Announcements

• Homework 7 due Wednesday 4/8 @ 11:59pm
Announcements

• Homework 7 due Wednesday 4/8 @ 11:59pm
  • Homework party Tuesday 4/7 5pm–6:30pm in 2050 VLSB
Announcements

• Homework 7 due Wednesday 4/8 @ 11:59pm
  ▪ Homework party Tuesday 4/7 5pm–6:30pm in 2050 VLSB
• Quiz 2 due Thursday 4/9 @ 11:59pm
Announcements

• Homework 7 due Wednesday 4/8 @ 11:59pm
  ▪ Homework party Tuesday 4/7 5pm–6:30pm in 2050 VLSB
• Quiz 2 due Thursday 4/9 @ 11:59pm
• Homework 8 due Wednesday 4/15 @ 11:59pm
Announcements

• Homework 7 due Wednesday 4/8 @ 11:59pm
  ▪ Homework party Tuesday 4/7 5pm–6:30pm in 2050 VLSB
• Quiz 2 due Thursday 4/9 @ 11:59pm
• Homework 8 due Wednesday 4/15 @ 11:59pm
• Project 1, 2, & 3 composition revisions due Monday 4/13 @ 11:59pm
Announcements

• Homework 7 due Wednesday 4/8 @ 11:59pm
  • Homework party Tuesday 4/7 5pm–6:30pm in 2050 VLSB
• Quiz 2 due Thursday 4/9 @ 11:59pm
• Homework 8 due Wednesday 4/15 @ 11:59pm
• Project 1, 2, & 3 composition revisions due Monday 4/13 @ 11:59pm
• Project 4 due Thursday 4/23 @ 11:59pm (Big!)
Scheme Recursive Art Contest: Start Early!
Scheme Recursive Art Contest: Start Early!

Fall 2012 Featherweight Winner
176 Scheme Tokens
Scheme Recursive Art Contest: Start Early!

Fall 2012 Featherweight Winner
176 Scheme Tokens

Fall 2013 Heavyweight Winner
1857 Scheme Tokens
Scheme Recursive Art Contest: Start Early!

Fall 2012 Featherweight Winner
176 Scheme Tokens

Fall 2013 Heavyweight Winner
1857 Scheme Tokens

Extra lecture on this image:
Thursday 4/9 5pm in 2050 VLSB
Dynamic Scope
Dynamic Scope
Dynamic Scope

The way in which names are looked up in Scheme and Python is called lexical scope (or static scope) [You can see what names are in scope by inspecting the definition]
Dynamic Scope

The way in which names are looked up in Scheme and Python is called lexical scope (or static scope) [You can see what names are in scope by inspecting the definition]

**Lexical scope:** The parent of a frame is the environment in which a procedure was defined
Dynamic Scope

The way in which names are looked up in Scheme and Python is called lexical scope (or static scope) [You can see what names are in scope by inspecting the definition]

**Lexical scope:** The parent of a frame is the environment in which a procedure was **defined**

**Dynamic scope:** The parent of a frame is the environment in which a procedure was **called**
Dynamic Scope

The way in which names are looked up in Scheme and Python is called lexical scope (or static scope) [You can see what names are in scope by inspecting the definition]

Lexical scope: The parent of a frame is the environment in which a procedure was defined

Dynamic scope: The parent of a frame is the environment in which a procedure was called

(define f (lambda (x) (+ x y)))
Dynamic Scope

The way in which names are looked up in Scheme and Python is called lexical scope (or static scope) [You can see what names are in scope by inspecting the definition]

**Lexical scope:** The parent of a frame is the environment in which a procedure was *defined*

**Dynamic scope:** The parent of a frame is the environment in which a procedure was *called*

```
(define f (lambda (x) (+ x y)))
(define g (lambda (x y) (f (+ x x))))
```
Dynamic Scope

The way in which names are looked up in Scheme and Python is called lexical scope (or static scope) [You can see what names are in scope by inspecting the definition]

**Lexical scope**: The parent of a frame is the environment in which a procedure was *defined*

**Dynamic scope**: The parent of a frame is the environment in which a procedure was *called*

```
(define f (lambda (x) (+ x y)))
(define g (lambda (x y) (f (+ x x))))
(g 3 7)
```
Dynamic Scope

The way in which names are looked up in Scheme and Python is called lexical scope (or static scope) [You can see what names are in scope by inspecting the definition]

Lexical scope: The parent of a frame is the environment in which a procedure was defined

Dynamic scope: The parent of a frame is the environment in which a procedure was called

(define f (lambda (x) (+ x y)))
(define g (lambda (x y) (f (+ x x))))
(g 3 7)

Lexical scope: The parent for f's frame is the global frame
**Dynamic Scope**

The way in which names are looked up in Scheme and Python is called **lexical scope** (or static scope) [You can see what names are in scope by inspecting the definition]

**Lexical scope:** The parent of a frame is the environment in which a procedure was defined

**Dynamic scope:** The parent of a frame is the environment in which a procedure was called

```
(define f (lambda (x) (+ x y)))
(define g (lambda (x y) (f (+ x x))))
(g 3 7)
```

**Lexical scope:** The parent for f's frame is the global frame
Dynamic Scope

The way in which names are looked up in Scheme and Python is called lexical scope (or static scope) [You can see what names are in scope by inspecting the definition]

Lexical scope: The parent of a frame is the environment in which a procedure was defined
Dynamic scope: The parent of a frame is the environment in which a procedure was called

```
(define f (lambda (x) (+ x y)))
(define g (lambda (x y) (f (+ x x))))
(g 3 7)
```

Lexical scope: The parent for f's frame is the global frame
Dynamic Scope

The way in which names are looked up in Scheme and Python is called lexical scope (or static scope) [You can see what names are in scope by inspecting the definition]

Lexical scope: The parent of a frame is the environment in which a procedure was defined

Dynamic scope: The parent of a frame is the environment in which a procedure was called

(define f (lambda (x) (+ x y)))
(define g (lambda (x y) (f (+ x x))))
(g 3 7)

Lexical scope: The parent for f's frame is the global frame

<table>
<thead>
<tr>
<th>Global frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
</tr>
<tr>
<td>g</td>
</tr>
<tr>
<td>((x) ...)</td>
</tr>
<tr>
<td>((x\ y) ...)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>f1: g [parent=global]</th>
</tr>
</thead>
<tbody>
<tr>
<td>x 3</td>
</tr>
<tr>
<td>y 7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>f2: f [parent=global]</th>
</tr>
</thead>
<tbody>
<tr>
<td>x 6</td>
</tr>
</tbody>
</table>
Dynamic Scope

The way in which names are looked up in Scheme and Python is called lexical scope (or static scope) [You can see what names are in scope by inspecting the definition]

**Lexical scope**: The parent of a frame is the environment in which a procedure was defined

**Dynamic scope**: The parent of a frame is the environment in which a procedure was called

```
(define f (lambda (x) (+ x y)))
(define g (lambda (x y) (f (+ x x))))
(g 3 7)
```

**Lexical scope**: The parent for f's frame is the global frame

*Error: unknown identifier: y*
Dynamic Scope

The way in which names are looked up in Scheme and Python is called lexical scope (or static scope) [You can see what names are in scope by inspecting the definition]

**Lexical scope:** The parent of a frame is the environment in which a procedure was defined

**Dynamic scope:** The parent of a frame is the environment in which a procedure was called

```
(define f (lambda (x) (+ x y)))
(define g (lambda (x y) (f (+ x x))))
(g 3 7)
```

**Lexical scope:** The parent for f's frame is the global frame

```
Error: unknown identifier: y
```

**Dynamic scope:** The parent for f's frame is g's frame

![Diagram of frame relationships]
Dynamic Scope

The way in which names are looked up in Scheme and Python is called lexical scope (or static scope) [You can see what names are in scope by inspecting the definition]

Lexical scope: The parent of a frame is the environment in which a procedure was defined

Dynamic scope: The parent of a frame is the environment in which a procedure was called

(define f (lambda (x) (+ x y)))
(define g (lambda (x y) (f (+ x x))))
(g 3 7)

Lexical scope: The parent for f's frame is the global frame

Error: unknown identifier: y

Dynamic scope: The parent for f's frame is g's frame
Dynamic Scope

The way in which names are looked up in Scheme and Python is called lexical scope (or static scope) [You can see what names are in scope by inspecting the definition]

Lexical scope: The parent of a frame is the environment in which a procedure was defined

Dynamic scope: The parent of a frame is the environment in which a procedure was called

(define f (lambda (x) (+ x y)))
(define g (lambda (x y) (f (+ x x))))
(g 3 7)

Lexical scope: The parent for f's frame is the global frame

Dynamic scope: The parent for f's frame is g's frame

Error: unknown identifier: y

Special form to create dynamically scoped procedures (mu special form only exists in Project 4 Scheme)
Dynamic Scope

The way in which names are looked up in Scheme and Python is called lexical scope (or static scope) [You can see what names are in scope by inspecting the definition]

Lexical scope: The parent of a frame is the environment in which a procedure was defined

Dynamic scope: The parent of a frame is the environment in which a procedure was called

(define f (lambda (x) (+ x y)))
(define g (lambda (x y) (f (+ x x))))
(g 3 7)

Lexical scope: The parent for f's frame is the global frame

Error: unknown identifier: y

Dynamic scope: The parent for f's frame is g's frame
**Dynamic Scope**

The way in which names are looked up in Scheme and Python is called lexical scope (or static scope) [You can see what names are in scope by inspecting the definition]

**Lexical scope**: The parent of a frame is the environment in which a procedure was defined

**Dynamic scope**: The parent of a frame is the environment in which a procedure was called

```
(define f (lambda (x) (+ x y)))
(define g (lambda (x y) (f (+ x x))))
(g 3 7)
```

**Lexical scope**: The parent for f's frame is the global frame

**Dynamic scope**: The parent for f's frame is g's frame

13
Tail Recursion
Functional Programming
Functional Programming

All functions are pure functions.
Functional Programming

All functions are pure functions.

No re-assignment and no mutable data types.
Functional Programming

All functions are pure functions.

No re-assignment and no mutable data types.

Name-value bindings are permanent.
**Functional Programming**

All functions are pure functions.

No re-assignment and no mutable data types.

Name-value bindings are permanent.

Advantages of functional programming:
Functional Programming

All functions are pure functions.

No re-assignment and no mutable data types.

Name-value bindings are permanent.

Advantages of functional programming:

• The value of an expression is independent of the order in which sub-expressions are evaluated.
Functional Programming

All functions are pure functions.

No re-assignment and no mutable data types.

Name-value bindings are permanent.

Advantages of functional programming:

• The value of an expression is independent of the order in which sub-expressions are evaluated.

• Sub-expressions can safely be evaluated in parallel or on demand (lazily).
Functional Programming

All functions are pure functions.

No re-assignment and no mutable data types.

Name-value bindings are permanent.

Advantages of functional programming:

- The value of an expression is independent of the order in which sub-expressions are evaluated.
- Sub-expressions can safely be evaluated in parallel or on demand (lazily).
- **Referential transparency**: The value of an expression does not change when we substitute one of its subexpression with the value of that subexpression.
Functional Programming

All functions are pure functions.

No re-assignment and no mutable data types.

Name-value bindings are permanent.

Advantages of functional programming:

- The value of an expression is independent of the order in which sub-expressions are evaluated.
- Sub-expressions can safely be evaluated in parallel or on demand (lazily).
- **Referential transparency**: The value of an expression does not change when we substitute one of its subexpression with the value of that subexpression.

But... no `for/while` statements! Can we make basic iteration efficient? Yes!
Recursion and Iteration in Python

In Python, recursive calls always create new active frames

```
factorial(n, k) computes: n! * k
```
Recursion and Iteration in Python

In Python, recursive calls always create new active frames

$$\text{factorial}(n, k) \text{ computes: } n! \times k$$

```
def factorial(n, k):
    if n == 0:
        return k
    else:
        return factorial(n-1, k*n)
```
Recursion and Iteration in Python

In Python, recursive calls always create new active frames

\[ \text{factorial}(n, k) \text{ computes: } n! \times k \]

```python
def factorial(n, k):
    if n == 0:
        return k
    else:
        return factorial(n-1, k*n)
```

```python
def factorial(n, k):
    while n > 0:
        n, k = n-1, k*n
    return k
```
Recursion and Iteration in Python

In Python, recursive calls always create new active frames

\[ \text{factorial}(n, k) \text{ computes: } n! \ast k \]

```
def factorial(n, k):
    if n == 0:
        return k
    else:
        return factorial(n-1, k*n)
```

```
def factorial(n, k):
    while n > 0:
        n, k = n-1, k*n
    return k
```
Recursion and Iteration in Python

In Python, recursive calls always create new active frames

\[ \text{factorial}(n, k) \text{ computes: } n! \times k \]

<table>
<thead>
<tr>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Θ(n)</td>
<td></td>
</tr>
</tbody>
</table>

```python
def factorial(n, k):
    if n == 0:
        return k
    else:
        return factorial(n-1, k*n)
```

```python
def factorial(n, k):
    while n > 0:
        n, k = n-1, k*n
    return k
```
Recursion and Iteration in Python

In Python, recursive calls always create new active frames.

\[ \text{factorial}(n, k) \text{ computes: } n! \times k \]

<table>
<thead>
<tr>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\Theta(n))</td>
<td>(\Theta(n))</td>
</tr>
</tbody>
</table>

```python
def factorial(n, k):
    if n == 0:
        return k
    else:
        return factorial(n-1, k*n)
```

```python
def factorial(n, k):
    while n > 0:
        n, k = n-1, k*n
    return k
```
Recursion and Iteration in Python

In Python, recursive calls always create new active frames

\[ \text{factorial}(n, k) \text{ computes: } n! \times k \]

<table>
<thead>
<tr>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\Theta(n))</td>
<td>(\Theta(n))</td>
</tr>
</tbody>
</table>

```python
def factorial(n, k):
    if n == 0:
        return k
    else:
        return factorial(n-1, k*n)

def factorial(n, k):
    while n > 0:
        n, k = n-1, k*n
    return k
```
Recursion and Iteration in Python

In Python, recursive calls always create new active frames

`factorial(n, k)` computes: \( n! \times k \)

```python
def factorial(n, k):
    if n == 0:
        return k
    else:
        return factorial(n-1, k*n)
```

<table>
<thead>
<tr>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Theta(n) )</td>
<td>( \Theta(n) )</td>
</tr>
</tbody>
</table>

```python
def factorial(n, k):
    while n > 0:
        n, k = n-1, k*n
    return k
```

<table>
<thead>
<tr>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Theta(n) )</td>
<td>( \Theta(1) )</td>
</tr>
</tbody>
</table>
Recursion and Iteration in Python

In Python, recursive calls always create new active frames

```
def factorial(n, k):
    if n == 0:
        return k
    else:
        return factorial(n-1, k*n)
```

```
def factorial(n, k):
    while n > 0:
        n, k = n-1, k*n
    return k
```

<table>
<thead>
<tr>
<th></th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Factorial</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
</tbody>
</table>

factorial(n, k) computes: n! * k
Tail Recursion

From the Revised\textsuperscript{7} Report on the Algorithmic Language Scheme:

```
def factorial(n, k):
    while n > 0:
        n, k = n−1, k∗n
    return k
```

<table>
<thead>
<tr>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Theta(n)$</td>
<td>$\Theta(1)$</td>
</tr>
</tbody>
</table>
Tail Recursion

From 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."

```
def factorial(n, k):
    while n > 0:
        n, k = n-1, k*n
    return k
```

<table>
<thead>
<tr>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Theta(n)$</td>
<td>$\Theta(1)$</td>
</tr>
</tbody>
</table>
Tail Recursion

From 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."

\[
\begin{align*}
(\text{define } (\text{factorial} \ n \ k) & \\
& \text{if } (\text{zero?} \ n) \ k \\
& \quad (\text{factorial} \ (-n \ 1) \ (* \ k \ n)))
\end{align*}
\]

<table>
<thead>
<tr>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Theta(n)$</td>
<td>$\Theta(1)$</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{def factorial}(n, k): & \\
\quad \text{while } n > 0: & \\
\quad \quad n, k = n-1, k*n & \\
\quad \text{return } k
\end{align*}
\]
Tail Recursion

From 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 (factorial n k)
  (if (zero? n) k
    (factorial (- n 1)
      (* k n))))
\end{verbatim}

\begin{verbatim}
def factorial(n, k):
    while n > 0:
        n, k = n-1, k*n
    return k
\end{verbatim}

\begin{tabular}{ll}
  Time & Space \\
  $\Theta(n)$ & $\Theta(1)$ \\
\end{tabular}
Tail Recursion

From 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."

\[
(\text{define } (\text{factorial} \ n \ k) \\
\hspace{1em} (\text{if } (\text{zero?} \ n) \ k \\
\hspace{2em} (\text{factorial} (- \ n \ 1) \\
\hspace{3em} (* \ k \ n))))
\]

Should use resources like

\[
\begin{array}{|c|c|}
\hline
\text{Time} & \text{Space} \\
\Theta(n) & \Theta(1) \\
\hline
\end{array}
\]

How? Eliminate the middleman!
Tail Recursion

From 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."

```scheme
(define (factorial n k)
  (if (zero? n) k
      (factorial (- n 1) (* k n))))
```

```python
def factorial(n, k):
    while n > 0:
        n, k = n-1, k*n
    return k
```

<table>
<thead>
<tr>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Θ(n)</td>
<td>Θ(1)</td>
</tr>
</tbody>
</table>

How? Eliminate the middleman!
Tail Calls
Tail Calls
Tail Calls

A procedure call that has not yet returned is active. Some procedure calls are tail calls. A Scheme interpreter 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. A Scheme interpreter 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. A Scheme interpreter 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 expression
Tail Calls

A procedure call that has not yet returned is active. Some procedure calls are tail calls. A Scheme interpreter 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 expression
- Sub-expressions 2 & 3 in a tail context if expression
Tail Calls

A procedure call that has not yet returned is **active**. Some procedure calls are **tail calls**. A Scheme interpreter 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 expression
- Sub-expressions 2 & 3 in a tail context if expression

```scheme
(define (factorial n k)
  (if (= n 0) k
      (factorial (- n 1)
        (* k n))))
```
Tail Calls

A procedure call that has not yet returned is active. Some procedure calls are tail calls. A Scheme interpreter 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 expression
• Sub-expressions 2 & 3 in a tail context if expression

```
(define (factorial n k)
  (if (= n 0) k
    (factorial (- n 1)
      (* k n))))
```
Tail Calls

A procedure call that has not yet returned is active. Some procedure calls are tail calls. A Scheme interpreter 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 expression

• Sub-expressions 2 & 3 in a tail context if expression

```scheme
(define (factorial n k)
  (if (= n 0) k
      (factorial (- n 1) (* k n))) )
```
Tail Calls

A procedure call that has not yet returned is active. Some procedure calls are tail calls. A Scheme interpreter 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 expression
- Sub-expressions 2 & 3 in a tail context if expression
- All non-predicate sub-expressions in a tail context cond

```
(define (factorial n k)
  (if (= n 0) k
      (factorial (- n 1)
                  (* k n)))
)
```
Tail Calls

A procedure call that has not yet returned is active. Some procedure calls are tail calls. A Scheme interpreter 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 expression
- Sub-expressions 2 & 3 in a tail context if expression
- All non-predicate sub-expressions in a tail context cond
- The last sub-expression in a tail context and or or or

```
(define (factorial n k)
  (if (= n 0) k
      (factorial (- n 1) (* k n)));
)```
Tail Calls

A procedure call that has not yet returned is active. Some procedure calls are tail calls. A Scheme interpreter 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 expression
- Sub-expressions 2 & 3 in a tail context if expression
- All non-predicate sub-expressions in a tail context cond
- The last sub-expression in a tail context and or or
- The last sub-expression in a tail context begin

```
(define (factorial n k)
  (if (= n 0) k
      (factorial (- n 1)
                  (* k n)))
)
```
Tail Calls

A procedure call that has not yet returned is active. Some procedure calls are tail calls. A Scheme interpreter 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 expression
- Sub-expressions 2 & 3 in a tail context if expression
- All non-predicate sub-expressions in a tail context cond
- The last sub-expression in a tail context and or or
- The last sub-expression in a tail context begin

```
(define (factorial n k)
  (if (= n 0) k
    (factorial (- n 1)
      (* k n))))
```
Example: Length of a List
Example: Length of a List

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

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

Linear recursive procedures can often be re-written to use tail calls
Example: Length of a List

(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 re-written to use tail calls.
Example: Length of a List

```
(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 re-written to use tail calls.
Example: Length of a List

(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 re-written to use tail calls.
Example: Length of a List

\[
\text{(define (length s)}
\]
\[
\text{(if (null? s) 0)} \quad \text{Not a tail context}
\]
\[
\text{(+) (length (cdr s))})}
\]

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

Linear recursive procedures can often be re-written to use tail calls.
Example: Length of a List

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

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

Linear recursive procedures can often be re-written to use tail calls

\[
\text{(define (length-tail s)}
\]
Example: Length of a List

\[
\text{(define (length s)}
\fontfamily{pzc}\selectfont
\text{(if (null? s) 0 Not a tail context)}
\text{(+ 1 (length (cdr s))) ) )}
\]

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

Linear recursive procedures can often be re-written to use tail calls

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

(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 re-written to use tail calls.

(define (length-tail s)

  (define (length-iter s n)
    (if (null? s) n
Example: Length of a List

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

Linear recursive procedures can often be re-written to use tail calls.

```scheme
(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)) ) )
```
Example: Length of a List

```
(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 re-written 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 of a List

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

Linear recursive procedures can often be re-written 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))
```
Example: Length of a List

```
(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 re-written 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 of a List

```
(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 re-written 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 of a List

\[
\text{(define (length s)} \\
\text{\quad (if (null? s) 0 \quad Not a tail context)} \\
\text{\quad (+ 1 (length (cdr s)) )) )}
\]

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

Linear recursive procedures can often be re-written to use tail calls.

\[
\text{(define (length-tail s)} \\
\text{\quad (define (length-iter s n)} \\
\text{\quad \quad (if (null? s) n \quad \quad Recursive call is a tail call)} \\
\text{\quad \quad \quad (length-iter (cdr s) (+ 1 n))) ))} \\
\text{\quad (length-iter s 0)) )}
\]
Eval with Tail Call Optimization
Eval with Tail Call Optimization

The return value of the tail call is the return value of the current procedure call.
Eval with Tail Call Optimization

The return value of the tail call is the return value of the current procedure call. Therefore, tail calls shouldn't increase the environment size.
Eval with Tail Call Optimization

The return value of the tail call is the return value of the current procedure call.

Therefore, tail calls shouldn't increase the environment size.

(Demo)
Tail Recursion Examples
Which Procedures are Tail Recursive?

Which of the following procedures run in constant space? \( \Theta(1) \)

;; Compute the length of s.
(define (length s)
  (+ 1 (if (null? s)
            -1
            (length (cdr s)))))

;; Return the nth Fibonacci number.
(define (fib n)
  (define (fib-iter current k)
    (if (= k n)
      current
      (fib-iter (+ current
                  (fib (- k 1)))
                 (+ k 1)))
  (if (= 1 n) 0 (fib-iter 1 2)))

;; Return whether s contains v.
(define (contains s v)
  (if (null? s)
      false
      (if (= v (car s))
          true
          (contains (cdr s) v))))

;; Return whether s has any repeated elements.
(define (has-repeat s)
  (if (null? s)
      false
      (if (contains? (cdr s) (car s))
          true
          (has-repeat (cdr s)))))
**Which Procedures are Tail Recursive?**

Which of the following procedures run in constant space? \( \Theta(1) \)

;; Compute the length of s.
(define (length s)
  (+ 1 (if (null? s)
          -1
          (length (cdr s)))))

;; Return the nth Fibonacci number.
(define (fib n)
  (define (fib-iter current k)
    (if (= k n)
      current
      (fib-iter (+ current
                 (fib (- k 1)))
                (+ k 1)))
  (if (= 1 n) 0 (fib-iter 1 2)))

;; Return whether s contains v.
(define (contains s v)
  (if (null? s)
    false
    (if (= v (car s))
      true
      (contains (cdr s) v))))

;; Return whether s has any repeated elements.
(define (has-repeat s)
  (if (null? s)
    false
    (if (contains? (cdr s) (car s))
      true
      (has-repeat (cdr s)))))

15
Which Procedures are Tail Recursive?

Which of the following procedures run in constant space? $\Theta(1)$

;;; Compute the length of s.
(define (length s)
  (+ 1 (if (null? s)
           -1
           (length (cdr s)))))

;;; Return the nth Fibonacci number.
(define (fib n)
  (define (fib-iter current k)
    (if (= k n)
      current
      (fib-iter (+ current
                 (fib (- k 1)))
                 (+ k 1)))
    (if (= 1 n) 0 (fib-iter 1 2)))

;;; Return whether s contains v.
(define (contains s v)
  (if (null? s)
    false
    (if (= v (car s))
      true
      (contains (cdr s) v))))

;;; Return whether s has any repeated elements.
(define (has-repeat s)
  (if (null? s)
    false
    (if (contains? (cdr s) (car s))
      true
      (has-repeat (cdr s))))
)
Which Procedures are Tail Recursive?

Which of the following procedures run in constant space? $\Theta(1)$

;; Compute the length of s.
(define (length s)
  (+ 1 (if (null? s)
         -1
         (length (cdr s)))))

;; Return the nth Fibonacci number.
(define (fib n)
  (define (fib-iter current k)
    (if (= k n)
       current
       (fib-iter (+ current (fib (- k 1)))
                  (+ k 1)))
  (if (= 1 n) 0 (fib-iter 1 2)))

;; Return whether s contains v.
(define (contains s v)
  (if (null? s)
      false
      (if (= v (car s))
          true
          (contains (cdr s) v))))

;; Return whether s has any repeated elements.
(define (has-repeat s)
  (if (null? s)
      false
      (if (contains? (cdr s) (car s))
          true
          (has-repeat (cdr s)))))

(define (contains? s f)
  (if (null? s)
      false
      (if (= f (car s))
          true
          (contains? (cdr s) f))))

(define (has-repeat? s)
  (if (null? s)
      false
      (if (contains? (cdr s) (car s))
          true
          (has-repeat? (cdr s)))))
Which Procedures are Tail Recursive?

Which of the following procedures run in constant space? \( \Theta(1) \)

;;; Compute the length of \( s \).
(define (length s)
  (+ 1 (if (null? s)
         -1
         (length (cdr s)))))

;;; Return whether \( s \) contains \( v \).
(define (contains s v)
  (if (null? s)
      false
      (if (= v (car s))
       true
       (contains (cdr s) v))))

;;; Return the \( n \)th Fibonacci number.
(define (fib n)
  (define (fib-iter current k)
    (if (= k n)
        current
        (fib-iter (+ current
                    (fib (- k 1)))
                  (+ k 1))
    (if (= 1 n) 0 (fib-iter 1 2)))
  (fib-iter 1 2))

;;; Return whether \( s \) has any repeated elements.
(define (has-repeat s)
  (if (null? s)
      false
      (if (contains? (cdr s) (car s))
        true
        (has-repeat (cdr s)))))
Which Procedures are Tail Recursive?

Which of the following procedures run in constant space? $\Theta(1)$

;; Compute the length of s.
(define (length s)
  (+ 1 (if (null? s)
    -1
    (length (cdr s))))))

;; Return the nth Fibonacci number.
(define (fib n)
  (define (fib-iter current k)
    (if (= k n)
      current
      (fib-iter (+ current
                    (fib (- k 1)))
                (+ k 1))
    )
  )
  (if (= 1 n) 0 (fib-iter 1 2)))

;; Return whether s contains v.
(define (contains s v)
  (if (null? s)
    false
    (if (= v (car s))
      true
      (contains (cdr s) v))))

;; Return whether s has any repeated elements.
(define (has-repeat s)
  (if (null? s)
    false
    (if (contains? (cdr s) (car s))
      true
      (has-repeat (cdr s)))))
Which Procedures are Tail Recursive?

Which of the following procedures run in constant space? $\Theta(1)$

;;; Compute the length of s.
(define (length s)
  (+ 1 (if (null? s)
            -1
            (length (cdr s)))))

;;; Return the nth Fibonacci number.
(define (fib n)
  (define (fib-iter current k)
    (if (= k n)
        current
        (fib-iter (+ current
                    (fib (- k 1)))
                  (+ k 1)))
  (if (= 1 n) 0 (fib-iter 1 2)))

;;; Return whether s contains v.
(define (contains s v)
  (if (null? s)
      false
      (if (= v (car s))
        true
        (contains (cdr s) v))))

;;; Return whether s has any repeated elements.
(define (has-repeat s)
  (if (null? s)
    false
    (if (contains? (cdr s) (car s))
      true
      (has-repeat (cdr s)))))

15
Which Procedures are Tail Recursive?

Which of the following procedures run in constant space? $\Theta(1)$

;; Compute the length of s.
(define (length s)
  (+ 1 (if (null? s)
          -1
          (length (cdr s))))))

;; Return the nth Fibonacci number.
(define (fib n)
  (define (fib-iter current k)
    (if (= k n)
      current
      (fib-iter (+ current
                  (fib (- k 1)))
                 (+ k 1))))
  (if (= 1 n) 0 (fib-iter 1 2)))

;; Return whether s contains v.
(define (contains s v)
  (if (null? s)
      false
      (if (= v (car s))
        true
        (contains (cdr s) v)))))

;; Return whether s has any repeated elements.
(define (has-repeat s)
  (if (null? s)
      false
      (if (contains? (cdr s) (car s))
        true
        (has-repeat (cdr s))))))
Which Procedures are Tail Recursive?

Which of the following procedures run in constant space? Θ(1)

;; Compute the length of s.
(define (length s)
 (+ 1 (if (null? s)
          -1
          (length (cdr s)))))

;; Return the nth Fibonacci number.
(define (fib n)
  (define (fib-iter current k)
    (if (= k n)
        current
        (fib-iter (+ current
                    (fib (- k 1)))
                   (+ k 1)))
  (if (= 1 n) 0 (fib-iter 1 2)))

;; Return whether s contains v.
(define (contains s v)
  (if (null? s)
      false
      (if (= v (car s))
          true
          (contains (cdr s) v))))

;; Return whether s has any repeated elements.
(define (has-repeat s)
  (if (null? s)
      false
      (if (contains? (cdr s) (car s))
          true
          (has-repeat (cdr s)))))
Which of the following procedures run in constant space? \( \Theta(1) \)

### Compute the length of s.

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

### Return the nth Fibonacci number.

```scheme
(define (fib n)
  (define (fib-iter current k)
    (if (= k n)
        current
        (fib-iter (+ current
                    (fib (- k 1)))
                  (+ k 1)))
  (if (= 1 n) 0 (fib-iter 1 2)))
```

### Return whether s contains v.

```scheme
(define (contains s v)
  (if (null? s)
      false
      (if (= v (car s))
          true
          (contains (cdr s) v))))
```

### Return whether s has any repeated elements.

```scheme
(define (has-repeat s)
  (if (null? s)
      false
      (if (contains? (cdr s) (car s))
          true
          (has-repeat (cdr s)))))
```
Which Procedures are Tail Recursive?

Which of the following procedures run in constant space? $\Theta(1)$

;; Compute the length of s.
(define (length s)
  (+ 1 (if (null? s)
          -1
          (length (cdr s))))))

;; Return the nth Fibonacci number.
(define (fib n)
  (define (fib-iter current k)
    (if (= k n)
        current
        (fib-iter (+ current (fib (- k 1)))
                  (+ k 1)))
  (if (= 1 n) 0 (fib-iter 1 2)))

;; Return whether s contains v.
(define (contains s v)
  (if (null? s)
      false
      (if (= v (car s))
          true
          (contains (cdr s) v)))))

;; Return whether s has any repeated elements.
(define (has-repeat s)
  (if (null? s)
      false
      (if (contains? (cdr s) (car s))
          true
          (has-repeat (cdr s))))))
Which Procedures are Tail Recursive?

Which of the following procedures run in constant space? $\Theta(1)$

;; Compute the length of s.
(define (length s)
  (+ 1 (if (null? s)
          -1 (length (cdr s)))))

;; Return the nth Fibonacci number.
(define (fib n)
  (define (fib-iter current k)
    (if (= k n)
        current
        (fib-iter (+ current (fib (- k 1)))
                 (+ k 1)))
  (if (= 1 n) 0 (fib-iter 1 2)))

;; Return whether s contains v.
(define (contains s v)
  (if (null? s)
      false
      (if (= v (car s))
          true
          (contains (cdr s) v))))

;; Return whether s has any repeated elements.
(define (has-repeat s)
  (if (null? s)
      false
      (if (contains? (cdr s) (car s))
          true
          (has-repeat (cdr s)))))

15
Which Procedures are Tail Recursive?

Which of the following procedures run in constant space? \(\Theta(1)\)

;; Compute the length of s.
(define (length s)
  (+ 1 (if (null? s)
         -1
         (length (cdr s))))))

;; Return the nth Fibonacci number.
(define (fib n)
  (define (fib-iter current k)
    (if (= k n)
      current
      (fib-iter (+ current
                 (fib (- k 1)))
               (+ k 1)))
  (if (= 1 n) 0 (fib-iter 1 2)))

;; Return whether s contains v.
(define (contains s v)
  (if (null? s)
    false
    (if (= v (car s))
      true
      (contains (cdr s) v)))))

;; Return whether s has any repeated elements.
(define (has-repeat s)
  (if (null? s)
    false
    (if (contains? (cdr s) (car s))
      true
      (has-repeat (cdr s))))))
Which Procedures are Tail Recursive?

Which of the following procedures run in constant space? $\Theta(1)$

;; Compute the length of s.
(define (length s)
  (+ 1 (if (null? s)
           -1
           (length (cdr s)))))

;; Return the nth Fibonacci number.
(define (fib n)
  (define (fib-iter current k)
    (if (= k n)
        current
        (fib-iter (+ current
                   (fib (- k 1)))
                  (+ k 1)))
  )
  (if (= 1 n) 0 (fib-iter 1 2)))

;; Return whether s contains v.
(define (contains s v)
  (if (null? s)
      false
      (if (= v (car s))
       true
       (contains (cdr s) v))))

;; Return whether s has any repeated elements.
(define (has-repeat s)
  (if (null? s)
      false
      (if (contains? (cdr s) (car s))
       true
       (has-repeat (cdr s)))))

Which Procedures are Tail Recursive?

Which of the following procedures run in constant space? \( \Theta(1) \)

`; Compute the length of s. 
(define (length s)
  (+ 1 (if (null? s)
           -1
           (length (cdr s)))))

`; Return the nth Fibonacci number. 
(define (fib n)
  (define (fib-iter current k)
    (if (= k n)
      current
      (fib-iter (+ current (fib (- k 1)))
                (+ k 1)))
  (if (= 1 n) 0 (fib-iter 1 2)))

`; Return whether s contains v. 
(define (contains s v)
  (if (null? s)
    false
    (if (= v (car s))
      true
      (contains (cdr s) v))))

`; Return whether s has any repeated elements. 
(define (has-repeat s)
  (if (null? s)
    false
    (if (contains? (cdr s) (car s))
      true
      (has-repeat (cdr s)))))


Which Procedures are Tail Recursive?

Which of the following procedures run in constant space? \( \Theta(1) \)

;;; Compute the length of s.
(define (length s)
  (+ 1 (if (null? s)
          -1
          (length (cdr s))))
)

;;; Return the nth Fibonacci number.
(define (fib n)
  (define (fib-iter current k)
    (if (= k n)
      current
      (fib-iter (+ current
                  (fib (- k 1)))
                (+ k 1))
    )
  
  (if (= 1 n) 0 (fib-iter 1 2)))

;;; Return whether s contains v.
(define (contains s v)
  (if (null? s)
      false
      (if (= v (car s))
        true
        (contains (cdr s) v)))
)

;;; Return whether s has any repeated elements.
(define (has-repeat s)
  (if (null? s)
      false
      (if (contains? (cdr s) (car s))
        true
        (has-repeat (cdr s)))))
Map and Reduce
Example: Reduce
Example: Reduce

(define (reduce procedure s start))
Example: Reduce

(define (reduce procedure s start)

(reduce * '(3 4 5) 2)
Example: Reduce

(define (reduce procedure s start)

(reduce * '(3 4 5) 2)  120
Example: Reduce

(define (reduce procedure s start)

(reduce * '(3 4 5) 2) 120

(reduce (lambda (x y) (cons y x)) '(3 4 5) '(2))
Example: Reduce

```
(define (reduce procedure s start)

(reduce * '(3 4 5) 2) 120
(reduce (lambda (x y) (cons y x)) '(3 4 5) '(2)) (5 4 3 2)
```
Example: Reduce

\[
\text{(define (reduce procedure s start)}
\]
\[
\text{(if (null? s) start)}
\]

\[
\text{(reduce * '(3 4 5) 2)} \quad 120
\]

\[
\text{(reduce (lambda (x y) (cons y x)) '(3 4 5) '(2))} \quad (5 4 3 2)
\]
Example: Reduce

```
(define (reduce procedure s start)
  (if (null? s) start
   (reduce procedure
        (reduce * '(3 4 5) 2) 120
   (reduce (lambda (x y) (cons y x)) '(3 4 5) '(2)) (5 4 3 2)
```
Example: Reduce

(define (reduce procedure s start)
  (if (null? s) start
      (reduce procedure
                (cdr s))

(reduce * '(3 4 5) 2) 120

(reduce (lambda (x y) (cons y x)) '(3 4 5) '(2)) (5 4 3 2)
Example: Reduce

```
(define (reduce procedure s start)
  (if (null? s) start
      (reduce procedure
                (reduce procedure
                          (cdr s)
                          (procedure start (car s)))
                start)
  )

(reduce * '(3 4 5) 2) 120
(reduce (lambda (x y) (cons y x)) '(3 4 5) '(2)) (5 4 3 2)
```
Example: Reduce

```
(define (reduce procedure s start)
  (if (null? s) start
   (reduce procedure
    (cdr s)
    (procedure start (car s)))))
```

(reduce * '(3 4 5) 2)  120

(reduce (lambda (x y) (cons y x)) '(3 4 5) '(2))  (5 4 3 2)
Example: Reduce

```
(define (reduce procedure s start)
  (if (null? s) start
    (reduce procedure
         (cdr s)
         (procedure start (car s)))))
```

(reduce * '(3 4 5) 2) 120

(reduce (lambda (x y) (cons y x)) '(3 4 5) '(2)) (5 4 3 2)
Example: Reduce

(define (reduce procedure s start)
  (if (null? s) start
    (reduce procedure
      (cdr s)
      (procedure start (car s))))))

(reduce * '(3 4 5) 2) 120
(reduce (lambda (x y) (cons y x)) '(3 4 5) '(2)) (5 4 3 2)
Example: Reduce

```
(define (reduce procedure s start)
  (if (null? s) start
      (reduce procedure
        (cdr s)
        (procedure start (car s))))))
```

Recursive call is a tail call

```
(reduce * '(3 4 5) 2) 120
(reduce (lambda (x y) (cons y x)) '(3 4 5) '(2)) (5 4 3 2)
```
Example: Reduce

```
(define (reduce procedure s start)
  (if (null? s) start
      (reduce procedure
        (cdr s)
        (procedure start (car s))))))
```

Recursive call is a tail call

Space depends on what `procedure` requires

```
(reduce * '(3 4 5) 2) 120
(reduce (lambda (x y) (cons y x)) '(3 4 5) '(2)) (5 4 3 2)
```
Example: Map with Only a Constant Number of Frames
Example: Map with Only a Constant Number of Frames

(define (map procedure s))
Example: Map with Only a Constant Number of Frames

(define (map procedure s)
  (if (null? s)
Example: Map with Only a Constant Number of Frames

```
(define (map procedure s)
  (if (null? s)
      nil
      (map procedure (cdr s)) ))
```
Example: Map with Only a Constant Number of Frames

\[
\text{(define (map procedure s)}
\text{  (if (null? s)}
\text{    nil)
\text{  (cons (procedure (car s)))}
\]

Example: Map with Only a Constant Number of Frames

\[
\text{(define (map procedure s)} \\
\text{  (if (null? s)} \\
\text{    nil} \\
\text{    (cons (procedure (car s))} \\
\text{      (map procedure (cdr s)))))})
\]
Example: Map with Only a Constant Number of Frames

```
(define (map procedure s)
  (if (null? s)
      nil
      (cons (procedure (car s))
            (map procedure (cdr s))))
)

(map (lambda (x) (- 5 x)) (list 1 2))
```
Example: Map with Only a Constant Number of Frames

(define (map procedure s)
  (if (null? s)
      nil
      (cons (procedure (car s))
            (map procedure (cdr s))))
)

(map (lambda (x) (- 5 x)) (list 1 2))
Example: Map with Only a Constant Number of Frames

```
(define (map procedure s)
  (if (null? s)
      nil
      (cons (procedure (car s))
            (map procedure (cdr s)))))

(map (lambda (x) (- 5 x)) (list 1 2))
```
Example: Map with Only a Constant Number of Frames

\[
\text{(define (map procedure s)}
\text{  (if (null? s)}
\text{    nil)
\text{  (cons (procedure (car s))
\text{    (map procedure (cdr s))}})
\text{  )})
\]

\[
\text{(map (lambda (x) (- 5 x)) (list 1 2))}
\]
Example: Map with Only a Constant Number of Frames

```
(define (map procedure s)
  (if (null? s)
      nil
      (cons (procedure (car s))
            (map procedure (cdr s)))))

(map (lambda (x) (- 5 x)) (list 1 2))
```
Example: Map with Only a Constant Number of Frames

```
(define (map procedure s)
  (if (null? s)
      nil
      (cons (procedure (car s))
            (map procedure (cdr s))))
)

(map (lambda (x) (- 5 x)) (list 1 2))
```
Example: Map with Only a Constant Number of Frames

```
(define (map procedure s)
  (if (null? s)
      nil
      (cons (procedure (car s))
            (map procedure (cdr s)))))
```

```
(map (lambda (x) (- 5 x)) (list 1 2))
```
Example: Map with Only a Constant Number of Frames

```
(define (map procedure s)
  (if (null? s)
      nil
      (cons (procedure (car s))
            (map procedure (cdr s))))
)
```

```
(map (lambda (x) (- 5 x)) (list 1 2))
```
Example: Map with Only a Constant Number of Frames

(define (map procedure s)
  (if (null? s)
      nil
      (cons (procedure (car s))
            (map procedure (cdr s)))))

(map (lambda (x) (- 5 x)) (list 1 2))
Example: Map with Only a Constant Number of Frames

```
(define (map procedure s)
  (if (null? s)
      nil
      (cons (procedure (car s))
            ((map procedure (cdr s))))))

(map (lambda (x) (- 5 x)) (list 1 2))
```
Example: Map with Only a Constant Number of Frames

```
(define (map procedure s)
  (if (null? s)
      nil
      (cons (procedure (car s))
            ((map procedure (cdr s)))))
)

(map (lambda (x) (- 5 x)) (list 1 2))
```
Example: Map with Only a Constant Number of Frames

```
(define (map procedure s)
  (if (null? s)
      nil
      (cons (procedure (car s))
            (map procedure (cdr s)))))

(map (lambda (x) (- 5 x)) (list 1 2))
```

```
(define (map procedure s)
  (if (null? s)
      nil
      (cons (procedure (car s))
            (map procedure (cdr s))))

(define (map-reverse s m)
  (if (null? s)
      nil
      (cons m (map-reverse (cdr s) m)))))
```
Example: Map with Only a Constant Number of Frames

```
(define (map procedure s)
  (if (null? s)
      nil
      (cons (procedure (car s))
            (map procedure (cdr s)))))

(map (lambda (x) (- 5 x)) (list 1 2))
```
Example: Map with Only a Constant Number of Frames

```
(define (map procedure s)
  (if (null? s)
      nil
      (cons (procedure (car s))
            (map procedure (cdr s)))))

(map (lambda (x) (- 5 x)) (list 1 2))
```

```
(define (map-reverse s m)
  (if (null? s)
      m
      (map-reverse (cdr s)...
```

Diagram:

```
4              3
Pair           Pair

2              3
Pair
1              nil
Pair
nil
```

```
s
Pair
```

```
s
Pair
```

```
s
Pair
```

```
Example: Map with Only a Constant Number of Frames

```
(define (map procedure s)
  (if (null? s)
      nil
      (cons (procedure (car s))
            (map procedure (cdr s))))
)

(map (lambda (x) (- 5 x)) (list 1 2))
```

```
(define (map procedure s)
  (if (null? s)
      nil
      (cons (procedure (car s))
            (map procedure (cdr s))))
)

(define (map-reverse s m)
  (if (null? s)
      m
      (map-reverse (cdr s)
                   (cons (procedure (car s)))))
)
```
Example: Map with Only a Constant Number of Frames

\[
\begin{align*}
\text{(define (map procedure s)} & \quad \text{(define (map-reverse s m)} \\
\text{  (if (null? s)} & \quad \text{  (if (null? s)} \\
\text{    nil)} & \quad \text{    m)} \\
\text{    (cons (procedure (car s))} & \quad \text{    (map-reverse (cdr s)) (cons (procedure (car s))} \\
\text{      (map procedure (cdr s)))))}) & \quad \text{m))})
\end{align*}
\]

(map (lambda (x) (- 5 x)) (list 1 2))
Example: Map with Only a Constant Number of Frames

\[
\text{(define (map \text{procedure} s)
(\text{if} (\text{null?} s)
  \text{nil}
  (\text{cons} (\text{procedure} (\text{car} s))
                (\text{map \text{procedure} (\text{cdr} s))))))
}
\]

\[
\text{(map (lambda (x) (- 5 x)) (list 1 2))}
\]

\[
\text{(define (map-reverse \text{procedure} s \text{m})
(\text{if} (\text{null?} s)
  \text{m}
  (\text{map-reverse} (\text{cdr} s)
                (\text{cons} (\text{procedure} (\text{car} s))
                             \text{m}))
  \text{nil}))
\]

\[
\text{(reverse (map-reverse s \text{nil}))}
\]
Example: Map with Only a Constant Number of Frames

\[
\text{(define (map procedure s)} \quad \begin{cases}
\text{nil} & \text{if (null? s)} \\
\text{(cons (procedure (car s)) (map procedure (cdr s)))} & \text{else}
\end{cases}
\text{)}
\]

\[
\text{(map (lambda (x) (- 5 x)) (list 1 2))}
\]
Example: Map with Only a Constant Number of Frames

\[
(\text{define (map procedure s)} \\
(\text{if (null? s)} \\
\quad \text{nil} \\
\quad (\text{cons (procedure (car s))} \\
\quad \quad (\text{map procedure (cdr s)}))))
\]

\[
(\text{map (lambda (x) (- 5 x)) (list 1 2))
\]

\[
(\text{define (map-reverse s m)} \\
(\text{if (null? s)} \\
\quad m \\
\quad (\text{map-reverse (cdr s)} \\
\quad (\text{cons (procedure (car s))} \\
\quad \quad m))) \\
\quad (\text{reverse (map-reverse s nil)}))
\]
Example: Map with Only a Constant Number of Frames

(define (map procedure s)
  (if (null? s)
      nil
      (cons (procedure (car s))
            (map procedure (cdr s)))))

(map (lambda (x) (- 5 x)) (list 1 2))

(define (map-reverse s m)
  (if (null? s)
      m
      (map-reverse (cdr s)
                   (cons (procedure (car s))
                          m))
      ))

(reverse (map-reverse s nil)))

(define (reverse s)
  (define (reverse-iter s r)
    ...)

...
Example: Map with Only a Constant Number of Frames

(define (map procedure s)
  (if (null? s)
      nil
      (cons (procedure (car s))
            ((map procedure (cdr s))))))

(map (lambda (x) (- 5 x)) (list 1 2))

(define (map procedure s)
  (define (map-reverse s m)
    (if (null? s)
        m
        (map-reverse (cdr s)
                    (cons (procedure (car s))
                          m))
      ))

(reverse (map-reverse s nil)))

(define (reverse s)
  (define (reverse-iter s r)
    (if (null? s)
        nil
        (cons (procedure (car s))
              (reverse-iter (cdr s) (cons procedure (car s))))))

(reverse (map-reverse s nil)))
(define (map procedure s)
  (if (null? s)
      nil
      (cons (procedure (car s))
        (map procedure (cdr s)))))

(map (lambda (x) (- 5 x)) (list 1 2))
Example: Map with Only a Constant Number of Frames

(define (map procedure s)
  (if (null? s)
      nil
      (cons (procedure (car s))
            (map procedure (cdr s)))))

(map (lambda (x) (- 5 x)) (list 1 2))

(define (map-reverse s m)
  (if (null? s)
      m
      (map-reverse (cdr s)
                   (cons (procedure (car s))
                         m))))

(reverse (map-reverse s nil)))

(define (reverse s)
  (define (reverse-iter s r)
    (if (null? s)
        r
        (reverse-iter (cdr s)))
  (reverse-iter s nil))

(define (reverse-iter s r)
  (if (null? s)
      r
      (reverse-iter (cdr s)))
  (reverse-iter s nil))
Example: Map with Only a Constant Number of Frames

(define (map procedure s)
  (if (null? s)
      nil
      (cons (procedure (car s))
            (map procedure (cdr s))))
)

(map (lambda (x) (- 5 x)) (list 1 2))

(define (map procedure s)
  (if (null? s)
      nil
      (map procedure (cdr s))))

(reverse (map-reverse s nil))

(define (reverse s)
  (define (reverse-iter s r)
    (if (null? s)
        r
        (reverse-iter (cdr s) (cons (car s) r))))
)
Example: Map with Only a Constant Number of Frames

\[
\text{(define (map procedure s)} \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\text{(if (null? s)} \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\text{nil)} \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\text{(cons (procedure (car s)))} \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\text{(map procedure (cdr s)))))})
\]

\[
\text{(define (map-reverse s m)} \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\text{(if (null? s)} \\\\\\\\\\\\\\\\\\\\\\\\\\\\\text{m)} \\\\\\\\\\\\\text{(map-reverse (cdr s)} \\\\\\\\\\\text{(cons (procedure (car s)))} \\\\\text{m))}) \\\text{))}
\]

\[
\text{(define (reverse s)} \\\\\\\\\\text{(define (reverse-iter s r)} \\\\\text{(if (null? s)} \\\\\text{r)} \\\\\text{(reverse-iter (cdr s)} \\\\\\\\\\text{(cons (car s) r)))))}) \\\text{(reverse-iter s nil))}
\]
Example: Map with Only a Constant Number of Frames

```
(define (map procedure s)
  (if (null? s)
      nil
      (cons (procedure (car s))
            (map procedure (cdr s))))
)

(map (lambda (x) (- 5 x)) (list 1 2))
```

```
(define (reverse s)
  (define (reverse-iter s r)
    (if (null? s)
        r
        (reverse-iter (cdr s) (cons (car s) r))))
  (reverse-iter s nil))
```

```
(define (map-reverse s m)
  (if (null? s)
      m
      (map-reverse (cdr s)
                   (cons (procedure (car s)) m)))
)

(reverse (map-reverse s nil))
```
Example: Map with Only a Constant Number of Frames

```
(define (map procedure s)
  (if (null? s)
      nil
      (cons (procedure (car s))
            (map procedure (cdr s)))))

(map (lambda (x) (- 5 x)) (list 1 2))
```

```
(define (map-reverse s m)
  (if (null? s)
      m
      (map-reverse (cdr s)
                   (cons (procedure (car s)) m))
   ))

(reverse (map-reverse s nil))
```

```
(define (reverse s)
  (define (reverse-iter s r)
    (if (null? s)
        r
        (reverse-iter (cdr s)
                      (cons (car s) r))
     ))

(reverse-iter s nil))
```
Example: Map with Only a Constant Number of Frames

```
(define (map procedure s)
  (if (null? s)
      nil
      (cons (procedure (car s))
            (map procedure (cdr s))))
)

(map (lambda (x) (- 5 x)) (list 1 2))
```

```
(define (map-reverse s m)
  (if (null? s)
      m
      (map-reverse (cdr s)
                   (cons (procedure (car s))
                         m)))
)

(reverse (map-reverse s nil))
```

```
(define (reverse s)
  (define (reverse-iter s r)
    (if (null? s)
        r
        (reverse-iter (cdr s)
                      (cons (car s) r)))
  )

(reverse-iter s nil))
```
Example: Map with Only a Constant Number of Frames

```
(define (map procedure s)
  (if (null? s)
      nil
      (cons (procedure (car s))
            (map procedure (cdr s)))))

(map (lambda (x) (- 5 x)) (list 1 2))
```

```
(define (map procedure s)
  (if (null? s)
      nil
      (cons (procedure (car s))
            (map procedure (cdr s)))))

(reverse (map-reverse s nil)))
```

```
(define (reverse s)
  (define (reverse-iter s r)
    (if (null? s)
        r
        (reverse-iter (cdr s) (cons (car s) r))
    )
  )

(reverse-iter s nil))
```
General Computing Machines
An Analogy: Programs Define Machines
An Analogy: Programs Define Machines

Programs specify the logic of a computational device
An Analogy: Programs Define Machines

Programs specify the logic of a computational device

factorial
An Analogy: Programs Define Machines

Programs specify the logic of a computational device

\[ \text{factorial} = \text{factorial} \times 1 \]
An Analogy: Programs Define Machines

Programs specify the logic of a computational device

\[
5 \rightarrow \frac{1}{\text{factorial}} = 1 \rightarrow 1 \rightarrow 1
\]

\[
- \rightarrow \text{factorial} \rightarrow *
\]
An Analogy: Programs Define Machines

Programs specify the logic of a computational device

$$5! = \text{factorial} \ast 1$$
Interpreters are General Computing Machine
Interpreters are General Computing Machine

An interpreter can be parameterized to simulate any machine
Interpreters are General Computing Machine

An interpreter can be parameterized to simulate any machine

5 \rightarrow \text{Scheme Interpreter} \rightarrow 120

\text{(define (factorial n) (if (zero? n) 1 (* n (factorial (- n 1)))))}
Interpreters are General Computing Machine

An interpreter can be parameterized to simulate any machine

\[
(\text{define} \ (\text{factorial} \ n) \\
(\text{if} \ (\text{zero?} \ n) \ 1 \ (* \ n \ (\text{factorial} \ (- n 1)))))
\]

Our Scheme interpreter is a universal machine
Interpreters are General Computing Machine

An interpreter can be parameterized to simulate any machine

```
(define (factorial n)
  (if (zero? n) 1 (* n (factorial (- n 1))))
```

Our Scheme interpreter is a universal machine

A bridge between the data objects that are manipulated by our programming language and the programming language itself
Interpreters are General Computing Machine

An interpreter can be parameterized to simulate any machine

\[
(\text{define } (\text{factorial } n) \\
(\text{if } (\text{zero? } n) 1 (* n (\text{factorial } (- n 1)))))
\]

Our Scheme interpreter is a universal machine

A bridge between the data objects that are manipulated by our programming language and the programming language itself

Internally, it is just a set of evaluation rules