The original frustrated comment

Emma made this in 1871 and hated every minute of it.
List abbreviations
Writing down large lists

What does the list containing 0 to 10 look like?
Writing down large lists

What does the list containing 0 to 10 look like?

(cons 0
  (cons 1
    (cons 2
      (cons 3
        (cons 4
          (cons 5
            (cons 6
              (cons 7
                (cons 8
                  (cons 9
                    (cons 10
                      '(()))))))))))}
Writing down large lists

What does the list containing 0 to 10 look like?

(cons 0 (cons 1 (cons 2 (cons 3 (cons 4 (cons 5 (cons 6 (cons 7 (cons 8 (cons 9 (cons 10 '())))))))))))
Writing down large lists

What does the list containing 0 to 10 look like?

(cons 0 (cons 1 (cons 2 (cons 3 (cons 4 (cons 5 (cons 6 (cons 7 (cons 8 (cons 9 (cons 10 '()))))))))))

Here’s a shortcut:

(list 0 1 2 3 4 5 6 7 8 9 10)

The list operator takes any number of arguments and constructs a list.
Writing down large lists

What does the list containing 0 to 10 look like?

(cons 0 (cons 1 (cons 2 (cons 3 (cons 4 (cons 5 (cons 6 (cons 7 (cons 8 (cons 9 (cons 10 '()))))))))))))

Here’s a shortcut:

(list 0 1 2 3 4 5 6 7 8 9 10)

The list operator takes any number of arguments and constructs a list.

But DrRacket still prints 11 conses as the value. 😞
Printing large lists

If you change DrRacket’s language level to *Beginning Student with List Abbreviations*, then DrRacket prints list values using the same abbreviation:

```scheme
> (list 0 1 2 3 4 5 6 7 8 9 10)
(list 0 1 2 3 4 5 6 7 8 9 10)
```
Printing large lists

If you change DrRacket’s language level to *Beginning Student with List Abbreviations*, then DrRacket prints list values using the same abbreviation:

```scheme
> (list 0 1 2 3 4 5 6 7 8 9 10)
(list 0 1 2 3 4 5 6 7 8 9 10)

> (cons 1 (cons 2 (cons 3 '()))
(list 1 2 3)
```
So, why have we been using this cumbersome notation of \texttt{cons}ing one element at a time?
So, why have we been using the notation of **cons**ing one element at a time?

**Lists are self-referential data, which we process by writing recursive functions.**

**Using cons makes this structure clear.**
**Using list abbreviations hides it.**
list and cons do not do the same thing

> (define L1 (list "b" "c"))
> (define L2 (list "d" "e" "f"))

> (cons "a" L1)
(list "a" "b" "c"
list and cons do not do the same thing

> (define L1 (list "b" "c"))
> (define L2 (list "d" "e" "f"))

> (cons "a" L1)
(list "a" "b" "c")

*Produce a new list by adding "a" to the front of L1*
list and cons do not do the same thing

> (define L1 (list "b" "c"))
> (define L2 (list "d" "e" "f"))

> (cons "a" L1)
(list "a" "b" "c")

> (list "a" L1)
(list "a" (list "b" "c"))

Produce a new list by adding "a" to the front of L1
list and cons do not do the same thing

> (define L1 (list "b" "c"))
> (define L2 (list "d" "e" "f"))

> (cons "a" L1)
(list "a" "b" "c")

> (list "a" L1)
(list "a" (list "b" "c"))

Produce a new list by adding "a" to the front of L1

Produce a new list with "a" as the first element and L1 as the second element
To add one element to a list, you still want to use \texttt{cons}, not \texttt{list}.

To define a fully formed list all at once, you can use \texttt{list}.
When to change language levels

1. You’re not tempted to write examples like this:

   (check-expect (feed-fish (cons 1 (cons 2 '())))
   2 3)
When to change language levels

1 You’re not tempted to write examples like this:
   
   (check-expect (feed-fish (cons 1 (cons 2 '())))
   2 3)

2 Your eyes hurt when you see
   
   (cons 1 (cons 2))

because it isn’t a *ListOfNumbers*.
When to change language levels

1. You’re not tempted to write examples like this:
   
   (check-expect (feed-fish (cons 1 (cons 2 '())))
   2 3)

2. Your eyes hurt when you see
   
   (cons 1 (cons 2))

   because it isn’t a **ListOfNumbers**.

3. When you see
   
   (list 1 2 3)
   (cons 1 (cons 2 (cons 3 '())))

   you recognize instantly that they’re the same.
When to change language levels

1 You’re not tempted to write examples like this:
   
   (check-expect (feed-fish (cons 1 (cons 2 '()))) 2 3)

2 Your eyes hurt when you see
   
   (cons 1 (cons 2))

   because it isn’t a ListOfNumbers.

3 When you see
   
   (list 1 2 3)
   
   (cons 1 (cons 2 (cons 3 '())))

   you recognize instantly that they’re the same.

Don’t switch until you understand how ListOf... functions match the shape of the data definition.
Even shorter

For the brave, there’s an even shorter shortcut!
Even shorter

For the brave, there’s an even shorter shortcut!

'(1 2 3)

is the same as

(list 1 2 3)

We use an apostrophe to quote Racket expressions we don’t want to be evaluated.

You’ve already seen this notation for the empty list, '(())

When we write an apostrophe before a non-empty list, the apostrophe gets distributed to everything inside.
We can use an apostrophe to quote symbols so Racket doesn’t try to look up a value for them, e.g.,

```scheme
> (define p 3)
> p
3
> 'p
p
```

For a list, any symbols inside are quoted when the list is quoted:

```
'(apple banana)
```

is the same as

```
(list 'apple 'banana)
```

But quoting numbers has no effect since they already evaluate to themselves.

```
> 1
1
> '1
1
> '(1 2 3)
(list 1 2 3)
```
Here’s a list of list-of-numbers using the shortcut:

`'( (1 2 3) (2 4 6 8) (3 9 27) )`
Here’s a list of list-of-numbers using the shortcut:

'((1 2 3) (2 4 6 8) (3 9 27))

which is the same as

(list (list 1 2 3)
     (list 2 4 6 8)
     (list 3 9 27))
Here’s a list of list-of-numbers using the shortcut:

`'((1 2 3) (2 4 6 8) (3 9 27))`

which is the same as

```lisp
(list (list 1 2 3)
     (list 2 4 6 8)
     (list 3 9 27))
```

which is the same as

```lisp
(cons (cons 1
       (cons 2
         (cons 3 '()))
    (cons (cons 2
           (cons 4
             (cons 6
               (cons 8 '()))))
  (cons (cons 3
         (cons 9
           (cons 27 '()))
   '())))
```
Natural numbers
Numbers to generate lists

Implement `countdown`, which takes a non-negative integer \( n \) and produces a list of numbers from \( n \) to 0, inclusive.

;; Number -> ListOfNumbers

(check-expect (countdown 0) (list 0))
(check-expect (countdown 3) (list 3 2 1 0))
Numbers to generate lists

Implement **countdown**, which takes a non-negative integer \( n \) and produces a list of numbers from \( n \) to 0, inclusive.

\[
\text{;; Number -> ListOfNumbers}
\]

\[
\text{(check-expect (countdown 0) (list 0))}
\]
\[
\text{(check-expect (countdown 3) (list 3 2 1 0))}
\]

The template for **Number** isn’t much help:

\[
\text{(define (func-for-num n)}
\]
\[
\text{...)}
\]

But **countdown** actually takes a *natural number*. 
Natural numbers

;;; A Natural is one of:
;;; - 0
;;; - (add1 Natural)
Natural numbers

;; A Natural is one of:
;;   - 0 base case
;;   - (add1 Natural)
Natural numbers

;; A Natural is one of:
;;  - 0 base case
;;  - (add1 Natural)
Natural numbers

;; A Natural is one of:
;; – 0 base case
;; – (add1 Natural)

Examples:

0
(add1 0)
(add1 (add1 0))
(add1 (add1 (add1 0)))
Natural numbers

;; A Natural is one of:
;; - 0
;; - (add1 Natural)

Examples:

0 → 0
(add1 0) → 1
(add1 (add1 0)) → 2
(add1 (add1 (add1 0))) → 3

These examples have shortcuts 0, 1, 2, and 3, but the long forms correspond to the template.
add1  cons

sub1  rest


add1 produces a natural number 1 bigger

sub1

cons produces a list 1 item longer

rest
add1 produces a natural number 1 bigger

sub1 produces a natural number 1 smaller

cons produces a list 1 item longer

rest produces a list 1 item shorter
Now that we have a data definition for natural numbers, we’ll need a template for functions that consume natural numbers.
Recall where the template for a list of numbers comes from:

```verbatim
;; A ListOfNumbers is one of:
;; - '()
;; - (cons Number ListOfNumbers)
```
Recall where the template for a list of numbers comes from:

```lisp
;; A ListOfNumbers is one of:
;;   - '()
;;   - (cons Number ListOfNumbers)
```

To derive a template, we used a `cond` for the two cases.

We broke up the non-empty list `(cons f r)` using the selector `first` to extract `f`

the selector `rest` to extract `r`, and

an application of the function on `r`. 
Recall where the template for a list of numbers comes from:

;; A ListOfNumbers is one of:
;; - '(())
;; - (cons Number ListOfNumbers)

(define (fn-for-lon lon)
 (cond [[(empty? lon) (....)]
        [[(cons? lon)
          (... (first lon)
           ... (fn-for-lon (rest lon)))]])

We also could have written `else` as the test for the second case
Recall where the template for a list of numbers comes from:

```
;; A ListOfNumbers is one of:
;; - '()
;; - (cons Number ListOfNumbers)
```

```scheme
(define (fn-for-lon lon)
  (cond [(empty? lon) (...)]
        [(cons? lon) (... (first lon) (... (fn-for-lon (rest lon)))))])
```

We also could have written `else` as the test for the second case.
Recall where the template for a list of numbers comes from:

```scheme
;; A ListOfNumbers is one of:
;; - '()  
;; - (cons Number ListOfNumbers)
```

```scheme
(define (fn-for-lon lon)
  (cond [(empty? lon) (...)]
        [(cons? lon)
         (... (first lon)
              ... (fn-for-lon (rest lon)))]))
```

We also could have written `else` as the test for the second case.
Similarly, we’ll use the data definition for a natural number:

```scheme
;; A Natural is one of:
;; 0
;; (add1 Natural)
```
Similarly, we’ll use the data definition for a natural number:

```ruby
;; A Natural is one of:
;;  - 0
;;  - (add1 Natural)
```

To derive a template for a natural number \( n \), we will use a `cond` for the two cases.

We will break up the non-zero case \( n = (\text{add1 } k) \) using

the function `sub1` to extract \( k \) and

an application of the function on \( k \), i.e., on

\( (\text{sub1 } n) \)
Similarly, we’ll use the data definition for a natural number:

```scheme
;; A Natural is one of:
;; - 0
;; - (add1 Natural)

(define (fn-for-nat n)
  (cond
   [(zero? n) (...)]
   [else
     (... n
     ... (fn-for-nat (sub1 n)))]))
```
Similarly, we’ll use the data definition for a natural number:

```scheme
;; A Natural is one of:
;; - 0
;; - (add1 Natural)
```

```scheme
(define (fn-for-nat n)
  (cond [(zero? n)
        (...)]
       [else
        (... n
             ... (fn-for-nat (sub1 n)))]))
```
Similarly, we’ll use the data definition for a natural number:

\[
\text{A Natural is one of:}
\]

\[
\begin{align*}
\text{– } & \text{ 0} \\
\text{– } & \text{(add1 Natural)}
\end{align*}
\]

\[
\text{define (fn-for-nat n)}
\]

\[
\text{(cond [\text{zero? n) \quad (...)] [else \quad (... n \quad \text{... (fn-for-nat (sub1 n))})]})}
\]
We haven’t defined *ListOfNaturals* as a data type! Don’t we need to?

Well, *ListOfNaturals* is the *produced* type (as opposed to the *consumed* type), so we won’t need a template for it.

You also know what the data definition would look like!

All of our *List* data definitions follow the same pattern. Starting next week, we’ll introduce the *[List-of …]* abstraction to capture this commonality.
;; countdown : Natural -> ListOfNaturals
;; Produces a decreasing list of Naturals starting at n and
;; ending with 0
(define (countdown n) '())

(check-expect (countdown 0) (list 0))
(check-expect (countdown 3) (list 3 2 1 0))
;;; countdown : Natural -> ListOfNaturals
;;; Produces a decreasing list of Naturals starting at n and
;;; ending with 0
(define (countdown n)
  (cond [(zero? n)
         (...)]
       [else
         (... n
         ... (countdown (sub1 n)))]))

(check-expect (countdown 0) (list 0))
(check-expect (countdown 3) (list 3 2 1 0))
;; countdown : Natural -> ListOfNaturals
;; Produces a decreasing list of Naturals starting at n and
;; ending with 0
(define (countdown n)
  (cond [(zero? n)
         (list 0)]
       [else
        (cons n (countdown (sub1 n)))]))

(check-expect (countdown 0) (list 0))
(check-expect (countdown 3) (list 3 2 1 0))
Condensed trace of countdown

(countdown 2)
→ (cons 2 (countdown 1))
→ (cons 2 (cons 1 (countdown 0)))
→ (cons 2 (cons 1 (cons 0 '())))
If the function countdown is applied to a negative argument, it won’t terminate.

We said its behavior is only defined when it’s given a natural number, but it’s easy to let the function handle negative arguments gracefully:

```scheme
(define (countdown n)
  (cond
   [(<= n 0) ; Was (zero? n)
     (list 0)]
   [else
     (cons n (countdown (sub1 n))))])
```
Beware

Some people use “natural number” to mean the non-negative integers 0, 1, 2, 3, …

Others use it to mean the positive integers 1, 2, 3, …

There’s no right or wrong answer; it’s just a difference in terminology. If you’re not sure what someone means, ask.
In a world without numbers...
Imagine Racket was created by someone with *numerophobia* – a fear of numbers.

It would still be a cool programming language, but you’d have a hard time writing a lot of programs.
Ask yourself: **What would MacGyver do?**
“No problem”, MacGyver says, “My mind is the ultimate weapon.”

“I remember the self-referential data definition for Natural. I’ve heard that add1 is kind of like cons and sub1 is kind of like rest. With some fishing line and chewing gum, I think I can make this work…”
Let’s define a new data type:

;;; A NaTuRaL is one of:
;;; - '()
;;; - (cons "!" NaTuRaL)
;;; interp. a natural number; the number
;;; of "!" in the list is the number

(define N0 '()) ; 0
(define N1 (cons "!" N0)) ; 1
(define N2 (cons "!" N1)) ; 2
(define N3 (cons "!" N2)) ; ...
(define N4 (cons "!" N3))
(define N5 (cons "!" N4))
(define N6 (cons "!" N5))
(define N7 (cons "!" N6))
(define N8 (cons "!" N7))
(define N9 (cons "!" N8))
These are the primitives that operate on \texttt{NaTuRaL} numbers:

\begin{verbatim}
;;; Any -> Boolean
(define (ZeRo? n) (empty? n))

;;; NaTuRaL -> NaTuRaL
(define (AdD1 n) (cons "!" n))

;;; NaTuRaL[>0] -> NaTuRaL
(define (SuB1 n) (rest n))
\end{verbatim}
With these primitives, the template for a function that takes a *NaTuRaL* number looks familiar:

```
(define (fn-for-nat n)
  (cond [(zero? n) (...)]
       [else (... n
              ... (fn-for-nat (sub1 n)))]))

(define (fn-for-NaTuRaL n)
  (cond [(ZeRo? n) (...)]
       [else (... n
              ... (fn-for-NaTuRaL (SuB1 n)))]))
```
Now that we have the template, we can define some functions for our ersatz natural numbers. We have \texttt{AdD1}, but not plain old addition of two numbers.
;; NaTuRaL NaTuRaL -> NaTuRaL
;; Produce a + b
(define (AdD a b) N0)

(check-expect (AdD N2 N0) N2)
(check-expect (AdD N0 N3) N3)
(check-expect (AdD N3 N4) N7)
;; NaTuRaL NaTuRaL -> NaTuRaL
;; Produce a + b
(define (AdD a b)
  (cond [[(ZeRo? b)
          (...)]
        [else
         (... b
          (... (AdD a (SuB1 b)))]))

(check-expect (AdD N2 N0) N2)
(check-expect (AdD N0 N3) N3)
(check-expect (AdD N3 N4) N7)

Note: We need to include a in the recursive call because AdD takes two arguments
;; NaTuRaL NaTuRaL -> NaTuRaL
;; Produce a + b
(define (AdD a b)
  (cond [(ZeRo? b)
         ;; Adding 0 to number 'a' gives us 'a'
         a]
       [else
        (... b
         (... (AdD a (SuB1 n)))]))

(check-expect (AdD N2 N0) N2)
(check-expect (AdD N0 N3) N3)
(check-expect (AdD N3 N4) N7)
For the recursive case, imagine addition as moving pebbles from one pile to the other:

\[ a = 3 \quad \text{and} \quad b = 2 \]
For the recursive case, imagine addition as moving pebbles from one pile to the other:

\[ a = 4 \quad b = 1 \]
For the recursive case, imagine addition as moving pebbles from one pile to the other:

\[ a = 5 \quad b = 0 \]
;; Natural Natural -> Natural
;; Produce a + b
(define (AdD a b)
  (cond [(ZeRo? b)
         ;; Adding 0 to number 'a' gives us 'a'
         a]
       [else
        (AdD (AdD1 a) (SuB1 b))])))

(check-expect (AdD N2 N0) N2)
(check-expect (AdD N0 N3) N3)
(check-expect (AdD N3 N4) N7)
How do we *Subtract* one *Natural* from another?
;; NatuRaL NatuRaL → NatuRaL
;; Produce a - b
(define (SubtRaCt a b)
  (cond [(ZeRo? b) a]
    [else
     (SubtRaCt (SuB1 a) (SuB1 b))]))

(check-expect (SubtRaCt N2 N0) N2)
(check-expect (SubtRaCt N6 N2) N4)
We can do arithmetic with natural numbers, without using any numbers:

\[
> \text{(AdD N2 N3)}\\
(\text{list } "!" "!" "!" "!" "!" "!" ) \ ; = N5\\
> \text{(SuBtRaCt N7 (AdD N2 N3))}\\
(\text{list } "!" "!" ) \ ; = N2
\]

MacGyvered!
The king is dead

Here ends the first part of the course:

*How to Design Programs, Parts I & II*

Exam 1

---

Long live the king

Here begins the second part of the course:

*How to Design Programs, Parts III & IV*

Exam 2
The king is dead

Here ends the first part of the course: 

*How to Design Programs, Parts I & II*

Exam 1

Long live the king

Here begins the second part of the course:

*How to Design Programs, Parts III & IV*

Exam 2

Well, approximately.

Exam 1 will emphasize earlier material.

And, because earlier concepts support the later ones, the exams are unavoidably cumulative.

We’ll review for Exam 1 next class.
Acknowledgments

This lecture incorporates material from:

- Matthias Felleisen
- Robert Bruce Findler
- Matthew Flatt
- Gregor Kiczales
- Shriram Krishnamurthi
- Marc Smith