Chapter 19

Handling errors

19.1 Error messages

Recall Exercise 11.6.1, in which you built pictures of houses. Once `build-house` was working, one could then build a whole village by writing something like

```
(place-image (build-house ...)  30 200
             (place-image (build-house ...)  105 220
                          (place-image (build-house ...)  130 60
                                       (empty-scene 300 300))))
```

Now suppose some foolish user provides a width or height that isn't a positive number: you'll get an unfriendly error message like

rectangle: expected <positive number> as second argument, given: -30.

You'd like to make this friendlier by giving the user a more informative message like *House height must be* > 0

instead. One way to do this would be for `build-house` to return that string as its value.

Of course, this violates the contract of `build-house`, which said it returns an image. No problem: now that we know about mixed data types, we can change the contract:

```
; build-house :  number number string -> image-or-string
```

We'll need some extra examples to test that it produces the appropriate string in the appropriate cases; I'll leave that to you. Finally, the body of the function will have an extra conditional somewhere:

```
...
(cond [(> height 0) ...]
      [else "House height must be > 0."])
```

But now when you try to build a village as before, you get the error message

place-image: expected <image> as first argument, given "House height must be > 0*"*

What happened? The `build-house` function promised, in its original contract, to return an image, and `place-image` only works on images so it relies on this promise. Now we've broken that promise by returning a string instead, so other programs that use `build-house` don't work any more. You could fix this by putting a conditional *everywhere* that `build-house` is used, to check whether it returned an image or a string, but that's a royal pain, and would make your `build-house` function *much* more inconvenient to use.

The problem is that *normally*, the `build-house` function returns an image, which is what `place-image` expects; the only time `build-house` returns a string is when something is very wrong and it doesn't make sense to call `place-image` at all. So the ideal solution would be for `build-house` to produce an error message *and never return to* `place-image` *at all.* (In Java, C++, and some other languages, this is referred to as "throwing an exception".) There's a built-in function named `error` that does this. It uses a new data type that we haven't seen before: *symbol.* We'll discuss it more in Chapter 29, but for now, think of it as a function name with an apostrophe in front (but *not* at the end!).

The `error` function has contract

```
; error :  object ...-> nothing
; The first argument is normally a symbol:  the name of the function
; that discovered the error, with an apostrophe in front.
; Any additional arguments go into the error message too.
; The function doesn't return, but stops the program.
```

Worked Exercise 19.1.1 *Modify the* `build-house` *function so that if the width or height is less than 0, it produces an appropriate error message and doesn't return to its caller.*

Solution: We don't need to change the contract, since *if* `build-house` returns at all, it will still return an image. We need to add some additional test cases:

```
  (build-house 0 100 "blue") "should produce an error message:"
  "build-house:  House width must be > 0."
  (build-house 100 0 "red") "should produce an error message:"
  "build-house:  House height must be > 0."
```

The skeleton and inventory don't change, but the body now looks like

```
...
(cond [ (<= width 0)
        (error 'build-house "House width must be > 0.")]
      [ (<= height 0)
        (error 'build-house "House height must be > 0.")]
      [ else
        ...])
```

In testing this function, you should get an error message in response to the first "bad" test case. Indeed, you'll never even see the `"should produce an error message:"` because the program stops before getting that far in the definitions pane. Likewise, you'll never get to the second "bad" test case at all, so you don't know whether it works correctly. One way to handle this is to test one "bad" test case, then once it works, comment it out and run again to test the next one. A better way to handle it is described below.

■

19.2 Testing for errors

You're already familiar with the `check-expect`, `check-within`, `check-member-of`, and `check-range` functions, which compare the actual answer from some expression with what you say it "should be". There's another function, `check-error`, which "expects" the expression to crash with a specific error message, and checks that this actually happens.

```
  ; check-error :  test-expression string -> nothing
  ; Checks whether evaluation of the test-expression produces
  ; the specified string as an error message
```

For example, the above "bad" test cases could be rewritten as

```
(check-error (build-house 0 100 "blue")
             "build-house:  House width must be > 0.")
(check-error (build-house 100 0 "red")
             "build-house:  House height must be > 0.")
```

and you don't need to comment out either of them, since `check-error` catches the first error, checks that it's correct, and goes on to the next.

Note that `check-error` will complain if the expression produces the wrong error message, or even if it *doesn't* produce an error message: try

```
  (check-error (error 'whatever "this error message")
               "that error message")
  (check-error (+ 3 4) "something went wrong")
```

19.3 Writing user-proof functions

Exercise 19.3.1 *Modify the solution to Exercise 15.1.4 so that if the input to* `reply` *isn't any of the known strings, it produces an error message and never returns, rather than returning* `"huh?"`*.*

Exercise 19.3.2 ***Modify*** *the solution to Exercise 9.2.3 so that if the input is an empty string, it produces the error message* chop-first-char: can't chop from an empty string *and never returns.*

Exercise 19.3.3 ***Modify*** *the solution to Exercise 9.2.4 so that if the input is an empty string, it produces the error message* first-char: can't get first character of an empty string *and never returns.*

Exercise 19.3.4 ***Develop a function*** `safe-double` *that takes in a number, a string, a boolean, or an image. If the input is a number, the function doubles it and returns the result. If the input is anything else, the function produces an appropriate error message,* e.g.
safe-double: This function expects a number, like 3; you gave it a picture.
or (even cooler)
safe-double: This function expects a number, like 3; you gave it the quoted string "five".

Hint: The second example calls for inserting the actual string you were given into your error message. This can be done using `string-append`, or using the `format` function, which I haven't told you about yet. If you wish, look it up and rewrite the function that way.

19.4 Review of important words and concepts

A function contract is a binding promise; if you don't return the type of result you said you would return, other people's programs will crash, and they'll blame you. But sometimes things go wrong, and there *is* no value of the promised return type that would be correct. In this case, often the best answer is to "throw an exception": to bail out of any functions that have called this one, and display an error message in the Interactions pane. The `error` function does this job for you; the `check-error` function in the `testing` teachpack helps you test it.

19.5 Reference: Built-in functions for signaling and testing errors

In this chapter, we introduced two new built-in functions:

- `error`
- `check-error`

(Technically, `check-error` is a special form rather than a function.)

We also mentioned the `format` function, which you are invited to look up for yourself.