Lecture 20: Scheme II

Brian Hou
July 26, 2016

Announcements

Announcements

- Project 3 is due today (7/26)

Announcements

- Project 3 is due today (7/26)
- Homework 8 is due tomorrow (7/27)

Announcements

- Project 3 is due today (7/26)
- Homework 8 is due tomorrow (7/27)
- Quiz 7 on Thursday (7/28) at the beginning of lecture

Announcements

- Project 3 is due today (7/26)
- Homework 8 is due tomorrow (7/27)
- Quiz 7 on Thursday (7/28) at the beginning of lecture
    - May cover mutable linked lists, mutable trees, or Scheme I

Announcements

- Project 3 is due today (7/26)
- Homework 8 is due tomorrow (7/27)
- Quiz 7 on Thursday (7/28) at the beginning of lecture
  - May cover mutable linked lists, mutable trees, or Scheme I
- Opportunities to earn back points

Announcements

- Project 3 is due today (7/26)
- Homework 8 is due tomorrow (7/27)
- Quiz 7 on Thursday (7/28) at the beginning of lecture
  - May cover mutable linked lists, mutable trees, or Scheme I
- Opportunities to earn back points
  - Hog composition revisions due tomorrow (7/27)

Announcements

- Project 3 is due today (7/26)
- Homework 8 is due tomorrow (7/27)
- Quiz 7 on Thursday (7/28) at the beginning of lecture
  - May cover mutable linked lists, mutable trees, or Scheme I
- Opportunities to earn back points
  - Hog composition revisions due tomorrow (7/27)
  - Maps composition revisions due Saturday (7/30)

Announcements

- Project 3 is due today (7/26)
- Homework 8 is due tomorrow (7/27)
- Quiz 7 on Thursday (7/28) at the beginning of lecture
  - May cover mutable linked lists, mutable trees, or Scheme I
- Opportunities to earn back points
  - Hog composition revisions due tomorrow (7/27)
  - Maps composition revisions due Saturday (7/30)
  - Homework 7 AutoStyle portion due tomorrow (7/27)

Roadmap

- Introduction
- Functions
- Data
- Mutability
- Objects
- Interpretation
- Paradigms
- Applications

Roadmap

- Introduction
- Functions
- Data
- Mutability
- Objects
- Interpretation
- Paradigms
- Applications

- This week (Interpretation), the goals are:

Roadmap

- Introduction
- Functions
- Data
- Mutability
- Objects
- Interpretation
- Paradigms
- Applications

- This week (Interpretation), the goals are:
    - To learn a new language, Scheme, in two days!

Roadmap

Introduction

Functions

Data

Mutability

Objects

Interpretation

Paradigms

Applications

- This week (Interpretation), the goals are:
  - To learn a new language, Scheme, in two days!
  - To understand how interpreters work, using Scheme as an example

The **let** Special Form

The **let** Special Form

- The **let** special form defines local variables and evaluates expressions in this new environment

The **let** Special Form (demo)

- The **let** special form defines local variables and evaluates expressions in this new environment

The **let** Special Form (demo)

- The **let** special form defines local variables and evaluates expressions in this new environment

```
scm> (define x 1)
x
scm> (let ((x 10) (y 20))
       (+ x y))
30
scm> x
1
```

Tail Recursion

Factorial (Again)

Factorial (Again)

```
(define (fact n)
```

Factorial (Again)

```
(define (fact n)
  (if (= n 0)
```

Factorial (Again)

```
(define (fact n)
  (if (= n 0)
    1
```

Factorial (Again)

```
(define (fact n)
  (if (= n 0)
    1
    (* n (fact (- n 1)))))
```

Factorial (Again)

```scheme
(define (fact n)
  (if (= n 0)
    1
    (* n (fact (- n 1)))))
```

```
scm> (fact 10)
```

Factorial (Again)

```
(define (fact n)
  (if (= n 0)
    1
    (* n (fact (- n 1)))))
```

```
scm> (fact 10)
scm> (fact 1000)
```

Factorial (Again)

```
(define (fact n)
  (if (= n 0)
    1
    (* n (fact (- n 1)))))
```

```
(define (fact n)
```

```
scm> (fact 10)
scm> (fact 1000)
```

Factorial (Again)

```scheme
(define (fact n)
  (if (= n 0)
    1
    (* n (fact (- n 1)))))
```

```scheme
(define (fact n)
  (define (helper n prod)
```

```
scm> (fact 10)
scm> (fact 1000)
```

Factorial (Again)

```
(define (fact n)
  (if (= n 0)
    1
    (* n (fact (- n 1)))))
```

```
(define (fact n)
  (define (helper n prod)
    (if (= n 0)
```

```
scm> (fact 10)
scm> (fact 1000)
```

Factorial (Again)

```
(define (fact n)
  (if (= n 0)
    1
    (* n (fact (- n 1)))))
```

```
(define (fact n)
  (define (helper n prod)
    (if (= n 0)
      prod
```

```
scm> (fact 10)
scm> (fact 1000)
```

Factorial (Again)

```scheme
(define (fact n)
  (if (= n 0)
    1
    (* n (fact (- n 1)))))
```

```scheme
(define (fact n)
  (define (helper n prod)
    (if (= n 0)
      prod
      (helper (- n 1) (* n prod))))
```

```
scm> (fact 10)
scm> (fact 1000)
```

Factorial (Again)

```scheme
(define (fact n)
  (if (= n 0)
    1
    (* n (fact (- n 1)))))
```

```scheme
(define (fact n)
  (define (helper n prod)
    (if (= n 0)
      prod
      (helper (- n 1) (* n prod))))
  (helper n 1))
```

```
scm> (fact 10)
scm> (fact 1000)
```

Factorial (Again) (demo)

```
(define (fact n)
  (if (= n 0)
    1
    (* n (fact (- n 1)))))
```

```
(define (fact n)
  (define (helper n prod)
    (if (= n 0)
      prod
      (helper (- n 1) (* n prod))))
  (helper n 1))
```

```
scm> (fact 10)
scm> (fact 1000)
```

Tail Recursion

Tail Recursion

The Revised^7 Report on the Algorithmic Language Scheme:

Tail Recursion

The Revised^7 Report on the Algorithmic Language Scheme:

> "Implementations of Scheme are required to be properly tail-recursive. This allows the execution of an iterative computation in constant space, even if the iterative computation is described by a syntactically recursive procedure."

Tail Recursion

The Revised[7] Report on the Algorithmic Language Scheme:

> "Implementations of Scheme are required to be properly tail-recursive. This allows the execution of an iterative computation in constant space, even if the iterative computation is described by a syntactically recursive procedure."

```scheme
(define (fact n)
  (define (helper n prod)
    (if (= n 0) prod (helper (- n 1) (* n prod))))
  (helper n 1))
```

Tail Recursion

The Revised[7] Report on the Algorithmic Language Scheme:

> "Implementations of Scheme are required to be properly tail-recursive. This allows the execution of an iterative computation in constant space, even if the iterative computation is described by a syntactically recursive procedure."

How? Eliminate the middleman!

```
(define (fact n)
  (define (helper n prod)
    (if (= n 0) prod (helper (- n 1) (* n prod))))
  (helper n 1))
```

Tail Calls

Tail Calls

- A procedure call that has not yet returned is *active*

Tail Calls

- A procedure call that has not yet returned is *active*
- Some procedure calls are *tail calls*

Tail Calls

- A procedure call that has not yet returned is *active*
- Some procedure calls are *tail calls*
- Scheme implementations should support an *unbounded number* of active tail calls using only a *constant* amount of space

Tail Calls

- A procedure call that has not yet returned is *active*
- Some procedure calls are *tail calls*
- Scheme implementations should support an *unbounded number* of active tail calls using only a *constant* amount of space
- A tail call is a call expression in a tail context:

Tail Calls

- A procedure call that has not yet returned is *active*
- Some procedure calls are *tail calls*
- Scheme implementations should support an *unbounded number* of active tail calls using only a *constant* amount of space
- A tail call is a call expression in a tail context:
  - The last body sub-expression in a **lambda**

Tail Calls

- A procedure call that has not yet returned is *active*
- Some procedure calls are *tail calls*
- Scheme implementations should support an *unbounded number* of active tail calls using only a *constant* amount of space
- A tail call is a call expression in a tail context:
  - The last body sub-expression in a **lambda**
  - The consequent and alternative in a tail context **if**

Tail Calls

- A procedure call that has not yet returned is *active*
- Some procedure calls are *tail calls*
- Scheme implementations should support an *unbounded number* of active tail calls using only a *constant* amount of space
- A tail call is a call expression in a tail context:
  - The last body sub-expression in a **lambda**
  - The consequent and alternative in a tail context **if**
  - All non-predicate sub-expressions in a tail context **cond**

Tail Calls

- A procedure call that has not yet returned is *active*
- Some procedure calls are *tail calls*
- Scheme implementations should support an *unbounded number* of active tail calls using only a *constant* amount of space
- A tail call is a call expression in a tail context:
  - The last body sub-expression in a **lambda**
  - The consequent and alternative in a tail context **if**
  - All non-predicate sub-expressions in a tail context **cond**
  - The last sub-expression in a tail context **and**, **or**, **begin**, or **let**

Tail Contexts

- A tail call is a call expression in a tail context:
  - The last body sub-expression in a **lambda**
  - The consequent and alternative in a tail context **if**
  - All non-predicate sub-expressions in a tail context **cond**
  - The last sub-expression in a tail context **and**, **or**, **begin**, or **let**

```
(define (fact n)
  (define (helper n prod)
    (if (= n 0) prod (helper (- n 1) (* n prod))))
  (helper n 1))
```

Tail Contexts

- A tail call is a call expression in a tail context:
  - The last body sub-expression in a **lambda**
  - The consequent and alternative in a tail context **if**
  - All non-predicate sub-expressions in a tail context **cond**
  - The last sub-expression in a tail context **and**, **or**, **begin**, or **let**

```
(define (fact n)
  (define (helper n prod)
    (if (= n 0) prod (helper (- n 1) (* n prod))))
  (helper n 1))
```

Tail Contexts

- A tail call is a call expression in a tail context:
  - The last body sub-expression in a **lambda**
  - The consequent and alternative in a tail context **if**
  - All non-predicate sub-expressions in a tail context **cond**
  - The last sub-expression in a tail context **and, or, begin**, or **let**

```
(define (fact n)
  (define (helper n prod)
    (if (= n 0) prod (helper (- n 1) (* n prod))))
  (helper n 1))
```

Tail Contexts

- A tail call is a call expression in a tail context:
  - The last body sub-expression in a **lambda**
  - The consequent and alternative in a tail context **if**
  - All non-predicate sub-expressions in a tail context **cond**
  - The last sub-expression in a tail context **and**, **or**, **begin**, or **let**

```
(define (fact n)
  (define (helper n prod)
    (if (= n 0) prod (helper (- n 1) (* n prod))))
  (helper n 1))
```

Tail Contexts

- A tail call is a call expression in a tail context:
  - The last body sub-expression in a **lambda**
  - The consequent and alternative in a tail context **if**
  - All non-predicate sub-expressions in a tail context **cond**
  - The last sub-expression in a tail context **and, or, begin**, or **let**

```
(define (fact n)
  (define (helper n prod)
   (if (= n 0) prod (helper (- n 1) (* n prod))))
  (helper n 1))
```

Example: Length

Example: Length

```
(define (length s)
```

Example: Length

```
(define (length s)
  (if (null? s) 0
```

Example: Length

```
(define (length s)
  (if (null? s) 0
    (+ 1 (length (cdr s)))))
```

Example: Length

```scheme
(define (length s)
  (if (null? s) 0
    (+ 1 (length (cdr s)))))
```

Example: Length

```
(define (length s)
  (if (null? s) 0
    (+ 1 (length (cdr s))) ) )
```

Example: Length

```
(define (length s)
  (if (null? s) 0
    (+ 1 (length (cdr s)))))
```

Example: Length

```
(define (length s)
  (if (null? s) 0
    (+ 1 (length (cdr s)))))
```

Not a tail context

Example: Length

```
(define (length s)
  (if (null? s) 0
    (+ 1 (length (cdr s)) ) ) )
```

Not a tail context

- A call expression is not a tail call if more computation is still required in the calling procedure

Example: Length

```
(define (length s)
  (if (null? s) 0
    (+ 1 (length (cdr s)))))
```

Not a tail context

- A call expression is not a tail call if more computation is still required in the calling procedure
- Linear recursive procedures can often be rewritten to use tail calls

Example: Length

```
(define (length s)
  (if (null? s) 0
    (+ 1 (length (cdr s))) ) )
```

Not a tail context

- A call expression is not a tail call if more computation is still required in the calling procedure
- Linear recursive procedures can often be rewritten to use tail calls

```
(define (length-tail s)
```

Example: Length

```
(define (length s)
  (if (null? s) 0
    (+ 1 (length (cdr s))) ) )
```

Not a tail context

- A call expression is not a tail call if more computation is still required in the calling procedure
- Linear recursive procedures can often be rewritten to use tail calls

```
(define (length-tail s)
  (define (length-iter s n)
```

Example: Length

```
(define (length s)
  (if (null? s) 0
    (+ 1 (length (cdr s))) ) )
```

Not a tail context

- A call expression is not a tail call if more computation is still required in the calling procedure
- Linear recursive procedures can often be rewritten to use tail calls

```
(define (length-tail s)
  (define (length-iter s n)
    (if (null? s) n
```

Example: Length

```
(define (length s)
  (if (null? s) 0
    (+ 1 (length (cdr s)))))
```

Not a tail context

- A call expression is not a tail call if more computation is still required in the calling procedure
- Linear recursive procedures can often be rewritten to use tail calls

```
(define (length-tail s)
  (define (length-iter s n)
    (if (null? s) n
      (length-iter (cdr s) (+ 1 n))))
```

Example: Length

```
(define (length s)
  (if (null? s) 0
    (+ 1 (length (cdr s)))))
```

Not a tail context

- A call expression is not a tail call if more computation is still required in the calling procedure
- Linear recursive procedures can often be rewritten to use tail calls

```
(define (length-tail s)
  (define (length-iter s n)
    (if (null? s) n
      (length-iter (cdr s) (+ 1 n))))
  (length-iter s 0))
```

Example: Length

```scheme
(define (length s)
  (if (null? s) 0
    (+ 1 (length (cdr s)))))
```

Not a tail context

- A call expression is not a tail call if more computation is still required in the calling procedure
- Linear recursive procedures can often be rewritten to use tail calls

```scheme
(define (length-tail s)
  (define (length-iter s n)
    (if (null? s) n
      (length-iter (cdr s) (+ 1 n))))
  (length-iter s 0))
```

Example: Length

```
(define (length s)
  (if (null? s) 0
    (+ 1 (length (cdr s)))))
```

Not a tail context

- A call expression is not a tail call if more computation is still required in the calling procedure
- Linear recursive procedures can often be rewritten to use tail calls

```
(define (length-tail s)
  (define (length-iter s n)
    (if (null? s) n
      (length-iter (cdr s) (+ 1 n))))
  (length-iter s 0))
```

Example: Length

```
(define (length s)
  (if (null? s) 0
    (+ 1 (length (cdr s)))))
```

Not a tail context

- A call expression is not a tail call if more computation is still required in the calling procedure
- Linear recursive procedures can often be rewritten to use tail calls

```
(define (length-tail s)
  (define (length-iter s n)
    (if (null? s) n
      (length-iter (cdr s) (+ 1 n))))
  (length-iter s 0))
```

Lazy Computation

Lazy Computation

Lazy Computation

- Lazy computation means that computation of a value is delayed until that value is needed

Lazy Computation

- Lazy computation means that computation of a value is delayed until that value is needed
  - In other words, values are computed *on demand*

Lazy Computation (demo)

- Lazy computation means that computation of a value is delayed until that value is needed
  - In other words, values are computed *on demand*

Lazy Computation (demo)

- Lazy computation means that computation of a value is delayed until that value is needed
  - In other words, values are computed *on demand*

```
>>> r = range(11111, 1111111111)
>>> r[20149616]
20160726
```

Streams

Streams

- Streams are lazy Scheme lists: the rest of a list is computed only when needed

Streams

- Streams are lazy Scheme lists: the rest of a list is computed only when needed

```
(car (cons 1 2)) -> 1
```

Streams

- Streams are lazy Scheme lists: the rest of a list is computed only when needed

```
(car (cons 1 2)) -> 1

(cdr (cons 1 2)) -> 2
```

Streams

- Streams are lazy Scheme lists: the rest of a list is computed only when needed

```
(car (cons 1 2)) -> 1

(cdr (cons 1 2)) -> 2

(cons 1 (cons 2 nil))
```

Streams

- Streams are lazy Scheme lists: the rest of a list is computed only when needed

```
(car (cons 1 2)) -> 1

(cdr (cons 1 2)) -> 2

(cons 1 (cons 2 nil))
```

Streams

- Streams are lazy Scheme lists: the rest of a list is computed only when needed

`(car (cons 1 2)) -> 1`	`(car        (cons-stream 1 2)) -> 1`
`(cdr (cons 1 2)) -> 2`	
`(cons 1 (cons 2 nil))`	

Streams

- Streams are lazy Scheme lists: the rest of a list is computed only when needed

`(car (cons 1 2)) -> 1`	`(car (cons-stream 1 2)) -> 1`
`(cdr (cons 1 2)) -> 2`	`(cdr-stream (cons-stream 1 2)) -> 2`
`(cons 1 (cons 2 nil))`	

Streams

- Streams are lazy Scheme lists: the rest of a list is computed only when needed

`(car (cons 1 2)) -> 1`	`(car (cons-stream 1 2)) -> 1`
`(cdr (cons 1 2)) -> 2`	`(cdr-stream (cons-stream 1 2)) -> 2`
`(cons 1 (cons 2 nil))`	`(cons-stream 1 (cons-stream 2 nil))`

Streams

- Streams are lazy Scheme lists: the rest of a list is computed only when needed
- Errors only occur when expressions are evaluated

Streams

- Streams are lazy Scheme lists: the rest of a list is computed only when needed
- Errors only occur when expressions are evaluated

```
(cons-stream 1 (/ 1 0))   -> (1 . #[promise (not forced)])
```

Streams

- Streams are lazy Scheme lists: the rest of a list is computed only when needed
- Errors only occur when expressions are evaluated

```
(cons-stream 1 (/ 1 0))    -> (1 . #[promise (not forced)])

(car        (cons-stream 1 (/ 1 0))) -> 1
```

Streams

- Streams are lazy Scheme lists: the rest of a list is computed only when needed
- Errors only occur when expressions are evaluated

```
(cons-stream 1 (/ 1 0))   -> (1 . #[promise (not forced)])

(car        (cons-stream 1 (/ 1 0))) -> 1

(cdr-stream (cons-stream 1 (/ 1 0))) -> ERROR
```

Streams (demo)

- Streams are lazy Scheme lists: the rest of a list is computed only when needed
- Errors only occur when expressions are evaluated

```
(cons-stream 1 (/ 1 0))   -> (1 . #[promise (not forced)])

(car        (cons-stream 1 (/ 1 0))) -> 1

(cdr-stream (cons-stream 1 (/ 1 0))) -> ERROR
```

Infinite Streams

Infinite Streams

- An integer stream is a stream of consecutive integers

Infinite Streams

- An integer stream is a stream of consecutive integers
- The rest of the stream is not computed when the stream is created

Infinite Streams

- An integer stream is a stream of consecutive integers
- The rest of the stream is not computed when the stream is created

```
(define (int-stream start)
```

Infinite Streams

- An integer stream is a stream of consecutive integers
- The rest of the stream is not computed when the stream is created

```
(define (int-stream start)
  (cons-stream
```

Infinite Streams

- An integer stream is a stream of consecutive integers
- The rest of the stream is not computed when the stream is created

```
(define (int-stream start)
  (cons-stream
    start
```

Infinite Streams

- An integer stream is a stream of consecutive integers
- The rest of the stream is not computed when the stream is created

```
(define (int-stream start)
  (cons-stream
    start
    (int-stream (+ start 1))))
```

Infinite Streams (demo)

- An integer stream is a stream of consecutive integers
- The rest of the stream is not computed when the stream is created

```
(define (int-stream start)
  (cons-stream
    start
    (int-stream (+ start 1))))
```

Recursively Defined Streams

Recursively Defined Streams

```
(define ones (cons-stream 1 ones))
```

Recursively Defined Streams

```
(define ones (cons-stream 1 ones))
```

1 1 1 1 1 1 ...

Recursively Defined Streams

```
(define ones (cons-stream 1 ones))
```

1 $\boxed{1 \quad 1 \quad 1 \quad 1 \quad 1 \quad \ldots}$

Recursively Defined Streams

```
(define ones (cons-stream 1 ones))

(define (add-streams s1 s2)
```

1 $\boxed{1 \quad 1 \quad 1 \quad 1 \quad 1 \quad \ldots}$

Recursively Defined Streams

```
(define ones (cons-stream 1 ones))

(define (add-streams s1 s2)
  (cons-stream
```

1 1 1 1 1 1 ...

Recursively Defined Streams

```
(define ones (cons-stream 1 ones))

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

1 1 1 1 1 1 ...

Recursively Defined Streams

```
(define ones (cons-stream 1 ones))

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

1 1 1 1 1 1 ...

Recursively Defined Streams

```
(define ones (cons-stream 1 ones))

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

1 1 1 1 1 1 ...

Recursively Defined Streams

```
(define ones (cons-stream 1 ones))

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

1 1 1 1 1 1 ...

Recursively Defined Streams

```scheme
(define ones (cons-stream 1 ones))

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

(define ints
```

1 1 1 1 1 1 ...

Recursively Defined Streams

```scheme
(define ones (cons-stream 1 ones))

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

(define ints
  (cons-stream 1
```

1 1 1 1 1 1 ...

Recursively Defined Streams

```
(define ones (cons-stream 1 ones))

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

(define ints
  (cons-stream 1
    (add-streams ones ints)))
```

1 1 1 1 1 1 ...

Recursively Defined Streams

```
(define ones (cons-stream 1 ones))

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

1 1 1 1 1 1 ...

```
(define ints
  (cons-stream 1
    (add-streams ones ints)))
```

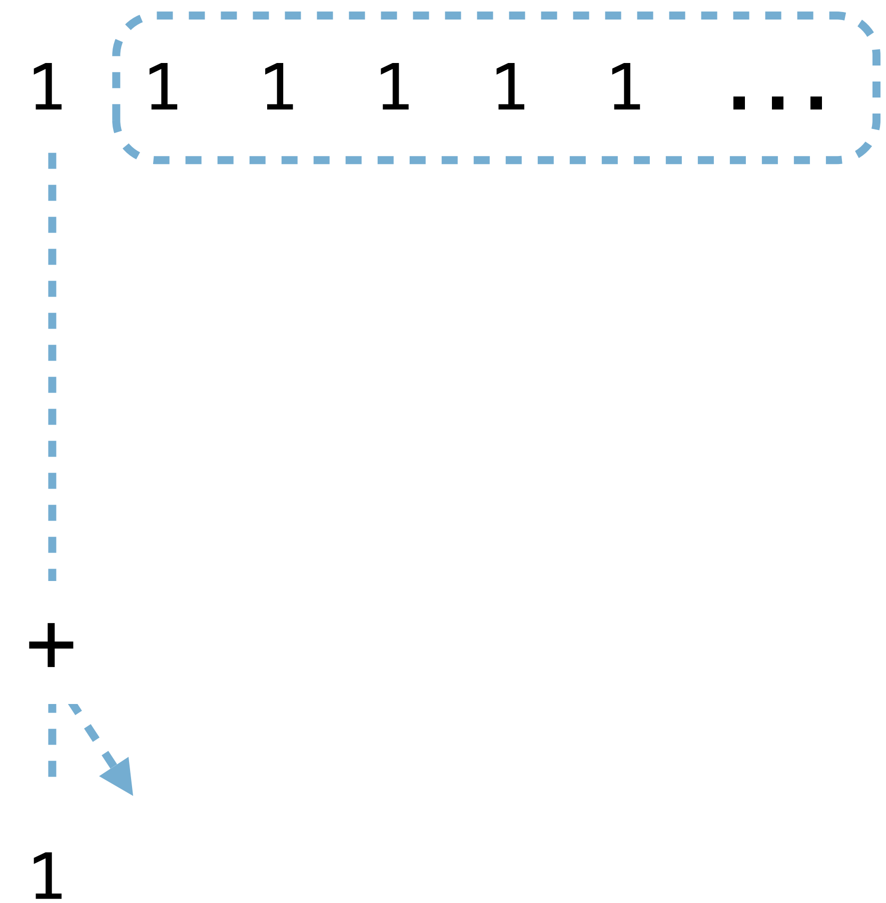
1

Recursively Defined Streams

```scheme
(define ones (cons-stream 1 ones))

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

(define ints
  (cons-stream 1
    (add-streams ones ints)))
```

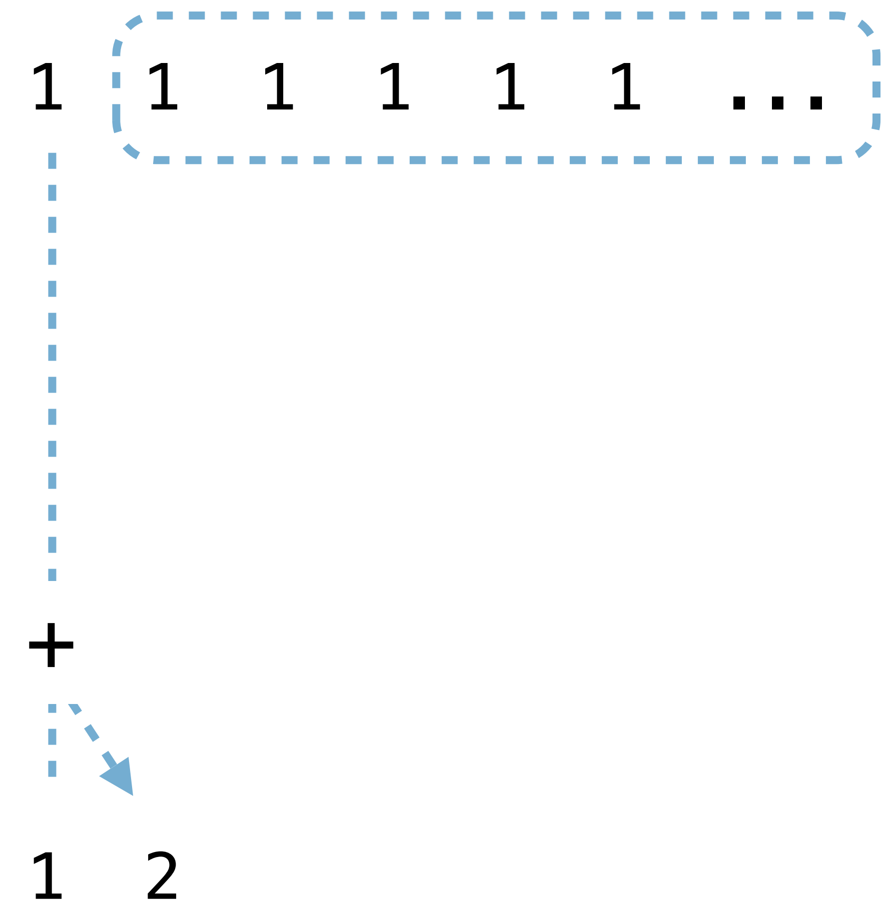


Recursively Defined Streams

```
(define ones (cons-stream 1 ones))

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

(define ints
  (cons-stream 1
    (add-streams ones ints)))
```

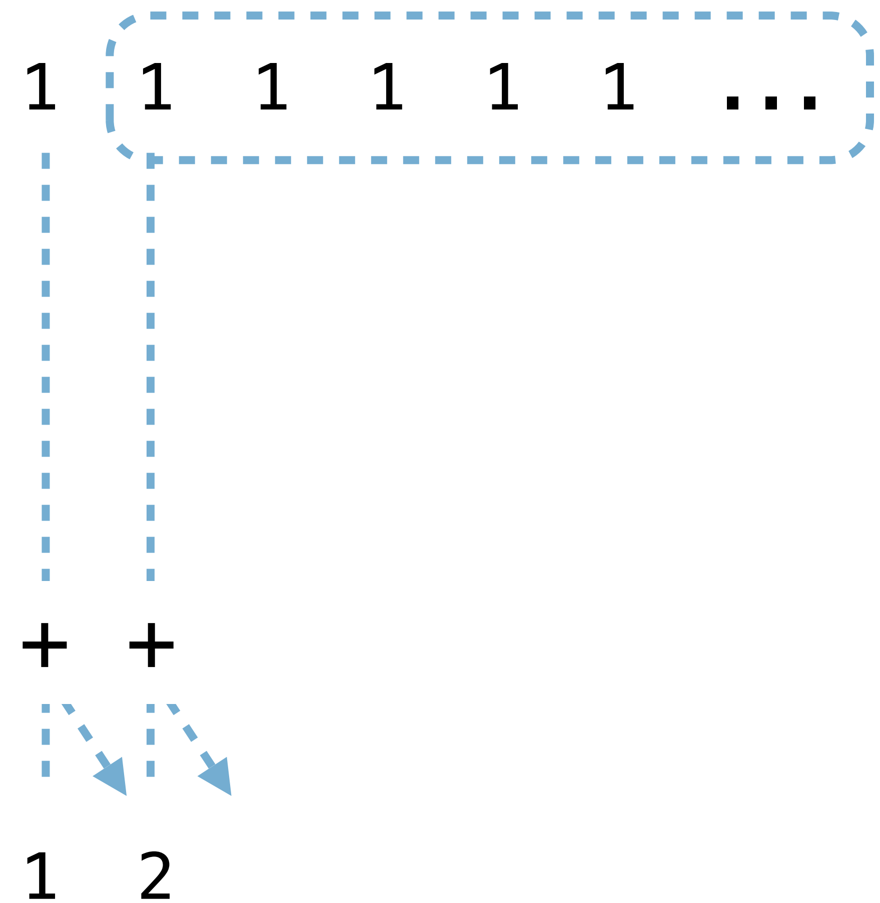


Recursively Defined Streams

```scheme
(define ones (cons-stream 1 ones))

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

(define ints
  (cons-stream 1
    (add-streams ones ints)))
```

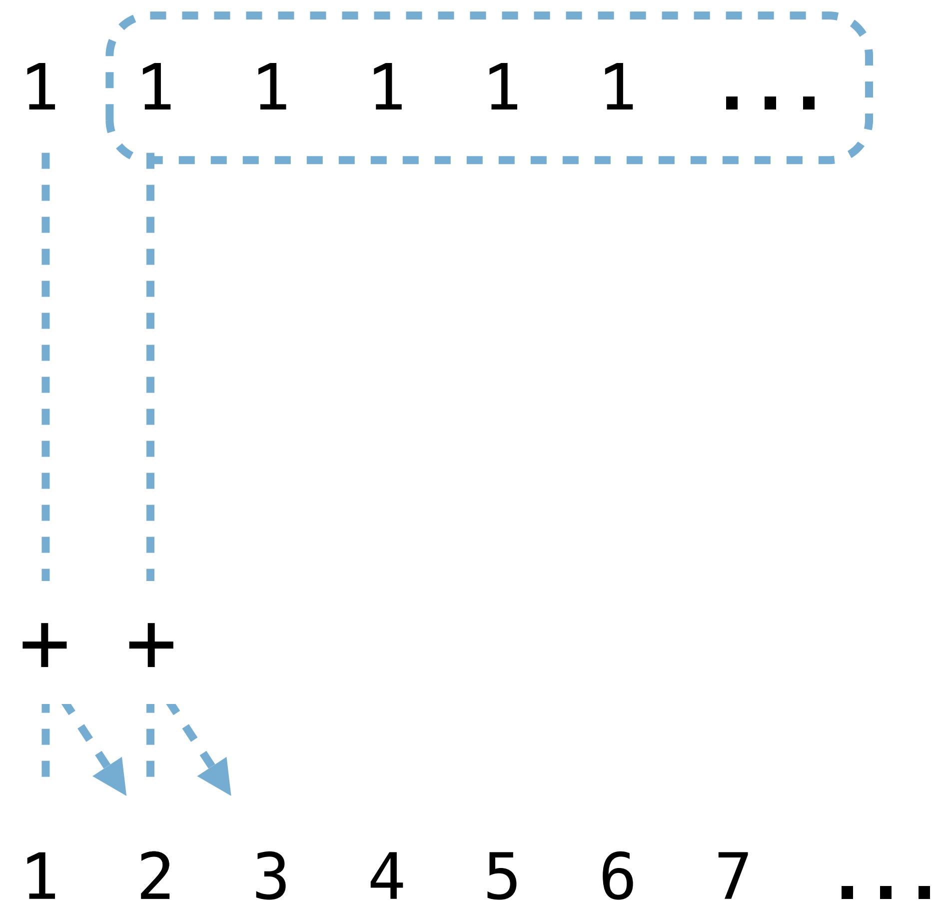


Recursively Defined Streams

```
(define ones (cons-stream 1 ones))

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

(define ints
  (cons-stream 1
    (add-streams ones ints)))
```

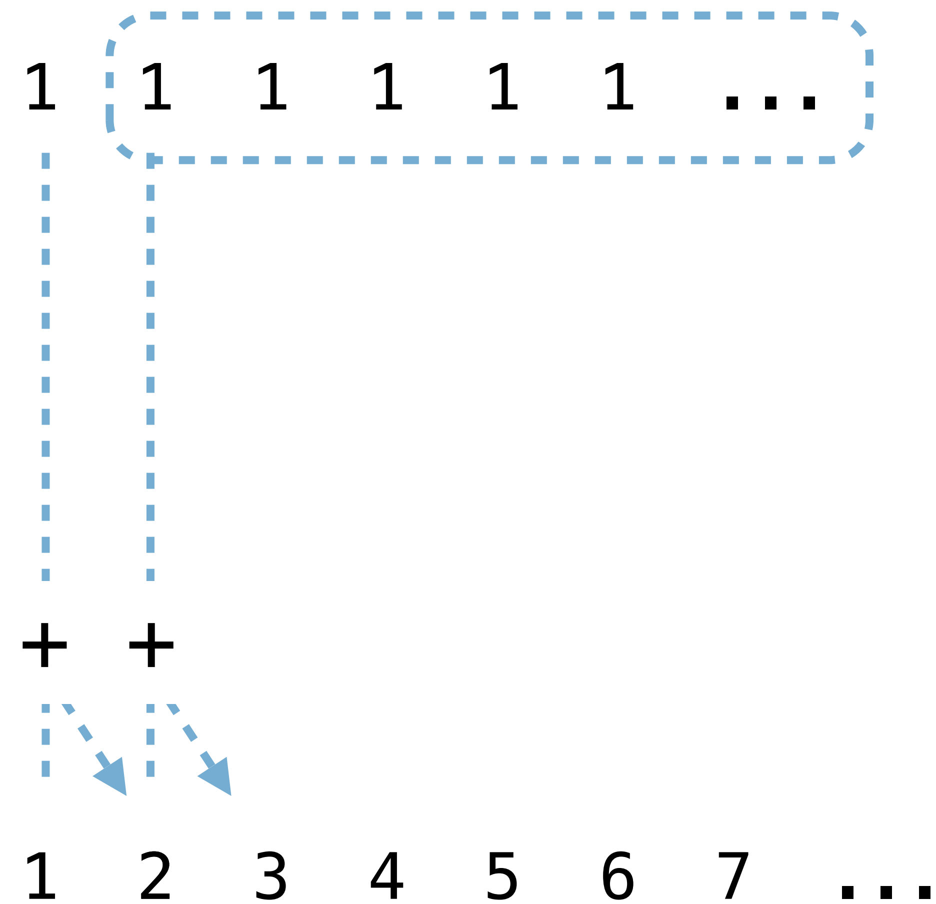


Recursively Defined Streams (demo)

```scheme
(define ones (cons-stream 1 ones))

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

(define ints
  (cons-stream 1
    (add-streams ones ints)))
```



A Stream of Primes

A Stream of Primes

- For a prime k, any larger prime cannot be divisible by k

A Stream of Primes

- For a prime k, any larger prime cannot be divisible by k
- Idea: Filter out all numbers that are divisible by k

A Stream of Primes

- For a prime k, any larger prime cannot be divisible by k
- Idea: Filter out all numbers that are divisible by k
- This idea is called the Sieve of Eratosthenes

A Stream of Primes

- For a prime k, any larger prime cannot be divisible by k
- Idea: Filter out all numbers that are divisible by k
- This idea is called the Sieve of Eratosthenes

2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13

A Stream of Primes

- For a prime k, any larger prime cannot be divisible by k
- Idea: Filter out all numbers that are divisible by k
- This idea is called the Sieve of Eratosthenes

2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13

A Stream of Primes

- For a prime k, any larger prime cannot be divisible by k
- Idea: Filter out all numbers that are divisible by k
- This idea is called the Sieve of Eratosthenes

2, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, 9, ~~10~~, 11, ~~12~~, 13

A Stream of Primes

- For a prime k, any larger prime cannot be divisible by k
- Idea: Filter out all numbers that are divisible by k
- This idea is called the Sieve of Eratosthenes

2, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, 9, ~~10~~, 11, ~~12~~, 13

A Stream of Primes

- For a prime k, any larger prime cannot be divisible by k
- Idea: Filter out all numbers that are divisible by k
- This idea is called the Sieve of Eratosthenes

2, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, ~~9~~, ~~10~~, 11, ~~12~~, 13

A Stream of Primes

- For a prime k, any larger prime cannot be divisible by k
- Idea: Filter out all numbers that are divisible by k
- This idea is called the Sieve of Eratosthenes

2, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, ~~9~~, ~~10~~, 11, ~~12~~, 13

A Stream of Primes (demo)

- For a prime k, any larger prime cannot be divisible by k
- Idea: Filter out all numbers that are divisible by k
- This idea is called the Sieve of Eratosthenes

2, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, ~~9~~, ~~10~~, 11, ~~12~~, 13


Break!

Symbolic Programming

Symbolic Programming

Symbolic Programming

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

Symbolic Programming

procedure

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

Symbolic Programming

procedure

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

```
'(define (square x) (* x x))
```

Symbolic Programming

procedure

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

```
'(define (square x) (* x x))
```

list

Symbolic Programming

procedure

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

```
'(define (square x) (* x x))
```

list

- Lists can be manipulated with **car** and **cdr**

Symbolic Programming

procedure

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

```
'(define (square x) (* x x))
```

list

- Lists can be manipulated with **car** and **cdr**
- Lists can created and combined with **cons, list, append**

Symbolic Programming

procedure

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

```
'(define (square x) (* x x))
```

list

- Lists can be manipulated with **car** and **cdr**
- Lists can created and combined with **cons, list, append**
- We can rewrite Scheme procedures using these tools!

List Comprehensions in Scheme

List Comprehensions in Scheme

```
((* x x) for x in '(1 2 3 4) if (> x 2))
```

List Comprehensions in Scheme

```
((* x x) for x in '(1 2 3 4) if (> x 2))
```

```
(map (lambda (x) (* x x))
     (filter (lambda (x) (> x 2)) '(1 2 3 4)))
```

List Comprehensions in Scheme

```
((* x x) for x in '(1 2 3 4) if (> x 2))
```

```
(map (lambda (x) (* x x))
     (filter (lambda (x) (> x 2)) '(1 2 3 4)))
```

List Comprehensions in Scheme

```
((* x x) for x in '(1 2 3 4) if (> x 2))
```

```
(map (lambda (x) (* x x))
     (filter (lambda (x) (> x 2)) '(1 2 3 4)))
```

List Comprehensions in Scheme

```
((* x x) for x in '(1 2 3 4) if (> x 2))
```

```
(map (lambda (x) (* x x))
     (filter (lambda (x) (> x 2)) '(1 2 3 4)))
```

List Comprehensions in Scheme

```
((* x x) for x in '(1 2 3 4) if (> x 2))
```

```
(map (lambda (x) (* x x))
     (filter (lambda (x) (> x 2)) '(1 2 3 4)))
```

List Comprehensions in Scheme

```
exp          ((* x x) for x in '(1 2 3 4) if (> x 2))
```

```
(map (lambda (x) (* x x))
     (filter (lambda (x) (> x 2)) '(1 2 3 4)))
```

List Comprehensions in Scheme

```
exp            ((* x x) for x in '(1 2 3 4) if (> x 2))
                                                (* x x)
```

```
(map (lambda (x) (* x x))
     (filter (lambda (x) (> x 2)) '(1 2 3 4)))
```

List Comprehensions in Scheme

```
exp          ((* x x) for x in '(1 2 3 4) if (> x 2))
(car exp)                                       (* x x)
```

```
(map (lambda (x) (* x x))
     (filter (lambda (x) (> x 2)) '(1 2 3 4)))
```

List Comprehensions in Scheme

```
exp             ((* x x) for x in '(1 2 3 4) if (> x 2))
(car exp)                                      (* x x)
                                                     x
```

```
(map (lambda (x) (* x x))
     (filter (lambda (x) (> x 2)) '(1 2 3 4)))
```

List Comprehensions in Scheme

```
exp                ((* x x) for x in '(1 2 3 4) if (> x 2))
(car exp)                                            (* x x)
(car (cddr exp))                                           x
```

```
   (map (lambda (x) (* x x))
        (filter (lambda (x) (> x 2)) '(1 2 3 4)))
```

List Comprehensions in Scheme

```
exp                 ((* x x) for x in '(1 2 3 4) if (> x 2))
(car exp)                                              (* x x)
(car (cddr exp))                                             x
                                                   '(1 2 3 4)
```

```
(map (lambda (x) (* x x))
     (filter (lambda (x) (> x 2)) '(1 2 3 4)))
```

List Comprehensions in Scheme

```
exp                  ((* x x) for x in '(1 2 3 4) if (> x 2))
(car exp)                                                (* x x)
(car (cddr exp))                                               x
(car (cddr (cddr exp)))                              '(1 2 3 4)
```

```
(map (lambda (x) (* x x))
     (filter (lambda (x) (> x 2)) '(1 2 3 4)))
```

List Comprehensions in Scheme

exp	((* x x) for x in '(1 2 3 4) if (> x 2))
(car exp)	(* x x)
(car (cddr exp))	x
(car (cddr (cddr exp)))	'(1 2 3 4)
	(> x 2)

```
(map (lambda (x) (* x x))
     (filter (lambda (x) (> x 2)) '(1 2 3 4)))
```

List Comprehensions in Scheme

```
exp                          ((* x x) for x in '(1 2 3 4) if (> x 2))
(car exp)                                                     (* x x)
(car (cddr exp))                                                    x
(car (cddr (cddr exp)))                                   '(1 2 3 4)
(car (cddr (cddr (cddr exp))))                                (> x 2)
```

```
(map (lambda (x) (* x x))
     (filter (lambda (x) (> x 2)) '(1 2 3 4)))
```

List Comprehensions in Scheme

`exp`	`((* x x) for x in '(1 2 3 4) if (> x 2))`
`(car exp)`	`(* x x)`
`(car (cddr exp))`	`x`
`(car (cddr (cddr exp)))`	`'(1 2 3 4)`
`(car (cddr (cddr (cddr exp))))`	`(> x 2)`

```
(map (lambda (x) (* x x))
     (filter (lambda (x) (> x 2)) '(1 2 3 4)))
```

List Comprehensions in Scheme

exp	((* x x) for x in '(1 2 3 4) if (> x 2))
(car exp)	(* x x)
(car (cddr exp))	x
(car (cddr (cddr exp)))	'(1 2 3 4)
(car (cddr (cddr (cddr exp))))	(> x 2)

```
(lambda (x) (* x x))
```

```
(map (lambda (x) (* x x))
     (filter (lambda (x) (> x 2)) '(1 2 3 4)))
```

List Comprehensions in Scheme

```
exp                     ((* x x) for x in '(1 2 3 4) if (> x 2))
(car exp)                                              (* x x)
(car (cddr exp))                                             x
(car (cddr (cddr exp)))                              '(1 2 3 4)
(car (cddr (cddr (cddr exp))))                         (> x 2)
```

```
(list 'lambda (list 'x) '(* x x))
                                    (lambda (x) (* x x))
```

```
(map (lambda (x) (* x x))
     (filter (lambda (x) (> x 2)) '(1 2 3 4)))
```

List Comprehensions in Scheme

```
exp                          ((* x x) for x in '(1 2 3 4) if (> x 2))
(car exp)                                                     (* x x)
(car (cddr exp))                                                    x
(car (cddr (cddr exp)))                                   '(1 2 3 4)
(car (cddr (cddr (cddr exp))))                                (> x 2)
```

```
(list 'lambda (list 'x) '(* x x))
                                    (lambda (x) (* x x))


                                    (lambda (x) (> x 2))


   (map (lambda (x) (* x x))
        (filter (lambda (x) (> x 2)) '(1 2 3 4)))
```

List Comprehensions in Scheme

```
exp                 ((* x x) for x in '(1 2 3 4) if (> x 2))
(car exp)                                            (* x x)
(car (cddr exp))                                           x
(car (cddr (cddr exp)))                           '(1 2 3 4)
(car (cddr (cddr (cddr exp))))                       (> x 2)
```

```
(list 'lambda (list 'x) '(* x x))
                                    (lambda (x) (* x x))
(list 'lambda (list 'x) '(> x 2))
                                    (lambda (x) (> x 2))
```

```
   (map (lambda (x) (* x x))
        (filter (lambda (x) (> x 2)) '(1 2 3 4)))
```

List Comprehensions in Scheme

```
exp                        ((* x x) for x in '(1 2 3 4) if (> x 2))
(car exp)                                                   (* x x)
(car (cddr exp))                                                  x
(car (cddr (cddr exp)))                                   '(1 2 3 4)
(car (cddr (cddr (cddr exp))))                              (> x 2)
```

```
(list 'lambda (list 'x) '(* x x))
                                    (lambda (x) (* x x))
(list 'lambda (list 'x) '(> x 2))
                                    (lambda (x) (> x 2))
```

```
   (map (lambda (x) (* x x))
        (filter (lambda (x) (> x 2)) '(1 2 3 4)))
```

List Comprehensions in Scheme (demo)

```
exp                      ((* x x) for x in '(1 2 3 4) if (> x 2))
(car exp)                                                 (* x x)
(car (cddr exp))                                                x
(car (cddr (cddr exp)))                                '(1 2 3 4)
(car (cddr (cddr (cddr exp))))                            (> x 2)
```

```
(list 'lambda (list 'x) '(* x x))
                                  (lambda (x) (* x x))
(list 'lambda (list 'x) '(> x 2))
                                  (lambda (x) (> x 2))
```

```
   (map (lambda (x) (* x x))
        (filter (lambda (x) (> x 2)) '(1 2 3 4)))
```

More Symbolic Programming

Rational numbers!

Summary

Summary

- Tail call optimization allows some recursive procedures to take up a constant amount of space – just like iterative functions in Python!

Summary

- Tail call optimization allows some recursive procedures to take up a constant amount of space – just like iterative functions in Python!
- Streams can be used to define implicit sequences

Summary

- Tail call optimization allows some recursive procedures to take up a constant amount of space — just like iterative functions in Python!
- Streams can be used to define implicit sequences
- We can manipulate Scheme programs (as lists) to create new Scheme programs

Summary

- Tail call optimization allows some recursive procedures to take up a constant amount of space – just like iterative functions in Python!
- Streams can be used to define implicit sequences
- We can manipulate Scheme programs (as lists) to create new Scheme programs
    - This is one huge language feature that has contributed to Lisp's staying power over the years

Summary

- Tail call optimization allows some recursive procedures to take up a constant amount of space — just like iterative functions in Python!
- Streams can be used to define implicit sequences
- We can manipulate Scheme programs (as lists) to create new Scheme programs
  - This is one huge language feature that has contributed to Lisp's staying power over the years
  - Look up "macros" to learn more!