Lists

CS 5010 Program Design Paradigms
Lesson 4.1
How to represent info of arbitrary size?

• a phone book with many listings
• a space-invaders game with many invaders
• a presentation with many slides

• Each of these can be represented as a sequence of information items.
• There may be better ways for some of these, but we will start with sequences
• This is our first example of recursive data
Module 04

Data Representations
- Basics
- Mixed Data
- Recursive Data
  - Functional Data
  - Objects & Classes
  - Stateful Objects

Design Strategies
- Combine simpler functions
- Use a template
- Divide into Cases
- Call a more general function
- Communicate via State

Generalization
- Over Constants
- Over Expressions
- Over Contexts
- Over Data Representations
- Over Method Implementations
Outline for the rest of this week

• The arithmetic of lists
• Using the list template
• Lists of Structures
At the end of this lesson, you should be able to:

• Write down a data definition for information represented as a list

• Notate lists using constructor, list, and write notations.

• Explain how lists are represented as singly-linked data structures, and how `cons`, `first`, and `rest` work on these structures

• Calculate with the basic operations on lists: `cons`, `first`, and `rest`.
Lists: A Handy Representation for Sequences

• Sequences of data items arise so often that most programming languages have a standard way of representing them.

• Sequence information in Racket is represented by \textit{lists}.

• We’ll see lots of examples:
  – ListOfNumbers
  – ListOfDigits
  – ListOfStrings
  – ListOfBooks
Lists of Numbers

A List of Numbers (LON) is one of:
-- empty
-- (cons Number LON)

List data is a kind of mixed data. Just as we did in our previous data definitions, the data definitions for lists shows the constructor for each case. Here we have two constructors: the constant empty and the function cons. A list of numbers (or "LON") is either empty or the value built by applying cons to a number and another LON.

cons is built into Racket. We don’t need a define-structure for it.

There’s no interpretation here because these lists don’t mean anything (yet). They do not refer to any real-world information.
A List of Numbers (LON) is one of:
-- empty
-- (cons Number LON)

Here are some examples of LONs.

**empty** is a LON by the data definition.

**(cons 11 empty)** is a LON because 11 is a number and **empty** is a LON.

**(cons 22 (cons 11 empty))** is a LON because 22 is a number and **(cons 11 empty)** is a LON.

And so on.
Lists of Digits

A Digit is one of
"0" | "1" | "2" | ... | "9"

A List of Digits (LOD) is one of:
-- empty
-- (cons Digit LOD)

Let's do it again, this time with digits.
We define a Digit to be one of the strings "0", "1", etc., through "9".
A List of Digits (LOD) is either empty or the cons of a Digit and a List of Digits.
Examples of LODs

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

• These are not LODs:
  - (cons 4 (cons "2" (cons "3" empty)))
  - (cons (cons "3" empty)
  - (cons "2" (cons "3" empty)))

A List of Digits (LOD) is one of:
- -- empty
- -- (cons Digit LOD)
Lists of Books

A Book is a (make-book ...).

A List of Books (LOB) is one of:
-- empty
-- (cons Book LOB)

We can build lists of more complicated data items. Imagine we had a data definition for Book. Then we can define a List of Books in the same way as we did for lists of numbers or lists of digits: a List of Books is either empty or the cons of a Book and a List of Books.
Examples of LOBs

(define book1 (make-book ...))
(define book2 (make-book ...))
(define book3 (make-book ...))

(empty)

(cons book1 empty)
(cons book2 (cons book1 empty))
(cons book2 (cons book2 (cons book1 empty))

• Not a LOB: (Why?)

(cons 4 (cons book2 (cons book1 empty))

A List of Books (LOB) is one of:
-- empty
-- (cons Book LOB)
This data definition is self-referential

A List of Numbers (LON) is one of:
-- empty
-- (cons Number LON)

The data definition for LONs contains something we haven't seen before: self-reference.
The second constructor uses LON, even though we haven't finished defining LONs yet. That's what we mean by self-reference.
In normal definitions, this would be a problem: you wouldn't like a dictionary that did this.
But self-reference the way we've used it is OK. We've seen in the examples how this works: once you have something that you know is a LON, you can do a cons on it to build another LON. Since that's a LON, you can use it to build still another LON.
We also call this a recursive data definition.
This one is self-referential, too

A Digit is one of
  "0"  |  "1"  |  "2"  |  ...  |  "9"

A List of Digits (LOD) is one of:
  -- empty
  -- (cons Digit LOD)
How Lists Represent Sequences

• If X is some data definition, we define a list of X's as either empty or the cons of an X and a list of X's.
• So a list of sardines is either empty or the cons of a sardine and a list of sardines.
• The interpretation is always "a sequence of X's".
  – empty represents a sequence with no elements
  – (cons x lst) represents a sequence whose first element is x and whose other elements are represented by lst.
• If we had some information that we wanted to represent as a list of X's (say a list of people), we would have to specify the order in which the X's appear (say "in increasing order of height"), or else say “in any order.”
A ListOfX is one of
-- empty
  interp: a sequence of X's with no elements
-- (cons X ListOfX)
  interp: (cons x lst) represents a sequence of X's whose first element is x and whose other elements are represented by lst.
List Notation

• There are several ways to write down lists.
• We've been using the constructor notation, since that is the most important one for use in data definitions.
• The second most important notation we will use is list notation. In Racket, you can get your output in this notation by choosing the language "Beginning Student with List Abbreviations".
• Internally, lists are represented as singly-linked lists.
• On output, lists may be notated in write notation.
Examples of List Notation

Constructor notation: 

\[(\text{cons}\ 11 \text{\ (cons}\ 22 \text{\ (cons}\ 33 \text{\ empty}))\)]

List notation: 

\[(\text{list}\ 11\ 22\ 33)\]

Internal representation: 

[Diagram of linked list with nodes 11, 22, and 33]

write-style (output only): 

\[(11\ 22\ 33)\]
Implementation of **cons**

Now that we've seen the internal representation of lists, we can see how **cons** creates a new list: it simply adds a new node to the front of the list. This operation takes a short, fixed amount of time.

\[
\begin{align*}
\text{lst} &= (\text{list } 22\ 33) \\
(\text{cons } 11\ \text{lst}) &= (\text{list } 11\ 22\ 33)
\end{align*}
\]
Operations on Lists

empty? : ListOfX -> Boolean
Given a list, returns true iff the list is empty

Racket provides 3 functions for inspecting lists and taking them apart. These are empty?, first, and rest.

The predicate empty? returns true if and only if the list is empty.
Operations on Lists

**first** : List0fX -> X

**GIVEN:** a list

**WHERE:** the list is non-empty

**RETURNS:** its first element

When we write down the template for lists, we will see that when we call **first**, its argument will always be non-empty.
Operations on Lists

\[ \text{rest} : \text{ListOfX} \rightarrow \text{ListOfX} \]

**GIVEN:** a list

**WHERE:** the list is non-empty

**RETURNS:** the list of all its elements except the first

When we write down the template for lists, we will see that when we call `rest`, its argument will always be non-empty.
Examples

(empty? empty) = true
(empty? (cons 11 empty)) = false
(empty? (cons 22 (cons 11 empty))) = false

(first (cons 11 empty)) = 11
(rest (cons 11 empty)) = empty

(first (cons 22 (cons 11 empty))) = 22
(rest (cons 22 (cons 11 empty))) = (cons 11 empty)

(first empty) ➔ Error! (Precondition failed)
(rest empty) ➔ Error! (Precondition failed)
first and rest simply follow a pointer in the singly-linked data structure.

\[
\text{lst2} = (\text{list 11 22 33})
\]
\[
\text{(first lst2)} = 11
\]
\[
\text{(rest lst2)} = (\text{list 22 33})
\]
Properties of \texttt{cons}, \texttt{first}, and \texttt{rest}

\begin{align*}
\text{(first (cons v l))} & = v \\
\text{(rest (cons v l))} & = l
\end{align*}

If \( l \) is non-empty, then

\[
\text{(cons (first l) (rest l))} = l
\]

Here are some useful facts about \texttt{first}, \texttt{rest}, and \texttt{cons}. Can you see why they are true?

These facts tell us that if we want to build a list whose \texttt{first} is \( x \) and whose \texttt{rest} is \( \texttt{lst} \), we can do this by writing \( \texttt{(cons x lst)} \).
At this point, you should be able to:

• Write down a data definition for information represented as a list
• Notate lists using constructor, list, and write notations.
• Explain how lists are represented as singly-linked data structures, and how `cons`, `first`, and `rest` work on these structures
• Calculate with the basic operations on lists: `cons`, `first`, and `rest`.
Next Steps

• If you have questions about this lesson, ask them on the Discussion Board
• Do Guided Practices 4.1 and 4.2
• Go on to the next lesson