05.1 Using Lists

Bookmark this page

In this lesson, we'll see how to put lists together, and how to extract the information from a pre-existing list. We'll also talk a little bit about ways to visualize lists that can help with intuition and, later in your programming career, with understanding how computers represent lists internally.

A key concept is that we don't need a complicated new type to handle every possible list length. Instead, we use the elegant idea that if we have some list with n things in it, we can add one more thing to obtain a new list with $n+1$ things. So in fact, we need just two new ideas: a way to describe a list with nothing in it, and a way to add one new thing to an existing list.

The empty list

We call a list with nothing in it the **empty list**. There's really only one empty list in the universe, and everyone who wants to build a list can use it as a starting point. We refer to it by using the special named constant `empty`:

```
> empty
empty
```

The constant `empty` isn't defined to contain a number, a string, or any other value. It's a bit like `true`: it's a special defined value that means only itself. In fact, `empty` is the only member of a special type, as we can see by experimenting with the `empty?` type predicate:

```
> (empty? empty)
true
> (empty? 5) ;; not an Int
false
> (empty? 'empty) ;; not a Sym
false
> (empty? "empty") ;; not a Str
false
> (empty? false) ;; not a Bool
false
```

Note: Racket will also sometimes use `'()` instead of `empty` to display the empty list. The reasons why both `empty` and `'()` are valid are beyond the scope of this course. In these lessons we'll use `empty` exclusively. You can use either in the code you write, and you can select which one DrRacket prefers by clicking the "Show Details" button in the "Choose Language..." dialog box.

"Consing" a list

Consider the following two informal rules (which we'll make more formal soon):

1. The empty list is a list.
2. Given an existing list, you can build a new list that has one more item added at the beginning.

We already have the constant `empty` to handle the first rule. For the second rule we introduce a built-in function `cons`, short for "construct". Suppose that a constant `lst` holds a previously constructed list, which we'll think of mathematically as $(v_1, v_2, \ldots, v_n)$. Let v be any other Racket value. Then the expression `(cons v lst)` will produce the new list $(v, v_1, v_2, \ldots, v_n)$. In general, `cons` consumes two arguments: the first is of any type at all, and the second must be a list. It produces a new list as a result.

Using `cons`, we can construct lists of any length:

```
;; empty is a list by definition (Rule 1), so we're allowed to
;; cons 5 onto it (Rule 2)
(define lst1 (cons 5 empty))
;; We know lst1 is itself a list, so we can cons onto it too.  Note that
;; lists can hold any types at all, even a mix -- here we added a string
;; to a list containing an integer.
(define lst2 (cons "hello" lst1))
;; And of course, we can keep going with a Boolean
(define lst3 (cons true lst2))
;; We don't need to define all our sub-lists as constants.  Here's a
;; single expression that conses together a list of four symbols.
(define suits
  (cons 'spades
    (cons 'hearts
      (cons 'diamonds
        (cons 'clubs empty)))))
```

Using a chain of `cons` applications to build a list of values is sometimes called "consing a list".

Like the constant `empty`, the `cons` function produces a special value of a new type, which we'll sometimes call a **cons cell**. We can verify that a cons cell is a new type by experimenting with its corresponding type predicate, `cons?`:

```
> (cons? empty) ;; the empty list isn't a cons
false
> (cons? 123) ;; other values aren't either
false
> (cons? lst3) ;; but our previously constructed list lst3 is
true
> (cons? suits) ;; and so is this list
true
> (empty? suits) ;; a non-empty list isn't empty!
false
```

Sometimes it's useful to know that a value "has list nature", i.e., it's either an empty or non-empty list, as opposed to some other value like a number. We can imagine writing our own type predicate for that:

```
(define (my-list? val)
  (or (empty? val) (cons? val)))
```

It turns out we don't have to write this function, because there's already a built-in function `list?` that does this:

```
> (list? empty)
true
> (list? lst2)
true
> (list? "I'm a list")
false
```

Concept Check 05.1.1

1/1 point (graded)

How many values are there in the list `(cons 4 empty)` ?

- ( ) 0
- (•) 1
- ( ) 2
- ( ) 3
- ( ) 4

✔

Save Show answer

Submit You have used 1 of 2 attempts

Concept Check 05.1.2

Mirepoix is a concoction created by lightly cooking aromatic vegetables in butter. It's an important flavouring component of soups and stews. The most typical recipe uses onions, carrots, and celery.

Define a constant `mirepoix` to be a list containing the three strings `"onion"`, `"carrot"`, and `"celery"`, in any order.

1/1 points
Attempts: 2 / Unlimited

```
(define mirepoix (cons "onion" (cons "carrot" (cons "celery" empty))))
```

Submit Code Reset Code

✔ Correct! 1/1 points

Deconstructing lists

You might think that because we can construct lists of any length, we'll need lots of different functions to extract elements from those lists. But in fact, we need just two: `first` and `rest`. These built-in functions have very simple behaviours—given any non-empty list of the form `(cons v lst)`, we have

- `(first (cons v lst))` ⇒ `v`
- `(rest (cons v lst))` ⇒ `lst`

Concept Check 05.1.3

1/1 point (graded)

What is the value of the following expression?

```
(first (cons false (cons true empty)))
```

- ( ) `first`
- ( ) `cons`
- (•) `false`
- ( ) `true`
- ( ) `empty`

✔

Save Show answer

Submit You have used 1 of 2 attempts

Concept Check 05.1.4

1/1 point (graded)

What is the value of the following expression?

```
(rest (cons false (cons true empty)))
```

- ( ) `false`
- ( ) `true`
- ( ) `empty`
- ( ) `cons`
- (•) `(cons true empty)`

✔

Save Show answer

Submit You have used 1 of 2 attempts

Of course, `lst` can itself be a list of any length (including empty); `rest` just produces everything beyond the first element as its result. Let's see that in action:

```
> (first suits)
'spades
> (rest suits)
(cons 'hearts (cons 'diamonds (cons 'clubs empty)))
```

It's illegal to use `first` or `rest` on the empty list:

```
> (first empty)
first: expects a non-empty list; given: empty
> (rest empty)
rest: expects a non-empty list; given: empty
```


But wait—how do we reach deeper into a list and extract elements past the first? We can accomplish by *combining* applications of first and rest. We just have to make sure to apply the functions in the correct order.

Concept Check 05.1.5

1/1 point (graded)

Recall our definition of `suits` above:

```
(define suits
  (cons 'spades
    (cons 'hearts
      (cons 'diamonds
        (cons 'clubs empty)))))
```

Which of the following expressions will produce the value `'hearts` ?

- ( ) `(first (first suits))`
- (•) `(first (rest suits))`
- ( ) `(rest (first suits))`
- ( ) `(rest (rest suits))`
- ( ) `(empty? (cons? suits))`

✔

Save Show answer

Submit You have used 1 of 2 attempts

Hopefully you can see the natural pattern that emerges from the combination of `first` and `rest` above: we can extract a value from any position in a list using a chain of `rest`s terminated by a final `first`. In fact, there are already built-in functions `second`, `third`, ..., `eighth` that follow the lead of `first`. But perhaps surprisingly, we almost never need to look beyond a list's second element in practice—as we'll see, recursion will do most of the work in helping us work our way down a list to the end.

Concept Check 05.1.6

Write a Racket predicate `starts-with-3?` that consumes a list of numbers `lon` and produces `true` if the list starts with the number `3`, and `false` otherwise. Remember that the list must also be non-empty for this to be possible.

1/1 points
Attempts: 1 / Unlimited

```
(define (starts-with-3? lon)
  (cond
    [(empty? lon) false]
    [(= 3 (first lon)) true]
    [else false]))
```

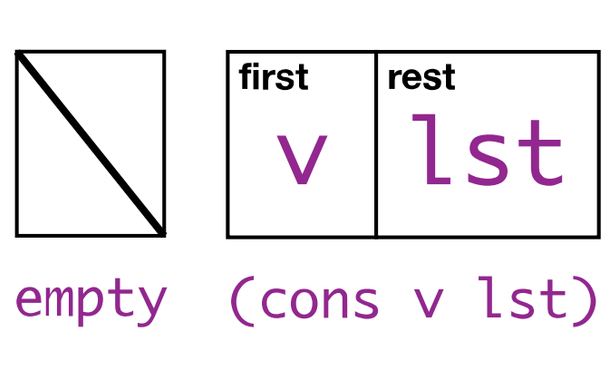
Submit Code Reset Code

✔ Correct! 1/1 points

Visualizing lists

Sometimes, it can be useful to build a mental image of the structure of a list. A mental picture (or a real picture that you draw on paper) can empower us to use spatial intuition when working with a list, which can help more and more as our lists get more complicated over the coming modules. Pictures can also give us some insight into how Racket deals with lists, both conceptually and in practice in the computer's memory. Therefore, we introduce a couple of simple **list visualizations**. These visualizations won't allow us to solve problems we couldn't solve before, and we won't ask about them often; but we'll sometimes offer a visualization as part of talking about lists of different kinds.

We use two fundamental building blocks to visualize lists, as shown here:



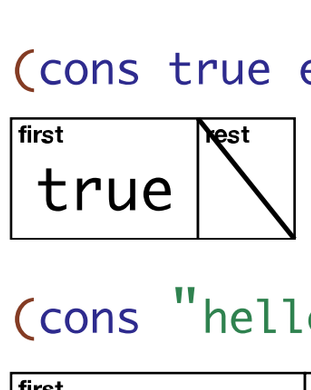
We always represent the empty list using a box with a slash through it. When we want to visualize a cons, we draw it as a box with two compartments, labeled "first" and "rest". We will use the compartments to visualize the contents of this cons cell in two different ways.

Visualization using nested boxes

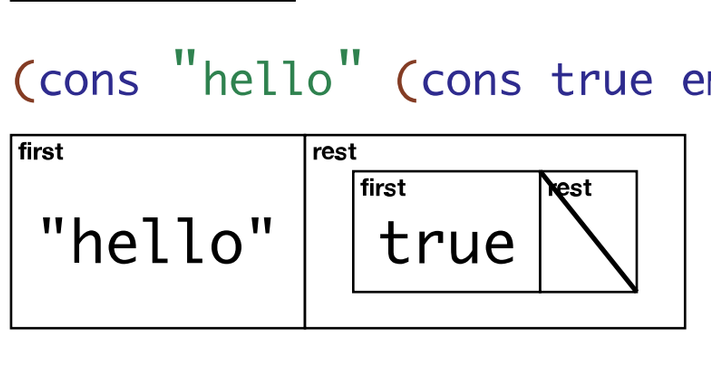
Fig 1. Matryoshka dolls

In a **nested visualization**, we draw boxes within boxes like matryoshka. In particular, if we have an expression of the form `(cons v lst)`, we first create visualizations for v and `lst`, and then place the two visualizations inside a new cons box large enough to hold them in its compartments. With that in mind, here are visualization of a few of the lists created above:

`(cons true empty)`



`(cons "hello" (cons true empty))`



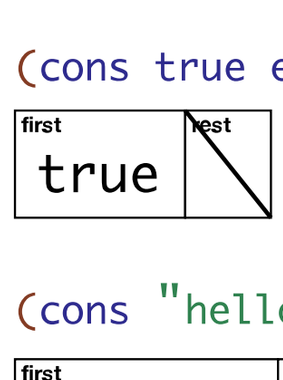
`(cons 5 (cons "hello" (cons true empty)))`

Visualization using boxes and arrows

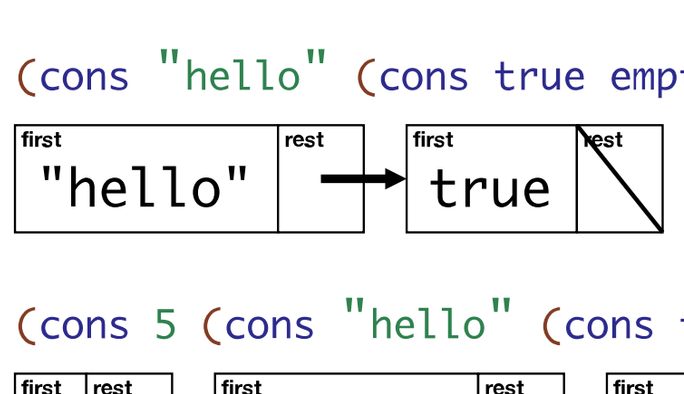
Fig 2. Passenger Train

The nested visualization is mathematically elegant, but with a long enough list, your visualization will either have to be gigantic, or have tiny nested cells towards the end. Besides—when we think about lists, we often imagine a parade of items in a line, not a nested structure. So, as an alternative, we sometimes use a **box-and-pointer visualization** for lists. Here, instead of nesting `lst` inside the box for `(cons v lst)`, we pull its box out and put it next to its parent, and then draw an arrow from parent to child to preserve the list's structure. The result looks like a train, where the train cars are the individual cons cells. Here are box-and-pointer visualizations of the same lists shown above:

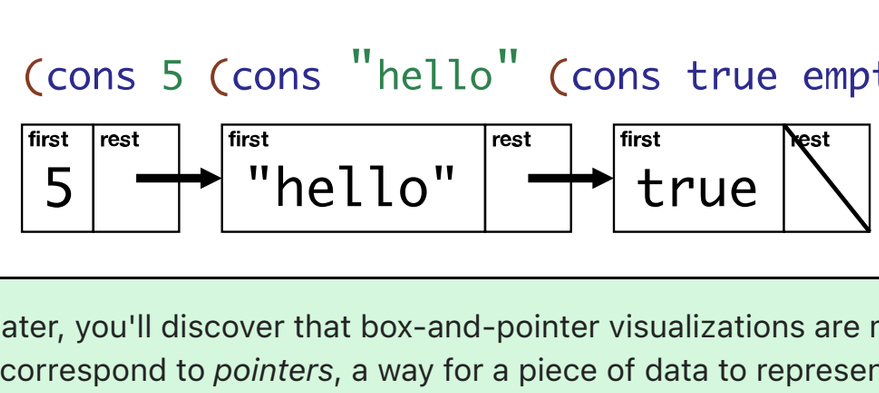
`(cons true empty)`



`(cons "hello" (cons true empty))`



`(cons 5 (cons "hello" (cons true empty)))`

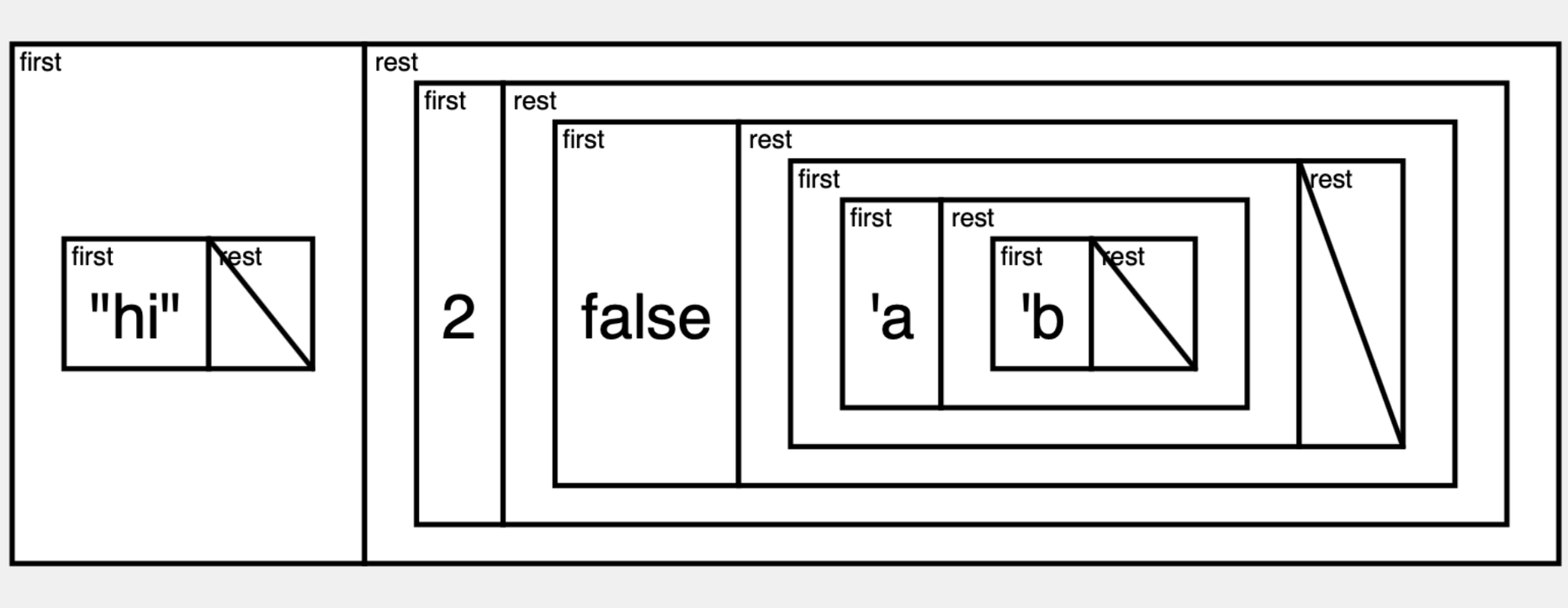


If you go on to study more programming later, you'll discover that box-and-pointer visualizations are more typical in a lot of other programming languages. The reason is that the arrows correspond to *pointers*, a way for a piece of data to represent the *location* of another piece of data within the computer's memory. But pointers are absolutely beyond the scope of this course, so we won't dwell on that connection any further.

Hopefully it's clear that these two visualizations are intimately connected—they show precisely the same information, just in slightly different ways. You can see how the two are related by animating smoothly between them. In the visualizer below, select the "Nested" or "Box-and-pointer" options to move back and forth between visualization styles.

```
(cons (cons "hi" empty)
  (cons 2
    (cons false
      (cons (cons 'a (cons 'b empty))
        empty))))
```

(•) Nested ( ) Box-and-pointer



Concept Check 05.1.7

Write a function `swap` that consumes a two-element list and produces a new two-element list containing the elements of the original list in the opposite order. For example, `(swap (cons 5 (cons "hello" empty)))` produces `(cons "hello" (cons 5 empty))`.

1/1 points
Attempts: 1 / Unlimited

```
(define (swap lst)
  (cons (first (rest lst))
        (cons (first lst) empty)))
```

Submit Code Reset Code

✔ Correct! 1/1 points

Summary

Based on the functions and types in this lesson, we now have a convenient way to hold unlimited amounts of data. I can define a constant to hold a list of numbers representing student exam marks in a course. If more students enrol in the course, I simply define a longer list.

What we can't do yet is expand the amount of work we do to accommodate data that doesn't have a predetermined size (like the list of students above). The next two lessons will show how to do that. First, we'll build up a conceptual framework for dealing with new types that have complicated structures (and add that framework to the Design Recipe), and then we'll see how to apply this framework to lists.

Discussion

Topic: Module 05 / 05.1 Using Lists

Hide Discussion

Add a Post

Show all posts | by recent activity

? 5.1.6 — 2
I am confused about how to go about this question, so far, for my code, I have (define (starts-with-3? lon) (equal? 3 (first lon)))

? Stuck on 05.1.7 — 2
I am confused because I am not sure how to print the code including the cond symbol. Here is what I have so far: (define (swap lst) (cons (rest lst) (cons (first lst) empty))) outp...