Working with more than one item at once

So far we have written only functions that consume one or a few values, and may combine them in various ways.

More often we have a collection of data to process.

Racket is a dialect of LISP, which was originally designed for List Processing.

Our principal way of grouping values is the list.

To easily work with lists, switch your Language Level to Intermediate Student.

What is a list?

The word list comes from Old English “líste”, meaning a strip (such a strip of cloth or paper).

“His targe wi þ gold list He carf atvo.”
(Guy of Warwick, ca. 1330)

→ A strip of paper with items written on it.
→ An ordered collection of items.

We can make a list really easily. A few examples:

```racket
(define wishes (list "comics" "turtle figures" "Donkey Kong" "Play-Doh Burger King"))
(define primes (list 2 3 5 7 11 13 17 19))
```

A value may be a list

Lists behave just like any other value.

We can define constants which are lists:

```racket
(define wishes (list "comics" "turtle figures" "Donkey Kong" "Play-Doh Burger King"))
(define primes (list 2 3 5 7 11 13 17 19))
```

We can have functions consume lists:

```racket
(length wishes) ⇒ 4
(first wishes) ⇒ "comics"
(rest wishes) ⇒ (list "turtle figures" "Donkey Kong" "Play-Doh Burger King")
(second wishes) ⇒ "turtle figures"
```

We can have functions produce lists:

```racket
(range 4 16 2) ⇒ (list 4 6 8 10 12 14)
(cons 6 (list 3 5 15)) ⇒ (list 6 3 5 15)
```
In the design recipe, we specify the type of values in a list as follows:

- **Use** `(listof Type)` for a single type.
  - `(listof Nat)` describes a list containing zero or more `Nat`. E.g. `(list 6 7 42)`
  - `(listof Str)` describes a list containing zero or more `Str`. E.g. `(list "hi" "there")`
- If a list may contain more than one type, use `(listof (anyof Type1 Type2 ...))`.
  - `(listof (anyof Num Str))` describes a list containing zero or more values, each of which is either a `Num` or a `Str`. E.g. `(list 3.14 "pie" "forty-two" -17)`
- If a list is of known length and types, use `(list Type1 Type2 ...)`.
  - `(list Nat Str)` describes a list containing two values. The first value is a `Nat`, and the second value is a `Str`. E.g. `(list 6 "foo")`.
  - `(list "foo" 6)` is not a `(list Nat Str)`. It is a `(list Str Nat)`.

### Exercise

For each set of lists, find a type that describes all the lists. Try to be as specific as possible.

For example, `(list 3 4 5)` is a `(listof Num)`, but it is also a `(listof Int)`, and even more specifically a `(listof Nat)`.

1. `(list 4 3 -7), (list 3 1)`
2. `(list "We're" "all" "fine here, now," "thank" "you."), (list "How" "are" "you?")
3. `(list "Joe" "Biden"), (list "Kamala" "Harris"), (list "Nancy" "Pelosi")
4. `(list 4 "*" 6 "=" 24), (list "sqrt" 4 "=" 2)
5. `(list 2.5 1.4142 5), (list)
6. `(list (list 1 2) (list 3 4 5)), (list (list 6) (list -5 3))

### Empty lists

A list does not need to contain anything. Hopefully, immediately after shopping, your shopping list is empty; you bought everything, and want nothing more.

We write a list with certain items in it by writing `(list` followed by all the items in the list, followed by `)`. So we can create an empty list simply by writing `(list)`; that is, with no items after the `(list` and before the `)`.

Our current version of Racket displays the empty list as `empty`. We can create that same value by writing `empty`. There are several ways to represent this value, including `empty`, `(list)`, and `()' .

For any type, a list of that type may contain **zero or more** values of that type. So `(list)`, that is, `empty`, counts as a list of any type.

If we have a function that works on a `(listof Num)`, for example, we generally don’t need to deal specially with the case that the list is empty.
Transforming items in a list using \texttt{map}

We can store data in a list, but what can we do with them?

There is a built-in function called \texttt{map} that transforms each item in a list, using a function.

\[
\text{(map } f \text{ (list } x_0 \ x_1 \ x_2 \ \ldots \ x_n) \Rightarrow \text{(list } (f x_0) \ (f x_1) \ (f x_2) \ \ldots \ (f x_n))
\]

For example: \text{(map \texttt{sqr} (list 0 1 2 3 4)) ⇒ (list 0 1 4 9 16)}

Exercise

Try out each use of \texttt{map} with the given list, and one or two other lists.

\textbf{What happens?}

\begin{itemize}
\item \text{(map \texttt{sqr} (list 0 1 2 3 4))}
\item \text{(define (\texttt{double-item} x) (* 2 x))}
\end{itemize}

\begin{itemize}
\item \text{(define (\texttt{double-each} L) (map \texttt{double-item} L))}
\item \text{(\texttt{double-each} (list 0 1 2 3 4))}
\end{itemize}

Strategy for working with \texttt{map}

To use \texttt{map} on a list of values of some type:

\begin{itemize}
\item write a function that consumes \textit{one single value} and transforms it as required.
\end{itemize}

For example, I wish to transform each item in a list by \( f(x) = 10\sqrt{x} \):

\begin{itemize}
\item Function to transform a single value:

\begin{verbatim}
;; (10rootx x) produce 10*sqrt(x)
;; Requires: x >= 0
;; Examples:
(check-expect (10rootx 49) 70)
(define (10rootx x) (* 10 (sqrt x)))
\end{verbatim}

\item Function to transform all items:

\begin{verbatim}
;; (10rootx-each L) transform each item in L by 10rootx.
;; Requires: each value is >= 0
;; Examples:
(check-expect (10rootx-each (list 49 81 100)) (list 70 90 100))
(define (10rootx-each L) (map 10rootx L))
\end{verbatim}
\end{itemize}
Working with **map**

To use **map** on a list of values of some type:
- write a function that consumes *one single value* and transforms it as required.

**Exercise**

Digital signals are often recorded as values between 0 and 255, but we often prefer to work with numbers between 0 and 1.
Write a function (**squash-range** `L`) that consumes a *(listof Nat)*, and produces a *(listof Num)* so numbers on the interval \([0, 255]\) are scaled to the interval \([0, 1]\).

\[(\text{squash-range } (\text{list } 0 204 255)) \Rightarrow (\text{list } 0 0.8 1)\]

**Exercise**

Write a function that consumes a *(listof Str)*, where each **Str** is a person’s name, and produces a list containing a greeting for each person.

\[(\text{greet-each } (\text{list } "\text{Ali}" "\text{Carlos}" "\text{Sai}")) \Rightarrow (\text{list } "\text{Hi Ali!}" "\text{Hi Carlos!}" "\text{Hi Sai!}"\]

Using **range** to build lists

\((\text{range } \text{start} \text{end} \text{step})\) produces the list that starts at \(\text{start}\), and steps by \(\text{step}\) until just before it reaches \(\text{end}\). This allows us to build new lists. Some examples:

\[(\text{range } 4 10 1) \Rightarrow (\text{list } 4 5 6 7 8 9)\]
\[(\text{range } 4 10 2) \Rightarrow (\text{list } 4 6 8)\]
\[(\text{range } 20 8 -3) \Rightarrow (\text{list } 20 17 14 11)\]
\[(\text{range } 20 8 3) \Rightarrow \text{empty} ; \text{the empty list}\]

To work with **range** and **map**:

1. get proper values from **range**; test it.
2. use **map** to transform these values as needed.

**Exercise**

Complete the function **list-cubes**.

\[;; \text{list-cubes } n \text{ produce the list of cubes from } 1*1*1 \text{ to } n*n*n.\]
\[;; \text{list-cubes: Nat } \to (\text{listof Nat})\]
\[;; \text{Examples:}\]
\[;(\text{check-expect } (\text{list-cubes } 4) (\text{list } 1 8 27 64))\]

Ignoring parameters!

A function does not need to use all its parameters. We could create a function that always produces the same value. Consider:

\[;; \text{always-zero } x \text{ Ignore } x, \text{ and produce } 0.\]
\[;; \text{Examples:}\]
\[;(\text{check-expect } (\text{always-zero } 42) 0)\]
\[;(\text{check-expect } (\text{always-zero } "\text{is this thing on?}" ) 0)\]
\[;; \text{always-zero: Any } \to \text{ Nat}\]
\[; (\text{define } (\text{always-zero } x) 0)\]
...This seems spectacularly useless. But maybe not...

;; (n-zeros n) produce a list containing n zeros.
;; Examples:
(check-expect (n-zeros 5) (list 0 0 0 0))

;; n-zeros: Nat -> (listof Nat)
(define (n-zeros n)
  (map always-zero (range 0 n 1)))

---

Exercise

Write a function `starry` that consumes a Nat and produces a list containing that many copies of "*". For example,

(starry 5) ⇒ (list "*" "*" "*" "*" "*")

---

Summarizing a list using `foldr`

`range` lets us create a list, and `map` lets us transform each item. What if I want to my result to be a combination of the items in the list, instead of the entire list?

What is the total of all the values in (list 6 5 8 5 14 4)?

(+ 6 (+ 5 (+ 8 (+ 5 (+ 14 4))))) ⇒ 42

To do this automatically, there is another function, `foldr`, meaning “fold right”.

(foldr f base (list x0 x1 ... xn)) ⇒ (f x0 (f x1 (f ... (f xn base))))

What does this mean?

We combine items, starting from the right, each time creating a new item to combine with.

(foldr + 0 (list 6 5 8 5 14 4))

1 The sum of nothing is zero, so the base is 0.

2 To calculate the sum of a value and another sum, just add the two values.

(define (add new old) (+ new old))
(define (sum lon) (foldr add 0 lon))
(sum empty) ⇒ 0
(sum (list 5 8 4)) ⇒ (add 5 (add 8 (add 4 0))) ⇒ 17
Working with \texttt{foldr}

\[
(foldr \ f \ base \ (\text{list} \ x_0 \ x_1 \ ... \ x_n)) \Rightarrow (f \ x_0 \ (f \ x_1 \ (f \ ... \ (f \ x_n \ base))))
\]

1. Figure out what the answer is when the list is empty. Use this as the base.
2. Write a function that consumes two values, \textit{new} and \textit{old}, where \textit{new} is a value from the list, and \textit{old} is an answer.

\textbf{Exercise}

Write a function \texttt{prod} that produces the product of a \texttt{(listof Num)}.

\[
(prod \ (\text{list} \ 2 \ 2 \ 3 \ 5)) \Rightarrow 60
\]

\textbf{Exercise}

Write a function \texttt{count-odd} that produces the number of odd numbers in a \texttt{(listof Nat)}.

\textbf{Hint: read the documentation on \texttt{remainder}}.

Can you do this using \texttt{map} and \texttt{foldr}? Just using \texttt{foldr}?

\textbf{Tracing foldr}

How’s that work again?

Each call to the function get a new value from the list, and an old answer from a the base or an earlier function call.

\[
(foldr \ \texttt{string-append} \ "2B" \\
\quad \ (\text{list} \ "To" \ "be" \ "or" \ "not")) \Rightarrow \texttt{string-append} \ "To" \\
\quad \ (\texttt{string-append} \ "be" \\
\quad \quad \ (\texttt{string-append} \ "or" \\
\quad \quad \quad \ (\texttt{string-append} \ "not" \ "2B"))) \\
\Rightarrow "Tobeornot2B"
\]

\textbf{Exercises}

\begin{itemize}
\item \textbf{Experiment with \texttt{fold-sub}.}
\item Describe how it behaves, and why.
\item Write the contract and a better purpose statement.
\end{itemize}

\begin{verbatim}
;; (fold-sub L) Do something mysterious with L.
;; fold-sub: (listof Int) -> ...

(define (fold-sub L) (foldr - 0 L))
(fold-sub (list 6 5 2)) ⇒ ?
\end{verbatim}

\begin{itemize}
\item \textbf{Read the documentation on \texttt{string-length}.}
\item Write a function \texttt{total-length} that produces the total length of all the values in a \texttt{(listof Str)}.
\end{itemize}

\begin{verbatim}
(total-length (list "hello" "how" "r" "u?")) ⇒ 11
\end{verbatim}
**Exercises**

**Exercise**

Write a function that produces the average (mean) of a non-empty `(listof Num)`.

```scheme
(check-expect (average (list 2 4 9)) 5)
(check-expect (average (list 4 5 6)) 5.25)
```

*Recall that `length L` produces the number of values in L.*

**Exercise**

The factorial function, \( n! \), produces the product of the numbers from 1 to \( n \). For example, \( 4! = 1 \times 2 \times 3 \times 4 = 24 \).

Write a function `(factorial n)` that produces \( n! \).

```scheme
(check-expect (factorial 5) 120)
(check-expect (factorial 1) 1)
```

---

**Non-empty lists?**

We said: “Figure out what what the answer is when the list is empty. Use this as the base.” But what if the question makes no sense for an empty list?

*What is the average of the empty list?*

If a problem makes no sense for an empty list, following the design recipe, we can write a `;; Requires` statement that the value is not empty. But then what can we use as the base?

Often, it is sufficient to use some value from the list. If we don’t care which, the first value may suffice. Consider:

```scheme
(define (sum-A lon) (foldr + 0 lon))
(define (sum-B lon) (foldr (+ (first lon) (rest lon))))
```

In `sum-A`, we need this value 0. But in `sum-B` we don’t need such a value; we work only with values from `lon`.

For summing values in a list, this makes no difference. But in some cases it will....

---

**range with other stuff**

**Exercise**

Write a function `(sum-square-difference n)` that consumes a `Nat` and produces the difference between the square of the sum of numbers from 0 to \( n \), and the sum of the squares of those numbers.

```scheme
(sum-square-difference 3) ⇒ (+ (sqr (+ 0 1 2 3)) (+ 0 1 4 9)) ⇒ 22
```

The square of the sum sum of the squares
A list is a value

A list contains values. But a list also is a value. So we can make a list of lists.

Imagine: I have some jars. Each jar contains some coins. How to express this in Racket?

Use a list of natural numbers for each jar; for example, \((\text{list } 5 \ 5 \ 10 \ 100)\) represents a jar containing four coins of various values, \((\text{list } 25)\) contains one coin, while \((\text{list})\) represents an empty jar.

Then use a list of these jars to represent the collection. The whole collection is:

\((\text{list (list } 5 \ 5 \ 10 \ 100) \ (\text{list } 25) \ (\text{list}))\).

Since this is a list, and each item in the list is a \((\text{listof Nat})\), this is a \((\text{listof (listof Nat)})\).

A list is a value

Exercise

Write a function \(\text{sum-coins}\) that consumes a \((\text{listof (listof Nat)})\) and produces the total value.

\((\text{check-expect (sum-coins (list (list 5 5 10 100) (list 25) empty)) 145})\)

Hint

Use the \(\text{sum}\) function we defined earlier as part of your solution.

A second folding function

\(\text{foldr}\) means “fold right”. This suggests that there should also be a “fold left”. And there is: \(\text{foldl}\).

\(\text{foldl}\) does the same sort of combining, but starting at the left:

\((\text{foldl } f \ base \ (\text{list } x0 \ x1 \ ... \ xn)) \Rightarrow (f \ xn \ (f \ ... \ (f \ x1 \ (f \ x0 \ base) \ x1))\))

The results are sometimes the same:

\((\text{foldr} + 0 \ (\text{list } 2 \ 3 \ 5 \ 7)) \Rightarrow 17\)
\((\text{foldl} + 0 \ (\text{list } 2 \ 3 \ 5 \ 7)) \Rightarrow 17\)

But not always:

\((\text{foldr} - 0 \ (\text{list } 2 \ 3 \ 5 \ 7)) \Rightarrow -3\)
\((\text{foldl} - 0 \ (\text{list } 2 \ 3 \ 5 \ 7)) \Rightarrow 3\)
Tracing foldl

foldl does almost the same thing that foldr does, just in the reverse direction.

\[
\text{foldl} \text{ string-append } \text{"2B"} \\
\quad \text{lst } \text{"To" } \text{"be" } \text{"or" } \text{"not"})
\]
\[
\quad \Rightarrow \text{string-append } \text{"not"} \\
\quad \quad \text{string-append } \text{"or"} \\
\quad \quad \quad \text{string-append } \text{"be"} \\
\quad \quad \quad \quad \text{string-append } \text{"To" } \text{"2B"})
\]
\[
\Rightarrow \text{"notorbeTo2B"}
\]

That's all we're going to say about foldl until much later. In the meantime, you may use it if you like. Generally, it will not make your code simpler to write.

Multi-argument map

So far we have used map only with functions that consume a single value: like \((\text{map } f \text{ lst})\), where \(f\) is a single-parameter function and \(\text{lst}\) is a list. But map can do so much more!

map works with any number of lists, all of the same length: \((\text{map } f \text{ lst1 lst2...})\)

For example, if we have two lists of equal length we can make a new list where the first value is the sum of the first values, the second is the sum of the second values, and so on.

\(\text{;; (elementwise-sum } L M) \text{ add each value in } L \text{ to the corresponding value in } M.\)
\(\text{;; Examples:}\)
\(\text{(check-expect (elementwise-sum (list 2 3 3) (list 7 4 1)) (list 9 7 4))}\)

\(\text{;; elementwise-sum: (listof Int) (listof Int) } \rightarrow \text{ (listof Int)}\)
\(\text{;; Requires: } L \text{ and } M \text{ are of equal length.}\)
\(\text{(define (elementwise-sum } L M) (\text{map } + L M))}\)

Multi-argument map

Exercise
Write a function \((\text{absdiff } a \text{ b})\) that consumes two \((\text{listof Int})\) and produces a \((\text{listof Nat})\) containing the absolute value of the difference between corresponding values.

\((\text{absdiff (list 1 3 5 7) (list 7 3 6 1)) } \Rightarrow \text{(list 6 0 1 6)}\)
Suppose we have two `(listof Str)`: one of first names, and one of matching last names:

```
(define gnames (list "Joseph" "Burt" "Douglas" "James" "David"))
(define snames (list "Hagey" "Matthews" "Wright" "Downey" "Johnston"))
```

**Exercise**

Complete `join-names`.

```
;; (join-names G S) Make a list of full names from G and S.
;; join-names: (listof Str) (listof Str) -> (listof Str)
;; Examples:
(Andrezej check-expect (join-names gnames snames)
(list "Joseph Hagey" "Burt Matthews" "Douglas Wright"
"James Downey" "David Johnston"))
```

**Multi-argument folding**

You can do something similar with `foldr` and `foldl`, again with lists of the same length:

```
(foldr f base (list a0 a1 a2 ... an) (list b0 b1 b2 ... bn))
⇒ (f a0 b0 (f a1 b1 (f a2 b2 (... (f an bn base))))))
```

**Exercise**

The *dot product* combines two vectors into a number, by multiplying corresponding elements, and adding them up. For example, the dot product of `[1, 2, 3]` and `[5, 2, 7]` is

```
1 × 5 + 2 × 2 + 3 × 7 = 30.
```

Write a function `(dot-product v1 v2)` that consumes two `(listof Num)`, of equal length. Treat these as vectors, and calculate the dot product. Use a single call to `foldr` to accomplish this.

For example:

```
(check-expect (dot-product (list 1 2 3) (list 5 2 7)) 30)
(check-expect (dot-product (list 4 3) (list 9 2)) 42)
```
Module Summary

- Start storing information in lists, and describe lists in contracts.
- Transform list values using map, and foldr.
- Construct new lists using range, especially in combination with map.
- Use foldr to combine a list to a single value. This can be especially powerful when combined with map.
- Understand the use of (listof ...) and (anyof ...) and be able to use them in your design recipes.

Summary: built-in functions

In this module we added the following to our toolbox:

first foldl foldr length list map range rest

These are the functions and special forms currently in our toolbox:

* + - / abs ceiling check-expect check-within define exp expt first floor foldl foldr
  length list map max min number->string quotient range remainder rest sqr sqrt
  string-append string-length substring