Chapter 16

New types and templates

16.1 Definition by choices

In Exercise 15.1.4, the contract said the input and output types were both *string*. This is a bit over-simplified. In fact, the input is supposed to be one of three possibilities, and the output will also be one of three possibilities.

In a sense, we've invented two new data types *greeting* and *answer*:

```
; A greeting is one of the strings "good morning",
; "good afternoon", or "good night"
; An answer is one of the strings "I need coffee",
; "I need a nap", or "bedtime!"

; reply :  greeting -> answer
; test cases as before
; definition as before
```

This may not seem important yet, but thinking of the input and the output as new data types actually helps us write the program. Since the input and output types are both three-way choices, there must be at least three test cases — one for each possibility — and the body of the method is probably a three-clause conditional. Furthermore, if we ever write *another* function that takes in or returns the *greeting* or *answer* type, it too will need at least three test cases, and its body will probably also involve a three-clause conditional.

The notion of defining a new data type as one of a specified set of choices is called "definition by choices". The predefined Boolean type can also be thought of as defined by choices: it has two choices, `true` and `false`, and as we've already seen, any function that returns a Boolean should have at least two test cases, one returning `true` and one returning `false`.

16.2 Inventories and templates

Suppose we were writing several functions that *each* took in a *greeting*, but all returned different kinds of things. The examples and function definitions would all look pretty similar: there would be three examples, using `"good morning"`, `"good afternoon"`, and

`"good night"` respectively, and the function definition would involve a conditional with three clauses, each question comparing the parameter with a different one of these strings.

Since so much of the code is identical from one function to another, it might save time to write the identical part once and for all. We'll put it in `#| ...|#` comments, for reasons that will become clear shortly.

```
#|
(check-expect (function-on-greeting "good morning") ...)
(check-expect (function-on-greeting "good afternoon") ...)
(check-expect (function-on-greeting "good night") ...)

(define (function-on-greeting greeting)
  ; greeting          a greeting, as defined above
  (cond [(string=?  greeting "good morning")   ...]
        [(string=?  greeting "good afternoon") ...]
        [(string=?  greeting "good night")     ...]
  ))
|#
```

This isn't a "real" function, obviously — the answers to the `cond`-clauses aren't filled in, and we don't even know what *types* they should be, much less the right answers — but rather a *template* for functions that take in a greeting. The template includes everything we can say about the function and its test cases just by knowing the input data type.

Now, every time you want to write a *real* function that takes in that data type, simply copy-and-paste everything between the `#|` and `|#`, change the name of the function, and you're 90% done.

Worked Exercise 16.2.1 ***Write a template*** *for functions that operate on bank balances, as defined in Exercise 15.4.1.*

Then ***use this template*** *to write* two functions*:* `bank-interest-rate` *(as before) and* `customer-type`*, which categorizes customers as* `"rich"`*,* `"moderate"`*,* `"poor"`*, or* `"college student"` *depending on the size of their bank account, using the same dividing lines.*

Solution: We'll start by defining the new data type *bank-balance*:

```
; A bank-balance is a number, in one of the categories
; 0-500 (not including 500); 500-1000 (not including 1000);
; 1000-4000 (not including 4000); and 4000-up.
```

Obviously, there are four choices. The template looks like

```
#|
(check-expect (function-on-bank-balance 200) ...)
(check-expect (function-on-bank-balance 500) ...)
(check-expect (function-on-bank-balance 800) ...)
(check-expect (function-on-bank-balance 1000) ...)
(check-expect (function-on-bank-balance 2000) ...)
(check-expect (function-on-bank-balance 4000) ...)
(check-expect (function-on-bank-balance 7500) ...)

(define (function-on-bank-balance balance)
  ; balance      a bank-balance
  (cond [(< balance 500)           ...]
        [(and (>= balance 500)
              (< balance 1000)     ...]
        [(and (>= balance 1000)
              (< balance 4000)     ...]
        [(>= balance 4000)         ...]
  ))
|#
```

The contract for `bank-interest` is

```
; bank-interest-rate :  bank-balance ->
                            number (either 0, 0.01, 0.02, or 0.03)
```

Next, copy-and-paste the template and change the name of the function:

```
(check-expect ( bank-interest-rate 200) ...)
(check-expect ( bank-interest-rate 500) ...)
(check-expect ( bank-interest-rate 800) ...)
(check-expect ( bank-interest-rate 1000) ...)
(check-expect ( bank-interest-rate 2000) ...)
(check-expect ( bank-interest-rate 4000) ...)
(check-expect ( bank-interest-rate 7500) ...)

(define ( bank-interest-rate balance)
  ; balance      a bank-balance
  (cond [(< balance 500)           ...]
        [(and (>= balance 500)
              (< balance 1000)     ...]
        [(and (>= balance 1000)
              (< balance 4000)     ...]
        [(>= balance 4000)         ...]
  ))
```

Replace the ... in the examples with the right answers for the problem you're trying to solve:

```
(check-expect (bank-interest-rate 200)  0.00)
(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 2000)  0.02)
(check-expect (bank-interest-rate 4000)  0.03)
(check-expect (bank-interest-rate 7500)  0.03)
```

Finally, replace the `...` in the `cond`-clause answers with the right answers for the problem you're trying to solve:

```
(define (bank-interest-rate balance)
  ; balance      a bank-balance
  (cond [(< balance 500)           0.00 ]
        [(and (>= balance 500)
              (< balance 1000)     0.01 ]
        [(and (>= balance 1000)
              (< balance 4000)     0.02 ]
        [(>= balance 4000)         0.03 ]
  ))
```

This should pass all its tests.

Now for `customer-type`. The contract is

```
; customer-type :  bank-balance -> string
;   ("rich", "moderate", "poor", or "college student")
```

By copying the template and changing the function name, we get

```
(check-expect ( customer-type 200) ...)
(check-expect ( customer-type 500) ...)
(check-expect ( customer-type 800) ...)
(check-expect ( customer-type 1000) ...)
(check-expect ( customer-type 2000) ...)
(check-expect ( customer-type 4000) ...)
(check-expect ( customer-type 7500) ...)

(define ( customer-type balance)
  ; balance      a bank-balance
  (cond [(< balance 500)           ...]
        [(and (>= balance 500)
              (< balance 1000)     ...]
        [(and (>= balance 1000)
              (< balance 4000)     ...]
        [(>= balance 4000)         ...]
  ))
```

We fill in the right answers in the examples:

```
(check-expect (customer-type 200)  "college student")
(check-expect (customer-type 500)  "poor")
(check-expect (customer-type 800)  "poor")
(check-expect (customer-type 1000)  "moderate")
(check-expect (customer-type 2000)  "moderate")
(check-expect (customer-type 4000)  "rich")
(check-expect (customer-type 7500)  "rich")
```

and then in the body of the function:

```
(define ( customer-type balance)
  ; balance      a bank-balance
  (cond [(< balance 500)           "college student" ]
        [(and (>= balance 500)
              (< balance 1000)     "poor" ]
        [(and (>= balance 1000)
              (< balance 4000)     "moderate" ]
        [(>= balance 4000)         "rich" ]
  ))
```

This should pass all its test cases. ∎

16.3 Outventories and templates

Likewise, suppose we were writing several functions that each *returned* an *answer*. They would probably all look like

```
#|
(check-expect (function-returning-answer ...)  "I need coffee")
(check-expect (function-returning-answer ...)  "I need a nap")
(check-expect (function-returning-answer ...)  "bedtime!")

(define (function-returning-answer whatever)
  (cond [...    "I need coffee"]
        [...    "I need a nap"]
        [...    "bedtime!"]
  ))
|#
```

Again, this obviously isn't a "real function", since this time the *questions* aren't filled in; it's only a *template* for functions that return a result of a particular type. It doesn't have an inventory, since we don't even know what type the input is, but it has what we might call an "outventory": the expressions likely to be needed to construct the right kind of answer.

Whereas an "inventory" answers the question "what am I given, and what can I do with it?", an "outventory" answers the question "what do I need to produce, and how can I produce it?". To use the cooking analogy, the "outventory" for a batch of cookies would involve observing that the *last* step of the process is baking, so we'd better find a cookie sheet and preheat the oven. Just as one can write a template based on an inventory, one can also write a template based on an outventory.

When you're writing a real function, you may have to choose between a template based on the input type and one based on the output type. In general, use the *more complicated* one. If the input type is more complicated than the output type, its template will be more detailed so you'll have less work left to do yourself. On the other hand, if the output type is more complicated than the input type (which happens less often), you should use an output-based template because it'll do more of the work for you.

16.4 Else and definition by choices

When we introduced an `else` case into Exercise 15.1.4, we were effectively changing the contract and data analysis: the function no longer took in one of three specific strings, but rather those three *or "any other string"*. In other words, the type definitions became something like

```
; A safe-greeting is one of four possibilities:  "good morning",
; "good afternoon", "good evening", or any other string.
; A safe-answer is one of four possibilities:  "I need coffee",
; "I need a nap", "bedtime!", or "huh?".
```

Technically, we could write the contract as

```
; replay :  string -> safe-answer
```

because the function now accepts *any* string, but it's more useful to think of it as

```
; reply :  safe-greeting -> safe-answer
```

since *safe-greeting*'s four possibilities tell us how to choose test cases: we need a `"good morning"`, a `"good afternoon"`, a `"good evening"`, and some other string. The four possibilities of the input type also tell us how to write the body of the function: a four-way conditional, checking whether the input is `"good morning"`, `"good afternoon"`, `"good evening"`, or any other string (which we can handle naturally using `else`); we need only to fill in the answers.

Alternatively, we could use the four cases of the result type *safe-answer* to tell us that we'll need four test cases — one returning each of the four legal answers. The outventory gives us a conditional with four clauses, with answers `"I need coffee"`, `"I need a nap"`, `"bedtime!"`, and `"huh?"`; we need only to fill in the questions.

16.5 A bigger, better design recipe

At this point I often find that students get confused between designing a function and designing a data type. Indeed, designing a function often requires that you design one or more data types first. The recipe in Figure 16.5 starts with the difference between function and data type, and then gives a series of steps for each one.

Exercise 16.5.1 *Re-do some of the problems from Chapter 15 in this style.*

16.6 Review of important words and concepts

When we write a function that makes decisions, it often helps to think of the input and/or output type as a *new data type defined by choices*. This helps us choose test cases, and helps again in getting from a function skeleton to a complete function body.

Figure 16.1: Design recipe, with definition by choices

Are you defining a function or a type?	
Functions are analogous to verbs in human languages: they represent *actions* that happen to particular things (the arguments) at a particular time. For example, `+` and `beside` are predefined functions, while `checkerboard2` and `cube` are user-defined functions.	**Data types** are like improper nouns (*e.g.* "computer", "student", "program") in human languages: they represent a *kind of thing*. Racket's built-in data types include "number", "boolean", "string", "image", *etc.* and you can define others like "bank-balance" and "letter-grade". A data type is not "called" at any particular time on any particular arguments, and it doesn't "return" a result; it just *is*.
Write a **contract** (and perhaps a **purpose statement**)	Identify the choices: how many distinct categories or values are there, and how can you detect each one? Are there borderlines to worry about?
Write **examples** of function calls, with correct answers, *e.g.* using `check-expect`. If you have a template for the input or output data type, use it as a starting point for the examples, skeleton and inventory.	Write **examples** of the new data type, one for each category. You don't need "correct answers", since the examples *are* the "correct answers". If your data type consists of sub-ranges, make sure to include examples at the boundaries.
Write a function skeleton and inventory. If you have a template for the input or output data type, use it as a starting point.	If you expect to write more than one function taking in the new type, write an **inventory template**. If you expect to write more than one function returning the new type, write an **outventory template**.
Fill in the function **body**. If it isn't obvious how to put together the pieces to get a right answer, try an **inventory with values** first.	
Proofread for errors that you can spot yourself	
Check Syntax for syntax errors that the computer can spot	
Test the program to make sure it produces correct answers	

If we expect to be writing *several* functions with the same input type or the same output type, it may save us time to write a *function template*: a function skeleton, with an inventory and/or outventory, but no actual code. A template should say as much as you can say about the function by knowing only its input and output types, but not knowing what specific problem it's supposed to solve. Once you've written one, you can copy-and-paste it as a starting point for *every* function you need to write that has that input type or that output type.

16.7 Reference

No new functions or syntax rules were introduced in this chapter.