# Chapter 29

# Input, output, and sequence

For this chapter, **switch languages** in DrRacket to "Advanced Student Language".

In the real world, we don't usually give a computer *all* the information it needs, all at once, and then ask it to go off and produce an answer. More often, we start a program and engage in a *dialogue* with it. For example, a word-processing program shows us the current state of the document; we tell it to add, delete, or move some more words, the program shows us the result, we request some more changes, and so on. We've seen some ways to *interact* with a computer program through animations and event handlers, and in this chapter we'll see another (more old-fashioned, but still useful) approach to interaction.

In an animation, the program typically goes on running all the time, but responds whenever we move or click the mouse, type a key, *etc.* In some problems, however, the program *can't go on* until it gets some information from the user. A familiar (if unpleasant) example are the dialogue boxes that pop up on your screen saying basically "something went wrong; should I try again, or give up?"

Here's another example. Suppose you were a mathematician who wanted a list of prime numbers. Of course, there are infinitely many prime numbers, so if you wrote a program to produce a list of *all* of them, it would never finish and you would never see any results at all. A more useful approach would be for the program to show you a prime number, then another, then another, and so on until you told it to stop.

Even a program that will eventually stop may need to show you some information along the way, then do some more computation, show you some more information, and so on. Of course, you can do this with an animation, but that seems like overkill for information that's basically textual.

But first, we'll introduce another data type, which has actually been available to us all along, but we haven't needed it until this chapter.

## 29.1 The symbol data type

Racket has a built-in type called *symbol* which behaves, in some ways, like *string*. The most obvious difference is the spelling rules: a symbol literal starts with an apostrophe and does *not* end with an apostrophe, but rather at the next space, parenthesis, *etc.* As a result, a symbol literal *cannot contain* spaces, parentheses, and certain other punctuation marks. Indeed, the spelling rules for symbol literals are basically the same as those for variable names and function names, except for the apostrophe at the beginning. (You've actually seen these before: the first argument to `error` is normally the name of the function that found the problem, as a symbol.)

Like image literals, string literals, number literals, and boolean literals, a symbol literal evaluates to itself; it doesn't "stand for" anything else:

```
(check-expect 'blah 'blah)
(check-expect 'this-is-a-long-name 'this-is-a-long-name)
```

The most common operation on symbols is to test whether two of them are equal, using the `symbol=?` function, which works analogously to the `string=?` function:

```
(check-expect (symbol=?  'blah 'snark) false)
(check-expect (symbol=?  'blah 'blah) true)
(define author 'Bloch)
(check-expect (symbol=?  author 'Hemingway) false)
(check-expect (symbol=?  author 'Bloch) true)
```

And as with all the other types we've seen, there's a built-in function to test whether something *is* a symbol, named (not surprisingly) `symbol?`. It works exactly as you would expect, by analogy with `number?`, `image?`, `posn?`, *etc.*

Unlike strings, symbols are not thought of as made up of individual characters strung together. A symbol is *atomic*, in the original sense of that word as meaning "not made up of smaller parts". So there is no `symbol-length` or `symbol-append` function analogous to `string-length` and `string-append`. And symbols have no *ordering*, so there's nothing analogous to `string<?` and its friends: two symbols are either the same or different, and that's all you can say about them.

In exchange for these restrictions, computations on symbols are typically a little faster than those on strings. However, this by itself wouldn't be enough reason to introduce them in this course. I'm mentioning them here because the built-in input and output functions treat symbols a little differently from strings.

**Exercise 29.1.1** ***Modify*** *the* `choose-picture` *function of Exercise 15.3.1 so it takes in a* symbol *rather than a* string *as its parameter,* e.g. `'baseball`, `'basketball`, `'Monopoly`.

Incidentally, the image functions that take in a color name (`circle`, `rectangle`, `triangle`, *etc.*) also accept the corresponding symbols: `'red`, `'orange`, `'purple`, `'black`, *etc.*.

**Exercise 29.1.2** ***Develop*** *a function named* `random-color` *that takes in a "dummy" argument and ignores it, but returns one of the symbols* `'red`, `'orange`, `'yellow`, `'green`, `'blue`, `'purple` *chosen at random.*

**Exercise 29.1.3** ***Develop*** *a function named* `different-color` *that takes in a color name as a symbol, and returns a different color name, also as a symbol. Which input color goes with which output color is up to you, as long as the result is always different from the input.*

**Hint:** DrRacket knows a *lot* of color names. You could try to write a `cond` with dozens or hundreds of cases, but that would be horrible, and it would no longer work if somebody added one more color name to Racket. Instead, think about how you can satisfy the requirements of the problem without knowing all the possible colors.

**Exercise 29.1.4** ***Modify*** *exercise 17.1.1 so it uses symbols rather than strings as the model.*

## 29.2 Console output

Racket has a built-in function named `display` that does simple textual output.

```
; display :  object -> nothing, but displays the object.
```

**Practice Exercise 29.2.1** *Try typing each of the following lines in the Interactions pane:*

```
(display 5)
(display "this is a string")
(display 'this-is-a-symbol)
(display (make-posn 3 4))
(display (list "a" "b" "c"))
(display (triangle 20 "solid" "blue"))
```

SIDEBAR:

Another built-in function, `write`, acts like `display`, but shows strings with double-quotes around them, so you can easily tell the difference between a string and a symbol.

So far this doesn't look very exciting. If anything, it's *less* useful than what we've been doing up until now, because you can't use the result of `display` in another expression:

```
(+ 1 (display 2))
```

produces an error message because `display` doesn't return anything.

The `display` function becomes much more useful if we *build* something for it to display from smaller pieces. For example,

**Worked Exercise 29.2.2** ***Develop a function*** `display-with-label` *that takes in a string (the "label") and an object, and prints the string followed by the object.*

**Solution:** The contract is

```
; display-with-label :  string object -> nothing
; Prints the string and the object.
```

A more-or-less realistic test case is

```
(define my-age 46)
(display-with-label "Age:  " my-age)
"should print" "Age:  46"
```

Make up some more test cases.

The skeleton and inventory are straightforward:

```
(define (display-with-label label thing)
  ; label     a string
  ; thing     an object of some kind
  ...
  )
```

We could easily display just the label, or just the thing (since `display` takes in *any* data type), but how can we combine them?

Recall the `format` function (first mentioned in Chapter 19), which is designed to build complex strings from a "template" with values filled in in various places, returning a string. Conveniently enough, each of the "values to fill in" can be of almost any data type. So we could try

```
(define (display-with-label label thing)
  ; label     a string
  ; thing     an object of some kind
  (display (format "~s~s" label thing))
  )
```

Try this on the example above, and it prints

```
"Age:  "46
```

Not bad, but the quotation marks are annoying. Fortunately, `format` has different "formatting codes": ~s shows strings *with* their quotation marks, and ~a shows strings *without* their quotation marks. (The main reason to use ~s is to allow the user to tell the difference between strings and symbols.) So

```
(define (display-with-label label thing)
  ; label     a string
  ; thing     an object of some kind
  (display (format  "~a~a" label thing))
  )
```

produces a better result:

```
Age:  46
```

This combination of `format` and `display` is common enough that Racket has a built-in function to do it: the `printf` function acts just like calling `display` on the result of `format`, so we could write the definition more briefly as

```
(define (display-with-label label thing)
  ; label     a string
  ; thing     an object of some kind
  (printf  "~a~a" label thing)
  )
```

█

SIDEBAR:

The `display` and `write` functions can indeed take in just about any data type, including images. However, `format`'s job is to build a string, and strings cannot contain images, so if you try `format` on an image, you'll get weird results.

**Testing functions that use console output**
How can we write test cases for a function like `display-with-label` that uses `display` or `write`? `check-expect` looks at the result *returned* by a function, but `display` and `write` don't return anything!

In Exercise 29.2.2, we used the "should be" approach. But as we already know, automated testing using `check-expect` is *much* more convenient. If only we could find out what the function printed, and compare it with a known right answer…

As it happens, there's a built-in function named `with-output-to-string` to do this: it evaluates an expression of your choice (presumably containing `display` or `write`), but *captures* whatever that expression tries to write, and puts it into a string instead; you can then check whether this string is what you expected with `check-expect`.

Its contract may seem a little strange at first:

```
; with-output-to-string :  (nothing -> anything) -> string
```

That is, you give it a *function of no parameters*; it calls this function, throws away any result it produces, and returns a string constructed from whatever the function `display`ed.

**Worked Exercise 29.2.3** *Write automated test cases for Exercise 29.2.2.*

**Solution:** We need a function of no arguments to pass into `with-output-to-string`. We could write one for each test case:

```
(define age 46)
(define last-name "Bloch")
(define (test-case-1) (display-with-label "Age:  " age))
(define (test-case-2) (display-with-label "Name:  " last-name))
(check-expect (with-output-to-string test-case-1) "Age:  46")
(check-expect (with-output-to-string test-case-2) "Name:  Bloch")
```

This seems silly. We can define the functions more simply using `lambda`:

```
(define age 46)
(define last-name "Bloch")
(check-expect
  (with-output-to-string
    (lambda () (display-with-label "Age:" age)))
  "Age:  46")
(check-expect
  (with-output-to-string
    (lambda () (display-with-label "Name:  " last-name)))
  "Name:  Bloch")
```

█

Functions of no arguments can be thought of as a way to pass around expressions without evaluating them until later. They come up often enough in Racket that they have a special name: they're called **thunks**.

**Exercise 29.2.4** *Recall the struct definition*

```
; An employee has a string (name) and two numbers (id and salary).
(define-struct employee [name id salary])
```

***Develop a function*** *`print-employee` that takes in an employee and returns nothing, but prints out the information about the employee, nicely formatted. For example,*

```
(print-employee (make-employee "Joe" 17 54000))
"should print" "Joe, employee #17, earns $54000/year"
```

## 29.3 Sequential programming

When you evaluate an expression like `(+ (* 3 4) (* 5 6))`, Racket needs to compute both $3 \cdot 4$ and $5 \cdot 6$, then add them. It doesn't really matter which of the two multiplications it does *first*, as long as it knows both answers before it tries to add them.

But `display` and `write` don't produce "answers", they produce *side effects*, and it matters very much which of two `display` expressions happens first. Racket has a syntax rule to specify doing things in a particular order:

**Syntax Rule 10** *(`begin` `expr1 expr2 ...exprn`) is an expression. To evaluate it, DrRacket evaluates each of the* `expr`*s in order, throwing away any results they produce except the last one, which it returns.*

For example, type the following into the Interactions pane:

```
(define result
  (begin
    (display (+ 12 5))
    (* 5 3)))
result
```

It prints out the number 17, but gives `result` the value 15.

Now let's try that in the opposite order:

```
(define other-result
  (begin
    (* 5 3))
    (display (+ 12 5)))
other-result
```

This still prints out the number 17, but `other-result` has *no value at all* (because `display` doesn't return anything). The result of `(* 5 3)` has been thrown away completely.

**Worked Exercise 29.3.1** ***Rewrite*** *the function `display-with-label` to use `begin` instead of `format`.*

**Solution:** The contract, test cases, skeleton, and inventory are exactly as before.

In the function body, clearly, we need to `display` both the label and the object:

```
(define (display-with-label label thing)
  ...
  (display label)
  ...
  (display thing)
  ...)
```

More specifically, we want to display the label *first, followed* by the thing. To do this, we'll use `begin`:

```
(define (display-with-label label thing)
  (begin
    (display label)
    (display thing)
  ))
```

■

**Controlling lines**

Sometimes you need to specify that the output should be on more than one line. There are several ways to do this:

- Use the built-in function

  ```
  ; newline :  nothing -> nothing
  ; advances the display to the next line
  ```

  in between `display`s in a `begin`, *e.g.*

  ```
  > (begin (display "abc")
           (newline)
           (display "def"))
  abc
  def
  ```

- Hit ENTER in the middle of a quoted string, *e.g.*

  ```
  > (display "abc
  def")
  abc
  def
  ```

- Some languages don't allow you to do this, so they use a third approach instead: you can put the special character \n in the middle of a quoted string to indicate a "new line":

  ```
  > (display "abc\ndef")
  abc
  def
  ```

Notice that all three produced the exact same output; which one you use is largely a matter of personal taste.

**Worked Exercise 29.3.2** ***Modify*** *the* `print-employee` *function to display its information on three separate lines,* e.g.

```
Joe
Employee #17
$54000/year
```

**Solution:** The contract, skeleton, and inventory are unchanged, but we'll need to modify the test cases. Here are two versions; either one should work.

```
(check-expect
  (with-output-to-string
    (lambda () (print-employee (make-employee "Joe" 17 54000))))
  "Joe\nEmployee #17\n$54000/year")

(check-expect
  (with-output-to-string
    (lambda () (print-employee (make-employee "Joe" 17 54000))))
  "Joe
Employee #17
$54000/year")
```

Next, we'll need to modify the function body. This can be done in any of several ways:

```
(begin
  (display (employee-name emp))
  (newline)
  (display "Employee #")
  (display (employee-id emp))
  (newline)
  (display "$")
  (display (employee-salary emp))
  (display "/year"))
```

```
(begin
  (display (employee-name emp))
  (display "
Employee #")
  (display (employee-id emp))
  (display "
$")
  (display (employee-salary emp))
  (display "/year"))
```

```
(begin
  (display (employee-name emp))
  (display "\nEmployee #")
  (display (employee-id emp))
  (display "\n$")
  (display (employee-salary emp))
  (display "/year"))
```

```
(printf "~a
Employee #~a
~a/year"
  (employee-name emp)
  (employee-id emp)
  (employee-salary emp))
```

```
(printf "~a\nEmployee #~a\n~a/year"
  (employee-name emp)
  (employee-id emp)
  (employee-salary emp))
```

Any of these five solutions should work; which one you use is largely a matter of personal taste. ▌

**Exercise 29.3.3** ***Develop a function*** `try` *that takes in a string (function-name), a function of one argument, and a value for that argument. It should print that it is "about to call" the function name on the specified argument, then call the function, then print that it has "returned from " the function and what the result was, and finally return that result. For example,*

```
(try "cube" cube 5)
```

*should print out*

```
About to call (cube 5)
Returned from (cube 5) with result 125
```

*and finally return the result 125. For another example,*

```
(try "display" display "blah")
```

*should print out*

```
About to call (display "blah")
blah
Returned from (display "blah") with result
```

**Exercise 29.3.4** ***Develop a function*** `count-down-display` *that takes in a whole number. It doesn't return anything, but displays the numbers from that number down to 0, one on each line, with "blastoff!" in place of the number 0. For example,*

```
> (count-down-display 5)
5
4
3
2
1
blastoff!
```

**Exercise 29.3.5** ***Modify*** *exercise 21.7.10 by adding two buttons, labelled "save" and "load". If the user clicks the "save" button, the current image (not including the color palette, "save" and "load" buttons) will be stored in the file "current.png" with* `save-image`*; the image on the screen shouldn't change. If the user clicks the "load" button, the image on the screen should be replaced with the contents of "current.png" (although the color palette, "save" and "load" buttons should be unaffected).*

## 29.4 Console input

### 29.4.1 The `read` function

The opposite of `display`, in a sense, is the built-in function `read`.

```
; read :  nothing -> object
; waits for the user to type an expression, and returns it
```

Try typing `(read)` into the Interactions pane. You should see a box with a typing cursor in it. Type a number like `17` into the box, and hit ENTER; the `read` function will return 17.

Type `(read)` again, and type a quoted string like `"hello there"` (complete with the quotation marks) into the box; `read` will return that string.

What happens when you type `(read)` and type a couple of words like `this is a test` (*without* quotation marks) into the box?

What happens when you type `(read)` and type a parenthesized expression like `(+ 3 4)` into the box? What about `(+ 3 (* 4 5))`?

What happens when you type `(read)` and type a Boolean literal (`true` or `false`) into the box?

What happens when you type `(read)` and type a comment like `; this is a comment` into the box?

What happens when you type `(read)` and type a symbol like `'snark` (with its apostrophe) into the box?

SIDEBAR:

This last example should come out looking similar to a function call, but with the "function" being named `quote`. In fact, there *is* a `quote` function; play with it to find out what it does, then look it up in the Help Desk.

There's also a function `read-line` which reads a whole line of input as a single string, even if it has spaces, parentheses, *etc.* inside it. **Try it.**

### 29.4.2 Testing functions that use console input

It's harder to write test cases for a function that involves input: some information may be provided as arguments, but some will be provided as input. So we could write, essentially, an actor's script: I'll say this, the program should say that, I'll say something else, the program should reply with such-and-such.

But that's even more of a pain than using "should be". So there's a built-in function

```
; with-input-from-string :  string (nothing -> anything) -> anything
```

which calls the specified thunk, but any time it tries to read from the console, it actually gets input from the string instead. `with-input-from-string` returns whatever the thunk returns.

**Worked Exercise 29.4.1** ***Develop a function*** *`ask` that takes in a string, prints it, waits for input, and returns that input.*

**Solution:** Contract:

```
; ask :  string -> object
; prints the string, waits for input, and returns it
```

Test cases, written as an "actor's script":

```
(ask "What is your name?)
; It prints "What is your name?".
; I type "Stephen" (without the quotation marks).
; It returns the symbol 'Stephen.
(define age (ask "How old are you?"))
; It prints "How old are you?".
; I type 46.
; It defines age to be 46.
```

Test cases, written using `check-expect` and `with-input-from-string`:

```
(check-expect
  (with-input-from-string "Stephen"
    (lambda () (ask "What is your name?")))
  'Stephen)
(define age
  (with-input-from-string "46"
    (lambda () (ask "How old are you?"))))
(check-expect age 46)
```

Definition:

```
(define (ask question)
  (begin
    (display question)
    (read)))
```

Remember that `begin` always returns the value of its *last* expression, which in this case is whatever `read` returns, which is whatever the user typed. ∎

**Note:** Even after your function has passed all its automated tests, it's probably a good idea to try a few tests in the Interactions pane, to make sure your program interacts with the user the way you want it to.

**Worked Exercise 29.4.2** ***Develop a function*** *`greet-by-name` that takes no parameters, asks for your name, then displays "Hello,* your-name-here*!".*

**Solution:** Since this function takes keyboard input and *also* prints to the screen, we'll need *both* `with-input-from-string` and `with-output-to-string`:

```
(check-expect
  (with-output-to-string
    (lambda ()
      (with-input-from-string "Steve" greet-by-name)))
  "What's your name?Hello, Steve!")
```

This works, but it's a bit of a pain. The `with-io-strings` function combines the jobs of `with-input-from-string` and `with-output-to-string`; its contract is

```
; with-io-strings:  string thunk -> string
```

For example, the above test case could be rewritten as

```
(check-expect (with-io-strings "Steve" greet-by-name)
              "What's your name?Hello, Steve!")
```

I leave the rest of the definition as an exercise for the reader (and it's in the Help Desk documentation for `with-io-strings`). █

### 29.4.3 Exercises

**Exercise 29.4.3** ***Develop a function*** *`repeat-input` that takes in a string (the "question"). It prints the question, waits for input, then prints the result twice, on separate lines, and returns nothing.*

**Hint:** You need to `read` only once, but use the result twice, so you'll need either a helper function or a `local`.

**Exercise 29.4.4** ***Develop a function*** *`ask-posn` that takes in no arguments, asks the user for an x coordinate and a y coordinate, and creates and returns a `posn` with those coordinates.*

**Hint:** This one does *not* require `local`.

**Exercise 29.4.5** ***Modify*** *exercise 29.3.5 so that when the user clicks the "load" or "save" button, the program asks you for a filename (using `ask` or something similar), then loads or saves that file rather than always using "current.png". You've now written a simple graphics editor.*

*An optional nice feature would be to have "load", "save", "load from", and "save to" buttons: "load from" and "save to" should behave as above, but "load" and "save" will operate on the last filename you used.*

**Hint:** You may want to use `read-line` rather than `read`, to avoid worrying about whether the input is treated as a symbol or a string, and to allow filenames to contain spaces.

## 29.5 Input streams

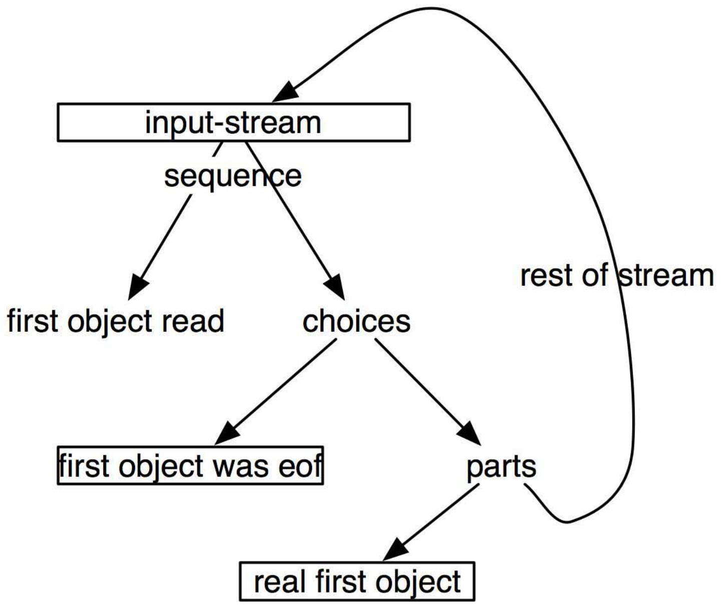
Many programs need to operate on a variable amount of information. We've seen how to do this with lists, but what if the information isn't provided in the form of a completed list?

Throughout this book, we've designed functions to correspond to the data type they take in. To handle a sequence of data coming from input, we'll need to describe it as a data type — an "input stream".

We've been using `read` to read information from the keyboard. But computer programs often read from files too: word processing documents, spreadsheets, image and music files, *etc.* Such files "feel" like lists: they're either empty or they have a sequence of finitely many objects. As you know, the end of a list is indicated by a special object named `empty`, which is recognized by the `empty?` function; similarly, the end of a file is indicated by a special object named `eof`, which is recognized by the `eof-object?` function.

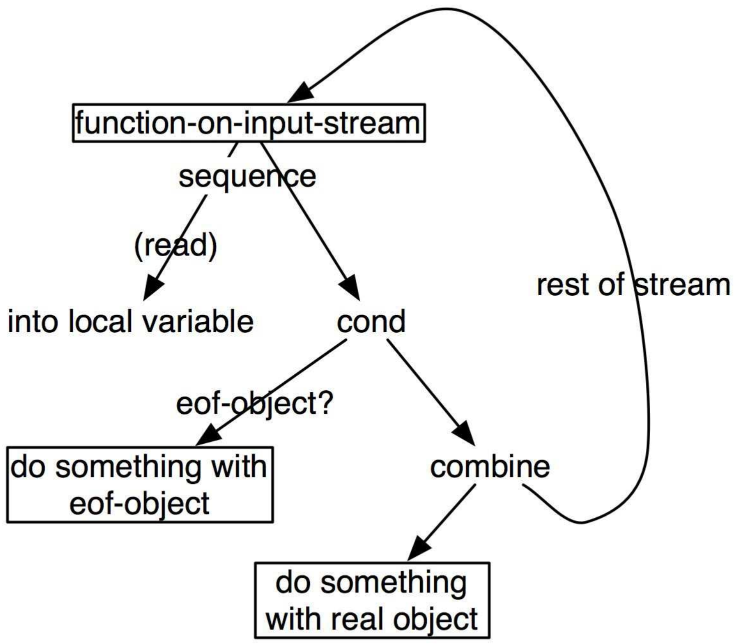
While practicing with the `read` function, you may have noticed an `eof` button next to the input box. Clicking this button causes `read` to return an `eof` object, as though at the end of a file. (You'll also get an `eof` object if you read past the end of the string in `with-input-from-string`.)

The `read` function, in essence, returns the first object, or `eof`, from an input stream, and has the side effect of "advancing" the input stream so that the next call to `read` will return the next object in the stream. As a result, if we want to use the result more than once, we'll need to store it in a `local` variable.



We can now write a function template for functions that operate on input streams.

```
#|
(define (function-on-input-stream)
  (local [(define obj (read))]
    (cond [(eof-object?  obj) ...]
          [else
           ; obj                          non-eof object
           ; (function-on-input-stream) whatever this returns
           ...
    )))
|#
```



**Worked Exercise 29.5.1** ***Develop a function*** *`add-from-input` which asks the user for a sequence of numbers, one at a time, terminated by `eof`, and returns their sum.*

**Solution:** The function takes no parameters, but reads from input and returns a number.

```
; add-from-input :  nothing -> number
; (reads zero or more numbers from input, terminated by eof)
```

We'll need several test cases. As with lists, we'll need an empty test case, and a one-element test case, and a more complicated test case. The function assumes that the inputs *are* numbers, so we don't need test cases with non-numbers.

First, I'll write them in the form of an "actor's script":

```
(add-from-input)
; It asks for a number.
; I hit the EOF button.
; It should return 0.
(add-from-input)
; It asks for a number.
; I type 7.
; It asks for a number.
; I hit the EOF.
; It should return 7.
(add-from-input)
; It asks for a number; I type 7.
; It asks for a number; I type -3.
; It asks for a number; I type 6.
; It asks for a number; I hit EOF.
; It should return 10.
```

But it's easier to run the test cases if we automate them with `check-expect`. Conveniently, `add-from-input` is *already* a thunk, so we don't need to wrap it up in a `lambda`:

```
(check-expect (with-input-from-string "" add-from-input)
              0)
(check-expect (with-input-from-string "7" add-from-input)
              7)
(check-expect (with-input-from-string "7 -3 6" add-from-input)
              10)
```

The template gives us a good deal of the definition:

```
(define ( add-from-input)
  (local [(define obj (read))]
    (cond [(eof-object?  obj)  0]
          [else
           ; obj                        number
           ; (add-from-input)           number
           ...
    )))
```

But this doesn't actually "ask" for numbers; to do this, let's replace the call to `read` with a call to `ask`. The only other thing left to do is add the two numbers:

```
(define (add-from-input)
  (local [(define obj  (ask "Next number?"))]
    (cond [(eof-object?  obj) 0]
          [else
           ; obj                        number
           ; (add-from-input)           number
           (+ obj (add-from-input))
    )))
```

If we want to make the function more "idiotproof", we can change the contract to read a sequence of *objects* rather than *numbers*, and have the function signal an error in that case.

```
(check-error (with-input-from-string "7 eight 9" add-from-input)
             "add-from-input:  That's not a number!")
...
(define (add-from-input)
    (local [(define obj  (ask "Next number?"))]
       (cond [(eof-object?  obj) 0]
             [(number?  obj)
              (+ obj (add-from-input))
             [else
              (error 'add-from-input "That's not a number!")
       )))
```

▮

**Exercise 29.5.2** ***Develop a function*** `read-objects` *which asks the user for a sequence of objects, terminated by* `eof`*, and returns a* list *of those objects, in the order that they were typed in.*

**Exercise 29.5.3** ***Develop a function*** `read-objects-until` *which takes in an object (which we'll call the "terminator") and acts just like* `read-objects` *above except that it stops when it gets* either *the* `eof` *object* or *the "terminator".*

*For example, suppose I type* `(read-objects-until 'quit)`*.*
*It asks me for an object; I type 3.*
*It asks me for an object; I type snark.*
*It asks me for an object; I type quit.*
*It returns the list* `(list 3 snark)`*.*

**Hint:** Since you don't know what type the terminator object will be, you'll need `equal?`.

**Exercise 29.5.4** ***Develop a function*** `echo` *which asks the user for a sequence of objects, terminated by* `eof`*, and displays each of them on a separate line, in the same order that they were typed in.*

**Hint:** You'll need `begin`.

## 29.6 Files

In the real world, programs read and write *files* at least as often as they read from the keyboard or write to the screen. There are predefined functions to make that easy:

```
; with-input-from-file :  string(filename) thunk -> anything
; Calls the thunk in such a way that if the thunk uses read or
; similar functions, it will read from the specified file instead
; of from the keyboard.
; Returns the result of the thunk.

; with-output-to-file :  string(filename) thunk -> anything
; Calls the thunk in such a way that if the thunk uses display,
; write, print, printf, etc., they will write to the specified file
; instead of to the screen.
```

Note: if you plan to write from a program into a file, and later read from the same file into a program, it's a good idea to use `write` rather than `display`; otherwise you may write out a string and read it in as a symbol. Also, `write` and `display` do a good job of showing images on the screen, but they don't know how to save an image to a file; if you need to store images in files, use `save-image` and `bitmap` instead.

**Exercise 29.6.1** ***Modify*** *exercise 21.7.9 to add "save" and "load" buttons as in exercise 29.3.5, and optionally "save-to" and "load-from" buttons as in exercise 29.4.5. Note that when you save to a file and re-load later from the same file, the cursor position (as well as the contents) should be preserved. You've now written a very simple word processor.*

## 29.7 The World Wide Web

Another common source of information to programs is the World Wide Web. There's a predefined function that helps you get information from Web sites:

```
; with-input-from-url :  string thunk -> anything
; Calls the thunk in such a way that if it uses read or similar functions,
; they'll read from the specified Web page instead of from the keyboard.
```

For example,

```
(with-input-from-url "http://www.google.com" read-line)
```

will give you back the first line of the Google website (headers, HTML, Javascript, the whole works).

Of course, extracting really useful information from the Web requires recognizing the structure of an HTML or XML document. DrRacket comes with libraries to help with that, but they're beyond the scope of this book; look up "XML" in the Help Desk.

## 29.8 Review of important words and concepts

Historically, most programs have expected to "read" their input, either from a file or from the user's keyboard, and "write" their results, either to a file or in text form to the user's screen. In this chapter we've learned how to do that in (Advanced Student Language) Racket. These are our first examples of functions with *side effects* (almost: `define` and `define-struct` can be thought of as functions with side effects), and hence the first for which the *order of calling* functions makes a big difference. To tell Racket that we want to evaluate a series of expressions in order, not for their values but for their side effects, we use the `begin` form.

## 29.9 Reference: input, output, and sequence

In this chapter, we've seen the following new built-in functions:

- display
- write
- printf
- begin
- newline
- read
- read-line
- with-input-from-string
- with-output-to-string
- with-io-strings
- with-input-from-file
- with-output-to-file
- with-input-from-url