Announcements

• Midterm survey due Monday 11/10 @ 11:59pm (Thanks!)
Announcements

• Midterm survey due Monday 11/10 @ 11:59pm (Thanks!)
• Homework 8 due Wednesday 11/12 @ 11:59pm (Scheme!)
Announcements

• Midterm survey due Monday 11/10 @ 11:59pm (Thanks!)
• Homework 8 due Wednesday 11/12 @ 11:59pm (Scheme!)
• Project 4 due Thursday 11/20 @ 11:59pm (Big!)
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)
Dynamic Scope

The way in which names are looked up in Scheme and Python is called lexical scope (or static scope)

**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)

**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)

**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)

**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)

**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)

**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**)

**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
Dynamic Scope

The way in which names are looked up in Scheme and Python is called lexical scope (or static scope)

**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)

**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

13
**Dynamic Scope**

The way in which names are looked up in Scheme and Python is called lexical scope (or static scope)

**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

```
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

\[
\text{factorial}(n, k) \text{ computes: } n! \times k
\]
Recursion and Iteration in Python

In Python, recursive calls always create new active frames

```
factorial(n, k) computes: n! * k
```

```python
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

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

```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.

`factorial(n, k)` computes: $n! * k$

```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></td>
</tr>
</tbody>
</table>
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

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

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

\(\Theta(n)\)
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></th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(\Theta(n))</td>
<td>(\Theta(n))</td>
</tr>
<tr>
<td>def factorial(n, k):</td>
<td></td>
<td></td>
</tr>
<tr>
<td>if n == 0:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>return k</td>
<td></td>
<td></td>
</tr>
<tr>
<td>else:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>return factorial(n-1, k*n)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(\Theta(n))</td>
<td>(\Theta(1))</td>
</tr>
<tr>
<td>def factorial(n, k):</td>
<td></td>
<td></td>
</tr>
<tr>
<td>while n &gt; 0:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>n, k = n-1, k*n</td>
<td></td>
<td></td>
</tr>
<tr>
<td>return k</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Recursion and Iteration in Python

In Python, recursive calls always create new active frames.

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

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

From the Revised\(^7\) Report on the Algorithmic Language Scheme:

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

<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."

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

\[
\text{def factorial(n, k):} \\
\text{  while n > 0:} \\
\text{    n, k = n-1, k*n} \\
\text{  return k}
\]

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

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

Should use resources like

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

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

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

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

How? Eliminate the middleman!

<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\(^7\) 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."

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

Should use resources like

\[
\textbf{def factorial}(n, k): \\
\text{ while } n > 0: \\
\quad n, k = n-1, k*n \\
\text{ return } k
\]

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

(Demo)
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

```
(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

```
(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

```
(define (factorial n k)
  (if (= n 0) k
      (* 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
      (* 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

(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

(define (length-tail s) ... )
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)
    ...
  )
)
```
Example: Length of a List

\begin{verbatim}
(define (length s)
  (if (null? s) 0 (+ 1 (length (cdr s)))))
\end{verbatim}

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.

\begin{verbatim}
(define (length-tail s)
  (define (length-iter s n)
    (if (null? s) n
\end{verbatim}
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.

```lisp
(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

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))))
  (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.

```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))))
  (length-iter s 0))
```
Example: Length of a List

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

Not a tail context

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

Linear recursive procedures can often be 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)))))

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 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)))))

Recursive call is a tail call

(define (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)))
  (fib-iter 0 1))

;; 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)))))


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)))))

14
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)))))

14
Which Procedures are Tail Recursive?

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

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

2. 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)))`

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

4. 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))))
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

\[(\text{define} \ (\text{reduce} \ \text{procedure} \ s \ \text{start}))\]
Example: Reduce

(\texttt{define (reduce procedure s start)})

(\texttt{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

\[
\text{\texttt{(define reduce procedure s start)}}
\]

\[
\begin{align*}
\text{(reduce \* '(3 4 5) 2)} & \quad 120 \\
\text{(reduce (lambda (x y) (cons y x)) '(3 4 5) '(2))} & \quad (5 4 3 2)
\end{align*}
\]
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

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

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

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

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

\[(\text{define} \ (\text{reduce} \ \text{procedure} \ s \ \text{start})\]
\[
\hspace{1em} (\text{if} \ (\text{null?} \ s) \ \text{start}
\hspace{1em} (\text{reduce} \ \text{procedure}
\hspace{2em} (\text{cdr} \ s)
\hspace{2em} (\text{procedure} \ \text{start} \ \text{(car} \ s))\))\))\]

Recursive call is a tail call

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

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

```
(define (reduce procedure s start)
  (if (null? s) start
   (reduce procedure
     (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
      ...))
```
Example: Map with Only a Constant Number of Frames

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

\[
\begin{align*}
&(\text{define } (\text{map} \ \text{procedure} \ s)) \\
&(\text{if } (\text{null?} \ s) \\
&\hspace{1em} \text{nil} \\
&\hspace{1em} (\text{cons} (\text{procedure} \ (\text{car} \ s)) \\
&\hspace{2em} (\text{map} \ \text{procedure} \ (\text{cdr} \ s))) )
\end{align*}
\]
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

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

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

```lisp
(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

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

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

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

Pair 4
Pair 3
Pair 1
Pair 2
Pair nil
Example: Map with Only a Constant Number of Frames

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

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

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

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

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

```
(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))acios))
```

```
Pair

4 3
```

```
Pair

1 2
```
Example: Map with Only a Constant Number of Frames

\[
\text{(define (map \textit{procedure} \textit{s})}
\]
\[
\text{  (if (null? \textit{s})}
\]
\[
\text{    \textit{nil}}
\]
\[
\text{    (cons (\textit{procedure} (\textit{car} \textit{s}))}
\]
\[
\text{      (map \textit{procedure} (\textit{cdr} \textit{s}))))}
\]
\[
\text{)}
\]
\[
\text{\text{map (lambda (x) (- 5 x)) (list 1 2))}
\]
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))
```

```
(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)) m))))
```

```
Pair

  4 3

Pair

  nil

Pair

  1 2

Pair

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

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

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

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

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

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

\[
\text{(reverse (map-reverse s nil))}
\]
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)
)
```
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))
                         m))))

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

```
(define (reverse s)
  (define (reverse-iter s r)
    nil)
```

```
(define (reverse s)
  (define (reverse-iter s r)
    nil)
```

Example: Map with Only a Constant Number of Frames

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

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

\[
\text{(define (reverse (map-reverse s nil))}
\]

\[
\text{(define (reverse s)}
\text{  (define (reverse-iter s r)}
\text{    (if (null? s)}
\text{      r)
\text{    (reverse-iter (cons (procedure (car s))}
\text{      s))})}
\]

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

```
(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)) m))))

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

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

(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 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)) m)))))

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

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

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

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

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

\begin{verbatim}
(define (map procedure s)
  (if (null? s)
    nil
    (cons (procedure (car s))
      (map procedure (cdr s)))))

(map (lambda (x) (- 5 x)) (list 1 2))
\end{verbatim}

\begin{verbatim}
(define (map-reverse s m)
  (if (null? s)
    m
    (map-reverse (cdr s)
      (cons (procedure (car s))
        m))
  )

(reverse (map-reverse s nil)))
\end{verbatim}

\begin{verbatim}
(define (reverse s)
  (define (reverse-iter s r)
    (if (null? s)
      r
      (reverse-iter (cdr s)
        (cons (car s) r))
    ))

(reverse s)
\end{verbatim}
Example: Map with Only a Constant Number of Frames

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

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

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

\[
(\text{reverse } (\text{map-reverse } s \text{ nil}))
\]

\[
(\text{define } (\text{reverse } s))
(\text{define } (\text{reverse-iter } s r))
(\text{if } (\text{null? } s)
\quad r
(\text{reverse-iter } (\text{cdr } s)
(\text{cons } (\text{car } s) r)))
\]

\[
(\text{reverse-iter } s \text{ nil})
\]
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))))
)

(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

\[
\begin{align*}
\text{(define (map procedure s)} &\\\\\\\\\\\\\\text{)(if (null? s)} &\\\\\\\\\\\\\\\text{nil)} &\\\\\\\\\\\\\\\text{(cons (procedure (car s)) &\\\\\\\\\\\\\\\text{(map procedure (cdr s)))))}} &\\\\\\\\\\\\\\text{)} &\\\\\\\\\\\\\\text{(map (lambda (x) (- 5 x)) (list 1 2))}
\end{align*}
\]

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

(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))
```
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

\[ \text{factorial} = 1 \]

5

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

\[ \text{factorial} \]

1

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

Programs specify the logic of a computational device

\[
5 \rightarrow 1 = 1 \downarrow \rightarrow * \rightarrow 120
\]

\[
\text{factorial} \quad 1 \rightarrow \quad 1 \quad 1 \rightarrow \quad * \rightarrow 120
\]

\[
- \quad \text{factorial} \quad \uparrow 1 \rightarrow\]

\[
\text{factorial} \quad 1 \rightarrow \quad 1 \rightarrow \quad * \rightarrow 120
\]
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

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

5 → Scheme Interpreter → 120
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
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

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

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

Internally, it is just a set of evaluation rules