Conses in Ruby: So Much More Than Lists

Christopher Mark Gore

cgore.com

Monday, September 12, AD 2016

Ruby is my ~~most~~ ~~second~~ third favorite programming language of all time.

1. My own super-awesome programming language, Teepee *(but it's not that awesome just yet)*

2. Common Lisp

3. Ruby

4. Clojure

   .

   .

   .

999. Java

Nearly every programming language has some way to do things to a list/array/vector/whatever of things.

to do list

1. Make vanilla pudding. Put in mayo jar. Eat in public.
2. Hire two private investigators. Get them to follow each other.
3. Wear shirt that says "Life." Hand out lemons on street corner.
4. Get into a crowded elevator and say "I bet you're all wondering why I gathered you here today."
5. Major in philosophy. Ask people WHY they would like fries with that.
6. Run into a store, ask what year it is. When someone answers, yell "It worked!" and run out cheering.
7. Become a doctor. Change last name to Acula.
8. Change name to Simon. Speak in third person.
9. Buy a parrot. Teach the parrot to say "Help! I've been turned into a parrot."
10. Follow joggers around in your car blasting "Eye of the Tiger" for encouragement.

Ruby has arrays.

```ruby
a = [1,2,3,4,5]
a.class # Array
a.length # 5
a.first # 1
a.map {|i| i*2} # [2,4,6,8,10]
```

Common Lisp prefers linked lists.

```lisp
1 (setf a '(1 2 3 4 5))
2 (class-of a) ; #<BUILT-IN-CLASS COMMON-LISP:CONS>
3 (length a) ; 5
4 (first a) ; 1
5 (mapcar (lambda (i) (* 2 i)) a) ; '(2 4 6 8 10)
```

But Common Lisp also has vectors, which are basically the same as Ruby arrays.

```
1 (let v (vector 1 2 3 4 5))
2 (class-of v)
3 ;; #<BUILT-IN-CLASS COMMON-LISP:SIMPLE-VECTOR>
4 (length v) ; 5
5 (elt v 0) ; 1
6 (map 'vector (lambda (i) (* 2 i)) v)
7 ;; #(2 4 6 8 10)
```

I'm wanting to add lisp-style lists to my language. *(There's no lists or arrays or anything like that yet.)*
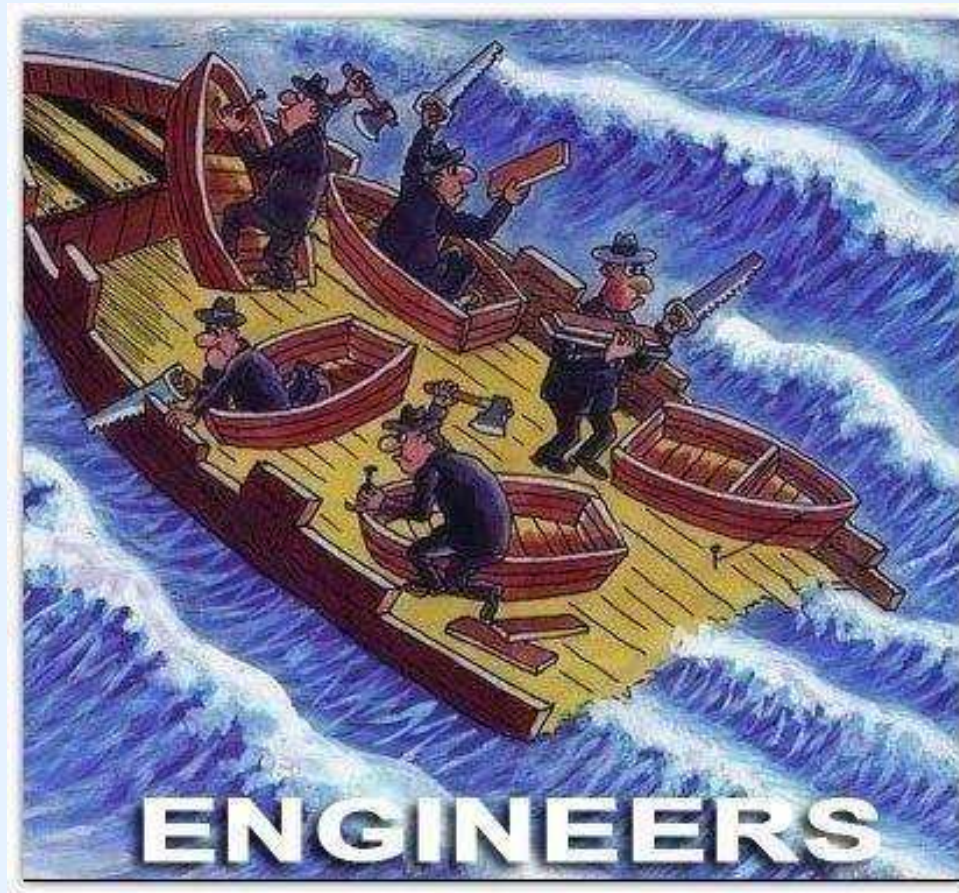
It's written in Ruby, so I'm writing a Cons gem!

```ruby
require 'cons'
c = [1,2,3,4,5].to_cons
c.class # Cons
c.length # 5
c.first # 1
c.map {|i| i*2} # Cons.list 2,4,6,8,10
```
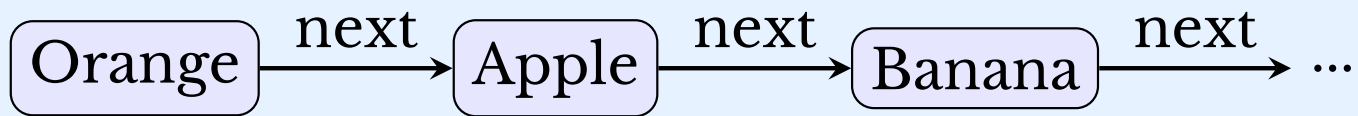
Once the gem is complete, I can then use it in the language's implementation.

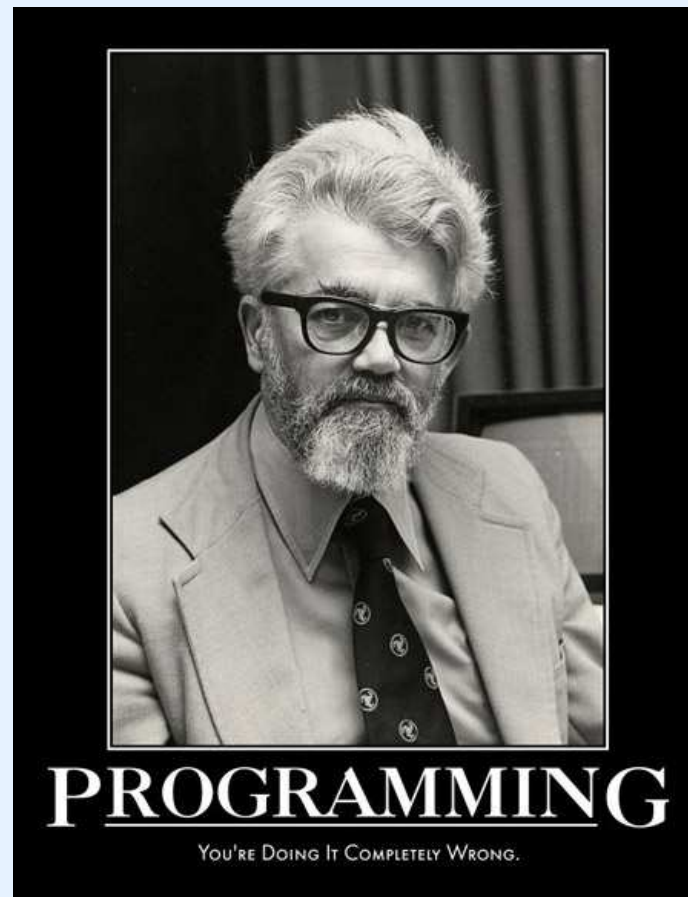**Just remember, yak shaving is only a bad thing if you are talking to a manager.**
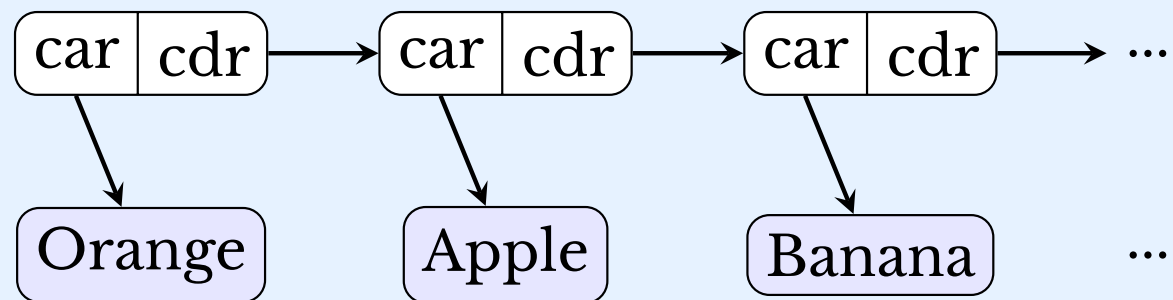
Remember linked lists?

Orange →next→ Apple →next→ Banana →next→ ...

We can do this something like this in Ruby:

```ruby
class LinkedList
  attr_accessor :data, :next
  def initialize data, next
    @data, @next = data, next
    ...
  end
  ...
end
```

Well, they don't work like that at all in Lisp.

Lisp uses conses instead.
The data isn't in the cell, it's two pointers.

| car | cdr | → | car | cdr | → | car | cdr | → … |

Orange   Apple   Banana   …

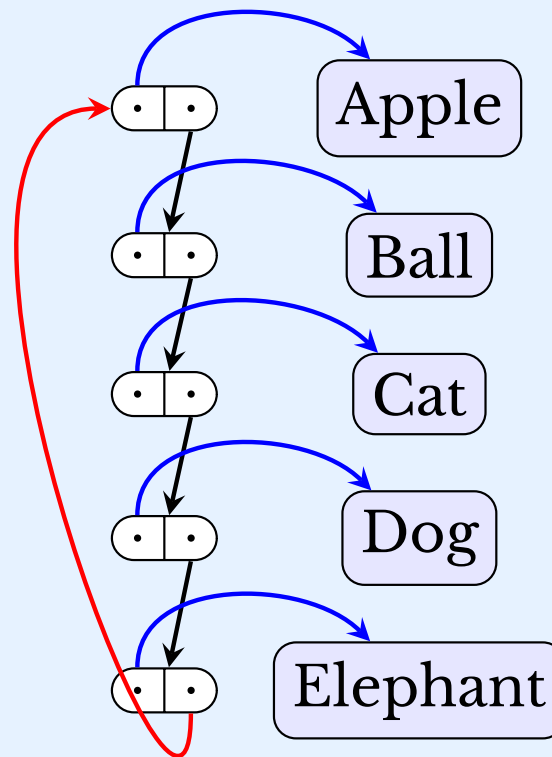# Say hello to the IBM Type 704
CAR: **Contents of the Address Register**
CDR: **Contents of the Data Register**

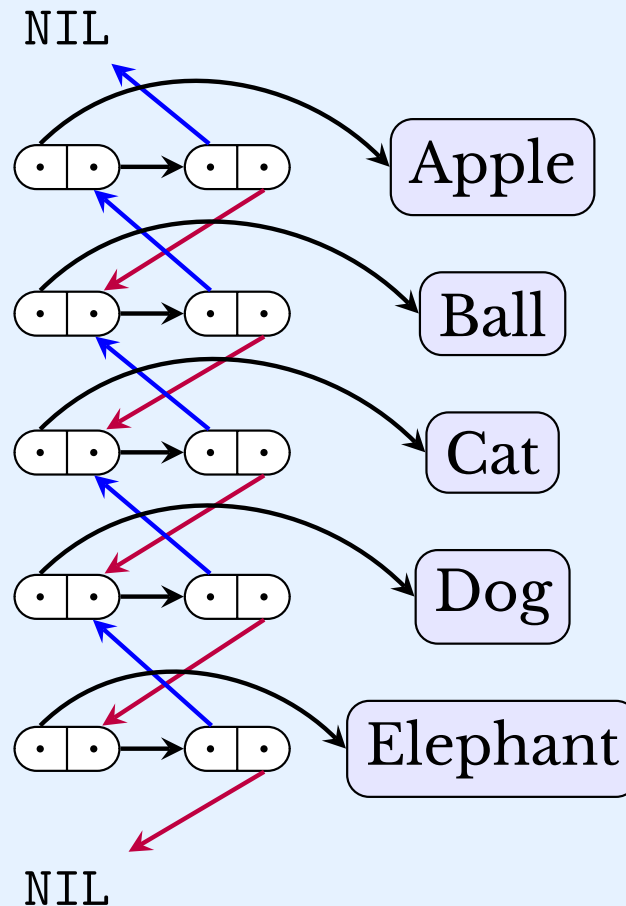# For similar reasons, every Linux box thinks your on one of these:

Why two pointers?
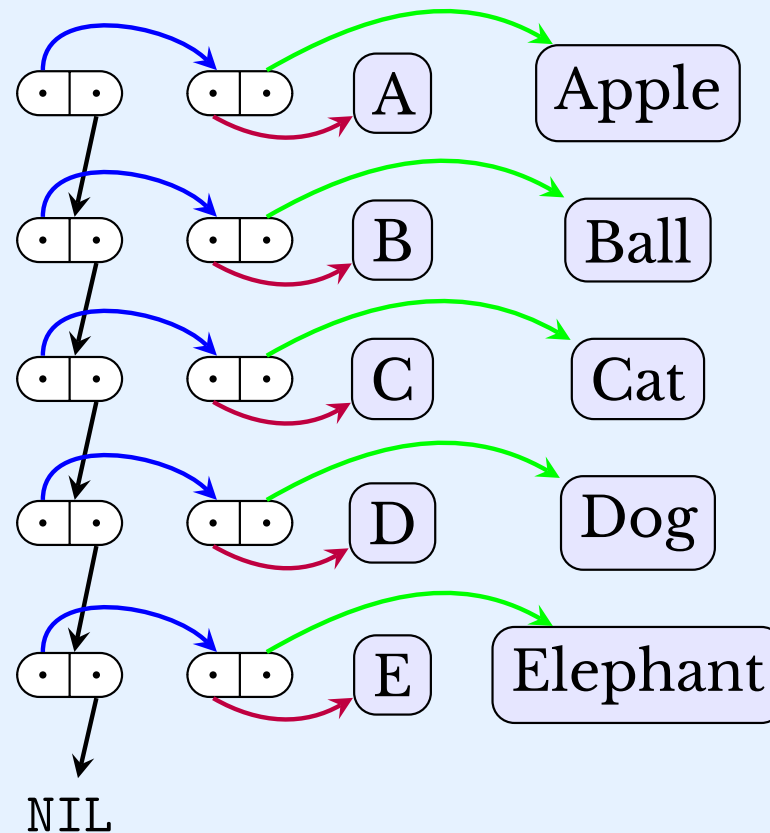## Because you can make ring buffers.

Why two pointers?
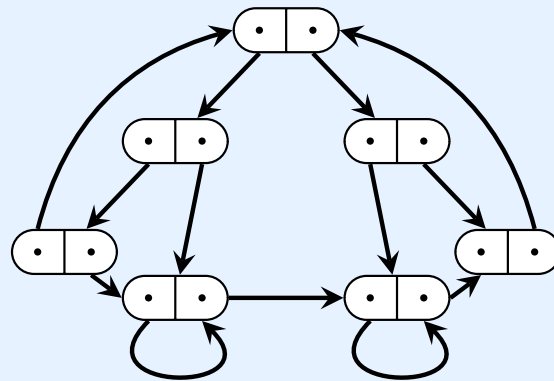## Because you can make doubly-linked lists.

Why two pointers?
Because you can make alists.

Alists, *association lists*, are how Lisp historically would create things we'd typically use hash maps for today.

```
1  '((A . Apple)
2    (B . Ball)
3    (C . Cat)
4    (D . Dog)
5    (E . Elephant))
```

Why two pointers?
Because then you can do this:

Sometimes, exploring your data structures should be an adventure!

```
YOU ARE IN A MAZE OF TWISTY PASSAGES, ALL ALIKE.
:GO SOUTH
YOU ARE IN A MAZE OF TWISTY PASSAGES, ALL ALIKE.
:GO SOUTH
YOU ARE IN A MAZE OF TWISTY PASSAGES, ALL ALIKE.
:GO EAST
YOU ARE IN A MAZE OF TWISTY PASSAGES, ALL ALIKE.
:GO NORTH
DEAD END.
:GO SOUTH
YOU ARE IN A MAZE OF TWISTY PASSAGES, ALL ALIKE.
:GO EAST
YOU ARE IN A MAZE OF TWISTY PASSAGES, ALL ALIKE.
:GO EAST
YOU ARE IN A MAZE OF TWISTY PASSAGES, ALL ALIKE.
:
```

Why do I want conses?
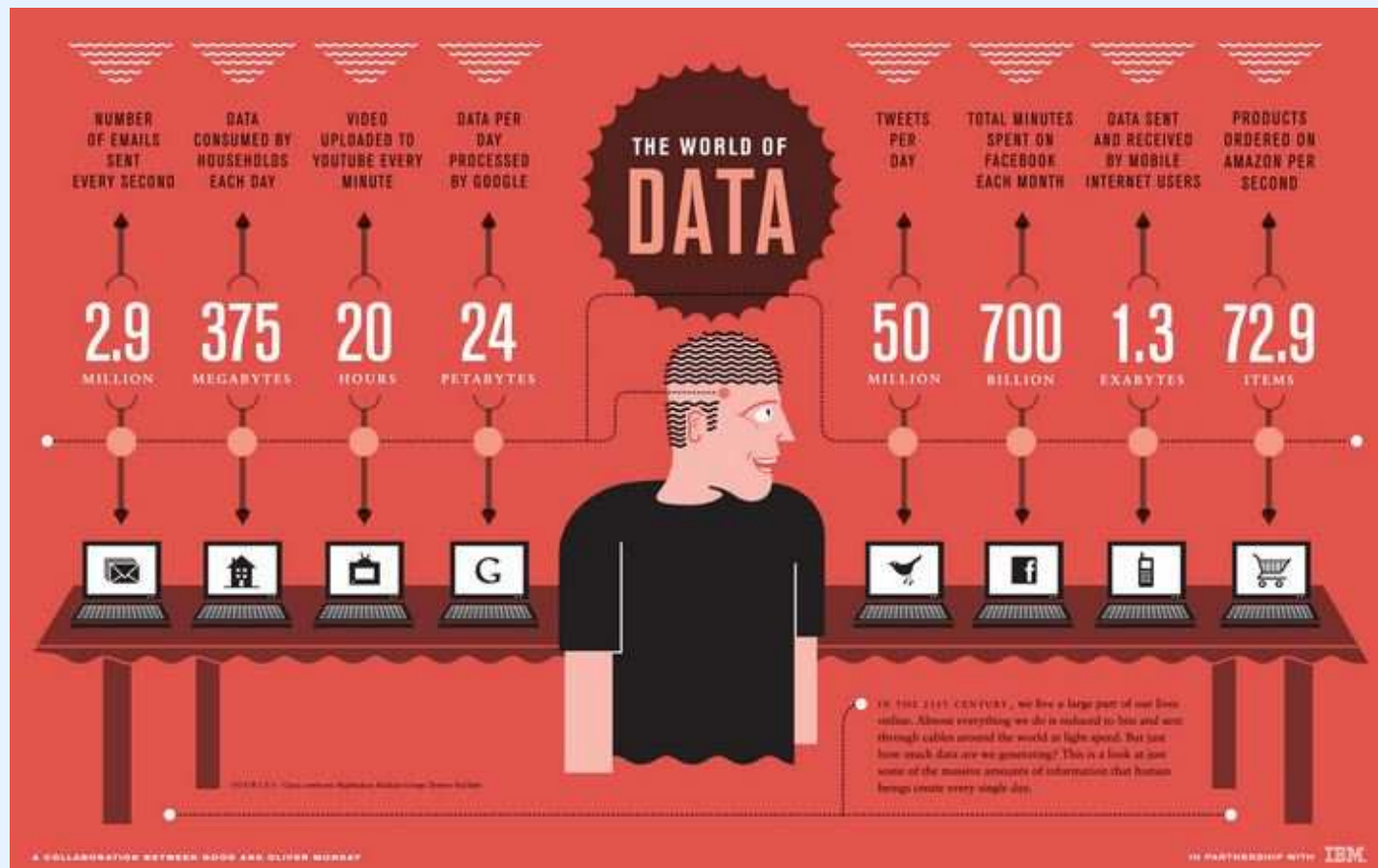
Big Data

- There's no reason why the `car` or the `cdr` need to point to local memory.

- Why not point them to the hard drive?

- Why not point them to S3?

- Why not have something cool that points to local memory, or the hard drive, or S3, at it's discretion?

Then we can have ninety-nine quintillion kitten pictures in a big list!

We can easily do Cons in Ruby

```ruby
1  class Cons
2    attr_accessor :car, :cdr
3    def initialize car, cdr
4      @car, @cdr = car, cdr
5      ...
6    end
7    ...
8  end
```

I've got a very basic implementation.

```ruby
require 'cons' # Yay gems!
Cons.new 1, nil # Basic constructor
Cons[1,nil] # Or use the brackets
Cons[1] # nil is default
Cons[1,Cons[2,Cons[3]]] # a linked list
Cons.from_array [1,2,3] # from an array
[1,2,3].to_cons # monkey patching
```

Comparisons, car, and cdr

```
1 c = [1,2,3,4,5].to_cons
2 c.length # 5, assumes a linked list
3 d = [1,2,3,4,5].to_cons
4 c == d # true, but they're not identical
5 c.car # 1
6 c.cdr.car # 2
7 c.cdr.cdr.car # 3
8 c.cdr.cdr.cdr.car # 4
9 c.cdr.cdr.cdr.cdr.car # 5
```

Because sometimes English is easier:

```
1 c = [1,2,3,4,5].to_cons
2 c.first  # 1
3 c.second  # 2
4 c.third  # 3
5 c.fourth  # 4
6 c.fifth  # 5
```

Umm, this is a thing?

```
1 c = [1,2,3,4,5].to_cons
2 c.car # 1
3 c.cdar # 2
4 c.cddar # 3
5 c.cdddar # 4
6 c.cddddar # What?  That's crazy talk!
```

It's easy to get the cdr's too.

```ruby
c = [1,2,3,4,5].to_cons
c.cdr.to_a # 2,3,4,5]
c.rest.to_a # [2,3,4,5]
c.cddr.to_a # [3,4,5]
c.cdddr.to_a # [4,5]
c.cddddr.to_a # [5]
```

You can easily make assignments.

```
1  c = [1,2,3,4,5].to_cons
2  c.car  # 1
3  c.car = "wow"
4  c.car  # "wow"
5  c.cdar = "hey"
6  c.cddar = "cool"
7  c.to_a  # ["wow","hey","cool",4,5]
```

Work in progress …

- Implement arbitrary `c*r` forms via `method_missing`? Maybe not?

- Smarter pretty printer.

- Lots of stuff for alists.

- Lots of stuff for lists as sets.

- Lots of other stuff from Common Lisp.

- Go to the hard drive!

- Go to S3!

Questions?