Lists

Readings: HtDP, sections 9 and 10.

- Avoid 10.3 (uses `draw.ss`).
- The textbook introduces “structures“ before lists. The discussion of lists makes a few references to structures that can be ignored.

Topics:

- Introducing lists
- Formalities: Contracts, syntax & semantics, data definitions, templates
- Processing lists
- Patterns of recursion
- Producing lists from lists
- Design recipe refinements
- Strings and lists of characters

Introducing lists

Numbers, strings and even Boolean values can represent a single data item.

But there are many circumstances in which we need more data: the names of all the students in a course, the weight of each bag loaded on an airplane, or the answers to a true/false multiple-choice quiz.

The amount of data is often unbounded, meaning it may grow or shrink – and you don't know how much. The order of values may also be important.

Many programming languages meet this need with **lists**.

A **list** is a recursive structure – it is defined in terms of a smaller list.

Consider a list of concerts:

- A list of 4 concerts is a concert followed by a list of 3 concerts.
- A list of 3 concerts is a concert followed by a list of 2 concerts.
- A list of 2 concerts is a concert followed by a list of 1 concert.
- A list of 1 concert is a concert followed by a list of 0 concerts.

A list of zero concerts is special. We'll call it the **empty list**.

Lists are created with `(cons v lst)`, which adds a value `v` to the beginning of list `lst`. `empty` is the empty list.

> Example lists

A sad state of affairs – no upcoming concerts to attend:

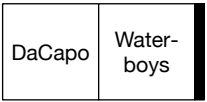
```
(define concerts0 empty)
```

A list with one concert to attend:

```
(define concerts1 (cons "Waterboys"
                        empty))
```

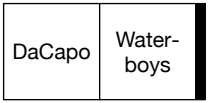
A new list just like `concerts1` but with a new concert at the beginning:

```
(define concerts2 (cons "DaCapo"
                        concerts1))
```



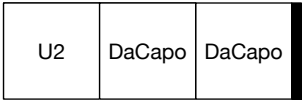
Another way to write `concerts2`:

```
(define concerts2alt (cons "DaCapo"
                           (cons "Waterboys"
                                 empty)))
```



A list with one U2 and two DaCapo concerts:

```
(define concerts3 (cons "U2"
                        (cons "DaCapo"
                              (cons "DaCapo"
                                    empty))))
```

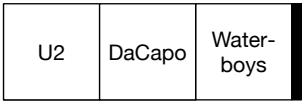


> Basic list constructs

- `empty`: A value representing an empty list.
- `(cons v lst)`: Consumes a value and a list; produces a new, longer list.
- `(first lst)`: Consumes a non-empty list; produces the first value.
- `(rest lst)`: Consumes a non-empty list; produces the same list without the first value.
- `(empty? v)`: Consumes a value; produces `true` if it is `empty` and `false` otherwise.
- `(cons? v)`: Consumes a value; produces `true` if it is a `cons` value and `false` otherwise.
- `(list? v)`: Equivalent to `(or (cons? v) (empty? v))`.

> Extracting values from a list

```
(define clst (cons "U2"
                   (cons "DaCapo" (cons "Waterboys" empty))))
```



First concert:

`(first clst)` ⇒ `"U2"`

Concerts after the first:

`(rest clst)` ⇒ `(cons "DaCapo" (cons "Waterboys" empty))`

Second concert:

`(first (rest clst))` ⇒ `"DaCapo"`

Exercise 1

Write a function `remove-second` that consumes a list of length at least 2, and produces a list containing the same items, with the second item removed.

```
(remove-second (cons 'Mercury (cons 'Venus empty))) ⇒ (cons 'Mercury
empty)
```

```
(remove-second (cons 2 (cons 4 (cons 6 (cons 0 (cons 1 empty)))))) ⇒
 (cons 2 (cons 6 (cons 0 (cons 1 empty))))
```

> Simple functions on lists

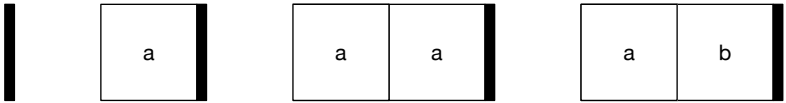
Using these built-in functions, we can write our own simple functions on lists.

```
;; (next-concert los) produces the next concert to attend or
;;     the empty string if los is empty
;; next-concert:  (listof Str) → Str
(check-expect (next-concert (cons "a" (cons "b" empty))) "a")
(check-expect (next-concert empty) "")

(define (next-concert los)
  (cond [(empty? los) ""]
        [else (first los)]))
```

```
;; (same-consec? los) determines if next two concerts are the same
;; same-consec?: (listof Str) → Bool
(check-expect (same-consec? empty) false)
(check-expect (same-consec? (cons "a" empty)) false)
(check-expect (same-consec? (cons "a" (cons "a" empty))) true)
(check-expect (same-consec? (cons "a" (cons "b" empty))) false)

(define (same-consec? los)
  (and (not (empty? los))
       (not (empty? (rest los)))
       (string=? (first los) (first (rest los)))))
```



Contracts involving lists

What is the contract for `(next-concert loc)`?

We could use "`List`" for `loc`.

However, we almost always need to answer the question "list of what?". A list of numbers? A list of strings? A list of any type at all?

> `(listof X)` notation in contracts

We'll use `(listof X)` in contracts, where `X` may be replaced with any type. For the concert list example in the previous slides, the list contains only strings and has type `(listof Str)`.

Other examples: `(listof Num)`, `(listof Bool)`, and `(listof Any)`.

Replace `X` with the most specific type available.

`(listof X)` always includes the empty list, `empty`.

Syntax and semantics: Values

List values are

- `empty`
- `(cons v l)`

where `v` is any Racket value (including list values) and `l` is a list value (which includes `empty`).

Note that values and expressions look very similar!

Value: `(cons 1 (cons 2 (cons 3 empty)))`

Expression: `(cons 1 (cons (+ 1 1) (cons 3 empty)))`

Racket list values are traditionally given using **constructor notation** – the same notation we would use to construct the value. We could represent list values differently (e.g. `[1, 2, 3]`), but choose not to.

Syntax and semantics: Expressions

The following are valid expressions:

- `(cons e1 e2)`, where `e1` and `e2` are expressions
- `(first e1)`
- `(rest e1)`
- `(empty? e1)`
- `(cons? e1)`
- `(list? e1)`

Syntax and semantics: Substitution rules

The substitution rules are:

- `(first (cons a b))` ⇒ `a`, where `a` and `b` are values.
- `(rest (cons a b))` ⇒ `b`, where `a` and `b` are values.
- `(empty? empty)` ⇒ `true`.
- `(empty? a)` ⇒ `false`, where `a` is any Racket value other than `empty`.
- `(cons? (cons a b))` ⇒ `true`, where `a` and `b` are values.
- `(cons? a)` ⇒ `false`, where `a` is any Racket value not created using `cons`.

Data defs & templates

Most interesting functions will process the entire consumed list. How many concerts are on the list? How many times does `"Waterboys"` appear? Which artists are duplicated in the list?

The structure of a function often mirrors the structure of the data it consumes. As we encounter more complex data types, we will find it useful to be precise about their structures.

We will do this by developing **data definitions**.

We can even go so far as developing function **templates** based on the data definitions of the values it consumes.

> List data definition

Informally: a list of strings is either empty, or consists of a **first** string followed by a list of strings (the **rest** of the list).

```
;; A (listof Str) is one of:
;; ⋆ empty
;; ⋆ (cons Str (listof Str))
```

This is a **recursive** data definition; the definition refers to itself. A **base** case does not refer to itself. A recursive (self-referential) case does refer to itself.

We can use this data definition to show rigourously that `(cons "a" (cons "b" empty))` is a `(listof Str)`.

We can generalize lists of strings to other types by using an X:

```
;; A (listof X) is one of:
;; ⋆ empty
;; ⋆ (cons X (listof X))
```

One of the main ideas of the HtDP textbook is that the form of a program often mirrors the form of the data.

A **template** is a general framework within which we fill in specifics.

We create a template once for each new form of data, and then apply it many times in writing functions that consume that type of data.

A template is derived from a data definition.

> Template for processing a `(listof X)`

We start with the data definition for a `(listof X)`:

```
;; A (listof X) is one of:
;; ⋆ empty
;; ⋆ (cons X (listof X))
```

A function consuming a `(listof X)` will need to distinguish between these two cases.

```
;; A (listof X) is one of:
;; ⋆ empty
;; ⋆ (cons X (listof X))
```

```
;; listof-X-template: (listof X) → Any
(define (listof-X-template lox)
  (cond [(empty? lox) ...]
        [(cons? lox) ...]))
```

The ... represents a place to fill in code specific to the problem.

In the last case we **know** from the data definition that there is a first X and the rest of the list of X's, so...

```
;; listof-X-template: (listof X) → Any
(define (listof-X-template lox)
  (cond [(empty? lox) ...]
        [(cons? lox) (... (first lox) ... (rest lox) ...)]))
```

Now we go a step further.

Because `(rest lox)` is of type `(listof X)`, we apply the same computation to it – that is, we apply `listof-X-template`.

```
;; listof-X-template: (listof X) → Any
(define (listof-X-template lox)
  (cond [(empty? lox) ...]
        [(cons? lox) (... (first lox) ...
                          (listof-X-template (rest lox)) ...)]))
```

This is the template for a function consuming a `(listof X)`. Its form parallels the data definition.

We can now fill in the dots for a specific example – counting the number of concerts in a list.

Processing lists: how many concerts?

We begin with writing the purpose, examples, contract, and then copying the template and renaming the function and parameters.

```
;; (count-concerts loc) counts the number of concerts in loc
;; count-concerts: (listof Str) → Nat
(check-expect (count-concerts empty) 0)
(check-expect (count-concerts (cons "a" (cons "b" empty))) 2)

(define (count-concerts loc)
  (cond [(empty? loc) ...]
        [else (... (first loc) ...
               ... (count-concerts (rest loc)) ...)]))
```

> Thinking about list functions

Here are three crucial questions to help think about functions consuming a list:

- What does the function produce in the base case?
- What does the function do to the first element in a non-empty list?
- How does the function combine the value produced from the first element with the value obtained by applying the function to the rest of the list?

```
;; (count-concerts loc) counts the number of concerts in loc
;; count-concerts: (listof Str) → Nat
(check-expect (count-concerts empty) 0)
(check-expect (count-concerts (cons "a" (cons "b" empty))) 2)

(define (count-concerts loc)
  (cond [(empty? loc) 0]
        [else (+ 1 (count-concerts (rest loc)))]))
```

This is a **recursive** function (it uses recursion).

A function is recursive when the body of the function involves an application of the same function.

This is an important technique which we will use quite frequently throughout the course.

Fortunately, our substitution rules allow us to trace such a function without much difficulty.

> Tracing count-concerts

```
 (count-concerts (cons "a" (cons "b" empty)))
⇒ (cond [(empty? (cons "a" (cons "b" empty)) 0]
        [else (+ 1 (count-concerts
                     (rest (cons "a" (cons "b" empty))))))])
⇒ (cond [false 0]
        [else (+ 1 (count-concerts
                     (rest (cons "a" (cons "b" empty))))))])
⇒ (cond [else (+ 1 (count-concerts
                     (rest (cons "a" (cons "b" empty))))))])
⇒ (+ 1 (count-concerts (rest (cons "a" (cons "b" empty)))))
⇒ (+ 1 (count-concerts (cons "b" empty)))
⇒ (+ 1 (cond [(empty? (cons "b" empty)) 0]
             [else (+ 1 (count-concerts (rest (cons "b" empty))))]))
```

```
⇒ (+ 1 (cond [false 0]
             [else (+ 1 (count-concerts (rest (cons "b" empty))))]))
⇒ (+ 1 (cond [else (+ 1 (count-concerts (rest (cons "b" empty))))]))
⇒ (+ 1 (+ 1 (count-concerts (rest (cons "b" empty)))))
⇒ (+ 1 (+ 1 (count-concerts empty)))
⇒ (+ 1 (+ 1 (cond [(empty? empty) 0]
                  [else (+ 1 (count-concerts (rest empty)))])))
⇒ (+ 1 (+ 1 (cond [true 0][else (+ 1 (count-concerts (rest empty)))])))
⇒ (+ 1 (+ 1 0))
⇒ (+ 1 1)
⇒ 2
```

Exercise 2

Write a recursive function `sum` that consumes a `(listof Int)` and returns the sum of all the values in the list.

`(sum (cons 6 (cons 7 (cons 42 empty))))` $\Rightarrow$ `55`

Consider:

- If I add up no items, what must the total be?
- If I have the first item, and a list containing all the other items, how many items do I have in total?

> Condensed traces

The full trace contains too much detail, so we instead use a **condensed trace** of the recursive function. This shows the important steps and skips over the trivial details.

This is a space saving tool we use in these slides, not a rule that you have to understand.

> The condensed trace of our example

```
(count-concerts (cons "a" (cons "b" empty)))
⇒ (+ 1 (count-concerts (cons "b" empty)))
⇒ (+ 1 (+ 1 (count-concerts empty)))
⇒ (+ 1 (+ 1 0))
⇒ 2
```

This condensed trace shows more clearly how the application of a recursive function leads to an application of the same function to a smaller list, until the base case is reached.

From now on, for the sake of readability, we will tend to use condensed traces. At times we will condense even more (for example, not fully expanding constants).

If you wish to see a full trace, you can use the Stepper.

But as we start working on larger and more complex forms of data, it becomes harder to use the Stepper, because intermediate expressions are so large.

Termination

It's important that our functions always **terminate** (stop running and produce an answer). Why does `count-concerts` always terminate?

There are two conditions. Either

- it's the base case, which produces `0` and immediately terminates
- or, it's the recursive case which applies `count-concerts` **to a shorter list**. Each recursive application is to a shorter list, which must eventually become empty and terminate.

We will eventually generalize "a shorter list" to "a smaller version of the same problem" where "a smaller version" depends on the nature of the problem. Perhaps a smaller number terminating at 0 or fewer elements that meet a certain criteria.

Does this remind you of induction? It should!

Thinking recursively

The similarity of recursion to induction suggests a way to think about developing recursive functions.

- Get the base case right.
- **Assume** that your function correctly solves a problem of size n (e.g. a list with n items).
- Figure out how to use that solution to solve a problem of size $n + 1$.

> Example: count-waterboys

```
;; (count-waterboys los) produces the number of occurrences
;;     of "Waterboys" in los
;; count-waterboys: (listof Str) → Nat
;; Examples:
(check-expect (count-waterboys empty) 0)
(check-expect (count-waterboys (cons "Waterboys" empty)) 1)
(check-expect (count-waterboys (cons "DaCapo"
                                     (cons "U2" empty))) 0)

(define (count-waterboys los) ...)
```

The template is a good place to start writing code. Write the template. Then, alter it according to the specific function you want to write.

For instance, we can generalize `count-waterboys` to a function which also consumes the string to be counted.

```
;; count-string: Str (listof Str) → Nat
(define (count-string s los) ...)
```

The recursive function application will be `(count-string s (rest los))`.

> More about list templates

Here are three crucial questions to help think about functions consuming a list **and filling in their templates**:

- What does the function produce in the base case? **Fill in that part of the template.**
- What does the function do to the first element in a non-empty list? **Fill in that part of the template.**
- How does the function combine the value produced from the first element with the value obtained by applying the function to the rest of the list? **Fill in that part of the template.**

> Refining the `(listof X)` template

Sometimes, each `X` in a `(listof X)` may require further processing. Indicate this with a template for `X` as a helper function.

```
;; listof-X-template: (listof X) → Any
(define (listof-X-template lox)
  (cond [(empty? lox) ...]
        [else (... (X-template (first lox)) ...
                   ... (listof-X-template (rest lox)) ...)]))
```

We assume this generic data definition and template from now on.

> Templates as generalizations

A template provides the basic shape of the code as suggested by the data definition.

Later in the course, we will learn about an abstraction mechanism (higher-order functions) that can reduce the need for templates.

We will also discuss alternatives for tasks where the basic shape provided by the template is not right for a particular computation.

Patterns of recursion

The list template has the property that the form of the code matches the form of the data definition.

We will call this **simple recursion**.

There are other patterns of recursion which we will see later on in the course.

Until we do, the functions we write (and ask you to write) will use simple recursion (and hence will fit the form described by such templates).

Use the templates.

> Simple recursion

In simple recursion, every argument in a recursive function application is either:

- unchanged, or
- *one step* closer to a base case according to a data definition

```
(define (func lst) ... (func (rest lst)) ...) ;; Simple
(define (func lst x) ... (func (rest lst) x) ...) ;; Simple
(define (func lst x) ... (func (process lst) x) ...) ;; NOT Simple
(define (func lst x)
 ... (func (rest lst) (math-function x)) ...) ;; NOT Simple
```

Useful list functions

A closer look at `count-concerts` reveals that it will work just fine on any list.

In fact, it is a built-in function in Racket, under the name `length`.

Another useful built-in function is `member?`, which consumes an element of any type and a list, and returns `true` if the element is in the list, or `false` if it is not present.

Producing lists from lists

Consider `negate-list`, which consumes a list of numbers and produces the same list with each number negated (3 becomes -3).

```
;; (negate-list lon) produces a list with every number in lon negated
;; negate-list: (listof Num) → (listof Num)
(check-expect (negate-list empty) empty)
(check-expect (negate-list (cons 2 (cons -12 empty)))
              (cons -2 (cons 12 empty)))
```

Since `negate-list` consumes a `(listof Num)`, we use the general list template to write it.

> negate-list with template

```
;; (negate-list lon) produces a list with every number in lon negated
;; negate-list: (listof Num) → (listof Num)
;; Examples:
(check-expect (negate-list empty) empty)
(check-expect (negate-list (cons 2 (cons -12 empty)))
              (cons -2 (cons 12 empty)))

(define (negate-list lon)
  (cond [(empty? lon) ...]
        [else (... (first lon) ... (negate-list (rest lon)) ... )]))
```

> negate-list completed

```
;; (negate-list lon) produces a list with every number in lon negated
;; negate-list: (listof Num) → (listof Num)
;; Examples:
(check-expect (negate-list empty) empty)
(check-expect (negate-list (cons 2 (cons -12 empty)))
              (cons -2 (cons 12 empty)))

(define (negate-list lon)
  (cond [(empty? lon) empty]
        [else (cons (- (first lon)) (negate-list (rest lon)))]))
```

> A condensed trace

```
 (negate-list (cons 2 (cons -12 empty)))
⇒ (cons (- 2) (negate-list (cons -12 empty)))
⇒ (cons -2 (negate-list (cons -12 empty)))
⇒ (cons -2 (cons (- -12) (negate-list empty)))
⇒ (cons -2 (cons 12 (negate-list empty)))
⇒ (cons -2 (cons 12 empty))
```

Exercise 3

Write a recursive function `keep-evens` that consumes a `(listof Int)` and returns the list of even values.

```
(keep-evens (cons 4 (cons 5 (cons 8 (cons 10 (cons 11 empty))))))
 (cons 4 (cons 8 (cons 10 empty)))
```

⇒

```
(keep-evens (cons 5 empty))
```

⇒ `empty`

```
(keep-evens (cons 4 empty))
```

⇒ `(cons 4 empty)`

> Non-empty lists

Sometimes a given computation makes sense only on a non-empty list — for instance, finding the maximum of a list of numbers.

Exercise: create a self-referential data definition for `(ne-listof X)`, a non-empty list of X. Develop a template for a function that consumes an `(ne-listof X)`. Finally, write a function to find the maximum of a non-empty list of numbers.

Design recipe refinements

When we introduce new types, like `(ne-listof X)`, we need to include it in the design recipe.

For each new type, place the following someplace between the top of the program and the first place the new type is used:

- The data definition
- The template derived from that data definition

This information is only needed **once**.

> Data definitions

Example:

```
;; A (listof X) is one of:
;; ⋆ empty
;; ⋆ (cons X (listof X))
```

In a self-referential data definition, at least one clause (possibly more) must not refer back to the definition itself; these are base cases.

Assignments do **not** need to include data definitions or templates for `(listof X)`. You do for `(ne-listof X)` and other types you may define.

The template follows directly from the data definition.

The overall shape of a self-referential template will be a `cond` expression with one clause for each clause in the data definition.

Self-referential data definition clauses lead to recursive expressions in the template.

Base case clauses will not lead to recursion.

The *per-function* part of the design recipe stays as before.

Exercise 4

Write a recursive function `list-max` that consumes a nonempty `(listof Int)` and returns the largest value in the list.

Strings and lists of characters

Processing text is an extremely common task for computer programs. Text is usually represented in a computer by strings.

In Racket (and in many other languages), a string is really a sequence of characters in disguise.

Racket provides the function `string->list` to convert a string to an explicit list of characters.

The function `list->string` does the reverse: it converts a list of characters into a string.

Racket's notation for the character 'a' is `#\a`.

The result of evaluating `(string→list "test")` is
the list `(cons #\t (cons #\e (cons #\s (cons #\t empty))))`.

This is unfortunately ugly, but the `#` notation is part of a more general way of specifying values in Racket.

> Counting characters in a string

Write a function to count the number of occurrences of a specified character in a string. Start by counting the occurrences in a list of characters.

```
;; (count-char/list ch loc) counts the number of occurrences
;;     of ch in loc.
;; count-char/list: Char (listof Char) → Nat
(check-expect (count-char/list #\e (string→list "")) 0)
(check-expect (count-char/list #\e (string→list "beekeeper")) 5)
(check-expect (count-char/list
        #\o (cons #\f (cons #\o (cons #\o (cons #\d empty))))) 2)
```

```
;; (count-char/list ch loc) counts the number of occurrences
;;     of ch in loc.
;; count-char/list: Char (listof Char) → Nat
(check-expect (count-char/list #\e (string→list "")) 0)
(check-expect (count-char/list #\e (string→list "beekeeper")) 5)

(define (count-char/list ch loc)
  (cond [(empty? loc) 0]
        [else (+ (cond [(char=? ch (first loc)) 1]
                       [else 0])
                 (count-char/list ch (rest loc)))]))
```

Exercise 5

Write a function `e→*` that consumes a `(listof Char)`, and replaces each `#\e` with a `#\*`.

```
(check-expect (e→* (cons #\h (cons #\e (cons #\y (cons #\! empty)))))
              (cons #\h (cons #\* (cons #\y (cons #\! empty)))))
```

Our functions should be easy to use. The problem statement was to count characters in a string, not in a list of characters.

We shouldn't expect the user of our function to know that to use `count-char/list` they need to convert their string to a list of characters.

In such cases it's good practise to include a **wrapper function** – a simple function that "wraps" the main function and takes care of housekeeping details like converting the string to a list.

```
;; (count-char ch s) counts the number of occurrences
;;     of ch in s.
;; count-char: Char Str → Nat
(check-expect (count-char #\e "") 0)
(check-expect (count-char #\e "beekeeper") 5)

(define (count-char ch s)
  (count-char/list ch (string→list s)))
```

» Characteristics of wrapper functions

Wrapper functions:

- are short and simple
- always call another function that does much more
- sets up the appropriate conditions for calling the other function, usually by transforming one or more of its parameters or providing a starting value for one of its arguments

Exercise 6

Write a wrapper function `drop-e` that consumes a `Str`, and replaces each `e` with `*`.

```
(check-expect (drop-e "hello world, how are you?")
              "h*llo world, how ar* you?")
```

Goals of this module

- You should understand the data definitions for lists, how the template mirrors the definition, and be able to use the template to write recursive functions consuming this type of data.
- You should understand the additions made to the semantic model of Beginning Student to handle lists, and be able to do step-by-step traces on list functions.
- You should understand and use `(listof X)` notation in contracts.
- You should understand strings, their relationship to characters and how to convert a string into a list of characters (and vice-versa).