Chapter 15

Conditionals

15.1 Making decisions

We saw in Chapter 13 how to write functions that answer yes/no questions, and we saw in Chapter 14 how to use such a function as the stopping criterion for an animation. But Booleans are used much more generally to *make decisions* about what computation to do.

For example, imagine an e-commerce site that sells games and game equipment of several different kinds: basketballs, baseballs, mitts, bats, Monopoly boards, chess boards and pieces, Nintendo consoles, *etc.*. One part of the site might give a list of the names of different games; a user would select one from the menu, and the site would then display a picture of the selected game.

You know how to display pictures, but this application requires displaying *one of several* pictures, depending on which menu item the user selected. The computer needs to do something like the following:

```
If the user selected "basketball",
  display the picture of a basketball.
If not, see whether the user selected "baseball";
  if so, display the picture of a baseball.
If not, see whether the user selected "Monopoly";
  if so, display the picture of a Monopoly set.
etc.
```

(We'll solve this problem in Exercise 15.3.1.)

This is such a common pattern in programming that Racket (and most other languages) provides a construct to do it, called a *conditional* (or `cond` for short). A Racket conditional includes a series of "question/answer" pairs: Racket evaluates the first question, and if the result is `true`, it returns the first answer. If not, it goes on to the second question: if the second question evaluates to `true`, it returns the second answer; if not, it goes on to the third question, and so on.

Syntax Rule 6 *Anything matching the following pattern is a legal expression:*
```
(cond [question1 answer1]
      [question2 answer2]
      [question3 answer3]
      ...
      [questionn answern]
      )
```
as long as each of the questions *is an expression of type Boolean, and each* answer *is an expression.*

Note that the questions and answers must come in pairs, surrounded by square brackets. Each question/answer pair is sometimes called a "`cond`-clause".

SIDEBAR:

In fact, you can use parentheses instead of the square brackets. To avoid confusion, I prefer to use parentheses only for calling functions, as we've been doing so far, and to use square brackets to group together each question/answer pair in a conditional.

Practice Exercise 15.1.1 *Type the following into the Interactions pane:*
```
(cond [(string=? "hello" "goodbye") "something is wrong!"]
      [(string=? "snark" "snark") "this looks better"]
      )
```
The result should be the string `"this looks better"`.

Type (or copy-and-paste) the same three lines into the Definitions pane and hit "Run"; you should get the same result.

Then hit "Step" instead of "Run": note that it first evaluates the expression `(string=? "hello" "goodbye")` to `false`, then throws away the answer `"something is wrong!"` and goes on to the second question/answer pair.

Practice Exercise 15.1.2 *Now try an example in which* none *of the questions comes out `true`:*
```
(cond [(string=? "hello" "goodbye") "something is wrong!"]
      [(string<? "snark" "boojum") "this isn't true either"]
      [(< 5 2) "nor this"]
      )
```
Type this into the Interactions pane; you should get an error message saying cond: all question results were false.

Copy the same four lines into the Definitions pane and hit "Run"; you should get the same error message.

Now try the Stepper; you'll see it evaluate each of the three questions in turn to `false`, discard each of the three answers, and then produce the error message.

Practice Exercise 15.1.3 *Here's an example in which a question other than the last one evaluates to `true`:*
```
(cond [(= (+ 2 5) (- 10 3))          "yes, this works"]
      [(string=? "hello" "goodbye") "this shouldn't happen"]
      [(string<? "goodbye" "hello")
       "this is true but it shouldn't get here"]
      )
```
Note that since the first question evaluates to `true`, Racket returns `"yes, this works"` and never bothers to even evaluate the other two questions, as you can see by using the

Stepper. This demonstrates that `cond`*, like* `define`*,* `and`*, and* `or`*, is a* special form *rather than a regular function.*

Of course, nobody really uses conditionals directly in Interactions; they're almost always used inside a function. Let's try a more realistic example:

Worked Exercise 15.1.4 ***Develop a function*** `reply` *which recognizes any one of the strings* `"good morning"`*,* `"good afternoon"`*, or* `"good night"`*, and returns either* `"I need coffee"`*,* `"I need a nap"`*, or* `"bedtime!"` *respectively. If the input isn't any of the three known strings, the function may produce an error message.*

Solution: Contract:

```
; reply :  string -> string
```

If we wished, we could be more specific about what input values are allowed, and what results are possible:

```
; reply:string("good morning","good afternoon",or "good night")->
; string ("I need coffee", "I need a nap", or "bedtime!")
```

Examples: There are exactly three legal inputs, and exactly three legal outputs, so to test the program adequately, we should try all three.

```
"Examples of reply:"
(check-expect (reply "good morning") "I need coffee")
(check-expect (reply "good afternoon") "I need a nap")
(check-expect (reply "good night") "bedtime!")
```

Skeleton: The input represents a greeting, so let's use that name for the parameter:

```
(define (reply greeting)
  ...)
```

Inventory: As usual, we need the parameter `greeting`. Since there are three possible cases, we can be pretty sure the function will need a conditional with three clauses, or question/answer pairs:

```
(define (reply greeting)
  ; greeting       a string
  ...)
```

Skeleton, revisited: Since there are three possible cases, we can be pretty sure the function will need a conditional with three clauses, or question/answer pairs:

```
(define (reply greeting)
  ; greeting      a string
  (cond [...    ...]
        [...    ...]
        [...    ...]
  ))
```

To complete the function definition, we need to fill in the "..." gaps. I usually recommend filling in either all the answers, then all the questions, or *vice versa*, depending on which looks easier. In this case, we know exactly what the possible answers are, so let's fill them in first:

```
(define (reply greeting)
  ; greeting      a string
  (cond [...      "I need coffee"]
        [...      "I need a nap"]
        [...      "bedtime!"]
  ))
```

We still need to fill in the questions, each of which should be a Boolean expression, probably involving `greeting`. Under what circumstances should the answer be `"I need coffee"`? Obviously, when `greeting` is `"good morning"`. Filling this in, we get

```
(define (reply greeting)
  ; greeting      a string
  (cond [ (string=?  greeting "good morning")   "I need coffee"]
        [...                                    "I need a nap"]
        [...                                    "bedtime!"]
  ))
```

We can do the same thing for the other two cases:

```
(define (reply greeting)
  ; greeting      a string
  (cond [(string=?  greeting "good morning")   "I need coffee"]
        [ (string=?  greeting "good afternoon")"I need a nap"]
        [ (string=?  greeting "good night")    "bedtime!"]
  ))
```

Now **test** the function on the three examples we wrote earlier; it should work. ∎

Incidentally, if you write a function definition with several `cond`-clauses and you provide test cases for only *some* of them, when you hit "Run" to see the results, the parts of the program you tested will be colored black, and the parts you *didn't* test will be reddish-brown.

Practice Exercise 15.1.5 *Try commenting out one or two of the test cases and hitting "Run" to see this.*

15.2 Else and error-handling

The `reply` function is unsatisfying in a way: if the input is anything other than one of the three known greetings, the user gets the ugly error message *cond: all question results were false*. It would be friendlier if we could write our program with an "anything else" case, producing a more appropriate message like `"I don't understand"` or `"huh?"` if the input isn't something we recognize.

Racket provides a keyword `else` for just this purpose. If you use `else` as the *last question* in a `cond`, it will catch all cases that haven't been caught by any of the earlier questions. For example,

```
(define (reply greeting)
  ; greeting      a string
  (cond [(string=?  greeting "good morning")   "I need coffee"]
        [(string=?  greeting "good afternoon") "I need a nap"]
        [(string=?  greeting "good night")     "bedtime!"]
        [else                                  "huh?"]
  ))
```

When this function is applied to a string like `"good evening"`, which doesn't match any of the known greetings, Racket will then go on to the last question, which is `else` so Racket returns the corresponding answer `"huh?"` rather than the error message.

SIDEBAR:

Technically, the `else` keyword isn't necessary: if you were to use `true` as the last question, it would (obviously) always evaluate to `true`, so it too would catch all cases that haven't been caught by any earlier question. Try the above function with `true` in place of `else`; it should work exactly as before.

However, `else` is considered easier to read and understand, so most Racket programmers use it in this situation.

By the way, `else` *cannot* appear anywhere in Racket *except* as the last question in a `cond`.

Note that the `else` is *not in parentheses*: it is not a function, and it cannot be applied to arguments. If you *put* it in parentheses, you'll get an error message.

15.3 Design recipe for functions that make decisions

To write functions on data types defined by choices, we'll add a few more details to the design recipe.

1. Write a contract (and perhaps a purpose statement).
2. Analyze input and output data types: *if either or both is made up of several choices, figure out how many and how to describe them.*
3. Write examples of how to use the function, with correct answers. *If an input or output data type is defined by choices, be sure there's at least one example for each choice.* If an input type involves sub-ranges, be sure there's an example at the borderline(s).
4. Write a function skeleton, specifying parameter names.
5. Write an inventory of available expressions, including parameter names and obviously relevant literals, along with their data types (and, if necessary, their values for a chosen example).
6. *Add some details to the skeleton: if an input or output data type is defined by several choices, write a* **`cond`** *with that many question/answer pairs, with "..." for all the questions and answers.*
7. Fill in the function body. *If the skeleton involves a* **`cond`***, fill in either all the answers or all the questions, whichever is easier, and then go back to fill in the other column. If one of the choices is "anything else", use `else` as the last question in the `cond`.*
8. Test the function.

To elaborate on step 7,

- if the answers are simple (*e.g.* a fixed set of known values), first fill in all the answers, then go back and figure out what question should lead to each answer; or

- if the questions are simpler than the answers (*e.g.* if the answers are complex expressions, but the questions are simply matching a parameter against a fixed set of known values), first fill in all the questions, then go back and figure out what expression will produce the right answer for each one. In particular, if one of the input choices is "anything else", detect this with `else` as the last question.

Recall that Syntax Rule 6 says the first expression in each `cond`-clause must be an expression of type Boolean, but it doesn't say anything about what type(s) the *second* expressions must be. In most cases, these should all be the same type as the return type of the function you're writing. For example, `reply` is supposed to return a string, so the second expression in each `cond`-clause is a string; in the e-commerce application we discussed earlier, the return type is a picture, so the second expression in each `cond`-clause should be a picture.

Exercise 15.3.1 ***Develop a function*** *`choose-picture` that takes in a string (either `"basketball"`, `"baseball"`, `"Monopoly"`,* etc.*; you can choose your own names if you wish, but don't choose more than about five) and returns a picture of that object (which you should be able to find on the Web).*

Exercise 15.3.2 ***Modify*** *exercise 15.3.1 so that if the input isn't any of the known games, it produces a picture of a question mark (or a person looking puzzled, or something like that).*

Exercise 15.3.3 ***Develop a function*** *named `random-bw-picture` that takes in a width and a height, and produces a rectangle that size and shape, in which each pixel is randomly either black or white.*

Exercise 15.3.4 *The town of Racketville needs a new computerized voting system. In an early version of the system, we assume there are exactly 4 voters (we'll see later how to handle an arbitrary number of voters).*
Develop a function *`count-votes-4` that takes in five strings. The first is the name of a candidate, and the other four are votes which might or might not be for that candidate. The function should return* how many *of the votes are for the specified candidate. For example,*

```
(check-expect
  (count-votes-4 "Anne" "Bob" "Charlie" "Bob" "Hortense") 0)
; since there are no votes for Anne
(check-expect
  (count-votes-4 "Anne" "Bob" "Anne" "Phil" "Charlie") 1)
(check-expect
  (count-votes-4 "Anne" "Anne" "Bob" "Anne" "Mary") 2)
(check-expect
  (count-votes-4 "Bob" "Anne" "Bob" "Charlie" "Bob") 2)
```

Hint: Write an auxiliary function that takes in two strings and returns either 1 (if they match) or 0 (if they don't).

Obviously, it's a pain passing around four votes as parameters, and it would be even worse if you had hundreds or thousands of votes. We'll see how to handle larger numbers of data in Chapter 22.

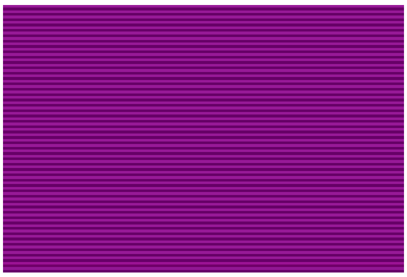
Exercise 15.3.5 ***Develop a function*** *`smallest-of-3` that takes in three numbers and returns the smallest of them.*

There is a built-in function `min` which does this job for you; for this exercise, use conditionals, not *`min` or anything like it.*

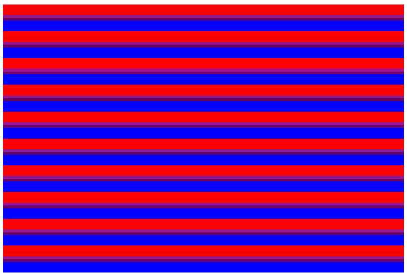
Exercise 15.3.6 ***Develop a function*** *`rough-age` that takes in a number representing a person's age and returns one of the strings `"child"`, `"teenager"`, or `"adult"` as appropriate. A "teenager" is at least 13 but less than 20 years old; a "child" is under 13; and an "adult" is at least 20.*

Exercise 15.3.7 *Using `build3-image`,* ***build a rectangular image*** *150 pixels wide by 100 pixels high which is yellow above the diagonal (from top-left to bottom-right corner) and blue below the diagonal.*

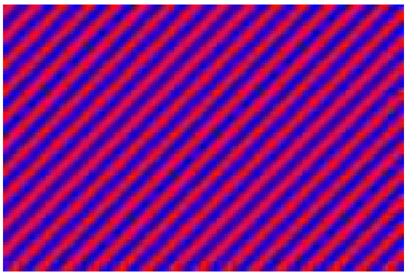
Exercise 15.3.8 ***Develop a function*** *`make-stripes` that takes in a width and height (in pixels), and produces a rectangle that size and shape in which all the even-numbered rows are red and the odd-numbered rows are blue. The result should be a bunch of narrow stripes.*



Exercise 15.3.9 *These stripes are really too narrow to see easily.* ***Develop a function*** *`make-wide-stripes` that does the same thing only with each stripe 5 pixels high: rows 0-4 are red, 5-9 are blue, 10-14 are red,* etc.



Exercise 15.3.10 ***Develop a function*** *`make-diag-stripes` that takes in a width and height (in pixels), and produces a rectangle that size and shape filled with stripes running from upper-right to lower-left.*



Exercise 15.3.11 ***Define a function*** `simplify-colors` *that takes in an image and produces an image the same size and shape: for each pixel in the given image, if it has more red than green or blue, make the resulting pixel pure red; if it has more green than red or blue, make it green; and if it has more blue than red or green, make it pure blue. In case of ties, you can decide what to do: pick one of the three colors arbitrarily, or make it white, or something like that.*

Exercise 15.3.12 *Come up with other cool things to do to images using conditionals. Go wild.*

15.4 Case study: bank interest

My bank offers savings accounts with a sliding scale of interest, depending on how much money is in the account: if you have less than $500 in the account, you earn no interest; for $500-$1000, you earn 1% per year interest; for $1000-$4000, you earn 2% per year, and for $4000 and up, you earn 3% per year.

Worked Exercise 15.4.1 ***Develop a function*** `bank-interest` *that computes the annual interest on a savings account with a specified balance.*

Solution: The function produces the amount of interest earned (a number), based on the balance in the account (a number). And it doesn't make sense, for purposes of this problem, to have a bank balance below 0, so we'll exclude that:

```
; bank-interest :  number(balance) -> number
; assumes balance is at least 0
```

The input and output types are both numbers. The output type doesn't break down in any obvious way into categories, but the input type does: there are four sub-ranges, 0-$500, $500-$1000, $1000-$4000, and $4000-up.

So how many examples will we need? There are four sub-ranges, so we'll need at least four examples; there are also three borderlines, at $500, $1000, and $4000, which need to be tested too. (If we wanted to be especially foolproof, we could test $0 and negative numbers too, but I'll skip that for this example.) So we need at least seven test cases: one inside each sub-range, and one at each borderline.

```
"Examples of bank-interest:"
(check-expect (bank-interest 200) 0)
(check-expect (bank-interest 500)  ?)
(check-expect (bank-interest 800) (* 800 .01)) ; or 8
(check-expect (bank-interest 1000)  ?)
(check-expect (bank-interest 2500) (* 2500 .02)) ; or 50
(check-expect (bank-interest 4000)  ?)
(check-expect (bank-interest 5000) (* 5000 .03)) ; or 150
```

The “right answers” inside each sub-range are obvious: $800 is in the 1% range, so we compute 1% of $800 and get $8, and so on. But the problem statement is rather vague about the borderlines.

For the $500 borderline, the problem actually said “if you have *less than* $500 in the account . . . ”, which suggests that *exactly* $500 should be treated as in the $500-$1000 category. Common sense agrees: if the bank paid no interest on a balance of $500, some customer with a balance of exactly $500 would complain, there’d be a TV news story about it, and the bank would come off looking petty and stingy. The other borderlines are less clear in the problem statement, but for consistency, and for the same public-relations reason as before, let’s assume that $1000 is treated as in the $1000-$4000 category, and $4000 is treated as in the $4000-up category. So now we can fill in the rest of the right answers:

```
"Examples of bank-interest:"
(check-expect (bank-interest 200) 0)
(check-expect (bank-interest 500) (* 500 .01)) ; or 5
(check-expect (bank-interest 800) (* 800 .01)) ; or 8
(check-expect (bank-interest 1000) (* 1000 .02)) ; or 20
(check-expect (bank-interest 2500) (* 2500 .02)) ; or 50
(check-expect (bank-interest 4000) (* 4000 .03)) ; or 120
(check-expect (bank-interest 5000) (* 5000 .03)) ; or 150
```

Note that each of these “right answers” is found by multiplying the balance by an appropriate interest rate.

The skeleton is straightforward:

```
(define (bank-interest balance)
  ...)
```

The inventory is easy too:

```
(define (bank-interest balance)
  ; balance       a number, in dollars
  ...)
```

Since the input type is one of four choices, we’ll probably need a four-clause `cond` . . . but wait! All the “right answers” match the same “pattern”: multiply the balance by the interest rate. So maybe this function should simply apply that formula, and leave the job of choosing the right interest rate to an *auxiliary function*, which might look something like this:

```
; bank-interest-rate :  number -> number
"Examples of bank-interest-rate:"
(check-expect (bank-interest-rate 200) 0)
(check-expect (bank-interest-rate 500) 0.01)
(check-expect (bank-interest-rate 800) 0.01)
(check-expect (bank-interest-rate 1000) 0.02)
(check-expect (bank-interest-rate 2500) 0.02)
(check-expect (bank-interest-rate 4000) 0.03)
(check-expect (bank-interest-rate 5000) 0.03)
```

Let’s pretend for a moment that we had this `bank-interest-rate` function. Then `bank-interest` would be quite simple: compute the interest rate, and multiply by the balance.

```
(define (bank-interest balance)
  ; balance      a number, in dollars
  (* balance (bank-interest-rate balance))
  )
```

We didn't need the four-clause conditional after all.

Of course, we're not done with the problem, since we haven't actually written the `bank-interest-rate` function. So let's write it. The next step in writing it is a skeleton:

```
(define (bank-interest-rate balance)
  ...)
```

The inventory contains a parameter name:

```
(define (bank-interest-rate balance)
  ; balance       a number, in dollars
  ...)
```

This time there's no obvious "pattern" that all the right answers fit; we actually need the conditional:

```
(define (bank-interest-rate balance)
  ; balance      a number in dollars
  (cond [...         ...]
        [...         ...]
        [...         ...]
        [...         ...]
  ))
```

Next, we need to fill in the questions and answers. The answers are easy:

```
(define (bank-interest-rate balance)
  ; balance      a number in dollars
  (cond [...       .00]
        [...       .01]
        [...       .02]
        [...       .03]
  ))
```

Next, under what conditions is the right answer 0? When the balance is under $500, *i.e.* `(< balance 500)`. The "$4000-and-up" case is similarly easy: `(>= balance 4000)`:

```
(define (bank-interest-rate balance)
  ; balance      a number in dollars
  (cond [(< balance 500)       .00]
        [...                   .01]
        [...                   .02]
        [(>= balance 4000)     .03]
  ))
```

The other two cases are a little trickier. The $500-$1000 bracket should include all the numbers that are at least 500, but strictly less than 1000, and the $1000-$4000 bracket should include all the numbers that are at least 1000, but strictly less than 4000. This calls for `and`:

```
(define (bank-interest-rate balance)
  ; balance      a number in dollars
  (cond [(< balance 500)              .00]
        [ (and(>= balance 500)
               (< balance 1000))      .01]
        [ (and(>= balance 1000)
               (< balance 4000))      .02]
        [(>= balance 4000)            .03]
  ))
```

We should now be able to test the `bank-interest-rate` function, and if it works, uncomment and test the `bank-interest` function. ▮

15.5 Ordering cases in a conditional

The program could be written somewhat shorter and simpler by taking advantage of the *order* in which DrRacket evaluates the cases of a conditional: it looks at the second question only if the first wasn't true, looks at the third only if the second wasn't true, *etc.* If the first question, `(< balance 500)`, isn't true, then we know that `(>= balance 500)` *must* be true, so we don't need to ask it. This simplifies the second question to `(< balance 1000)`. Likewise, if this isn't true, then `(>= balance 1000)` *must* be true, so we can simplify the third question to `(< balance 4000)`. If this in turn isn't true, then the fourth question `(>= balance 4000)` *must* be true, so we can simplify it to just `else`. The result is

```
(define (bank-interest-rate balance)
  ; balance      a number in dollars
  (cond [(< balance 500)  .00]
        [ (< balance 1000).01]
        [ (< balance 4000).02]
        [ else            .03]
  ))
```

This sort of simplification isn't always a good idea. In the original definition, the order of the cases in the `cond` doesn't matter: we could scramble them up, as in

```
(define (bank-interest-rate balance)
  ; balance      a number in dollars
  (cond [(and (>= balance 500)
              (< balance 1000))      .01]
        [(and (>= balance 1000)
              (< balance 4000))      .02]
        [(>= balance 4000)           .03]
        [(< balance 500)             .00]
  ))
```

and the function would work just as well, although it might be slightly harder to read. This is because *the cases don't overlap*: there's no possible value of `balance` for which two different questions would be true, so it doesn't matter in what order we ask the questions. By looking at any one of the cases, you can tell when it will happen.

However, in the "simplified" version above, the second question *includes* the first question, the third includes the second, and the fourth includes everything. As a result, if you scrambled the order of the cases in the "simplified" definition, you would get wrong answers. And to understand when any one of the cases will happen, you need to read not only that case but all the ones before it as well. This is no big deal for this program, which has only four cases, but imagine a program with dozens or hundreds of cases, added by several programmers over the course of weeks or months: to understand under what circumstances the 46th case will happen, you would have to read the first 45 as well!

I generally recommend writing the questions of a conditional so that *no two overlap*, and each one completely describes the situations in which it will happen. I have three exceptions to this rule:

- if one of the cases really is best described as "anything else", then I would use an `else` as the last question;
- if there are only two cases, I would use `else` as the second question rather than repeating the whole first question with a `not` around it (or, better, use another Racket construct named `if` instead of `cond` — look it up!); and
- if I'm extremely concerned about the speed of the program, I'll take full advantage of the order of the questions to simplify the later ones, in order to save a few microseconds.

Different teachers have different opinions on this: if your instructor prefers the version that takes advantage of the order of questions, go ahead and do it that way.

Exercise 15.5.1 *A carpet store needs a function to compute how much to charge its customers. Carpeting costs $5/yard, but if you buy 100 yards or more, there's a* 10% *discount on the whole order, and if you buy 500 yards or more, the discount becomes* 20% *on the whole order.*

Develop a function `carpet-price` *that takes in the number of yards of carpeting and returns its total price.*

Exercise 15.5.2 ***Develop a function*** *named* `digital-thermometer` *that takes in a temperature (in degrees Fahrenheit) and produces an image of the temperature as a number, colored either green (below* 99°*), yellow (at least* 99° *but less than* 101°*) or red (at least* 101°*).*

For example,

```
(digital-thermometer 98.3)
98.3
(digital-thermometer 99.5)
99.5
(digital-thermometer 102.7)
102.7
```

Hint: To convert a number to a string, use `number->string`. However, if you try it on a number like 98.6, you may get a fraction rather than a decimal. If you want it in decimal form, first make it inexact, using `exact->inexact`.

Hint: Use an auxiliary function to choose the color.

Exercise 15.5.3 ***Develop a function*** *named* `letter-grade` *that takes in a grade average on a 100-point scale and returns one of the strings* `"A"`, `"B"`, `"C"`, `"D"`, *or* `"F"`, *according to the rule*

- *An average of 90 or better is an A;*
- *An average of at least 80 but less than 90 is a B;*
- *An average of at least 70 but less than 80 is a C;*
- *An average of at least 60 but less than 70 is a D;*
- *An average of less than 60 is an F.*

Exercise 15.5.4 *Three candidates (Anne, Bob, and Charlie) are running for mayor of Racketville, which, by court order, has a new computerized voting system.* ***Develop a function*** *named* `who-won` *that takes in three numbers (the number of votes for Anne, the number of votes for Bob, and the number of votes for Charlie, respectively) and returns a string indicating who won – either* `"Anne"`, `"Bob"`, *or* `"Charlie"`. *If two or more candidates tied for first place, the function should return* `"tie"`.

Exercise 15.5.5 ***Develop a function*** *named* `4-votes->winner` *that takes in four strings representing votes, and returns the name of the winner (or* `"tie"` *if there was a tie). You may assume that the only candidates in the race are* `"Anne"`, `"Bob"`, *and* `"Charlie"` *(this makes it much easier!)*

Hint: This should be short and simple if you re-use previously-defined functions.

Exercise 15.5.6 *Some credit card companies give you a refund at the end of the year depending on how much you've used the card. Imagine a company that pays back*

- *0.25% of the first $500 you charge;*
- *0.50% of the next $1000 you charge (*i.e. *anything you charge between $500 and $1500);*
- *0.75% of the next $1000 you charge (*i.e. *between $1500 and $2500);*
- *1% of anything you charge over $2500.*

*For example, a customer who charged $400 would get back $1.00, which is 0.25% of $400. A customer who charged $1400 would get back 0.25% of the first $500 (making $1.25), plus 0.50% of the next $900 (*i.e. *$4.50), for a total refund of $5.75.*

Develop a function `card-refund` *to determine how much refund will be paid to a customer who has charged a specified amount on the card.*

15.6 Unnecessary conditionals

The above recipe may seem to contradict the way we wrote functions in Chapter 13: in most of the problems in that chapter, there were two or more categories of input, and two categories of output (`true` and `false`), yet we didn't need any conditionals. For example, recall Worked Exercise 13.3.1, whose definition was

```
(define (may-drive?  age)
  ; age      a number
  ; 16       a fixed number we're likely to need
  (>= age 16)
  )
```

In fact, we could have written this one using a conditional too:

```
(define (may-drive?  age)
  ; age       a number
  ; 16        a fixed number we're likely to need
  (cond [(>= age 16) true]
        [(< age 16) false]
  ))
```

and it would work perfectly well, but it's longer and more complicated than the previous version. Indeed, *every* function in Chapter 13 could have been written using a conditional, and would be longer and more complicated that way.

Rule of thumb: Functions that return Boolean can usually be written more simply without a conditional than with one.

Since `string=?`, `=`, and so on return Booleans, their results can be compared using `boolean=?`:

```
(define (reply greeting)
  ; greeting      a string
  (cond [(boolean=?  (string=?  greeting "good morning")  true)
         "I need coffee"]
        [ (boolean=?  (string=?  greeting "good afternoon")  true)
         "I need a nap"]
        [ (boolean=?  (string=?  greeting "good night")  true)
         "bedtime!"]
  ))
```

This works perfectly well, and passes all its test cases, but it's longer and more complicated than necessary. Likewise

```
(define (not-teenager?  age)
  ; age       a number
  (boolean=?  (and (>= age 13) (< age 20))  false)
  )
```

could be more briefly written as

```
(define (not-teenager?  age)
  ; age       a number
  (not (and (>= age 13) (< age 20)))
  )
```

or as

```
(define (not-teenager?  age)
  ; age       a number
  (or ( < age 13) ( >= age 20)))
  )
```

Rule of thumb: If you're using `boolean=?`, you're probably making things longer and more complicated than they need to be.

(The only time I can imagine needing `boolean=?` is when I have two Boolean expressions, and I don't care whether either one is true or false as long as they match. This isn't very common.)

For another example, recall Exercise 13.7.1, which we wrote as follows:

```
(define (18-to-25?  age)
  ; 18              a fixed number we'll need
  ; 25              another fixed number we'll need
  ; (>= age 18)     a Boolean
  ; (<= age 25)     a Boolean
  (and (>= age 18)
       (<= age 25))
  )
```

We *could* have written this using a conditional:

```
(define (18-to-25?  age)
  ; age             a number
  ; 18              a fixed number we'll need
  ; 25              another fixed number we'll need
  ; (>= age 18)     a Boolean
  ; (<= age 25)     a Boolean
  (cond [(>= age 18) (<= age 25)]
        [(< age 18) false])
  )
```

or even (putting one conditional inside another)

```
(define (18-to-25?  age)
  ; age             a number
  ; 18              a fixed number we'll need
  ; 25              another fixed number we'll need
  ; (>= age 18)     a Boolean
  ; (<= age 25)     a Boolean
  (cond [(>= age 18)(cond [(<= age 25) true]
                          [(> age 25) false])
        [(< age 18) false])
  )
```

but again, these definitions are longer and more complicated than the one that doesn't use a conditional.

15.7 Nested conditionals

Yes, you can put one conditional inside another. The "answer" part of each `cond`-clause is allowed to be any expression, and a `cond` is an expression, so why not? In the example above, it wasn't necessary, and even made the program longer and harder to understand. But there are situations in which nested conditionals are the most natural way to solve a problem.

Worked Exercise 15.7.1 *Imagine that you work for a company that sells clothes over the Internet: a Web page has a menu from which customers can choose which item of clothing, and which color, they're interested in. For simplicity, let's suppose there are only three items of clothing: pants, shirt, and shoes. The pants are available in black or navy; the shirt is available in white or pink; and the shoes are available in pink, burgundy, or navy. Your company photographer has given you pictures of all seven of* *these items, which you've copied and pasted into DrRacket under the variable names `black-pants`, `navy-pants`, `pink-shirt`, `white-shirt`, `pink-shoes`, `burgundy-shoes`, and `navy-shoes`.*

Develop a function *`show-clothing` that takes in two strings representing the item of clothing and the color, and returns a picture of the item of clothing. If the requested combination of item and color doesn't exist, it should return an appropriate error message.*

Solution: The **contract** is clearly

```
; show-clothing:  string(item) string(color) -> image
```

But wait: sometimes the function is supposed to return an error message instead! There are at least two ways we can handle this: we could either build an image of the error message (using the `text` function), or we could change the contract to return "image or string". For now, we'll opt for consistency and always return an image. (In Chapter 19, we'll see another way to handle this.)

There are seven legal **examples**, and to be really confident we should test them all:

```
(check-expect (show-clothing "pants" "black") black-pants)
(check-expect (show-clothing "pants" "navy") navy-pants)
(check-expect (show-clothing "shirt" "white") white-shirt)
(check-expect (show-clothing "shirt" "pink") pink-shirt)
(check-expect (show-clothing "shoes" "pink") pink-shoes)
(check-expect (show-clothing "shoes" "burgundy") burgundy-shoes)
(check-expect (show-clothing "shoes" "navy") navy-shoes)
```

In addition, we should have some illegal examples to test the handling of nonexistent items, unrecognized colors, etc.

```
(check-expect (show-clothing "hat" "black")
  (text "What's a hat?" 12 "red"))
(check-expect (show-clothing "pants" "burgundy")
  (text "We don't have pants in burgundy" 12 "red"))
```

The **skeleton** is easy:

```
(define (show-clothing item color)
  ...)
```

The **inventory** is fairly straightforward too:

```
(define (show-clothing item color)
  ; item          string
  ; color         string
  ; "pants"       string
  ; "shirt"       string
  ; "shoes"       string
  ; "black"       string
  ; "navy"        string
  ; "white"       string
  ; "pink"        string
  ; "burgundy"    string
  ...)
```

(Writing an inventory entry for every one of these literal strings is really boring, and if it's OK with your instructor, feel free to skip this.) We may also need some other expressions in order to build the error messages, but we'll come back to that later.

We know that `item` is supposed to be either `"pants"`, `"shirt"`, or `"shoes"` (or "anything else"), so the body of the function will need a conditional with four cases:

```
(define (show-clothing item color)
  ; item     string
  ; color    string
  (cond [(string=?  item "pants") ...]
        [(string=?  item "shirt") ...]
        [(string=?  item "shoes") ...]
        [else                     ...])
  )
```

If the item is in fact `"pants"`, the color can be either `"black"`, `"navy"`, or "anything else", which can be most naturally represented by *another* `cond` *inside the first one*:

```
(define (show-clothing item color)
  ; item     string
  ; color    string
  (cond [ (string=?  item "pants")
          (cond [(string=?  color "black")    ...]
                [(string=?  color "navy")     ...]
                [else ...])]
        [ (string=?  item "shirt") ...]
        [ (string=?  item "shoes") ...]
        [ else                     ...])
  )
```

We can do the same thing for `"shirt"` and `"shoes"`:

```
(define (show-clothing item color)
  ; item     string
  ; color    string
  (cond [ (string=?  item "pants")
          (cond [(string=?  color "black")    ...]
                [(string=?  color "navy")     ...]
                [else                         ...])]
        [(string=?  item "shirt")
          (cond [(string=?  color "pink")     ...]
                [(string=?  color "white")    ...]
                [else                         ...])]
        [(string=?  item "shoes")
          (cond [(string=?  color "pink")     ...]
                [(string=?  color "burgundy") ...]
                [(string=?  color "navy")     ...]
                [else                         ...])]
        [ else  ...])
  )
```

After figuring out all of these conditions, the legal answers are easy:

```
(define (show-clothing item color)
  ; item     string
  ; color    string
  (cond [ (string=?  item "pants")
          (cond [(string=?  color "black")    black-pants ]
                [(string=?  color "navy")     navy-pants ]
                [else                         ...])]
        [(string=?  item "shirt")
          (cond [(string=?  color "pink")     pink-shirt ]
                [(string=?  color "white")    white-shirt ]
                [else                         ...])]
        [(string=?  item "shoes")
          (cond [(string=?  color "pink")     pink-shoes ]
                [(string=?  color "burgundy") burgundy-shoes ]
                [(string=?  color "navy")     navy-shoes ]
                [else                         ...])]
        [ else  ...])
  )
```

All that remains is constructing the error messages, which we can do using `text` and `string-append`. But this function definition is getting pretty long already; since building these error messages really is a completely different sort of job from what we've been doing so far, let's have it done by auxiliary functions. Here's one to handle unrecognized items:

```
; bad-item :  string(item) -> image
(define (bad-item item)
  ; item                                    string
  ; (string-append "What's a " item "?")    string
  ; 12                                      number (font size)
  ; "red"                                   string (text color)
  (text (string-append "What's a " item "?") 12 "red")
  )
"Examples of bad-item:"
(check-expect (bad-item "hat") (text) "What's a hat?" 12 "red")
(check-expect (bad-item "belt") (text) "What's a belt?" 12 "red")
```

The analogous function for unrecognized colors is left for you to do; see Exercise 15.7.2 below.

The final definition of `show-clothing` looks like

```
(define (show-clothing item color)
  ; item     string
  ; color    string
  (cond [ (string=?  item "pants")
          (cond [(string=?  color "black")    black-pants ]
                [(string=?  color "navy")     navy-pants ]
                [else             (bad-color item color) ])]
        [(string=?  item "shirt")
          (cond [(string=?  color "pink")     pink-shirt ]
                [(string=?  color "white")    white-shirt ]
                [else             (bad-color item color) ])]
        [(string=?  item "shoes")
          (cond [(string=?  color "pink")     pink-shoes ]
                [(string=?  color "burgundy") burgundy-shoes ]
                [(string=?  color "navy")     navy-shoes ]
                [else             (bad-color item color) ])]
        [ else            (bad-item item) ])
  )
```

■

Exercise 15.7.2 ***Write*** *the* `bad-color` *function needed in the above example.*

Exercise 15.7.3 ***Develop a function*** `make-shape` *that takes in three strings: a* shape *(either* `"circle"` *or* `"triangle"`*), a* size *(either* `"small"`*,* `"medium"`*, or* `"large"`*), and a* color *(any color that DrRacket recognizes), and produces an appropriate image.*
Note: *Make sure that a "medium circle" and a "medium triangle" are about the same size.*

15.8 Decisions among data types

Most the functions we've written so far have expected one specific type of input, and produced one specific type of output — number, string, image, boolean, *etc.* But sometimes a function needs to be able to handle input of *several different types.* We'll see more useful applications of this ability in the next few chapters, but for now simply imagine a function that takes in a number, which a confused user mistakenly puts in quotation marks. The user would get an unfriendly error message like
+: expects type <number> as 1st argument, given "4"; other arguments were: 3
It would be *really* nice if our program could figure out that by `"4"` the user probably meant the number 4. Even if we didn't want to go that far, or if there *were* nothing reasonable our program could do with the incorrect input, it would be nice if our program could produce a friendlier message like
This program expects a number, like 3. You typed a quoted string, "4".
To do this, our program would need to recognize that the input was a string rather than a number as expected.

The ability to make decisions based on the types of our inputs (which computer scientists call *polymorphism*) will be useful in a number of ways. In Chapter 19 we'll see how to produce friendly error messages like the above. But first, how do we detect different types?

Recall from Chapter 13 that Racket has built-in *type predicates*, functions that take in *any* type of input (including types you haven't seen yet) and return either `true` or `false` depending on whether the input is of that particular type. For example,

```
; number?  :  anything -> boolean
; image?  :  anything -> boolean
; string?  :  anything -> boolean
; boolean?  :  anything -> boolean
; integer?  :  anything -> boolean
...
```

With these functions in hand, and the decision-making ability provided by `cond`, one can easily write functions that operate differently on different types of inputs.

Worked Exercise 15.8.1 ***Develop a function*** *`classify` that tells what type its input is, by returning one of the strings `"image"`, `"string"`, `"number"`, or `"other"` as appropriate.*

(The only "other" type you've seen so far is boolean*, but we'll see more in the next few chapters.)*

Solution: The contract is

```
; classify :  anything -> string
```

In the data analysis step, we observe that not just any string can be produced; the result is always one of four choices, so a more informative contract would be

```
; classify :  anything -> string
; ("image", "string", "number", or "other")
```

For that matter, the input is indeed "anything", but we're interested in which of four categories it falls into (image, string, number, or anything else), so we could even write

```
; classify :  anything (image, string, number, or anything else) ->
; string ("image", "string", "number", or "other")
```

(Again, we could write an inventory template for this, but we don't expect to be writing lots of functions that divide the world up in this particular way, so we won't bother.)

Since the input and output data types each fall into four categories, there should be at least four examples:

```
"Examples of classify:"
(check-expect (classify (circle 5 "solid" "green")) "image")
(check-expect (classify "hello there") "string")
(check-expect (classify 74) "number")
(check-expect (classify true) "other")
```

The skeleton is simple:

```
(define (classify thing)
  ...)
```

For the inventory, we normally start by listing parameters and labeling each one with its type. In this case, we don't *know* what data type thing is.

```
(define (classify thing)
  ; thing       anything
  ...)
```

However, we know that there are four categories, both of input and of output, so we can reasonably guess that the body of the function will involve a cond with four clauses:

```
(define (classify thing)
  ; thing      anything
  (cond [...               ...]
        [...               ...]
        [...               ...]
        [...               ...]
        )
  )
```

The next step in writing a function involving conditionals is normally to fill in either all the questions or all the answers, whichever is easier. We know that the answers are "image", "string", and so on, so we can fill them in easily:

```
(define (classify thing)
  ; thing      anything
  (cond [...    "image"]
        [...    "string" ]
        [...    "number" ]
        [...    "other" ]
        )
  )
```

We still need to fill in the questions. The only expression we have available to work with is the parameter thing, so we must ask questions about it. Under what circumstances is the right answer "image"? Obviously, when thing is an image. Conveniently, the image? function tests this. We can test the other types similarly.

```
(define (classify thing)
  ; thing      anything
  (cond [  (image?  thing)      "image"]
        [  (string?  thing)     "string" ]
        [  (number?  thing)     "number" ]
        [  else                 "other" ]
        )
  )
```

Note the else in the last clause, which catches any input that hasn't matched any of the previous criteria. ∎

Exercise 15.8.2 ***Define a function*** *named* size *that takes in a number, a string, or an image, and returns "how big it is". For a number, this means the absolute value of* *the number. For a string, it means the length of the string. For an image, it means the number of pixels,* i.e. *the width times the height.*

Exercise 15.8.3 ***Define a function*** *named* `big?` *that takes in either a number or a string, and tells whether the argument is "big". What does "big" mean? For numbers, let's say it means at least 1000, and for strings, let's say it's any string of length 10 or more.*

Hint: The function needs to handle two kinds of input, and for each kind of input there are two possible answers and a "borderline", so you'll need six test cases.

Exercise 15.8.4 ***Develop a function*** *named* `same?` *that takes in two arguments, each of which is either a number or a string, and tells whether they're "the same". If one is a number and the other a string, they're obviously* not *"the same"; if both are numbers, you can compare them using* `=`*; and if both are strings, you can compare them using* `string=?`*.*

There's actually a built-in function `equal?` that does this and more: it compares *any* two objects, no matter what types, to tell whether they're the same. You may not use it in writing Exercise 15.8.4. You may use it in the rest of the book (except where I specifically tell you not to), but in most cases it's a better idea to use something more specific like `=`, `string=?`, `key=?`, *etc.* because if you accidentally violate a contract and call one of these on the wrong type of input, you'll get an error message immediately rather than the program going on as if everything were OK. Eventually it would probably produce wrong answers, which are *much harder to track down and fix than error messages.*

Exercise 15.8.5 ***Develop a function*** *named* `smart-add` *that takes two parameters and adds them. The trick is that the parameters can be* either *numbers (like* `17`*) or strings of digits (like* `"17"`*); your function has to be able to handle both.*

Hint: There are two parameters, each of which could be either of two types, so you'll need at least four examples.

15.9 Review of important words and concepts

Much of the power of computers comes from their ability to *make decisions* on their own, using *conditionals*. A Racket conditional consists of the word `cond` and a sequence of *question/answer* clauses: it evaluates each *question* in turn is evaluated, and as soon as one comes out `true`, it returns the value of the corresponding *answer*. If none of the *question*s evaluates to `true`, you get an error message. If you want to avoid this error message, you can add another *question/answer* clause at the end, with the *question* being simply the word `else`, which guarantees that if none of the previous *answer*s was true, this one will be. This is often used for error-handling.

In order to design functions that make decisions, we add some details to the *skeleton* and *body* steps of the design recipe: we write the skeleton of a `cond`, with the right number of cases, and then fill in the questions and the answers (I recommend either all the questions, then all the answers, or *vice versa*, depending on which is easier).

Functions that return `boolean` can usually be written shorter and simpler *without* a conditional than *with* one. Almost any time you use the `boolean=?` function, you could have accomplished the same thing more simply without it.

If a function has two or more inputs that *each* come in multiple categories, or if a type has sub-categories of a specific category, often the most natural way to write the function is with *nested conditionals*: the *answer* part of a conditional is itself another whole conditional. However, if you find yourself doing this, there may be a shorter and simpler way that *doesn't* require nested conditionals.

People normally write Racket functions to take in particular types of arguments, but you can also design a function to be more flexible, checking for itself what types its arguments were and handling them appropriately. Every built-in Racket type has a *discriminator* function, also known as a *type predicate*, whose name is the name of the type, with a question-mark at the end (*e.g.* `number?`, `string?`, `image?`) which takes in anything and tells whether or not it is of that type. You can use these discriminator functions in a conditional to write *polymorphic functions*, functions that work differently on different types of inputs.

15.10 Reference: Built-in functions for making decisions

In this chapter, we've introduced one new function: `equal?`, which takes in two parameters of *any* types and tells whether they're the same. In general, it's a good idea to use something more specific, like `string=?` or `=`, because you'll catch mistakes faster that way.

This chapter also introduced the Syntax Rule 6, with the *reserved words* `cond` and `else`. (A *reserved word* is a word whose meaning is built into the Racket language and can't be changed, but which isn't called like a function. In particular, the reserved word `else` can *only* appear as the *last question* in a `cond`.)

For the common case of a `cond` with only two cases, the second of which is `else`, there's a shorter form:

```
(if question answer-if-true answer-if-false)
```

Look it up in the Help Desk.