Chapter 23

Functions that return lists

If you did exercises 22.5.15 or 22.5.16, you've already written some functions that return lists, but only in a very simple way: adding one new element to the front of an existing list. In this chapter we'll discuss functions that construct an entire list as their results.

23.1 Doing something to each element

Worked Exercise 23.1.1 ***Develop a function*** `add1-each` *that takes in a list of numbers and returns a list of numbers of the same length, with each element of the answer equal to 1 more than the corresponding element of the input. For example,*

```
(check-expect (add1-each (cons 3 (cons -12 (cons 7 empty))))
              (cons 4 (cons -11 (cons 8 empty))))
```

Solution: For brevity, I'll write this as a single function; the two-function version is quite similar. The contract, test cases, skeleton, and inventory are straightforward:

```
; add1-each :  list-of-numbers -> list-of-numbers
(check-expect (add1-each empty) empty)
(check-expect (add1-each (cons 3 empty)) (cons 4 empty))
(check-expect (add1-each (cons 3 (cons -12 (cons 7 empty))))
              (cons 4 (cons -11 (cons 8 empty))))

(define (add1-each nums)
  ; nums                           lon
  (cond [(empty?  nums)      ...]
        [(cons?  nums)
         ; nums                    nelon
         ; (first nums)            number
         ; (rest nums)             lon
         ; (add1-each (rest nums)) lon
         ...]
  ))
```

The answer to the empty case is obviously `empty` (since the result has to be the same length as the input). To fill in the non-empty case, let's do an inventory with values:

```
  [(cons?  nums)
   ; nums                    (cons 3 (cons -12 (cons 7 empty)))
   ; (first nums)            3
   ; (rest nums)             (cons -12 (cons 7 empty))
   ; (add1-each (rest nums)) (cons -11 (cons 8 empty))
   ; right answer            (cons 4 (cons -11 (cons 8 empty)))
   ...]
  ))
```

Notice that the recursive call `(add1-each (rest nums))` gives you most of the right answer, but it's missing a 4 at the front. Where could the 4 come from? Since `(first nums)` in this example is 3, an obvious choice is `(+ 1 (first nums))`.

```
  [(cons?  nums)
   ; nums                    (cons 3 (cons -12 (cons 7 empty)))
   ; (first nums)            3
   ; (rest nums)             (cons -12 (cons 7 empty))
   ; (add1-each (rest nums)) (cons -11 (cons 8 empty))
   ; right answer            (cons 4 (cons -11 (cons 8 empty)))
   (cons (+ 1 (first nums))
         (add1-each (rest nums))) ]
  ))
```

Test this, and it should work on all legal inputs. ∎

Exercise 23.1.2 *Develop a function* `square-each` *that takes in a list of numbers and returns a list of their squares, in the same order.*

Exercise 23.1.3 *Develop a function* `string-lengths` *that takes in a list of strings and returns a list of their (numeric) lengths, in the same order.*

Exercise 23.1.4 *Develop a function* `salaries` *that takes in a list of* `employee` *structures (from Exercise 21.2.1) and returns a list containing only their salaries, in the same order.*

Exercise 23.1.5 *Develop a function* `give-10%-raises` *that takes in a list of* `employee` *structures and returns a list of the same* `employees`*, but each earning* 10% *more than before.*

Exercise 23.1.6 *Develop a function* `stutter` *that takes in a list of anything (it doesn't matter whether they're strings, numbers, or something else) and returns a list twice as long, with each element repeated twice in a row. For example,*

```
(check-expect (stutter (cons 5 (cons 2 (cons 9 empty))))
  (cons 5 (cons 5 (cons 2 (cons 2 (cons 9 (cons 9 empty)))))))
```

Exercise 23.1.7 *Develop a function* `list-each` *that takes in a list (of numbers, strings, it doesn't matter) and returns a* list of one-element lists*, each containing a different one of the elements in the original list. For example,*

```
(check-expect (list-each (cons "a" (cons "b" empty)))
  (cons (cons "a" empty) (cons (cons "b" empty) empty)))
```

Exercise 23.1.8 ***Develop a function*** `suffixes` *that takes in a list (of numbers, strings, it doesn't matter) and returns a* list of lists *comprising all the* suffixes *of the list (that is, "the last 3 elements," "the last 17 elements,", "the last 0 elements,"* etc. *of the given list). For example,*

```
(check-expect (suffixes (cons "a" (cons "b" (cons "c" empty))))
  (cons (cons "a" (cons "b" (cons "c" empty)))
        (cons (cons "b" (cons "c" empty))
              (cons (cons "c" empty)
                    (cons empty
                          empty))))))
```

Exercise 23.1.9 *Recall the built-in* `string-append` *function, which takes in two strings and produces a single string by combining them end to end.* ***Develop a function*** `list-append` *that takes in two* lists *(of numbers, strings, it doesn't matter) and combines them end-to-end into one list. For example,*

```
  (check-expect
    (list-append (cons "a" (cons "b" (cons "c" empty)))
                 (cons "d" (cons "e" empty)))
    (cons "a" (cons "b" (cons "c" (cons "d" (cons "e" empty))))))
```

Hint: This function takes in *two* lists, so one might wonder what template to use. We'll discuss this more fully in Chapter 25, but for now, use the template on the first of the two lists, treating the second as just a simple variable.

There's a built-in function `append` that does this, but you are *not* allowed to use `append` in writing your function; the point is that if `append` weren't built in, you could have written it yourself. After you've finished this exercise, feel free to use `append` in the rest of the book.

Exercise 23.1.10 *Define a function* `backwards` *that takes in a list (of anything) and returns a list of the same objects in the opposite order.*

There's a built-in function named `reverse` which does this, but I want you to define it yourself without using `reverse`. After you've finished this exercise, feel free to use `reverse` in the rest of the book.

23.2 Making decisions on each element

In some problems, you need to make a decision about each element of a list, using a conditional. As with Exercises 22.5.5, 22.5.7, *etc.*, this conditional is usually nested inside the one that decides whether the list is empty or not.

Exercise 23.2.1 ***Develop a function*** `substitute` *that takes in two strings and a list of strings, and returns a list the same length as the given list, but with all occurrences of the first string replaced by the second. For example,*

```
(check-expect
  (substitute "old" "new" (cons "this" (cons "that" (cons "old"
    (cons "new" (cons "borrowed" (cons "old" (cons "blue"
    empty))))))))
  (cons "this (cons "that" (cons "new" (cons "new"
    (cons "borrowed" (cons "new" (cons "blue" empty)))))))
```

Exercise 23.2.2 ***Develop a function*** `remove-all` *that takes in a string and a list of strings, and returns the same list but with all occurrences (if there are any) of the given string removed. For example,*

```
(check-expect
  (remove-all "old" (cons "this (cons "that" (cons "old"
    (cons "new" (cons "borrowed" (cons "old"
    (cons "blue" empty))))))))
  (cons "this" (cons "that" (cons "new" (cons "borrowed"
    (cons "blue" empty)))))))
```

Exercise 23.2.3 ***Develop a function*** `remove-first` *that takes in a string and a list of strings, and returns the same list but with the first occurrence (if any) of the given string removed. For example,*

```
(check-expect
  (remove-first "old" (cons "this (cons "that" (cons "old"
    (cons "new" (cons "borrowed" (cons "old"
    (cons "blue" empty))))))))
  (cons "this" (cons "that" (cons "new" (cons "borrowed"
    (cons "old" (cons "blue" empty)))))))
```

Exercise 23.2.4 ***Develop a function*** `unique` *that takes in a list of objects and returns a list of the same objects, but each appearing only once each.*

Hint: There are several ways to do this. Probably the easiest way, given what you've seen so far, produces a result in the order in which each string *last* appeared in the input; for example,

```
(check-expect (unique (cons "a" (cons "b" (cons "a" empty))))
  (cons "b" (cons "a" empty)))
; not (cons "a" (cons "b" empty)))
(check-expect
  (unique (cons 3 (cons 8 (cons 5 (cons 5 (cons 8 empty))))))
  (cons 3 (cons 5 (cons 8 empty))))
; not (cons 3 (cons 8 (cons 5 empty)))
```

We'll discuss other approaches in later chapters.

Since you don't know what kind of objects you're dealing with, you'll need to use `equal?` to compare them.

Exercise 23.2.5 ***Develop a function*** `fire-over-100K` *that takes in a list of* `employee` *structures and returns a list of those who earn at most $100,000/year, leaving out the ones who earn more. The remaining employees should be in the same order they were in before.*

Exercise 23.2.6 ***Develop a function*** `add-vote-for` *that takes in a string (representing a candidate's name) and a list of* `candidate` *structures, and returns a list of* `candidate` *structures in which that candidate has one more vote (and all the others are unchanged). You may assume that no name appears more than once in the list.*

Hint: What should you do if the name doesn't appear in the list at all?

Exercise 23.2.7 ***Develop a function*** `tally-votes` *that takes in a list of strings (Voter 1's favorite candidate, Voter 2's favorite candidate,* etc.*) and produces a list of* `candidate` *structures in which each candidate's name appears once, with how many votes were cast for that candidate.*

23.3 A shorter notation for lists

23.3.1 The list function

Writing `(cons "a" (cons "b" (cons "c" empty)))` for a three-element list is technically correct, but it's tedious. Fortunately, Racket provides a shorter way to accomplish the same thing. There's a built-in function named `list` that takes zero or more parameters and constructs a list from them. For example,

```
> (list "a" "b" "c")
(cons "a" (cons "b" (cons "c" empty)))
```

Note that `list` is *just a shorthand*: it produces the exact same list as the `cons` form, and any function that works on one of them will still work on the other.

Common beginner mistake
I've frequently seen students simply replace every `cons` in their code with `list`, getting results like
`(list "a" (list "b" (list "c" empty)))`
Think of `cons` as *adding one element to the front* of a list, whereas `list` builds a list from scratch. If you call `cons` on two arguments, the result will be a list one element longer than the second argument was; if you call `list` on two arguments, the result will be a list of exactly two elements. For example,

```
> (define my-list (list "x" "y" "z"))
> (cons "w" my-list)
(cons "w" (cons "x" (cons "y" (cons "z" empty))))
> (list "w" my-list)
(cons "w" (cons (cons "x" (cons "y" (cons "z") empty) empty))
```

Practice Exercise 23.3.1 ***Translate*** *each of the following lists from* `list` *notation into nested-*`cons` *notation.* ***Check*** *that your answers are correct by typing each expression into DrRacket (Beginning Student language) and comparing the result with your answer.*

```
(list)
(list "a")
(list "a" "b")
(list (list "Mary" "Joe") (list "Chris" "Phil"))
(list empty "a" empty)
```

23.3.2 List abbreviations for display

The `list` function makes it easier and more convenient to *type in* lists (especially lists of structs, lists of lists, *etc.*), but it's still a pain to *read* them. If you want lists to *print out* in `list` notation rather than nested-`cons` notation, simply go to the "Language" menu in DrRacket, select "Choose Language", and then (under the *How to Design Programs* heading) select "Beginning Student with List Abbreviations".

```
> (list "a" "b" "c")
(list "a" "b" "c")
> (cons "a" (cons "b" (cons "c" empty)))
(list "a" "b" "c")
> (define my-list (list "x" "y" "z"))
> (cons "w" my-list)
(list "w" "x" "y" "z")
> (list "w" my-list)
(list "w" (list "x" "y" "z"))
```

Again, note that this is only a change in output convention: both `cons` and `list` still work, and any correct function on lists will still work no matter which way you type in the examples, and no matter which language you're in.

Practice Exercise 23.3.2 ***Translate*** *each of the following lists from nested-`cons` notation into `list` notation.* ***Check*** *that your answers are correct by typing each expression into DrRacket (Beginning Student with List Abbreviations language) and comparing the result with your answer.*

```
(cons "a" empty)
empty
(cons 3 (cons 4 (cons -2 empty)))
(cons (cons 3 empty) empty)
```

There's an even shorter form called "quasiquote notation", using the apostrophe:

```
> '(1 2 (3 4) 5)
(list 1 2 (list 3 4) 5)
> '("abc" "de" ("f" "g") "h")
(list "abc" "de" (list "f" "g") "h")
> '()
empty
```

Quasiquote notation is *not* available in Beginning Student language.

SIDEBAR:

If you want to see your *output* in this even-shorter notation, "Choose Language", choose "Beginning Student with List Abbreviations", click "Show Details" at the bottom of the window, find the "Output Style" section on the right, choose "Quasiquote", then click "OK" and then "Run" (which you have to do whenever you change languages).

You'll notice that the output has not an ordinary apostrophe but rather a "backquote". For now, you can treat these two characters as the same. Backquote allows you to do some other neat things, but we won't use it in this textbook; if you're really curious, look it up in the Help Desk.

Practice Exercise 23.3.3 *For each exercise from Chapters 22 and 23 that you've already done, rewrite the test cases using `list` or `quasiquote` notation, and try the function again. The results should be especially nice for functions that take in or return lists of lists or lists of structs, like Exercises 22.8.5 and 23.1.8.*

23.4 Animations with lists

Exercise 23.4.1 ***Write an animation*** *of a bunch of balls, each moving around the screen with constant velocity and bouncing off walls. Pressing the* `+` *key should create one more ball, with random initial location (inside the animation window) and random velocity (say, from -10 to +10 in each dimension). Pressing the* `-` *key should remove the most recently-added ball, unless there are no balls, in which case it should do nothing. Clicking with the mouse inside a ball should remove the ball you clicked on, leaving the rest of the balls unchanged.*

Hint: You'll need each ball to have a location and a velocity, as in exercise 21.7.6, and use a list of structs as your model, as in exercise 22.6.7.

Hint: What should happen if you click with the mouse in a place where two or more balls overlap? The assignment doesn't say exactly; you should decide in advance what you *want* to happen in this case, and make it work.

23.5 Strings as lists

In Section 22.7, we showed two different ways to write functions on strings: using an input template for them analogous to the input template for lists, and using the built-in function `string->list`, which converts a string to a list of characters.

One can do the exact same thing for functions that *return* a string: either use an output template analogous to that for returning a list, or use the built-in function `list->string`, which converts a list of characters to a string. (If any of the things in the list is *not* a character, it produces an error message.) I recommend trying some of the following problems each way.

Exercise 23.5.1 ***Develop a function*** `string-reverse` *that takes in a string and returns a string of the same characters in reverse order.*

Exercise 23.5.2 ***Develop a function*** `string-suffixes` *that takes in a string and returns a list of all its suffixes. For example,*

```
(check-expect (string-suffixes "abc")
  (list "abc" "bc" "c" ""))
```

Exercise 23.5.3 ***Develop a function*** `replace-char` *that takes in a character and two strings (*replacement *and* target*). It returns a string formed by replacing each occurrence of the character in* target *with the entire string* replacement*. For example,*

```
(check-expect (replace-char #\r "fnord" "reference librarian")
  "fnordefefnordence libfnordafnordian")
```

Exercise 23.5.4 ***Develop a function*** *named* `ubby-dubby` *which translates a given string into "ubby-dubby talk". This is defined as follows: insert the letters "ubb" in front of each vowel in the original string. For example,*

```
(check-expect (ubby-dubby "hello there")
              "hubbellubbo thubberubbe")
```

You may assume *for simplicity that all the letters are lower-case. You may find it useful to write a* `vowel?` *helper function.*

Exercise 23.5.5 ***Modify*** *your solution to exercise 23.5.4 so it inserts the letters "ubb" only once in front of each* group of consecutive *vowels. For example,*

```
(check-expect (ubby-dubby "hello friends out there")
  "hubbellubbo frubbiends ubbout thubberubbe")
```

Hint: See Exercise 22.7.4.

Exercise 23.5.6 ***Develop a function*** `words` *that takes in a string and returns a* list *of strings, one for each word in the input string, leaving out any spaces, punctuation, numbers,* etc. *A "word" is defined as in Exercise 22.7.4: a sequence of one or more letters. For example,*

```
(check-expect (words "This is chapter 26, or is it 25?")
              (list "This" "is" "chapter" "or" "is" "it"))
```

Exercise 23.5.7 ***Develop a function*** `pig-latin` *that takes in a string and converts it to "Pig Latin": for each word, if it starts with a vowel, add "way" at the end of the word, and if it starts with a consonant, move that consonant to the end of the word and add "ay". You may* assume *that the input string has no upper-case letters, numbers, or punctuation. For example,*

```
(check-expect (pig-latin "hi boys and girls")
              "ihay oysbay andway irlsgay")
```

Exercise 23.5.8 ***Modify*** *your solution to exercise 23.5.7 so that if a word starts with* more than one *consonant, the function moves* all *of them to the end, followed by "ay". For example,*

```
(check-expect (pig-latin "this is a strange function")
              "isthay isway away angestray unctionfay")
```

Exercise 23.5.9 ***Modify*** *your solution to exercise 23.5.7 or 23.5.8 to handle capital letters correctly: any word that started with a capital letter before should still start with a capital letter after converting it to Pig Latin, and capital letters moved from the start to the end of a word should no longer be capitalized. For example, if you made both this modification and the one in exercise 23.5.8,*

```
(check-expect (pig-latin "My name is Stephen Bloch")
  "Ymay amenay isway Ephenstay Ochblay")
```

Hint: To do this, you may need some of the built-in functions `char-upper-case?`, `char-lower-case?`, `char-upcase`, and `char-downcase`. Look them up in the Help Desk.

Exercise 23.5.10 ***Design a function*** `basic-mad-lib` *that takes in a string (the* template*) and a non-empty list of strings, and returns a string. The template may contain ordinary words and punctuation, as well as the hyphen character (-). The output of the function should be based on the template, but it should replace each - with a randomly-chosen word from the list of words. For example,*

```
(basic-mad-lib "The - bit the - and took a - home."
  (list "dog" "cat" "senator" "taxi" "train" "chair"))
; could be "The chair bit the dog and took a senator home."
```

Exercise 23.5.11 ***Design a function*** `mad-lib` *similar to the above, but it takes a string (the* template*) and* three *non-empty lists (which we'll call* nouns*,* verbs*, and* adjectives*). The template may contain the "special" words* `-noun-`*,* `-verb-`*, and* `-adjective-`*; each* `-noun-` *should be replaced by a randomly chosen element of* nouns*, and so on. For example,*

```
(mad-lib
  "The -noun- -verb- the -adjective- -noun- and -verb- quickly."
  (list "dog" "cat" "senator" "taxi" "train" "chair")
  (list "tall" "green" "expensive" "chocolate" "overrated")
  (list "ate" "drank" "slept" "wrote"))
; could be
"The senator slept the overrated cat and drank quickly."
```

23.6 More complex functions involving lists

Lists and recursion allow us to solve much more interesting and complicated problems than we could solve before. Sometimes such problems require "helper" or "auxiliary" functions. For each of the following problems, you'll need *at least* one helper function. To figure out what helper functions you need, just start writing the main function, following the design recipe. When you reach the "inventory with values" point, you'll probably find that there is no built-in function to produce the right answer from the available expressions. So decide what such a function would need to do. Then write it, following the design recipe. This function in turn may need yet another helper function, and so on.

Exercise 23.6.1 ***Develop a function*** `sort` *that takes in a list of numbers and returns a list of the same numbers in increasing order.*

Hint: There are several possible ways to do this. If you use an input template, you'll probably need a helper function that inserts a number in order into an already-sorted list. If you use an output template, you'll probably need two helper functions: one to find the smallest element in an unsorted list, and one to remove a specified element from an unsorted list. In either case, I recommend treating `list-of-numbers` and `sorted-list-of-numbers` as two separate types: when a function produces a sorted list, or assumes that it is given a sorted list, say so in the contract and inventory, and make sure your test cases satisfy the assumption.

Exercise 23.6.2 ***Develop a function*** *named* `sort-candidates` *that takes in a list of* `candidate` *structures and returns a list of the same* `candidate` *structures in decreasing order by number of votes, so the winner is first in the list, the second-place winner is second,* etc.. *In case of ties, either order is acceptable.*

Exercise 23.6.3 ***Develop a function*** `subsets` *that takes in a list (of numbers, strings, it doesn't matter) and returns a list of lists representing* all possible subsets *of the elements in the original list, once each. For example,* `(subsets (list "a" "b" "c"))` *should produce something like*

```
(list (list)
      (list "a")
      (list "b")
      (list "a" "b")
      (list "c")
      (list "a" "c")
      (list "b" "c")
      (list "a" "b" "c"))
```

You may assume *that all the things in the input list are different.*

Exercise 23.6.4 ***Develop a function*** `scramble` *that takes in a list (of numbers, strings, it doesn't matter) and returns a list of lists representing* all possible orderings *of the elements in the original list, once each. For example,* `(scramble (list "a" "b" "c" ))` *should produce something like*

```
(list (list "a" "b" "c")
      (list "b" "a" "c")
      (list "a" "c" "b")
      (list "c" "a" "b")
      (list "b" "c" "a")
      (list "c" "b" "a"))
```

Again, you may assume *that all the things in the input list are different. Even better,* decide *what the function should do if there are duplicate elements in the input list, and make sure it does the right thing.*

Hint: This will probably require *more than one* helper function. Take it one step at a time: try to write the main function, figure out what you need to do to the recursive result, invent a function to do that, and repeat until what you need to do is built-in.

Exercise 23.6.5 ***Develop a function*** `scramble-word` *that takes in a string and returns a list of strings representing* all possible orderings *of the characters in the string.*

For a basic version of this function, you may include the same string more than once: for example, `(scramble-word "foo")` *might produce* `(list "foo" "ofo" "oof" "foo" "ofo" "oof")` *Once you have this working, try rewriting it so it doesn't produce any duplicates:* `(scramble-word "foo")` *might produce* `(list "foo" "ofo" "oof")` *instead.*

Hint: Re-use functions you've already written!

Exercise 23.6.6 ***Modify*** *the* `scramble-word` *function so that, even if there are repeated characters in the string, the result won't contain the same word twice:* `(scramble-word "foo")` *might produce* `(list "foo" "ofo" "oof")`*.*

23.7 Review of important words and concepts

Many of the most interesting things one can do with a list involve producing another list. Sometimes we do the same thing to every element of a list, producing a list of the results. Sometimes we select some of the elements of a list, producing a shorter list. And sometimes we do more complicated things like `scramble` or `subsets`.

Recall from Chapter 20 that the inventory-with-values technique tends to be more useful the more complicated the output type is. Lists, and especially lists of lists or lists of structs, are the most complicated types we've seen yet, and the inventory-with-values technique is extremely helpful in writing these functions.

23.8 Reference: built-in functions that return lists

We've seen several new built-in functions in this chapter:

- `append`
- `char-upper-case?`
- `char-lower-case?`
- `char-upcase`
- `char-downcase`
- `reverse`
- `list->string`