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)
This week (Interpretation), the goals are:
This week (Interpretation), the goals are:

- To learn a new language, Scheme, in two days!
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  

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

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

```scheme
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)

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

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

(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)
    ...
  )

(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) 
      (helper (- n 1) (* prod n)))))

scm> (fact 10)

scm> (fact 1000)
(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)))))

scm> (fact 10)
scm> (fact 1000)
Factorial (Again)

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

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

scm> (fact 10)
scm> (fact 1000)
(define (fact n)
  (if (= n 0)
      1
      (* n (fact (- n 1)))))

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

(helper n 1))

scm> (fact 10)
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
      (helper (- n 1) (* n prod))))
  (helper n 1))

scm> (fact 10)
scm> (fact 1000)
Tail Recursion
Tail Recursion

The Revised\textsuperscript{7} Report on the Algorithmic Language Scheme:
Tail Recursion

The Revised 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\textsuperscript{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."

\begin{verbatim}
(define (fact n)
    (define (helper n prod)
        (if (= n 0) prod (helper (- n 1) (* n prod)))
    (helper n 1))
\end{verbatim}
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
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))
```
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`

```scheme
(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

\[
\text{(define} \quad (\text{length} \ s) \\
\begin{align*}
&= (\text{if} \ (\text{null?} \ s) \ 0 \\
&\quad (+ 1 (\text{length} \ (\text{cdr} \ s))))
\end{align*}
\]
Example: Length

\[
\text{(define (length s)}
\]
\[
\text{(if (null? s) 0}
\]
\[
\text{(+ 1 (length (cdr s))));)}
\]

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

\[
\text{(define (length s)} \\
\text{  (if (null? s) 0} \\
\text{    (+ 1 (length (cdr s)))))}
\]

- 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

\[
\text{(define (length s)}\nonumber \\
\quad \text{(if (null? s) 0)}\nonumber \\
\quad \quad \quad \quad \text{(+ 1 (length (cdr s)))))}\nonumber \\
\]

- 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.

\[
\text{(define (length-tail s)}\nonumber \\
\quad \text{(if (null? s) 0)}\nonumber \\
\quad \quad \quad \quad \text{(+ 1 (length (cdr s)))))}\nonumber \\
\]

Not a tail context
Example: Length

\[(\text{define} \ (\text{length} \ s))\]
\[(\text{if} \ (\text{null?} \ s) 0\]
\[(+\ 1\ (\text{length} \ (\text{cdr} \ s))))]\]

• 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:

\[(\text{define} \ (\text{length}-\text{tail} \ s))\]
\[(\text{define} \ (\text{length}-\text{iter} \ s \ n)]
Example: Length

\[
\text{(define (length s)} \\
\quad \text{(if (null? s) 0)} \\
\quad \quad \text{(+ 1 (length (cdr s))))})
\]

- 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.

\[
\text{(define (length-tail s)} \\
\quad \text{(define (length-iter s n)} \\
\quad \quad \text{(if (null? s) n}))}
\]
Example: Length

\[
\begin{align*}
(\text{define} & \ (\text{length} \ s) \\
(\text{if} & \ (\text{null?} \ s) \ 0 \\
& \ (+ \ 1 \ (\text{length} \ (\text{cdr} \ s)))))
\end{align*}
\]

- 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.

\[
\begin{align*}
(\text{define} & \ (\text{length-tail} \ s) \\
(\text{define} & \ (\text{length-iter} \ s \ n) \\
(\text{if} & \ (\text{null?} \ s) \ n \\
& \ (\text{length-iter} \ (\text{cdr} \ s) \ (+ \ 1 \ n)))))
\end{align*}
\]
Example: Length

\[
\text{(define (length } s) \\text{\texttt{(define (length } s) \texttt{(if (null? } s \texttt{0) (define (length-cdr s))))})}
\]

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.

\[
\text{(define (length-tail } s) \\text{(define (length-iter } s n) (define (length-iter } s n) \text{(define (length-iter } s n) \texttt{(if (null? } s n (length-iter (cdr s) (+ 1 n))))})
\]

(length-iter s 0))
Example: Length

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

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

Not a tail context
Example: Length

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

• 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

\[
\text{(define (length s)}
\begin{align*}
& (\text{if (null? s) 0} \\
& \quad (\text{+ 1 (length (cdr s)))))
\end{align*}
\)
\]

- 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

\[
\text{(define (length-tail s)}
\begin{align*}
& \text{(define (length-iter s n)}
\begin{align*}
& (\text{if (null? s) n} \\
& \quad ((\text{length-iter (cdr s) (+ 1 n))))))
\end{align*}
\)
\]

\[
\text{(length-iter s 0)}
\]
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

- 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

• 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

\[
\begin{align*}
\text{(car (cons 1 2))} & \rightarrow 1 & \text{(car (cons-stream 1 2))} & \rightarrow 1 \\
\text{(cdr (cons 1 2))} & \rightarrow 2 \\
\text{(cons 1 (cons 2 nil))} & 
\end{align*}
\]
Streams

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

\[
\begin{align*}
(car \ (cons \ 1 \ 2)) & \rightarrow 1 \\
(cdr \ (cons \ 1 \ 2)) & \rightarrow 2 \\
(cons \ 1 \ (cons \ 2 \ nil)) &
\end{align*}
\]
Streams

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

\[
\begin{align*}
\text{(car (cons 1 2))} & \rightarrow 1 \\
\text{(cdr (cons 1 2))} & \rightarrow 2 \\
\text{(cons 1 (cons 2 nil))} & \\
\end{align*}
\]

\[
\begin{align*}
\text{(car (cons-stream 1 2))} & \rightarrow 1 \\
\text{(cdr-stream (cons-stream 1 2))} & \rightarrow 2 \\
\text{(cons-stream 1 (cons-stream 2 nil))} & \\
\end{align*}
\]
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

```scheme
(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

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

\[(\text{cons-stream} \ 1 \ (/ \ 1 \ 0)) \rightarrow (1 . \#[\text{promise (not forced)}])\]

\[(\text{car} \ (\text{cons-stream} \ 1 \ (/ \ 1 \ 0))) \rightarrow 1\]

\[(\text{cdr-stream} \ (\text{cons-stream} \ 1 \ (/ \ 1 \ 0))) \rightarrow \text{ERROR}\]
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

\[
\text{(define (int-stream start))} \\
\text{(cons-stream}
\]
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
    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

- 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

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

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

1 1 1 1 1 1 1 ...
Recursively Defined Streams

(define ones (cons-stream 1 ones))

1 1 1 1 1 1 1 ...
Recursively Defined Streams

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

\[ \text{(define (add-streams s1 s2)} \]

\[ 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad ... \]
Recursively Defined Streams

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

(define (add-streams s1 s2)
  (cons-stream 1 (add-streams (cdr s1) (cdr s2))))
```

```
1 1 1 1 1 1 1 ...
```
Recursively Defined Streams

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

\[
\text{(define (add-streams s1 s2)}
\text{  (cons-stream \( + (\text{car s1)} (\text{car s2)}) \))}
\]
Recursively Defined Streams

\[
\begin{align*}
\text{(define ones (cons-stream 1 ones))} \\
\text{(define (add-streams s1 s2)} & \\
& \text{(cons-stream} \quad \text{1 1 1 1 1 1 1 ...} \\
& \text{(+ (car s1) (car s2))} \\
& \text{(add-streams}}
\end{align*}
\]
Recursively Defined Streams

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

\[
\text{(define (add-streams s1 s2)} \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad ... \\
\text{ (cons-stream)} \\
\text{ (+ (car s1) (car s2))} \\
\text{ (add-streams)} \\
\text{ (cdr-stream s1)}
\]
Recursively Defined Streams

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

\[
\text{define (add-streams s1 s2)}
\]

\[
\begin{align*}
\text{(cons-stream} & + (\text{car s1}) (\text{car s2})) \\
\text{(add-streams} & \text{(cdr-stream s1)}} \\
& \text{(cdr-stream s2))}
\end{align*}
\]
Recursively Defined Streams

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

\[ (\text{define} \ \text{(add-streams s1 s2)} \]
\[ \quad (\text{cons-stream} \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ \ldots) \]
\[ \quad (+ \ (\text{car s1}) \ (\text{car s2})) \]
\[ \quad \text{(add-streams} \]
\[ \quad \quad (\text{cdr-stream s1}) \]
\[ \quad \quad (\text{cdr-stream s2}))) \]

\[ (\text{define ints} \]
Recursively Defined Streams

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

\[
\text{ints} = (\text{cons-stream} \ 1 \ \text{ints})
\]

\[
\text{add-streams} \ s1 \ s2 = (\text{cons-stream} \ (+ \ (\text{car} \ s1) \ (\text{car} \ s2)) \ (\text{add-streams} \ (\text{cdr-stream} \ s1) \ (\text{cdr-stream} \ s2)))
\]
Recursively Defined Streams

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

\[(\text{define} \ (\text{add-streams} \ s1 \ s2) \rightarrow \ (\text{cons-stream} \ (+ \ (\text{car} \ s1) \ (\text{car} \ s2)) \ (\text{add-streams} \ (\text{cdr-stream} \ s1) \ (\text{cdr-stream} \ s2))))\]

\[(\text{define} \ \text{ints} \rightarrow \ (\text{cons-stream} \ 1 \ (\text{add-streams} \ \text{ones} \ \text{ints})))\]
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

(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

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

\[(\text{define} \ \text{ints} \ (\text{cons-stream} \ 1 \ \text{ints} \ (\text{add-streams} \ \textones \ \text{ints})))\]

\[(\text{define} \ \text{add-streams} \ s1 \ s2) \rightarrow (\text{cons-stream} \ (+ \ (\text{car} \ s1) \ (\text{car} \ s2)) \ (\text{add-streams} \ (\text{cdr-stream} \ s1) \ (\text{cdr-stream} \ s2))))\]
Recursively Defined Streams

\[
\begin{align*}
&\text{(define ones (cons-stream } 1 \text{ ones))} \\
&\text{(define (add-streams s1 s2)} \\
&\quad \quad \text{(cons-stream} \\
&\quad \quad \quad (+ \text{(car s1) (car s2)}) \\
&\quad \quad \quad \text{(add-streams} \\
&\quad \quad \quad \quad \text{(cdr-stream s1)} \\
&\quad \quad \quad \quad \quad \text{(cdr-stream s2)}))))) \\
&\text{(define ints } \\
&\quad \text{(cons-stream } 1 \\
&\quad \quad \quad \text{(add-streams ones ints))})
\end{align*}
\]
Recursively Defined Streams  

\[
\begin{align*}
\text{(define ones (cons-stream 1 ones))} \\
\text{(define (add-streams s1 s2)} \\
\quad \text{(cons-stream} \\
\quad \quad \text{(+ (car s1) (car s2))} \\
\quad \text{(add-streams} \\
\quad \quad \text{(cdr-stream s1)} \\
\quad \quad \quad \text{(cdr-stream s2))})) \\
\text{(define ints} \\
\quad \text{(cons-stream 1} \\
\quad \quad \text{(add-streams ones ints)))}
\end{align*}
\]
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

(define (square x) (* x x))
Symbolic Programming

\[
\text{procedure}
\]

\[
\text{(define } \text{ (square x) \ ((\ast \ x \ x))})
\]

\[
\text{'}\text{(define } \text{ (square x) \ ((\ast \ x \ x))})
\]
Symbolic Programming

(procedure list)

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

'(define (square x) (* x x))
Symbolic Programming

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

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

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

- Lists can be manipulated with `car` and `cdr`
- Lists can be created and combined with `cons`, `list`, and `append`
Symbolic Programming

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

\[(\ast \ x \ x) \text{ for } x \text{ in } '(1 2 3 4) \text{ if } (> x \ 2)\]
List Comprehensions in Scheme

\[
((\ast \enspace x \enspace x) \enspace \text{for} \enspace x \ \text{in} \ (1 \ 2 \ 3 \ 4) \ \text{if} \ (x > 2))
\]

(map (lambda (x) (\ast \enspace x \enspace x))
     (filter (lambda (x) (x > 2)) '1 2 3 4))
List Comprehensions in Scheme

$$\left(\left(\ast \, x \, x\right) \text{ for } x \text{ in } \left(1 \ 2 \ 3 \ 4\right) \text{ if } (\gt x \ 2)\right)$$

(map (lambda (x) (\ast \ x \ x)) (filter (lambda (x) (\gt x \ 2)) \'(1 2 3 4)))
List Comprehensions in Scheme

\[
(\text{map } \text{lambda} \ x \ (\text{* } x \ x) \ \text{for } x \ \text{in } '(1 \ 2 \ 3 \ 4) \ \text{if } (> x \ 2))
\]
List Comprehensions in Scheme

\[
((\star x x) \text{ for } x \text{ in } '(1 2 3 4) \text{ if } (> x 2))
\]

(map (lambda (x) (\star x x))
(filter (lambda (x) (> x 2)) '(1 2 3 4)))
List Comprehensions in Scheme

\[ ((\ast \ x \ x) \ for \ x \ in \ ' (1 \ 2 \ 3 \ 4) \ if \ (> \ x \ 2)) \]

\( (\text{map} \ (\lambda x \ (\ast x x)) \ (\text{filter} \ (\lambda x \ (> x 2)) \ ' (1 \ 2 \ 3 \ 4))) \)
List Comprehensions in Scheme

\[
\text{exp} \quad \left( (\ast \ x \ x) \ \text{for} \ x \ \text{in} \ '1\ 2\ 3\ 4' \ \text{if} \ (> x 2) \right)
\]

\[
\begin{align*}
(\text{map} & \ (\text{lambda} (x) (\ast x x)) \\
(\text{filter} & \ (\text{lambda} (x) (> x 2)) '1\ 2\ 3\ 4))
\end{align*}
\]
List Comprehensions in Scheme

```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))
(car exp)
```

```
(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)

(map (lambda (x) (* x x))
(filter (lambda (x) (> x 2)) (1 2 3 4)))
```
List Comprehensions in Scheme

\[
\text{exp} \quad \left(\left(\ast \ x \ x\right) \text{ for } x \text{ in } (1 \ 2 \ 3 \ 4) \text{ if } (\gt x \ 2)\right)
\]

(car exp)

(car (cddr exp))

(map (lambda (x) (\ast \ x \ x)) (filter (lambda (x) (\gt x \ 2)) (list 1 2 3 4)))
List Comprehensions in Scheme

\[
\text{exp} \quad ((* \ x \ x) \ \text{for} \ x \ \text{in} \ '1 \ 2 \ 3 \ 4 \ \text{if} \ (> \ x \ 2))
\]

(car exp)

(car (cddr exp))
List Comprehensions in Scheme

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

(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

\[
\text{exp} \quad \left( (\ast \ x \ x) \ \text{for} \ x \ \text{in} \ (1 \ 2 \ 3 \ 4) \ \text{if} \ (> \ x \ 2) \right)
\]

\[
(\text{car exp}) \quad (\ast \ x \ x)
\]

\[
(\text{car (cddr exp)}) \quad x
\]

\[
(\text{car (cddr (cddr exp)))} \quad (1 \ 2 \ 3 \ 4)
\]

\[
(\text{car (cddr (cddr (cddr exp))))} \quad (> \ x \ 2)
\]

\[
(\text{map (lambda (x) (\ast \ x \ x))})
\]

\[
(\text{filter (lambda (x) (> x 2))} \quad (1 \ 2 \ 3 \ 4))
\]
List Comprehensions in Scheme

\[
\text{exp} = \left( (\times x x) \text{ for } x \text{ in } '(1 2 3 4) \text{ if } (> x 2) \right)
\]

\[
(\text{car exp}) = (\times x x)
\]

\[
(\text{car (cddr exp)}) = x
\]

\[
(\text{car (cddr (cddr exp))}) = '(1 2 3 4)
\]

\[
(\text{car (cddr (cddr (cddr exp))}) = (> x 2)
\]

\[
(\text{map (lambda (x) (\times x x))})
\]

\[
(\text{filter (lambda (x) (> x 2))} ' (1 2 3 4))
\]
List Comprehensions in Scheme

\[
\text{exp} = ((\ast x x) \text{ for } x \text{ in '}(1\ 2\ 3\ 4) \text{ if } (x > 2))
\]

\[
\text{(car exp)}
\]

\[
\text{(car (cddr exp))}
\]

\[
\text{(car (cddr (cddr exp)))}
\]

\[
\text{(car (cddr (cddr (cddr exp)))))}
\]

\[
\text{(lambda (x) (\ast x x))}
\]

\[
\text{(map (lambda (x) (\ast x x))}
\]

\[
\text{(filter (lambda (x) (x > 2)) '}(1\ 2\ 3\ 4))}
\]
List Comprehensions in Scheme

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

(list 'lambda (list '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)
(car (cddr exp))
(car (cddr (cddr exp)))
(car (cddr (cddr (cddr exp)))))

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

(map (lambda (x) (* x x))
(filter (lambda (x) (> x 2)) '(1 2 3 4)))
```
List Comprehensions in Scheme

\[ \text{exp} \quad \text{((} \ast \ x \ x \text{)} \text{ for } x \text{ in } '1 \ 2 \ 3 \ 4 \text{ if } (> x \ 2))} \]

\[ (\text{car exp}) \quad (\ast \ x \ x) \]

\[ (\text{car (cddr exp)}) \quad x \]

\[ (\text{car (cddr (cddr exp))}) \quad '1 \ 2 \ 3 \ 4 \]

\[ (\text{car (cddr (cddr (cddr exp))))} \quad (> x \ 2) \]

\[ (\text{list 'lambda (list 'x) '((} \ast \ x \ x\text{)})} \quad (\text{lambda (x) (} \ast \ x \ x\text{)}) \]

\[ (\text{list 'lambda (list 'x) '(} (> \ x \ 2)\text{)} \quad (\text{lambda (x) (} (> \ x \ 2)\text{)}) \]

\[ (\text{map (} \text{lambda (x) (} \ast \ x \ x\text{)}) \quad (\text{filter (} \text{lambda (x) (} (> \ x \ 2)\text{) '}(1 \ 2 \ 3 \ 4)\text{))}) \]
List Comprehensions in Scheme

\[
\begin{align*}
\text{exp} & \quad \left(\left(* \ x \ x\right) \text{ for } x \text{ in } \left(1 \ 2 \ 3 \ 4\right) \text{ if } (\ > \ x \ 2)\right) \\
(\text{car exp}) & \quad \left(* \ x \ x\right) \\
(\text{car (cddr exp)}) & \quad x \\
(\text{car (cddr (cddr exp))}) & \quad \left(1 \ 2 \ 3 \ 4\right) \\
(\text{car (cddr (cddr (cddr exp))})) & \quad (\ > \ x \ 2) \\
\end{align*}
\]

\[
\begin{align*}
(\text{list 'lambda (list 'x) '(* x x)}) & \quad (\text{lambda } \ x \ (\ * \ x \ x)) \\
(\text{list 'lambda (list 'x) ' (> x 2)}) & \quad (\text{lambda } \ x \ (\ > \ x \ 2)) \\
\end{align*}
\]

\[
\begin{align*}
(\text{map (lambda (x) (* x x))}) \\
(\text{filter (lambda (x) (> x 2)) ' (1 2 3 4))}
\end{align*}
\]
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))
(lambdax(* x x))
(list 'lambda (list 'x) ' (> x 2))
(lambdax(> 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!