We’ve already seen how *evaluation order* can change behavior when we program with *state*.

Now we want to investigate how evaluation order can change *efficiency* -- and more important, how changing evaluation order can open the door to a whole new perspective on data structures. In particular, *infinite data structures*...
Streams and stream processing

We have already seen the "signal processing" or "circuit" style of programming:

```
(sum (map square
    (filter prime?
      (enumerate-interval 2 1000000))))
```

Now, what about the "waste motion" of the following?

```
(car (cdr (map square
    (filter prime?
      (enumerate-interval 2 1000000))))))
```

Instead: produce elements of a new kind of list -- a stream -- on a "need to know" basis, while preserving the "standard invariants"

```
(stream-car (cons-stream x y)) = x
(stream-cdr (cons-stream x y)) = y
```

Also introduce the-empty-stream, stream-null? etc.
Building blocks:

(define (stream-ref s n)
  (if (= n 0)
      (stream-car s)
      (stream-ref (stream-cdr s) (- n 1))))

(define (stream-map proc s)
  (if (stream-null? s)
      the-empty-stream
      (cons-stream (proc (stream-car s))
                   (stream-map proc (stream-cdr s)))))

(define (stream-for-each proc s) ...)

(define (display-stream s)
  (stream-for-each (lambda (x) (display x) (newline))
                   s))

What changes by changing just a name? Nothing. **We need a new implementation...**
We need a new implementation...

\[(\text{cons-stream } a \ b) \ \text{=def=} \ (\text{cons } a \ (\text{delay } b))\]

...but how do we delay evaluation?...

\[(\text{delay } b) \ \text{=def=} \ (\text{lambda } () \ b)\]

Thus

\[(\text{cons-stream } a \ b) \ \text{=def=} \ (\text{cons } a \ (\text{lambda } () \ b))\]

[Think of this as \textit{syntactic sugar}.] \textbf{Cons-stream} gets a pair, evaluates the first argument, but builds a procedure which can evaluate the second argument (if called). \textit{What does this mean in terms of environment diagrams?}

\[(\text{define } (\text{stream-car stream}) \ (\text{car stream}))\]

or if you prefer,

\[(\text{define stream-car car})\]

\[(\text{define } (\text{stream-cdr stream}) \ (\text{force } (\text{cdr stream})))\]

where

\[(\text{force } s) \ \text{=def=} \ (s)\]

thus

\[(\text{define } (\text{stream-cdr stream}) \ ((\text{cdr stream})))\]
Stream implementation in action:

(stream-car
  (stream-cdr
   (stream-enumerate-interval 2 1000000))))

(stream-car
  (stream-cdr
   (stream-filter prime?
     (stream-enumerate-interval 2 1000000)))))

Note

(define (stream-enumerate-interval from to)
  (if (> from to)
      the-empty-stream
      (cons-stream from (stream-enumerate-interval (+ from 1) to))))

Efficiency issues

(define s (stream-filter prime? (stream-enumerate-interval 2 1000000)))

(+ (select 1000 s) (select 1001 s))

Notice work gets repeated. Also, we are using:

(define (select n stream)
  (if (= n 0)
      (stream-car stream)
      (select (- n 1) (stream-cdr stream))))

Answer: memoization
A memoized version of streams

(define head car)
;Value: head

(define (tail s)
  (let ((t (cdr s)))
    (if (or (pair? t)
            (null? t))
      t
      (begin
        (set-cdr! s (t))
        (cdr s))))))
;Value: tail

(define (ints n)
  (cons n
    (lambda ()
      (ints (1+ n))))))
;Value: ints

(define N (ints 1))
;Value: N

N
;Value: (1 . #[compound-procedure 2])

(tail N)
;Value: (2 . #[compound-procedure 4])

N
;Value: (1 2 . #[compound-procedure 4])

(tail (tail N))
;Value: (3 . #[compound-procedure 6])

N
;Value: (1 2 3 . #[compound-procedure 6])
Memoization of a parameterless procedure:

(define (memo-proc proc)
  (let ((already-run? #f) (result #f))
    (lambda ()
      (if (not already-run?)
        (begin (set! result (proc))
               (set! already-run? #t)
               result)
        result))))

Question: what does this look like in the environment model?
Now instead, use the syntactic sugar:

(delay b) =def= (memo-proc (lambda () b))
Infinite streams

Lists are *inductively defined*, and every one of them has *finite length*. (Prove it!) In contrast, streams may be *infinite* in length!

\[
\text{(define ones (cons-stream 1 ones))}
\]

Compare this with

\[
\text{(define ones (cons 1 ones))}
\]
[what happens here?]

\[
\text{(define (integers-from n)}
\]
\[
\quad \text{(cons-stream n (integers-from (+ n 1))))}
\]

\[
\text{(select 100 (integers-from 101))}
\]
Infinite streams, cont.

Integers not divisible by 7:

(define integers (integers-from 0))

(define (divisible? x y) (= (remainder x y) 0))

(define no-sevens
  (stream-filter (lambda (x) (not (divisible? x 7))) integers))

Fibonacci numbers, yet again:

(define (fibgen a b) (cons-stream a (fibgen b (+ a b))))

(define fibs (fibgen 0 1))

Primes:

(define (sieve stream)
  (cons-stream
    (stream-car stream)
    (sieve (stream-filter (lambda (x)
                                (not (divisible? x (stream-car stream)))
                              (stream-cdr stream))))))

(define primes (sieve (integers-from 2)))
Defining streams implicitly

Assume the streams are infinite...

(define (add-streams s1 s2)
  (cons-stream (+ (stream-car s1) (stream-car s2))
    (add-streams (stream-cdr s1)
      (stream-cdr s2))))

(define integers (cons-stream 1
  (add-streams ones
    integers)))

(define fibs
  (cons-stream 0
    (cons-stream 1
      (cons-stream 1
        (add-streams (stream-cdr fibs) fibs)))))
Defining streams implicitly: coroutining a predicate and a stream

(define primes
  (cons-stream 2
    (stream-filter prime?
      (integers-from 3)))))

(define (prime? n)
  (define (iter ps)
    (if (> (square (stream-car ps)) n)
      #t
      (if (divisible? n (stream-car ps))
        #f
        (iter (stream-cdr ps))))
    (iter primes))

Note the list of primes is use to generate candidate prime divisors -- we just need to "prime the pump" (no pun intended...).
Constructing the real numbers...

In mathematics, the reals are constructed from infinite sequences (i.e., streams) of rational numbers. For example, one you know:

```scheme
(define (sqrt n)
  (define (improve g)
    (average g (/ n g)))
  (define (seq g)
    (cons-stream g (seq (improve g n)))
    (seq 1))
```

This leads to infinite-precision arithmetic—goodbye, floating point! How many significant digits do you want?

*What problems arise? (how to +, x, ÷, test =0,...)*
Limits and convergence

If \( a_0, a_1, a_2, \ldots \) is a sequence, then \( \lim_{n \to \infty} a_n = a \) if:

for every \( \varepsilon > 0 \)

there exists an \( N \) (depending on \( \varepsilon \))

such that:

\[ n > N \Rightarrow |a_n - a| < \varepsilon \]

What does this mean about the stream?
ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO
THE ENTSCHEIDUNGSPROBLEM

By A. M. TURING.

[Received 28 May, 1936.—Read 12 November, 1936.]

The "computable" numbers may be described briefly as the real
numbers whose expressions as a decimal are calculable by finite means.
Although the subject of this paper is ostensibly the computable numbers,
it is almost equally easy to define and investigate computable functions
of an integral variable or a real or computable variable, computable
predicates, and so forth. The fundamental problems involved are,
however, the same in each case, and I have chosen the computable numbers
for explicit treatment as involving the least cumbersome technique. I hope
shortly to give an account of the relations of the computable numbers,
functions, and so forth to one another. This will include a development
of the theory of functions of a real variable expressed in terms of computable
numbers. According to my definition, a number is computable
if its decimal can be written down by a machine.

For instance, let us assume as given the computation of the function of

1. THE CONCEPT OF A FUNCTION. Underlying the formal calculi which we shall develop is the concept of a function, as it appears in various branches of mathematics, either under that name or under one of the synonymous names, "operation" or "transformation." The study of the general properties of functions, independently of their appearance in any particular mathematical (or other) domain, belongs to formal logic or lies on the boundary line between logic and mathematics. This study is the original motivation for the calculi — but they are so formulated that it is possible to abstract from the intended meaning and regard them merely as formal systems.

A function is a rule of correspondence by which when anything is given (as argument) another thing (the value of the function for that argument) may be obtained. That is, a function is an operation which may be applied on one thing (the argument) to yield another thing (the value of the function). It is not, however, required that the operation shall necessarily be applicable to everything whatsoever; but for each function there is a class, or range, of possible arguments — the class of things to which the operation is significantly applicable — and this we shall call the range of arguments, or range of the independent variable, for that function. The class of all values of the function, obtained by taking all possible arguments, will be called the range of values, or range of the dependent variable.

If \( f \) denotes a particular function, we shall use the notation \( (f\alpha) \) for the value of the function \( f \) for the argument \( \alpha \). If \( \alpha \) does not belong to the range of arguments of \( f \), the notation \( (f\alpha) \) shall be meaningless.

It is, of course, not excluded that the range of arguments or range of values of a function should consist wholly or partly of functions. The derivative, as this notion appears in the elementary differential calculus, is a familiar mathematical example of a function for which both ranges consist of functions. Or, turning to the integral calculus, if in the expression \( \int_0^1 (f\alpha) dx \) we take the function \( f \) as independent variable, we are led to a function for which the range of arguments consists of functions and the range of values, of numbers. Formal logic provides other examples; thus the existential quantifier, according to the present account, is a function for which the range of arguments consists of propositional functions, and the range of values consists of truth-values.
Continuity

We say that function $f$ is continuous at $a$ if for any sequence $a_0, a_1, a_2, \ldots$ where $\lim_{n \to \infty} a_n = a$, it’s true that $\lim_{n \to \infty} f(a_n) = f(a)$.

If a real number is a stream, what does this say about function evaluation?

Example of discontinuity: let $a_n = 2^{-n}$, so $\lim_{n \to \infty} a_n = 0$, and consider the function $f = \text{not-positive}.$

```scheme
(define (not-positive? r)
  (if (<= r 0) #t #f))

(define a
  (cons-stream 1 (map-stream (lambda (x) (/ x 2)) a)))

(not-positive? 0) = #t

(map-stream not-positive? a) = ???
```
Continuity

We say that function \( f \) is **continuous at** \( a \) if for any sequence \( a_0, a_1, a_2, \ldots \) where \( \lim_{n \to \infty} a_n = a \), it’s true that \( \lim_{n \to \infty} f(a_n) = f(a) \).

If a real number is a **stream**, what does this say about function evaluation?

**Example of discontinuity**: let \( a_n = 2^{-n} \), so \( \lim_{n \to \infty} a_n = 0 \), and consider the function \( f = \text{not-positive?} \)?

```
(define (not-positive? r)
  (if (<= r 0) #t #f))

(define a
  (cons-stream 1 (map-stream (lambda (x) (/ x 2)) a)))

(not-positive? 0) = #t

(map-stream not-positive? a) = ???
```
Defining infinite data structures

An *infinite data structure* is just a finite piece of “ordinary” data structure, together with a rule (i.e., procedure) for generating more of the structure.
A simple example: *infinite binary trees.*

Syntactic sugar:

```
(make-tree root left right) =def= (list root
                    (lambda () left)
                    (lambda () right))
```

```
(define root car)
(define (left-tree tree) ((cadr tree)))
(define (right-tree tree) ((caddr tree)))
```

Suppose S1 and S2 are streams, and you want to make a tree of all the possible interleavings of S1 and S2 ...

```
(define (interleave rootvalue S1 S2)
  (make-tree rootvalue
    (interleave (stream-car S1) (stream-cdr S1) S2)
    (interleave (stream-car S2) S1 (stream-cdr S2))))
```
Joint bank accounts: “Don’t cross the streams!”

One-client bank account, with a stream of deposits (- means withdrawal)

```
(define (make-deposit-account balance deposit-stream)
  (cons-stream
   balance
   (make-deposit-account (+ balance (head deposit-stream))
                        (tail deposit-stream))))
```

Two-client account (with two streams of deposits):

```
(define (joint-account balance harry-stream anne-stream)
  (make-tree balance
              (joint-account (+ balance (head harry-stream))
                            (tail harry-stream)
                            anne-stream)
              (joint-account (+ balance (head anne-stream))
                            harry-stream
                            (tail anne-stream))))
```

Every path through the tree indicates a specific interleaving...
Dr. Egon Spengler: There's something very important I forgot to tell you.
Dr. Peter Venkman: What?
Dr. Egon Spengler: Don't cross the streams.
Dr. Peter Venkman: Why?
Dr. Egon Spengler: It would be bad.
Dr. Peter Venkman: I'm fuzzy on the whole good/bad thing. What do you mean, "bad"?
Dr. Egon Spengler: Try to imagine all life as you know it stopping instantaneously and every molecule in your body exploding at the speed of light.
Dr. Ray Stantz: Total protonic reversal.
Dr. Peter Venkman: Right. That's bad. Okay. All right. Important safety tip. Thanks, Egon.
\[ h = <h_0, h_1, h_2, \ldots > \]
\[ a = <a_0, a_1, a_2, \ldots > \]
State versus streams

Can state (bank accounts) model phenomena that streams cannot?

Contrast

Account object (as we’ve discussed)

versus

Shared bank account (two streams of deposits)
Mapping a function over an infinite tree:

```
(define (map-tree proc tree)
  (list (proc (root tree))
    (lambda () (map-tree proc (left-tree tree)))
    (lambda () (map-tree proc (right-tree tree))))
```

Picking a (stream) path through an infinite tree:

```
(define (random-path-stream tree)
  (cons-stream (root tree)
    (random-path-tree
     ((pick-random (subtrees tree))))))
```
What’s the difference between programs and data?

Here’s a program:

```
(define (square n) (* n n))
```
What’s distinguishes programs from data?

Philosophical analogue: do we represent knowledge \textit{extensionally} [list], \textit{declaratively} [specification of what is], or \textit{procedurally} [how to]? Here’s a program:

\begin{verbatim}
(define (square n) (* n n))
\end{verbatim}

\textbf{No}… this is data! You see, there’s an infinite stream of squares, and \texttt{(square n)} selects the $n$-th element of this stream…

Maybe a better differentiation is… \textit{static vs. dynamic}? Or: \textit{extensional versus intensional}?
Programs vs. data (see: predicates vs. sets): on the precipice of paradox

Philosophical analogue: *How do we represent knowledge?* Extensionally [list], declaratively [specification of what is], or procedurally [how to]?

The set: \textbf{Even} = \{0, 2, 4, 6, ...\}

The predicate:
\[
\Phi = \text{(define (even? n) (zero? (mod n 2)))}
\]

Are \(\Phi\) and \textbf{Even} equivalent, and in what way?

Is \(\{ x \mid \Phi(x)\}\) a set? Sure... the even numbers!
On the precipice of paradox

Is \{ x \mid \text{Even?}(x) \} a set? Sure... the even numbers!

Try a different property:

\[ \Phi(x) \equiv x \not\in x \]

Now if

\[ R = \{ x \mid \Phi(x) \} = \{ x \mid x \notin x \} \]

is a set,

then \( R \in R \iff R \notin R \)!

How to fix this? (Data) types...
Turing and Wittgenstein: a seminar exchange...
**Wittgenstein:**... Think of the case of the Liar. It is very queer in a way that this should have puzzled anyone---much more extraordinary than you might think... Because the thing works like this: if a man says 'I am lying' we say that it follows that he is not lying, from which it follows that he is lying and so on. Well, so what? You can go on like that until you are black in the face. Why not? It doesn't matter. ...it is just a useless language-game, and why should anyone be excited?

**Turing:** What puzzles one is that one usually uses a contradiction as a criterion for having done something wrong. But in this case one cannot find anything done wrong.

**W:** Yes---and more: nothing has been done wrong, ... where will the harm come?

**T:** The real harm will not come in unless there is an application, in which a bridge may fall down or something of that sort.
W: ... The question is: Why are people afraid of contradictions? It is easy to understand why they should be afraid of contradictions, etc., outside mathematics. The question is: Why should they be afraid of contradictions inside mathematics? Turing says, `Because something may go wrong with the application.' But nothing need go wrong. And if something does go wrong---if the bridge breaks down---then your mistake was of the kind of using a wrong natural law. ...

T: You cannot be confident about applying your calculus until you know that there are no hidden contradictions in it.

(What would Turing have said if he was a programmer?)
W: There seems to me an enormous mistake there. ...
Suppose I convince Rhees of the paradox of the Liar, and he says, `I lie, therefore I do not lie, therefore I lie and I do not lie, therefore we have a contradiction, therefore $2 \times 2 = 369$.'
Well, we should not call this 'multiplication,' that is all...

T: Although you do not know that the bridge will fall if there are no contradictions, yet it is almost certain that if there are contradictions it will go wrong somewhere.

W: But nothing has ever gone wrong that way yet...

Indeed they have! Software errors...