                      a number
  ; MOTEL-PRICE-PER-NIGHT     a number
  ; (nights-in-motel days)    a number
  (* MOTEL-PRICE-PER-NIGHT (nights-in-motel days))
  )
```

Test this on the already-written examples. If it produces correct answers in every case, go on to the next function. If it doesn't, use the Stepper to decide whether the mistake is in `nights-in-motel` (it shouldn't be, since we've already tested that function) or in this one (much more likely); fix the problem and re-test.

The only function we can do next is `rental-cost`. Uncomment its examples and write a skeleton and (a start on an) inventory:

```
; rental-cost :  number (miles) number (days) -> number
(define (rental-cost miles days)
  ; miles      a number
  ; days       a number
  ...)
```

According to Figure 11.1, it also depends on three fixed numbers: `CAR-RENTAL-FIXED-FEE`, `CAR-RENTAL-PER-DAY`, and `CAR-RENTAL-PER-MILE`, so we'll add these to the inventory:

```
; rental-cost :  number (miles) number (days) -> number
(define (rental-cost miles days)
  ; miles                   a number
  ; days                    a number
  ; CAR-RENTAL-FIXED-FEE    a number
  ; CAR-RENTAL-PER-DAY      a number
  ; CAR-RENTAL-PER-MILE     a number
  ...)
```

The "daily charge" obviously needs to be multiplied by the number of days, and the "per mile charge" obviously needs to be multiplied by the number of miles; add these expressions to the inventory. (If this isn't "obvious", try the "inventory with values" technique.)

```
; rental-cost :  number (miles) number (days) -> number
(define (rental-cost miles days)
  ; miles                          a number
  ; days                           a number
  ; CAR-RENTAL-FIXED-FEE           a number
  ; CAR-RENTAL-PER-DAY             a number
  ; CAR-RENTAL-PER-MILE            a number
  ; (* days CAR-RENTAL-PER-DAY) > a number
  ; (* miles CAR-RENTAL-PER-MILE) a number
  ...)
```

These last two expressions represent the amount the rental company charges for days, and for miles, respectively. If we add up these two and the fixed fee, we should get the final answer:

```
; rental-cost :  number (miles) number (days) -> number
(define (rental-cost miles days)
  ; miles                             a number
  ; days                              a number
  ; CAR-RENTAL-FIXED-FEE              a number
  ; CAR-RENTAL-PER-DAY                a number
  ; CAR-RENTAL-PER-MILE               a number
  ; (* days CAR-RENTAL-PER-DAY)       a number
  ; (* miles CAR-RENTAL-PER-MILE)     a number
  (+ (* days CAR-RENTAL-PER-DAY)
     (* miles CAR-RENTAL-PER-MILE)
     CAR-RENTAL-FIXED-FEE)
  )
```

Test this on the already-written examples. If it produces correct answers in every case, go on to the next function. If not, use the Stepper to locate the problem (as before); fix it and re-test.

The only function remaining is `road-trip-cost` itself. We follow the same procedure:

```
; road-trip-cost :  number (miles) number (days) -> number
(define (road-trip-cost miles days)
  ; miles      a number
  ; days       a number
  ...)
```

We know that the answer will involve what we spend on gas, what we spend on motels, and what we spend on car rental. Fortunately, there are functions that compute each of these, and the only inputs those functions require are miles and/or days, both of which we have. So we can add calls to those functions to the inventory:

```
; road-trip-cost :  number (miles) number (days) -> number
(define (road-trip-cost miles days)
  ; miles                        a number
  ; days                         a number
  ; (gas-cost miles)             a number
  ; (motel-cost days)            a number
  ; (rental-cost miles days)     a number
  ...)
```

With these expressions in hand, the answer is obvious: add them up.

```
; road-trip-cost :  number (miles) number (days) -> number
(define (road-trip-cost miles days)
  ; miles                        a number
  ; days                         a number
  ; (gas-cost miles)             a number
  ; (motel-cost days)            a number
  ; (rental-cost miles days)     a number
  (+ (gas-cost miles)
     (motel-cost days)
     (rental-cost miles days))
  )
```

Test this on the already-written examples. If it produces correct answers in every case, congratulate yourself: we've developed a fairly complex program by breaking it down into small, digestible pieces. ▮

Exercise 11.5.2 ***Choose one of the fixed numbers*** *in the above problem: either* `MILES-PER-GALLON`*,* `PRICE-PER-GALLON`*,* etc. ***Change its numeric value.*** Before re-running the program*,* ***predict*** *which examples are affected, and* ***recalculate*** *(by hand or calculator) their new correct values.* ***Test*** *the program to see if your predictions were right.*

By the way, we *could* in principle have written the whole function at once, without breaking it down into small pieces, and the result might have looked like this:

```
(define (monolithic-rtc miles days)
  (+ (* (/ miles MILES-PER-GALLON) PRICE-PER-GALLON)
     (* MOTEL-PRICE-PER-NIGHT (- days 1))
     (+ (* days CAR-RENTAL-PER-DAY)
        (* miles CAR-RENTAL-PER-MILE)
        CAR-RENTAL-FIXED-FEE)
     )  )
```

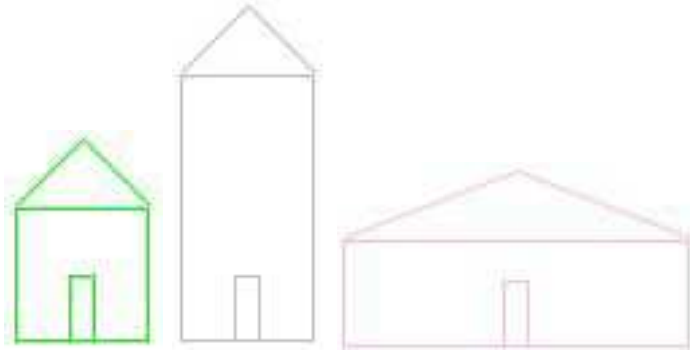
If we got everything right on the first try, this would actually be quicker and easier than writing seven separate functions . . . but computer programs are almost *never* right on the first try, especially if they're more than two or three lines long. If something were wrong in this definition, it would be quite difficult to track down the mistake(s).

The approach we took, breaking the problem into several small functions, has at least two major advantages: one can test each function individually, and some of the functions may be re-usable from one project to another (*e.g.* the `gas-needed`, `cost-of-gallons`, and `gas-cost` functions which we just copied from a previous problem).

11.6 Practicalities of multi-function programs

As you've seen, a multi-function program is written by applying the design recipe to each of the functions in turn, and testing them "bottom-up" — that is, the functions that don't depend on any others first, then the ones that depend on the first few, and finally the main function that depends on all the others. And you have a certain amount of flexibility how far to go on which function in which order.

However, the final result should look as though you had written one function at a time: the contract, skeleton-turned-body, and examples for one function should appear together with no other function definitions in between. In other words, you may need to do a certain amount of moving forward and backwards to find the right places to type things.

Exercise 11.6.1 ***Develop a function*** `build-house` *that draws a picture of a house, like these:*  *Note that houses can be of different widths, heights, and colors, but the door is always the same size, and centered on the floor of the house. The roof is also always the same height.*

Hint: You may find it helpful to write some auxiliary functions that produce images, and others that produce numbers.

Exercise 11.6.2 *A small commercial airline company wants to figure out how much to charge for its tickets. Specifically, for any given ticket price, they'd like to be able to predict how much profit they'll make. Given that ability, they can try various different ticket prices, see which produces the most profit, and select that as their ticket price.*

Profit, of course, is income minus expenses. There are two major expenses: paying the salaries of the pilot, copilot, and two flight attendants (these four salaries add up to $450 per flight, regardless of how many passengers are on the flight), and jet fuel, at $2.999/gallon. The amount of jet fuel consumed is one gallon per twenty pounds of loaded weight, which is the weight of the airplane itself plus the weight of the people and luggage on it. The airplane itself weighs 50000 pounds. Each passenger and his/her luggage, on average, weighs 250 pounds; same for the four crew members (pilot, copilot, two flight attendants).

The airline is currently charging $200/ticket, and at that price they can typically sell 120 tickets. Raising the price means they make more money on each ticket, but it also

causes fewer people to buy tickets; conversely, lowering the price means they take in less money on each ticket, but they can sell more of them. To be precise, they estimate that for each $10 they raise (lower) the ticket price, they'll lose (gain) 4 paying passengers.

Develop a function `airline-profit` *that takes in a proposed ticket price, and returns the estimated profit the airline will make at that price.*

Use *your function to determine the profit-maximizing ticket price. Also find the* least *they could charge and make any profit at all.*

Change *one or two of the constants (*e.g. *the price of jet fuel, the number of people who change their minds about buying tickets when the price goes up or down, the size of the crew, the crew salaries,* etc.*) and repeat the previous paragraph.*

Exercise 11.6.3 ***Develop a function*** *that takes in the name of a color (*e.g. `"green"`*) and produces that word, followed by a randomly-chosen numeric font size (say, between 10 and 30 points inclusive), in text of that color and font size, surrounded by a box of the same color which is 10 pixels wider and 6 pixels higher than the text. For example,*

Hint: Since this function has random results, it'll be difficult to write test cases for. I did it with two helper functions, both of which contained no randomness and therefore could be tested using `check-expect`.

11.7 Re-using definitions from other files

By this time you've probably gotten used to me saying things like
Hint: Re-use a function you've already written!

If the function you need to re-use is in the same Definitions pane, this is no problem: you can just call it in the definition of your new function. But what if you want to re-use something you wrote weeks ago, *e.g.* for a previous homework assignment? For example, suppose you did some problems from chapter 9, saved them in the file `chap9.rkt`, then started working on chapter 10 in the file `chap10.rkt` and realized that problem 10.2.4 would be easier if you re-used the `number->image` function you wrote for problem 9.2.7.

You could do this by opening `chap9.rkt`, copying the relevant definition (and its test cases), pasting it into `chap10.rkt`, then using the function as usual. But doing this should bother you: remember the rule *if you write almost the exact same thing over and over, you're doing something wrong.* (In fact, this time you're writing *exactly* the same thing over and over, just in different files.) One problem with this is that if you discover

a bug in that function definition, now it's in two different files so you have to remember to fix it in both places.

It's quite common for professional programmers to realize, while working on one program, that they need a function that they originally wrote as part of a different program. And every modern programming language provides a way to re-use such things without copying them; Racket is no exception.

11.7.1 `require` and `provide`

Depending on what version of DrRacket you have, this may not work. If not, you can either use copy-and-paste, or download a newer version of DrRacket.

In a sense, you've already seen how Racket does this: `require`. Since the beginning of this book, you've been writing

```
(require picturing-programs)
```

to tell DrRacket that you want to be able to use all the variables and functions defined in the `picturing-programs` library. You can do something similar to tell DrRacket that you want to be able to use things defined in a previous program of your own:

```
(require "chap9.rkt")
```

Note: I've put the name `chap9.rkt` in quotation marks. When you do this, DrRacket expects it to be the name of a file you wrote yourself. On the other hand, if you `require` something that's not in quotation marks, DrRacket expects it to be the name of a standard built-in library, like `picturing-programs`.

But just putting

```
(require "chap9.rkt")
```

into `chap10.rkt` isn't quite enough: if you try to use `image->number` in `chap10.rkt`, you'll still get an error message like *`number->image: this variable is not defined`*. This is because DrRacket respects the privacy of other files, and doesn't read anything from them for which it hasn't specifically been given permission.

Each Racket file has its own "privacy settings": each Racket file is expected to specify which of the things defined in that file can be used in other files. Open `chap9.rkt` and add the line

```
(provide number->image)
```

somewhere (I usually put things like this near the beginning, just after the `(require picturing-programs)`.) This tells DrRacket that other files are allowed to see the `number->image` function defined in this file — but nothing else. Now save `chap9.rkt`, go back to `chap10.rkt`, which should still have the line

```
(require "chap9.rkt")
```

in it, and run it; it should now be able to use `image->number` as though it were defined in the same Definitions pane.

Note: When you `require` a file of your own like this, DrRacket looks for it in the same folder that you're already in. So if you want to take advantage of this feature, make sure to save all your `.rkt` files in the same folder. There are ways to refer to files in other folders; if you need to do that, you can read about it in the Help Desk.

This also means that if you open a new DrRacket window, start writing, and try to run it without saving first, DrRacket doesn't *know* what "folder you're already in," so it

can't find the other file. So before you try to run a file that requires another file, make sure you've saved both files.

Practice Exercise 11.7.1 *Create two files* `part-a.rkt` *and* `part-b.rkt`*.*

In `part-a.rkt`*, define a variable* `my-pic` *to stand for some picture you've built, and define the* `counterchange` *and* `surround` *functions from exercises 4.2.3 and 4.2.4. You should* `provide` `my-pic` *and* `surround`*, but not* `counterchange`*.*

In `part-b.rkt`*, define a function* `surround-with-my-pic` *that takes in a picture and surrounds it with two copies of* `my-pic`*, re-using (but not copying) the definitions of* `surround` *and* `my-pic` *from* `part-a.rkt`*. Make sure this works, then add a call to* `counterchange` *and confirm that it doesn't pass a syntax check (much less run).*

Hint: In `part-a.rkt`, you can write two separate `provide` lines:

```
(provide my-pic)
(provide surround)
```

or you can combine them into one:

```
(provide my-pic surround)
```

11.7.2 `provide`-ing everything

Now suppose you've written a file that contains a lot of definitions that you want to use in other files. You can `provide` them all one by one:

```
(provide this-function
         that-function
         the-other-function
         yet-another-function
         lots-of-other-functions)
```

but for the common situation that you want to `provide` *everything* in the file, there's a shorthand:

```
(provide (all-defined-out))
```

Practice Exercise 11.7.2 *Replace the* `(provide my-pic surround)` *in* `part-a.rkt` *from exercise 11.7.1 with* `(provide (all-defined-out))`*, and confirm that the* `counterchange` *call in* `part-b.rkt` *works again.*

Hint: If it doesn't, the most likely reason is that you haven't saved the modified version of `part-a.rkt` yet.

11.8 Review of important words and concepts

Program requirements change, so it's in your interest to prepare for such change. One way to do this is to use *symbolic constants*: variables with meaningful names to represent important or likely-to-change values. When these values *do* change, you can just change the variable definition in one place, and all the functions and programs that use this variable will automatically be corrected. Another way is to *design large programs so that each likely change affects only one function.*

To paraphrase John Donne, "no program is an island." Every program you ever write has the potential to be re-used to make subsequent programs easier, and it's in your interest to design the current program to maximize the likelihood that you can re-use it later. In particular, *don't make unnecessary assumptions about your input*, and *design each function to do one clear, simple task.*

A large program is often made up of a "main function" and one or more *auxiliary* or "helper" functions. When faced with a large program to write, *break it down into manageable pieces* Identify the important *quantities* in the problem, and categorize each as an *input*, an *output*, a *fixed value*, or an *intermediate computation.* The inputs will probably become parameters to your main function (and to some of the auxiliary functions); the output will probably be the result of your main function; the fixed values will probably become symbolic constants; and the intermediate computations will probably become auxiliary functions. It's often a good idea to write down contracts (and perhaps examples) for all of these functions at once, so you have a clear idea what they're all supposed to do. Then start writing skeletons, inventories, and bodies, and testing the functions one at a time, starting with the ones that *don't* depend on any other functions.

Really large programs are often spread out over several files, either as an organizational technique to keep closely-related parts of the program together, or simply because you wrote some of the functions for other purposes and want to re-use them. In Racket, the `require` function allows you to specify that you want to re-use the definitions from a particular other file, while the `provide` function allows you to specify which definitions from this file can be re-used in others. (Other languages like Java have different mechanisms to accomplish the same thing.)

11.9 Reference

The only new function introduced in this chapter is `provide`, which takes one or more names defined in this file and makes them available to other files that `require` this file. `provide` can also be given the shorthand `(all-defined-out)` to provide *everything* that was defined in this file. There are other such shorthands — `all-from-out`, `rename-out`, `prefix-out`, `struct-out` — but I don't want to go into them here; you can read about them if you're interested.

We also learned a new way of using `require`: with a filename in quotation marks, for which DrRacket will look in the same folder as the file that contains the `require` line.