Chapter 25

Multiple recursive data

The inventories and template functions we've been using so far are generally based on the data type of an input (or, in a few cases, on the data type of the result). If an input is a list, we can use the input template of Chapter 22; if the output is a list, we can use the output template of Chapter 23; if the input is a natural number, we can use the input template of Chapter 24; *etc.*

But what template should we use when we have *two or more* inputs of complex types?

If your function takes two parameters x and y of complex types, there are four possibilities:

		y	
		simple	complex
x	simple	1	2
	complex	3	4

1. x and y are both simple;
2. x is simple but y is complex;
3. x is complex but y is simple;
4. x and y are both complex.

25.1 Separable parameters

In exercise 23.1.9, we wrote a `list-append` function that took in two list parameters. We used the usual list template for the first and pretended the second was of a simple type. In other words, we lumped cases 1 and 2 together, and cases 3 and 4 together.

		list2	
		simple	complex
list1	simple	1	2
	complex	3	4

Likewise, in exercise 24.3.1, we wrote an `wn-add` function that took in two natural number parameters. In that case, we used the usual natural-number template for one of them and pretended the other one was of a simple type. The rest of the exercises in section 24.3 can also be done in this way: a function that does different things depending on whether x is simple or complex, perhaps calling a helper function that does different things depending on whether y is simple or complex.

I call this situation "separable parameters", because we can separate the question of whether x is simple or complex from the question of whether y is simple or complex.

25.2 Synchronized parameters

That approach works frequently, but not all the time. For example,

Worked Exercise 25.2.1 *Develop a function* `pay-list` *that takes in two lists: one specifies the number of hours worked by each of a bunch of employees, and the other specifies the corresponding hourly wage for each of those employees. It should produce a list of the amounts each employee should be paid, in the same order.*

Solution: The contract is straightforward:

```
; pay-list :  list-of-numbers(hours)
              list-of-numbers(hourly-wages)
              -> list-of-numbers
```

For test cases, as usual, we'll start with the simplest cases and then build up:

```
(check-expect (pay-list empty empty) empty)
(check-expect (pay-list (list 3) empty)  ???)
```

The two lists we were given are of different sizes, and it's not clear from the problem assignment what should happen in this case. In fact, the problem doesn't even make sense if the lengths of the two lists are different. So let's revise the contract:

```
; pay-list :  list-of-numbers(hours)
              list-of-numbers(hourly-wages)
              -> list-of-numbers
; Assumes the two input lists are the same length.
; Result will be the same length.
```

Now that we've excluded inputs that make no sense, we can get back to the test cases:

```
(check-expect (pay-list empty empty) empty)
(check-expect (pay-list (list 30) (list 8)) (list 240))
(check-expect (pay-list (list 30 20 45) (list 8 10 10))
              (list 240 200 450))
```

The skeleton is easy. For the inventory, we'll try the usual list template on `hours`, and treat `hourly-wages` as simple.

```
(define (pay-list hours hourly-wages)
  ; hours                list of numbers (list 30 20 45)
  ; hourly-wages         list of numbers (list 8 10 10)
  (cond [(empty?  hours) ...]
        [(cons?  hours)
         ; (first hours) number          30
         ; (rest hours)  list of numbers (list 20 45)
         ; (pay-list (rest hours) hourly-wages)
         ;               list of numbers ???
         ; right answer  list of numbers (list 240 200 450)
         ]))
```

What is the "right answer" to the recursive call? It doesn't *have* a right answer, because it's being called on two lists of different lengths!

We'll have to try something different. It doesn't make sense to call `(pay-list (rest hours) hourly-wages)`, or for that matter `(pay-list hours (rest hourly-wages))`, because both of those calls break the contract by passing in lists of different lengths. But `(pay-list (rest hours) (rest hourly-wages))` *would* make sense, if we knew that both of those things existed — that is, if *both* `hours` and `hourly-wages` were non-empty lists. Similarly, if *both* of them were empty lists, the answer would be empty. So here's a revised inventory:

```
(define (pay-list hours hourly-wages)
  ; hours                       list of numbers (list 30 20 45)
  ; hourly-wages               list of numbers (list 8 10 10)
  (cond [ (and (empty?  hours) (empty?  hourly-wages)) ...]
        [ (and (cons?  hours) (cons?  hourly-wages))
         ; (first hours)        number          30
         ; (first hourly-wages) number          8
         ; (rest hours)         list of numbers (list 20 45)
         ; (rest hourly-wages)  list of numbers (list 10 10)
         ; (pay-list (rest hours)  (rest hourly-wages))
         ;                      list of numbers (list 200 450)
         ; right answer         list of numbers (list 240 200 450)
         ]))
```

Now it's easy:

```
(define (pay-list hours hourly-wages)
  ; hours                       list of numbers (list 30 20 45)
  ; hourly-wages               list of numbers (list 8 10 10)
  (cond [(and (empty?  hours) (empty?  hourly-wages))  empty]
        [(and (cons?  hours) (cons?  hourly-wages))
         ; (first hours)        number          30
         ; (first hourly-wages) number          8
         ; (rest hours)         list of numbers (list 20 45)
         ; (rest hourly-wages)  list of numbers (list 10 10)
         ; (pay-list (rest hours) (rest hourly-wages))
         ;                      list of numbers (list 200 450)
         ; right answer         list of numbers (list 240 200 450)
         (cons (* (first hours) (first hourly-wages))
               (pay-list (rest hours) (rest hourly-wages)))
         ]))
```

One thing might bother you about this definition. (At least it bothers *me*!) We're checking whether *both* parameters are empty (case 1), and whether *both* parameters are non-empty (case 4), but what if one is empty and the other isn't (cases 2 and 3)? That would of course be an illegal input, but if some user tried it, (s)he would get an unfriendly error message like *cond: all question results were false*, and the user might conclude that *our program* was wrong, when in fact it's the user's fault. Instead, let's handle this case specifically with a more-informative error message:

```
(define (pay-list hours hourly-wages)
  (cond [(and (empty?  hours) (empty?  hourly-wages)) ...]
        [(and (cons?  hours) (cons?  hourly-wages))
         (cons (* (first hours) (first hourly-wages))
               (pay-list (rest hours) (rest hourly-wages)))
         ]
        [else
         (error 'pay-list
                "Number of hours and hourly wages must match.")
        ]))

(check-error (pay-list (list 3) empty)
       "pay-list:  Number of hours and hourly wages must match.")
(check-error (pay-list (list 3) (list 8 7.50))
       "pay-list:  Number of hours and hourly wages must match.")
```

In this example, it wasn't enough to treat one of the two list inputs as if it were simple: we had to go through the two lists *in lock-step*, looking at the `first` of both at the same time, and the `rest` of both at the same time. It's usually easy to spot such situations, because they *don't make sense unless the inputs are the same size.* When you see such a problem, not only can you use an inventory like the above, but you can also include an error-check that produces an informative error message if the inputs *aren't* the same size.

For this sort of problem, cases 2 and 3 (in which one of x and y is simple and the other complex) are lumped together — they're both illegal — and we only really need to worry about cases 1 and 4.

		hourly-wages	
		simple	complex
hours	simple	1	2
	complex	3	4

25.3 Interacting parameters

There are problems that don't seem to fit either of the preceding patterns: it's reasonable for either of the parameters to be simple and the other complex, and whatever one of them is, it makes a difference what the other one is. The parameters *interact* with one another.

Worked Exercise 25.3.1 *Suppose the* `list-ref` *function weren't built into DrRacket; how would we write it? Since* `list-ref` *is built in, we'll name ours* `pick-element` *instead.*

Develop *a function* `pick-element` *that takes in a natural number and a non-empty list, and returns one of the elements of the list. If the number is 0, it returns the first element of the list; if 1, it returns the second element;* etc. *If there is no such element, it should produce an appropriate error message.*

Solution: The contract is

```
; pick-element :  natural non-empty-list -> object
```

A *non-empty list* is defined to be either `(cons object empty)` or `(cons object non-empty list)`; the template for it looks like

```
#|
(check-expect (function-on-nel (list "a")) ...)
(check-expect (function-on-nel (list "a" "b")) ...)
(check-expect (function-on-nel (list 3 1 4 1 5)) ...)

(define (function-on-nel L)
  (cond [(empty?  (rest L)) ...]
        [(cons?  (rest L))
         ; L                           non-empty list
         ; (first L)                   object
         ; (rest L)                    non-empty list
         ; (function-on-nel (rest L)) whatever
         ...
        ]))
|#
```

For our problem, the test cases clearly need to include long-enough and not-long-enough lists.

```
(check-expect (pick-element 0 (list "a")) "a")
(check-expect (pick-element 0 (list 5 -2 17)) 5)
(check-error (pick-element 1 (list "a"))
             "pick-element:  no such element")
(check-expect (pick-element 1 (list 5 17)) 17)
(check-expect (pick-element 1 (list 5 -2 17)) -2)
(check-error (pick-element 4 (list "a"))
             "pick-element:  no such element")
(check-expect
   (pick-element 4 (list "a" "b" "c" "d" "e"))
   "e")
(check-expect
   (pick-element 4 (list "a" "b" "c" "d" "e" "f" "g"))
   "e")
```

As a first attempt, let's try the separable-parameters approach, treating n as complex and the list as simple.

```
(define (pick-element n things)
  ; n                                    natural
  ; things                              non-empty list
  (cond [(zero?  n) ...]
        [(positive?  n)
         ; (sub1 n)                     natural
         ; (pick-element (sub1 n) things) object
         ...
        ]))
```

In the base case ($n = 1$), the right answer is either `(first things)` or an error message, depending on whether `things` is empty. We can do that if necessary, using a nested `cond`.

The recursive case is a problem: knowing the answer to `(pick-element (sub1 n) things)` tells you *nothing* about the right answer to `(pick-element n things)`. This won't work.

Maybe if, instead of treating n as complex and the list as simple, we treat the list as complex and n as simple?

```
(define (pick-element n things)
  ; n                                    natural
  ; things                               non-empty list
  (cond [(empty?  (rest things)) ...]
        [(cons?  (rest things))
         ; things                        non-empty list
         ; (rest things)                 list
         ; (pick-element n (rest things)) object
         ...
        ]))
```

The base case, again, will need a nested conditional to check whether n is 1 or larger. The recursive case still has a problem: knowing the answer to `(pick-element n (rest things))` tells you nothing about the right answer to `(pick-element n things)`.

There seems to be a sort of synchronization going on here: the list length doesn't have to be *exactly the same* as the number, but it does have to be *at least as much*. So let's try the synchronized approach. I'll also throw in an "inventory with values" while we're at it.

```
(define (pick-element n things)
  ; n                    natural        1
  ; things               non-empty list (list "a" "b" "c")
  (cond [(and (zero?  n) (empty?  (rest things)))
         (first things)]
        [(and (positive?  n) (cons?  (rest things)))
         ; (sub1 n)      natural        0
         ; (rest things) non-empty list (list "b" "c")
         ; (pick-element (sub1 n) (rest things))
         ;                object         "b"
         ; right answer  object         "b"
         ...]
        [else (error 'pick-element "no such element")]
```

This looks a little more promising: the result of the recursive call is the same as the right answer to the problem at hand. So let's make that the answer in the second `cond` clause:

```
(define (pick-element n things)
  ; n                    natural        1
  ; things               non-empty list (list "a" "b" "c")
  (cond [(and (zero?  n) (empty?  (rest things)))
         (first things)]
        [(and (positive?  n) (cons?  (rest things)))
         ; (sub1 n)      natural        0
         ; (rest things) non-empty list (list "b" "c")
         ; (pick-element (sub1 n) (rest things))
         ;                object         "b"
         ; right answer  object         "b"
         (pick-element (sub1 n) (rest things)) ]
        [else (error 'pick-element "no such element")]
```

This passes several of its tests, but not all: in particular, it fails tests in which the element to be picked *isn't the last*. By following the "synchronized" approach, we've effectively forced the length of the list to be *exactly one more* than the number. To get this function to work right, it must be more permissive: if the number is down to 0 and the list isn't only one element, that's fine.

```
(define (pick-element n things)
  ; n                    natural        1
  ; things               non-empty list (list "a" "b" "c")
  (cond [(and (zero?  n) (empty?  (rest things)))
         (first things)]
        [(and (positive?  n) (cons?  (rest things)))
         (first things)]
        [(and (positive?  n) (cons?  (rest things)))
         ; (sub1 n)      positive-natural
         ; (rest things) non-empty list (list "b" "c")
         ; (pick-element (sub1 n) (rest things))
         ;                object         "b"
         ; right answer  object         "b"
         (pick-element (sub1 n) (rest things)) ]
        [else (error 'pick-element "no such element")]
```

The conditional now explicitly identifies and deals with *all four* cases:

		n	
		simple	complex
things	simple	1	2
	complex	3	4

For this particular problem, two of the cases produce the same answer, so we can combine them. Removing the inventory comments, we're left with

```
(define (pick-element n things)
  (cond [ (zero?  n)     (first things)]
        [(and (positive?  n) (cons?  (rest things)))
         (pick-element (sub1 n) (rest things)) ]
        [else (error 'pick-element "no such element")]))
```

which does in fact pass all its tests.

In this case, we've actually combined cases 1 and 3:

		n	
		simple	complex
things	simple	1	2
	complex	3	4

But in other situations, we might actually need all four cases to do four different things. See the Exercises below.

The above function definition isn't foolproof, in that if someone violates the contract by passing in the number 0, or a negative number, or an empty list, it'll produce an ugly error message. We could remedy this by re-formulating the problem as accepting *any* number and *any* list; the result would be

```
(define (pick-element n things)
  (cond [(< n 0)
         (error 'pick-element "illegal element number")]
        [(empty?  things)
         (error 'pick-element "no such element")]
        [(zero?  n)                    (first things)]
        [(and (positive?  n) (cons?  things))
         (pick-element (sub1 n) (rest things))]))
```

∎

25.4 Exercises

Some of these exercises involve separable parameters; some involve synchronized parameters; some involve interacting parameters, and may require treating all four possible combinations of simple and complex separately.

Exercise 25.4.1 *Develop a function* `cart-prod` *(short for "Cartesian product") that takes in two lists and returns a list of two-element lists, each with an element from the first input list and an element from the second in that order. The result should include* all possible pairs *of elements. You may assume that there are no duplicate elements in the first list, and no duplicate elements in the second list (although there might be things that are in both input lists.)*

Hint: You'll need a helper function.

Exercise 25.4.2 *Develop a function* `make-posns` *that takes in two lists of numbers, the same length, and produces a list of* `posns` *with x coordinates taken from the first list, in order, and y coordinates from the corresponding elements of the second list.*

Exercise 25.4.3 *Develop a function* `label-names` *that takes in a list of strings and a natural number, which should be how many strings there are. It produces a list of two-element lists, each comprising a different natural number and one of the strings from the list. The numbers should be in decreasing order, starting with the given number and ending with 1, and the strings should be in the order they were given. For example,*

```
(check-expect (label-names (list "anne" "bob" "charlie") 3)
  (list (list 3 "anne") (list 2 "bob") (list 1 "charlie")))
```

If the above exercise feels sorta strange and artificial, here's a more natural (but slightly harder) version, which will probably require a helper function:

Exercise 25.4.4 *Develop a function* `label-names-2` *that takes in a list of strings, and produces a list of two-element lists, each comprising a different natural number and one of the strings from the list. The numbers should be in increasing order, starting with 1 and ending with the number of strings in the list, and the strings should be in the order they were given. For example,*

```
(check-expect (label-names-2 (list "anne" "bob" "charlie"))
  (list (list 1 "anne") (list 2 "bob") (list 3 "charlie")))
```

Exercise 25.4.5 *Develop a function* `intersection` *that takes in two lists of strings and returns a list of the strings that appear in* both *of them, leaving out any string that appears in only one or the other. The strings in the result should be in the same order that they appeared in the first list. You may* assume *that no string appears more than once in either list.*

Exercise 25.4.6 *Develop a function* `union` *that takes in two lists of strings and returns a list of the strings that appear in* either *list, but only once each. You may* assume *that no string appears more than once in either list.*

Exercise 25.4.7 *Develop a function* `set-diff` *that takes in two lists of strings and returns a list of the strings that appear in* the first but not the second. *You may* assume *that no string appears more than once in either list.*

Exercise 25.4.8 *Re-do exercises 25.4.5, 25.4.6, and 25.4.7 without the assumption that there are no duplicates in the inputs.*

Exercise 25.4.9 ***Develop a function*** `binary-add` *that takes in two natural numbers and returns their sum, using the binary template (and not using the built-in* `+` *function).*

Note: Somebody had to do this, not in Racket but in wires and transistors, in order for your computer to be able to add.

Exercise 25.4.10 ***Develop a function*** `binary-mult` *that takes in two natural numbers and returns their product, using the binary template (and not using the built-in* `+` *or* `*` *functions).*

Note: Somebody had to do this, not in Racket but in wires and transistors, in order for your computer to be able to multiply.

Exercise 25.4.11 ***Develop a function*** `binary-raise` *that takes in two natural numbers m and n and returns m^n, using the binary template (and not using the built-in* `+`*,* `*`*, or* `expt` *functions).*

Exercise 25.4.12 *Develop a function* `substring?` *that takes in two strings and tells whether the first one appears in the second as a substring. For example,*

```
(check-expect (substring?  "bob" "") false)
(check-expect (substring?  "" "bob") true)
(check-expect (substring?  "b" "bob") true)
(check-expect (substring?  "c" "bob") false)
(check-expect (substring?  "bob" "bob") true)
(check-expect (substring?  "bob" "bobs") true)
(check-expect (substring?  "bob" "brats and snobs") false)
(check-expect (substring?  "no rat" "brats and snobs") false)
(check-expect (substring?  "bob" "thingbobs") true)
(check-expect (substring?  "bob" "I botched it but bob fixed it")
  true)
(check-expect (substring?  "bob" "I botched it but amy fixed it")
  false)
```

(There is a function in some dialects of Racket that does this job, but I want you to do it using only `char=?`*, comparing one character at a time.)*

Exercise 25.4.13 *Develop a function* `subsequence?` *that takes in two strings and tells whether the characters of the first appear in the same order in the second (but possibly with some other characters in between). For example,*

```
(check-expect (subsequence?  "bob" "") false)
(check-expect (subsequence?  "" "bob") true)
(check-expect (subsequence?  "b" "bob") true)
(check-expect (subsequence?  "c" "bob") false)
(check-expect (subsequence?  "bob" "bob") true)
(check-expect (subsequence?  "bob" "bobs") true)
(check-expect (subsequence?  "bob" "brats and snobs") true)
(check-expect (subsequence?  "no rat" "brats and snobs") false)
(check-expect (subsequence?  "bob" "thingbobs") true)
(check-expect (subsequence?  "bob" "I botched it but bob fixed it")
true)
(check-expect (subsequence?  "bob" "I botched it but amy fixed it")
  true)
```

I don't think there's a built-in Racket function that will help much with this, but in any case, I want you to do this using only `char=?`*, comparing one character at a time.*

Hint: Perhaps surprisingly, this problem is *easier* than `substring?`.

Exercise 25.4.14 *Develop a function* `lcsubstring` *("longest common substring") that takes in two strings and returns the longest string which is a substring of both of them. For example,*

```
(check-expect (lcsubstring "mickey mouse" "minnie mouser") "mouse")
```

The answer may not be unique: for example,

`(lcsubstring "mickey mouse" "minnie mush")`

could legitimately be either `"mi"`*,* `" m"`*, or* `"us"`*.*

Hint: Different approaches to this can differ radically in efficiency. My first attempt took several *minutes* to solve the `(lcsubstring "mickey mouse" "minnie mush")` problem.

A different approach, using only things that you've seen already, solved the same problem in 0.01 seconds; the difference is even more dramatic for longer strings.

Exercise 25.4.15 *Develop a function* `lcsubsequence` *("longest common subsequence") that takes in two strings and returns the longest string which is a subsequence of both of them. For example,*

```
(check-expect (lcsubsequence "mickey mouse" "minnie moose")
  "mie mose")
```

The answer may not be unique: for example, `(lcsubsequence "abc" "cba")` *could legitimately be any of the strings* `"a"`, `"b"`, *or* `"c"`.

Hint: As in exercise 25.4.14, your program may be slow. My first attempt took about 2.5 seconds to solve `(lcsubsequence "mickey mouse" "minnie moose")`, and I don't know of a more efficient way to do it using what you've already seen. A technique called *dynamic programming* or *memoization*, which we'll discuss in Chapter 30, enabled me to do it in about 0.01 seconds. Again, the difference is more dramatic for longer strings.

Exercise 25.4.16 *A common task in computer science is* pattern-matching*: given a pattern, ask whether a particular string matches it. In our pattern language, a* `"?"` *stands for "any single character," while* `"*"` *stands for "any zero or more characters." For example, the pattern* `"c?t"` *would match "cat" and "cut" but not "colt", "cats", or "dog". Similarly, the pattern* `"cat*"` *would match the strings "cat", "cats", "catastrophe", etc. but not "caltrop" or "dog". The pattern* `"a??a*r"` *would match "abbatoir", "akbar", and "araaar", etc. but not "almoner", "alakazam", or "fnord". The pattern* `"*.docx"` *would match the name of any Word 2007 file (and thus could be used to decide which filenames to show in a file dialog).*

Define *a function* `pattern-match?` *that takes in two strings: the* pattern *and the* target*, and tells whether the target matches the pattern.*

Note that the special characters `"?"` *and* `"*"` *are special only when they appear in the* pattern*; if they appear in the* target*, they should be treated as ordinary characters.*

SIDEBAR:

Exercises 25.4.12 through 25.4.16 resemble problems that come up in biology: the "strings" in that case are sequences of DNA bases in a gene, or sequences of amino acids in a protein. The efficiency of such programs determines how quickly a genome can be sequenced, a drug interaction predicted, a virus identified, *etc.*

25.5 Review of important words and concepts

When you have to write a function that takes in two or more complex parameters (lists, strings, natural numbers, *etc.*), you can take several different approaches.

- If the problem "doesn't make sense" unless the parameters are "the same size" (or have some particular relationship to one another), then you can generally take the "synchronized parameters" approach: your function will check whether both are simple (the base case) and whether both are complex (the recursive case). If one is simple but the other complex, it produces an error message.

		p2	
		simple	complex
p1	simple	1	2
	complex	3	4

```
(define (function-on-synch-params p1 p2)
  (cond [(and (simple?  p1) (simple?  p2)) ...]
        [(and (complex?  p1) (complex?  p2))
         ; (function-on-synch-params (simplify p1)
                                      (simplify p2))
         ...]
        [else (error ...)]))
```

(The functions *simple?*, *complex?*, and *simplify* in the above aren't real functions; they stand for however you identify simple and complex elements of the data type in question, and how you simplify it. For example, if you were dealing with lists, *simple?* would stand for `empty?`; *complex?* for `cons?`; and *simplify* for `rest`.)

- If the expression necessary to produce the right answer is the same for both simple and complex second parameters, we call the parameters "separable", and you can just use a template on the first parameter, treating the second parameter as a simple type.

		p2	
		simple	complex
p1	simple	1	2
	complex	3	4

```
(define (function-on-separable-params p1 p2)
  (cond [(simple?  p1) ...]
        [(complex?  p1)
         ; (function-on-separable-params (simplify p1) p2)
         ...]))
```

Likewise, if the expression is the same for simple and complex *first* parameters, you can use a template on the second parameter, treating the first as a simple type.

- If neither of the previous situations applies, you'll probably need to identify all four possible combinations of simple and complex parameters and treat them individually. Furthermore, in the case that both are complex, there are several different reasonable recursive calls you could make.

		p2	
		simple	complex
p1	simple	1	2
	complex	3	4

```
(define (function-on-interacting-params p1 p2)
  (cond [(and (simple?  p1) (simple?  p2)) ...]
        [(and (simple?  p1) (complex?  p2))
         ; (function-on-interacting-params p1 (simplify p2))
         ...]
        [(and (complex?  p1) (simple?  p2))
         ; (function-on-interacting-params (simplify p1) p2)
         ...]
        [(and (complex?  p1) (complex?  p2))
         ; (function-on-interacting-params p1 (simplify p2))
         ; (function-on-interacting-params (simplify p1) p2)
         ; (function-on-interacting-params
                     ; (simplify p1) (simplify p2))
         ...]))
```

Some functions that you can write using these techniques are correct, but surprisingly slow and inefficient; a technique called *dynamic programming* or *memoization*, which we can't discuss until chapter 30, can improve the efficiency enormously.

25.6 Reference

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