We realized the various complex problems that assignment raises. One could argue that the OOP paradigm was not good at modeling state and thus one needs an alternative approach to modeling state.

In OOP, we identified
time variation in the real world with
time variation in the computer.

If time is measured in discrete steps, then we can model a time function as a (possibly infinite) sequence.

We realized the various complex problems that assignment raises. One could argue that the OOP paradigm was not good at modeling state and thus one needs an alternative approach to modeling state.

In OOP, we identified
time variation in the real world with
time variation in the computer.

If time is measured in discrete steps, then we can model a time function as a (possibly infinite) sequence.

To accomplish this, we introduce new data structure called **streams**.

From an abstract point of view, a stream is simply a sequence. However, straightforward implementation of streams as lists doesn't fully reveal the power of stream processing.

From an abstract point of view, a stream is simply a sequence. However, straightforward implementation of streams as lists doesn't fully reveal the power of stream processing.

As an alternative, we introduce the technique of **delayed evaluation**, which enables us to represent very large (possibly infinite) sequences as streams.

Stream processing lets us model systems that have state without using assignment or mutable data.

In theory, the stream framework can avoid the drawbacks inherent in introducing assignment but it raises difficulties of its own.

Sequences (aka lists) served as standard interfaces for combining program modules.
$\rightarrow$ For example, procedure-1 took a list as input and produced another list as output for the subsequent procedure-2 to use.

We formulated powerful abstractions for manipulating sequences, such as map, filter, and accumulate/reduce.

These abstractions allowed us to write succinct and elegant procedures.

Sequences (aka lists) served as standard interfaces for combining program modules.
$\rightarrow$ For example, procedure-1 took a list as input and produced another list as output for the subsequent procedure-2 to use.

We formulated powerful abstractions for manipulating sequences, such as map, filter, and accumulate/reduce.

These abstractions allowed us to write succinct and elegant procedures.

However, this elegance is bought at the price of severe inefficiency with respect to both the time and space required by our computations.

```
(define (sum-primes a b)
  (define (iter count accum)
    (cond ((> count b) accum)
          ((prime? count)
             (iter (+ count 1) (+ count accum)))
          (else (iter (+ count 1) accum))))
  (iter a 0))
```

```
(define (sum-primes a b)
  (accumulate +
              0
              (filter prime?
                      (enumerate-interval a b))))
```

Figure: Two versions with same output

The first program needs to store only the sum being accumulated. In contrast, in the second program

1. the interval (a,b) is enumerated
2. then you filter out all the primes
3. then you perform the sum.

Each step has to wait for the previous step to finish and each step requires large storage.

This problem is not apparent when the range is small. For large/infinite ranges that can be a problem.

The inefficiency in using lists becomes painfully apparent if we think of using the sequence paradigm to compute the second prime in the interval from 10^5 to 10^{10}

Why produce the entire interval before checking for primality?

When we represent manipulations on sequences as transformations of lists, our programs must construct and copy data structures (which may be huge) at every step of a process.

In a more traditional programming style, we would interleave the enumeration and the filtering, and stop when we reached the second prime.
i.e one would probably come up with a code which resembles the first solution.

In a more traditional programming style, we would interleave the enumeration and the filtering, and stop when we reached the second prime.
i.e one would probably come up with a code which resembles the first solution.

Streams allow us to use sequence manipulations without incurring the costs of manipulating sequences as lists.

With streams we can achieve the best of both worlds:

1. formulate programs elegantly as sequence manipulations.
2. attain the efficiency of incremental computation.

The basic idea is to arrange to construct a stream only partially, and to pass the partial construction to the program that consumes the stream.

The basic idea is to arrange to construct a stream only partially, and to pass the partial construction to the program that consumes the stream.

- What if the procedure attempts to access a part of stream not yet constructed?

The basic idea is to arrange to construct a stream only partially, and to pass the partial construction to the program that consumes the stream.

- What if the procedure attempts to access a part of stream not yet constructed?

  The stream will automatically construct just enough more of itself to produce the required part, thus preserving the illusion that the entire stream exists.

The basic idea is to arrange to construct a stream only partially, and to pass the partial construction to the program that consumes the stream.

- What if the procedure attempts to access a part of stream not yet constructed?

  The stream will automatically construct just enough more of itself to produce the required part, thus preserving the illusion that the entire stream exists.

We will write programs as if we were processing complete sequences.

We will design our stream implementation to automatically and transparently interleave

- the construction of the stream with
- the use of the stream.

We will write programs as if we were processing complete sequences.

We will design our stream implementation to automatically and transparently interleave

- the construction of the stream with
- the use of the stream.

On the surface, streams are just lists with different names for the procedures that manipulate them.

- Constructor $\rightarrow$ cons-stream,
- Two selectors $\rightarrow$ stream-car and stream-cdr.

There is a distinguishable object, **the-empty-stream** (like nil in lists),
However, this object cannot be the result of any cons-stream operation, and it can be identified with the predicate stream-null?
Thus we can make and use streams in just the same way as we can make and use lists.

• How to make the stream implementation automatically and transparently interleave the construction of a stream with its use?
$\rightarrow$ Evaluate the cdr of a stream when it is accessed by the stream-cdr procedure rather than when the stream is constructed by cons-stream.

• How to make the stream implementation automatically and transparently interleave the construction of a stream with its use?
→ Evaluate the cdr of a stream when it is accessed by the stream-cdr procedure rather than when the stream is constructed by cons-stream.

As a data abstraction, streams are the same as lists.
The difference is the time at which the elements are evaluated.

- Lists → car and cdr are evaluated at construction time.
- Streams → cdr is evaluated at selection time.

This is analogous to the choice of reducing a rational number at time of constructing it or at time of displaying it.

Our implementation of streams will be based on a special form called **delay**.
Evaluating (delay ⟨exp⟩) does not evaluate the expression ⟨exp⟩, but rather returns a so-called delayed object, which we can think of as a "promise" to evaluate ⟨exp⟩ at some future time.

Our implementation of streams will be based on a special form called **delay**.
Evaluating (delay ⟨exp⟩) does not evaluate the expression ⟨exp⟩, but rather returns a so-called delayed object, which we can think of as a "promise" to evaluate ⟨exp⟩ at some future time.

The counter-part to delay is a procedure called **force** that takes a delayed object as argument and performs the evaluation—in effect, forcing the delay to fulfill its promise.

- (delay <exp>) is syntactic sugar for (λ () <exp>)
- force simply calls the procedure (of no arguments) produced by delay, so we can implement force as a procedure:

  (define (force delayed-object) (delayed-object))

- (delay <exp>) is syntactic sugar for (λ () <exp>)
- force simply calls the procedure (of no arguments) produced by delay, so we can implement force as a procedure:

  (define (force delayed-object) (delayed-object))

- stream-cons is a special form defined so that (stream-cons $< a >< b >$) $\equiv$ (cons $< a >$ (delay $< b >$))

```
; Creates a pair whose's second part is a promise
(define s (cons 10 (delay ((λ (x) (* x x)) 5))))

s
(car s)
; (cdr s) does not evaluate the expression
(cdr s)
; One has to use force to evaluate a promise
(force (cdr s))
```

• If one makes use of the cons used for constructing immutable pairs, it would have the following effect

```
(10 . #<promise:.../delay-and-force.rkt:6:19>)
10
#<promise:.../delay-and-force.rkt:6:19>
25
```

Delay **decouples**
the apparent order of events in our programs from
the actual order of events that happened in the machine.

For example,

- Suppose we have a recipe (set of instructions) to prepare a dish. The recipe implicitly defines an order in which the items need to prepared.
  However the notion of stream would then allow us to add salt at the beginning and if we need more salt, then we can add more salt at any later stage.

  We are not confined to add the recipe recommmended salt at some fixed moments in the execution of the recipe.

By devising delay(streams) we are giving up the idea that our procedures as they run mirror some notion of time.

This gives delay the freedom to arrange the order of events in the computation the way it likes.

If one had created s as a pair with the delay

$$(define\ s\ (cons\ 10\ (\underline{delay}\ (\lambda\ (x)\ (*\ x\ x))\ 5)))$$

One might be required to recompute the second element everytime we try to access it.

For infinite length streams, this can be computationally very inefficient.

If one had created s as a pair with the delay

$$(define\ s\ (cons\ 10\ (\underline{delay}\ (\lambda\ (x)\ (*\ x\ x))\ 5)))$$

One might be required to recompute the second element everytime we try to access it.

For infinite length streams, this can be computationally very inefficient.

The solution is to build delayed objects so that the first time they are forced, they store the value that is computed.
This is called **memoization**.

We implement delay as a special-purpose memoized procedure.

One way to accomplish this is to use the following procedure, which

- takes as argument a procedure (of no arguments) and
- returns a memoized version of the procedure.

The first time the memoized procedure is run, it saves the computed result.

On subsequent evaluations, it simply returns the result.

```
(define (memo-proc proc)
  (let ((already-run? false) (result false))
    (lambda ()
      (if (not already-run?)
          (begin (set! result (proc))
                 (set! already-run? true)
                 result)
          result))))
```

```
(define (memo-proc proc)
  (let ((already-run? false) (result false))
    (lambda ()
      (if (not already-run?)
          (begin (set! result (proc))
                 (set! already-run? true)
                 result)
          result))))
```

delay is then defined so that (delay <exp>) is equivalent to

```
(memo-proc (lambda () ⟨exp⟩))
```

Memo-proc is a type of one time assignment which possibly has less side-effects compared to a normal assignment (using set!).
The library racket/stream by default uses memoization.

One could create infinitely long sequences using streams For example,

```
; Infinite sequence of 3s aka (3 3 3 ...)
(define threes (stream-cons 3 threes))


; Natural numbers starting from 4
(define integers
  (stream-cons 4
     (stream-map (lambda (x) (+ x 1))
        integers)))
```

Notice that the first sequence is constructed implicitly, while the second sequence has an explicit procedure to compute each element.

One could also make use of some external procedure to create elements of the stream.

```
; Stream consisting of perfect squares
; Stream uses the i^th element to produce (i+1)^th element.

(define (square x) (* x x))

(define perfect-squares
  (stream-cons 1
     (stream-map (lambda (x) (square (+ 1 (sqrt x))))
        perfect-squares)))
```

In case of fibgen, the last two elements are used to produce a new element of the sequence.

```
; Generating infinite sequence of fibonacci numbers

(define (fibgen a b) (stream-cons a (fibgen b (+ a b))))
(define fibs (fibgen 0 1))

; fibs -> (fibgen 0 1)
;      -> (stream-cons 0 (fibgen 1 1))
;      -> (stream-cons 0 (stream-cons 1 (fib-gen 1 2)))
;      -> (stream-cons 0 (stream-cons 1 (stream-cons 1 (fib-gen 2 3))
```

One could also generate these streams implicitly.

```
(define (add-streams s1 s2) (stream-map + s1 s2))

(define implicit-fibs
  (stream-cons 0
               (stream-cons 1
                            (add-streams (stream-rest implicit-fibs) implicit-fibs))))
```

This definition says that fibs is a stream beginning with 0 and 1, such that the rest of the stream can be generated by adding fibs to itself shifted by one place

```
          1 1 2 3  ... (stream-rest fibs)
          0 1 1 2  ...  fibs
      0 1 1 2 3 5       fibs
index 0 1 2 3 4 6
```

Streams with delayed evaluation can be a powerful modeling tool, providing many of the benefits of local state and assignment.

Moreover, they avoid some of the theoretical tangles that accompany the introduction of assignment into a programming language.

The stream approach allows us to build systems with different module boundaries than systems organized around assignment to state variables.

For example,
we can think of an entire account statement as a focus of interest, rather than the values of the balance at individual moments.

This makes it convenient to combine and compare components of state from different moments.

We have previously looked at iterative processes, which proceed by updating state variables.

We know now that we can represent state as

- a "timeless" stream of values rather than as a set of variables to be updated.

We had the idea to generate a sequence of better and better guesses for the square root of x by applying over and over again the procedure that improves guesses

```
(define (sqrt-improve guess x)
  (average guess (/ x guess)))
```

```
(define (sqrt-stream x)
  (define guesses
    (cons-stream
     1.0
     (stream-map (lambda (guess) (sqrt-improve guess x))
                 guesses)))
  guesses)
```

This would produce a stream of guesses for the square root of the input.

```
(display-stream (sqrt-stream 2))
1.
1.5
1.4166666666666665
1.4142156862745097
```

We can generate more and more terms of the stream to get better and better guesses.