Introducing lists
Contracts involving lists
Processing lists: Data definitions & templates
Patterns of recursion
Producing lists from lists
Design recipe refinements
Strings and lists of characters

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.
Introducing lists
Introduction

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.

Furthermore, the amount of data is often unbounded, meaning it may grow or shrink – and you do not know how much. The order of values may also be important.

Most programming languages meet this need with lists.
Naïve definition

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

• A list of 3 concerts is a concert followed by a list of 2 concerts.

\[
\text{List of 3 concerts} = \text{Kamelot @ The Phoenix} + \text{List of 2 concerts}
\]
Naïve definition

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

\[
\text{List of 2 concerts} = \text{Delain @ Opera House} + \text{List of 1 concert}
\]
Naïve definition

A list is a **recursive structure** – it is defined in terms of a smaller list. Consider a list of 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 call it the **empty list**.

```
List of 1 concert = Eluveitie @ The REBEL + empty
```
Naïve definition

A list is a **recursive structure** – it is defined in terms of a smaller list. Consider a list of 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 call it the **empty list**.

Lists are created with \((\text{cons} \ v \ \text{lst})\), which adds a value \(v\) to the beginning of list \(\text{lst}\). The empty list is simply called **empty**.

(That means **empty** is technically a list as well!)
Examples

No upcoming concerts to attend:
(define concerts0 empty)
Examples

No upcoming concerts to attend:
(define concerts0 empty)

A list with one concert to attend:
(define concerts1 (cons "Kamelot" empty))

concerts1 = Kamelot + empty
Examples

No upcoming concerts to attend:
(\textit{define concerts0 empty})

A list with one concert to attend:
(\textit{define concerts1 (cons "Kamelot" empty)})

\textit{cons} tells Racket to create a list
"Kamelot" is the value added to the list
\textit{empty} is the list to which "Kamelot" is added

\textcolor{red}{\textbf{concerts1}} \quad = \quad \textcolor{orange}{\textbf{Kamelot}} \quad + \quad \textcolor{blue}{\textbf{empty}}
Examples

No upcoming concerts to attend:
(\texttt{define concerts0 empty})

A list with one concert to attend:
(\texttt{define concerts1 (cons "Kamelot" empty)})

A new list just like \texttt{concerts1} but with a new concert at the beginning:
(\texttt{define concerts2 (cons "Delain" concerts1)})

\[
\texttt{concerts2} = \texttt{Delain} + \texttt{concerts1}
\]

\[
\texttt{concerts1} = \texttt{Kamelot} + \texttt{empty}
\]
Examples

No upcoming concerts to attend:
(\texttt{define concerts0 empty})

A list with one concert to attend:
(\texttt{define concerts1 (cons "Kamelot" empty)})

A new list just like \texttt{concerts1} but with a new concert at the beginning:
(\texttt{define concerts2 (cons "Delain" concerts1)})

\texttt{concerts2} = \texttt{Delain} + \texttt{Kamelot} + \texttt{empty}
Examples

No upcoming concerts to attend:
(\texttt{define concerts0 empty})

A list with one concert to attend:
(\texttt{define concerts1 (cons "Kamelot" empty)})

A new list just like \texttt{concerts1} but with a new concert at the beginning:
(\texttt{define concerts2 (cons "Delain" concerts1)})

Another way to write \texttt{concerts2}:
(\texttt{define concerts2alt (cons "Delain"
           (cons "Kamelot" empty)))}

\[
\begin{array}{c}
\text{concerts2alt} = \quad \text{Delain} + \quad \text{Kamelot} + \quad \text{empty}
\end{array}
\]
Examples

No upcoming concerts to attend:

\[(\text{define concerts0 empty})\]

A list with one concert to attend:

\[(\text{define concerts1 (cons "Kamelot" empty)})\]

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

\[(\text{define concerts2 (cons "Delain" concerts1)})\]

Another way to write concerts2:

\[(\text{define concerts2alt (cons "Delain" (cons "Kamelot" empty)))}\]

\[
\text{concerts2alt} = \text{Delain} + \text{Kamelot} + \text{empty}
\]
Examples (cont)

A list with one HammerFall and two Nightwish concerts:

```
(define concerts3 (cons "Nightwish"
         (cons "HammerFall"
         (cons "Nightwish" empty))))
```
Visualization – Nested boxes

(cons v lst) consumes a value v and a list lst and produces a new, longer list. It can be visualized two ways: the first one uses nested boxes.
Visualization – Nested boxes

(cons v lst) consumes a value v and a list lst and produces a new, longer list. It can be visualized two ways: the first one uses nested boxes.

(cons "Kamelot" empty)
Visualization – Nested boxes

\((\text{cons } v \ 1\text{st})\) consumes a value \(v\) and a list \(1\text{st}\) and produces a new, longer list. It can be visualized two ways: the first one uses nested boxes.

\((\text{cons } "Kamelot" \ \text{empty})\)

![Nested boxes visualization for (cons "Kamelot" empty)]

\((\text{cons } "\text{Delain}" \ (\text{cons } "\text{Kamelot}" \ \text{empty}))\)

![Nested boxes visualization for (cons "Delain" (cons "Kamelot" empty))]
Visualization – Nested boxes

\((\text{cons } v \ 1\text{st})\) consumes a value \(v\) and a list \(1\text{st}\) and produces a new, longer list. It can be visualized two ways: the first one uses nested boxes.

\((\text{cons } "\text{Kamelot}" \ \text{empty})\)

\begin{align*}
\text{first} & : \text{Kamelot} \\
\text{rest} & : \text{empty}
\end{align*}

\((\text{cons } "\text{Delain}" \ (\text{cons } "\text{Kamelot}" \ \text{empty}))\)

\begin{align*}
\text{first} & : \text{Delain} \\
\text{rest} & : \begin{align*}
\text{first} & : \text{Kamelot} \\
\text{rest} & : \text{empty}
\end{align*}
\end{align*}

\((\text{cons } "\text{Eluveitie}" \ (\text{cons } "\text{Delain}" \ (\text{cons } "\text{Kamelot}" \ \text{empty})))\)

\begin{align*}
\text{first} & : \text{Eluveitie} \\
\text{rest} & : \begin{align*}
\text{first} & : \text{Delain} \\
\text{rest} & : \begin{align*}
\text{first} & : \text{Kamelot} \\
\text{rest} & : \text{empty}
\end{align*}
\end{align*}
\end{align*}
Visualization – Boxes and pointers

(cons v 1st) consumes a value and a list and produces a new, longer list. It can be visualized two ways: the second one uses boxes and pointers.
Visualization – Boxes and pointers

\((\text{cons} \ v \ \text{1st})\) consumes a value and a list and produces a new, longer list. It can be visualized two ways: the second one uses boxes and pointers.

\((\text{cons} \ "\text{Kamelot}" \ \text{empty})\)
Visualization – Boxes and pointers

\((\text{cons} \hspace{1em} v \hspace{1em} \text{1st})\) consumes a value and a list and produces a new, longer list. It can be visualized two ways: the second one uses boxes and pointers.

\((\text{cons} \hspace{1em} "\text{Kamelot}" \hspace{1em} \text{empty})\)

\((\text{cons} \hspace{1em} "\text{Delain}" \hspace{1em} (\text{cons} \hspace{1em} "\text{Kamelot}" \hspace{1em} \text{empty}))\)
(cons v 1st) consumes a value and a list and produces a new, longer list. It can be visualized two ways: the second one uses boxes and pointers.

(cons "Kamelot" empty)

(cons "Delain" (cons "Kamelot" empty))

(cons "Eluveitie" (cons "Delain" (cons "Kamelot" empty)))
Basic 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)**: Predicate that consumes a value v; produces true if it is **empty** and false otherwise.

**(cons? v)**: Predicate that consumes a value v; produces true if it is a **cons** value and false otherwise.
Basic constructs

**empty**: a value representing an empty list
Basic constructs

(cons v lst): Consumes a value and a list; produces a new, longer list.

(define lst (cons "Delain" (cons "Kamelot" empty)))
Basic constructs

(first lst): Consumes a non-empty list; produces the first value.

(define lst (cons "Delain" (cons "Kamelot" empty)))
(first lst) => "Delain"

```
  first  rest
  Delain
      first  rest
         first Kamelot rest empty
```
Basic constructs

(rest 1st): Consumes a non-empty list; produces the same list without the first value.

(define lst (cons "Delain" (cons "Kamelot" empty)))
(rest lst) => (cons "Kamelot" empty)
(rest (rest lst)) => empty
Basic constructs

\[(\text{empty? } v)\]: Predicate that consumes a value \(v\); produces true if \(v\) is empty and false otherwise.

\[
\text{(define lst (cons "Delain" (cons "Kamelot" empty)))}
\]
\[
(\text{empty? lst}) \Rightarrow \text{false}
\]
\[
(\text{empty? (rest (rest lst))}) \Rightarrow \text{true}
\]
Basic constructs

(cons? v): Predicate that consumes a value v; produces true if v is a cons value and false otherwise.

(define lst (cons "Delain" (cons "Kamelot" empty)))
(cons? lst) => true
(cons? (rest (rest lst))) => false
Basic constructs – Extracting values

(define clst
  (cons "Eluveitie" (cons "Delain" (cons "Kamelot" empty))))

First concert:
(first clst) => "Eluveitie"
Basic constructs – Extracting values

\[(\textbf{define} \ \textit{clst} \n\quad (\textbf{cons} \ "\textit{Eluveitie}" \ (\textbf{cons} \ "\textit{Delain}" \ (\textbf{cons} \ "\textit{Kamelot}" \ \textit{empty}))))\]

First concert:

\[(\textbf{first} \ \textit{clst}) \Rightarrow \ "\textit{Eluveitie}"\]
Basic constructs – Extracting values

(define clst
  (cons "Eluveitie" (cons "Delain" (cons "Kamelot" empty)))))

Concerts after the first:
(rest clst) => (cons "Delain" (cons "Kamelot" empty))

\[
\begin{array}{c|c}
\text{first} & \text{rest} \\
\hline
\text{Eluveitie} & \text{Delain} \\
\hline
\text{Kamelot} & \text{empty}
\end{array}
\]
Basic constructs – Extracting values

(define clst
  (cons "Eluveitie" (cons "Delain" (cons "Kamelot" empty))))

Concerts after the first:

(rest clst) => (cons "Delain" (cons "Kamelot" empty))
Basic constructs – Extracting values

(define clst
  (cons "Eluveitie" (cons "Delain" (cons "Kamelot" empty)))))

Second concert:
(first (rest clst)) => "Delain"
Basic constructs – Extracting values

(define clst
  (cons "Eluveitie" (cons "Delain" (cons "Kamelot" empty)))))

Second concert:
(first (rest clst)) => "Delain"
Basic constructs – Extracting values

(define clst
  (cons "Eluveitie" (cons "Delain" (cons "Kamelot" empty)))))

Second concert:

(first (rest clst)) => "Delain"
Custom functions

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 clst)
  (cond
    [(empty? clst) false]
    [else (first clst)]))
Custom functions

;; (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? clst)
  (and
    (not (empty? clst))
    (not (empty? (rest clst)))
    (string=? (first clst) (first (rest clst))))))
Contracts involving lists
Contracts

What is the contract for \(\text{next-concert } \text{clst}\)?

We could use “List” for \text{clst}.

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?
Contracts – (listof X) notation

We 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 include:
• (listof Num)
• (listof Bool), and
• (listof Any).

Replace X with the most specific type available.
(listof X) always includes the empty list, empty.
Contracts – (anyof ...) notation

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 include:
• (anyof Num Str)
• (anyof Str Num Bool)
• (anyof 1 2 3), and
• (listof (anyof Str 'SomeSymbol))