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 [3–13]
• Contracts involving lists [14–18]
• Processing lists: Data definitions & templates [19–40]
• Patterns of recursion [41–43]
• Producing lists from lists [44–48]
• Design recipe refinements [49–51]
• Strings and lists of characters [52-55]
• Wrapper functions [56-58]
Introducing lists

Numbers and Boolean values represent a single item.

But there are many circumstances in which we need to track many items: the names of all the students in a course, the weight of each bag loaded on an airplane, etc.

Furthermore, the amount of data *may change over time* (grow or shrink) without a predetermined maximum size (i.e. its size is *unbounded*).

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 structure* (i.e. 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 aValue aList), which adds aValue to the beginning of aList. The constant empty is the empty list.
Example lists with 0, 1 or 2 items

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))
Example lists with 2 or 3 items

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 aValue aList)**: Consumes `aValue` and `aList` producing a new, longer list.
- **(first lst)**: Consumes a nonempty list; produces the first value.
- **(rest lst)**: Consumes a nonempty list; produces the same list without the first value.
- **(empty? aValue)**: Consumes `aValue`; produces `true` if it is empty and `false` otherwise.
- **(cons? aValue)**: Consumes `aValue`; 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 "Waterboys"
                 (cons "DaCapo" (cons "Waterboys" empty))))

First concert:
(first clst) ⇒ "Waterboys"

Concerts after the first:
(rest clst) ⇒ (cons "DaCapo" (cons "Waterboys" empty))

Second concert:
(first (rest clst)) ⇒ "DaCapo"
Nested boxes visualization

`cons` can be thought of as producing value with two parts.

It can be visualized two ways. The first:

\[(\text{cons } "\text{Waterboys}" \text{ empty})\]

\[
\begin{array}{c|c}
"\text{Waterboys}" & \text{empty} \\
\end{array}
\]

\[(\text{cons } "\text{DaCapo}" (\text{cons } "\text{Waterboys}" \text{ empty}))\]

\[
\begin{array}{c|c}
\text{first} & \text{rest} \\
"\text{DaCapo}" & \begin{array}{c|c}
\text{first} & \text{rest} \\
"\text{Waterboys}" & \text{empty} \\
\end{array}
\end{array}
\]
(cons "Waterboys"
  (cons "DaCapo"
    (cons "Waterboys"
      empty))))
Box-and-pointer visualization

(cons "Waterboys" empty)

(cons "DaCapo" (cons "Waterboys" empty))

(cons "Waterboys" (cons "DaCapo" (cons "Waterboys" empty))))
Simple functions on lists: next-concert

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

`; (next-concert loc) produces the next concert to attend or false if loc is empty
(check-expect (next-concert (cons "a" (cons "b" empty))) "a")
(check-expect (next-concert empty) false)

(define (next-concert loc)
  (cond [(empty? loc) false]
        [else (first loc)]))
Simple functions on lists: \texttt{same-consec}

;; (same-consec? loc) determines if next two concerts are the same
(check-expect (same-consec? (cons "a" (cons "b" empty))) false)
(check-expect (same-consec? (cons "a" (cons "a" empty))) true)
(check-expect (same-consec? (cons "a" empty)) false)

(define (same-consec? loc)
  (and (not (empty? loc))
       (not (empty? (rest loc)))
       (string=? (first loc) (first (rest loc))))))
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.
(anyof . . . ) notation in contracts

What about the value produced by next-concert? It might be a string or it might be false.

Use (anyof X Y . . . ) to mean any of the listed types or values.

Examples:

- (anyof Num Str)
- (anyof Str Num Bool)
- (anyof 1 2 3)
- (listof (anyof Str 'SomeSymbol))
Syntax for lists

*Lists are values* (i.e. they cannot be simplified).

The following are valid expressions:

- `empty`
- `(cons aValue aList)`
- `(first (cons aValue aList))`
- `(rest (cons aValue aList))`
- `(empty? e)`
- `(cons? e)`

Here *e* is any Racket expression, *aValue* is any Racket value (including lists) and *aList* is a list (which includes `empty`).
Semantics for lists

The substitution rules are:

- \((\text{first } (\text{cons } a\text{Value } a\text{List})) \Rightarrow a\text{Value}\).
- \((\text{rest } (\text{cons } a\text{Value } a\text{List})) \Rightarrow a\text{List}\).
- \((\text{empty? } \text{empty}) \Rightarrow \text{true}\).
- \((\text{empty? } a\text{Value}) \Rightarrow \text{false}\), where \(a\text{Value}\) is any Racket value other than \text{empty}.
- \((\text{cons? } (\text{cons } a\text{Value } a\text{List})) \Rightarrow \text{true}\).
- \((\text{cons? } a\text{Value}) \Rightarrow \text{false}\), where \(a\text{Value}\) is any Racket value not created using \text{cons}.
Processing lists: 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**, i.e. it has a *base case* and a **recursive (self-referential) case**.

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))
Templates and data-directed design

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 \((\text{listof } X)\)

We start with the data definition for a \((\text{listof } X)\):

\[
;; \text{ A (listof } X) \text{ is one of:}

;; \ast \text{ empty}

;; \ast (\text{cons } X \text{ (listof } X))
\]

A function consuming a \((\text{listof } X)\) will *need to distinguish between these two cases.*
Template for processing a (listof X)

Data Definition

;; A (listof X) is one of:

;; ★ empty

;; ★ (cons X (listof X))

Template

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

The . . . represents a place to fill in code for what expression to return in the case of an empty or a non-empty list.

In the last case (i.e. (cons? lox)) we know from the data definition that there is a first X and the rest of the list of X’s, so ...
Template for processing a (listof X) refined

;;; 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.
Template for processing a (listof X) refined

;; 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.
Example: 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:

1. What does the function produce in the *base case*?

2. What does the function produce for the *first element* in a non-empty list?

3. 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*?
Example: how many concerts?

;; (count-concerts los) counts the number of concerts in los
;; count-concerts: (listof Str) → Nat
(check-expect (count-concerts empty) 0)
(check-expect (count-concerts (cons "a" (cons "b" empty)))) 2)
(define (count-concerts los)
  (cond [(empty? los) 0]
        [(else (+ 1 (count-concerts (rest los)))]))

The only parts not in the template are: 0 and + 1.

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.