Announcements

- HW11 due next Wednesday
- Scheme project out
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 \textit{lexical scope} (or \textit{static scope})

\textbf{Lexical scope}: The parent of a frame is the environment in which a procedure was \textit{defined}

\textbf{Dynamic scope}: The parent of a frame is the environment in which a procedure was \textit{called}

\begin{verbatim}
(define f (lambda (x) (+ x y)))
(define g (lambda (x y) (f (+ x x))))
\end{verbatim}
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*.

```scheme
(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*

```scheme
(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
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 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 lazily

• Referential transparency: The value of an expression does not change when we substitute one of its sub-expression with the value of that sub-expression
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 lazily

• Referential transparency: The value of an expression does not change when we substitute one of its sub-expression with the value of that sub-expression

But... Can we make basic loops efficient?
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 lazily

- Referential transparency: The value of an expression does not change when we substitute one of its sub-expression with the value of that sub-expression

But... Can we make basic loops efficient?

Yes!
Iteration Versus Recursion in Python

In Python, recursive calls always create new active frames.

<table>
<thead>
<tr>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
</table>

Iteration Versus Recursion in Python

In Python, recursive calls always create new active frames.

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

<table>
<thead>
<tr>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
In Python, recursive calls always create new active frames

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

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

In Python, recursive calls always create new active frames

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

def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)

def factorial(n):
    total = 1
    while n > 0:
        n, total = n - 1, total * n
    return total
Iteration Versus Recursion in Python

In Python, recursive calls always create new active frames.

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

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

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

In Python, recursive calls always create new active frames.

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

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

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

In Python, recursive calls always create new active frames.

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

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

<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></td>
<td>$\Theta(n)$</td>
<td>$\Theta(1)$</td>
</tr>
</tbody>
</table>
Iteration and Recursion
Iteration and Recursion

Reminder: Iteration is a special case of recursion
Reminder: Iteration is a special case of recursion

Idea: The state of iteration can be passed as parameters
Reminder: Iteration is a special case of recursion

Idea: The state of iteration can be passed as parameters

```python
def factorial(n):
    total = 1
    while n > 0:
        n, total = n - 1, total * n
    return total
```
Reminder: Iteration is a special case of recursion
Idea: The state of iteration can be passed as parameters

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

Local names become...
Reminder: Iteration is a special case of recursion

Idea: The state of iteration can be passed as parameters

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

def factorial(n, total):
    if n == 0:
        return total
    return factorial(n - 1, total * n)
Reminder: Iteration is a special case of recursion

Idea: The state of iteration can be passed as parameters

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

def factorial(n, total):
    if n == 0:
        return total
    return factorial(n - 1, total * n)
```
Reminder: Iteration is a special case of recursion

Idea: The state of iteration can be passed as parameters

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

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

But this converted version still uses linear space in Python
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."
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."

```
(define (factorial n total)
  (if (= n 0) total
    (factorial (- n 1) (* total n))))
```
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 total)
  (if (= n 0) total
    (factorial (- n 1) (* total n))))
```

```
def factorial(n, total):
    if n == 0:
        return total
    return factorial(n - 1, total * 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.
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.

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.

A tail call is a call expression in a *tail context*:

- The last body sub-expression in a *lambda* expression
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.

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.

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`
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.

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**
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.

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**
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.

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 total)
  (if (= n 0) total
    (factorial (- n 1) (* total n)) ))
```
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.

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 total)
   (if (= n 0) total
      (factorial (- n 1) (* total n)) ))
```
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.

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

(define (length s)
    (if (null? s) 0
        (+ 1 (length (cdr s)))))
Example: Length of a List

(define (length s)
  (if (null? s) 0
      (+ 1 (length (cdr s)))))
Example: Length of a List

```
(define (length s)
  (if (null? s) 0
   (+ 1 (length (cdr s)))))
```
Example: Length of a List

\[
\text{(define (length s)}
\]

\[
\quad (\text{if (null? s) 0}
\]

\[
\quad \quad (+ 1 (\text{length (cdr s)}))
\]

\[
\text{)}
\]
Example: Length of a List

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

A call expression is not a tail call if more computation is still required in the calling procedure.
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 recursions can often be rewritten to use tail calls.
Example: Length of a List

```scheme
(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 recursions can often be rewritten to use tail calls.

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

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

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

Linear recursions can often be rewritten 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 recursions can often be rewritten to use tail calls.

```
(define (length-tail s)
  (define (length-iter s n)
    (if (null? s) n
```
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 recursions can often be rewritten to use tail calls.

\[
\text{(define (length-tail s)}
\]
\[
\text{(define (length-iter s n)}
\]
\[
\text{(if (null? s) n}
\]
\[
\text{(length-iter (cdr s) (+ 1 n)) ) )}
\]
Example: Length of a List

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

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

Linear recursions can often be rewritten to use tail calls.

\[
\text{(define (length-tail s)}\\
\quad \text{(define (length-iter s n) }\\
\quad \quad (\text{if (null? s) n}\\
\quad \quad \quad \quad \text{(length-iter (cdr s) (+ 1 n)) }}) ) )\\
\quad \text{(length-iter s 0) )}
\]
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 recursions can often be rewritten to use tail calls.

```
(define (length-tail s)
  (define (length-iter s n)
    (if (null? s) n
        (length-iter (cdr s) (+ 1 n)) ))
  (length-iter s 0))
```
Example: Length 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 recursions can often be rewritten to use tail calls.

```
(define (length-tail s)
  (define (length-iter s n)
    (if (null? s) n
        (length-iter (cdr s) (+ 1 n)) ))
  (length-iter s 0))
```
A call expression is not a tail call if more computation is still required in the calling procedure.

Linear recursions can often be rewritten to use tail calls.

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

```scheme
(define (length-tail s)
  (define (length-iter s n)
    (if (null? s) n
      (length-iter (cdr s) (+ 1 n)) )
  )
  (length-iter s 0) )
)```
Example: Length of a List

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

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

Linear recursions can often be rewritten to use tail calls.

\[
\text{(define (length-tail s)}
\]
\[
\text{\hspace{1cm} (define (length-iter s n)}
\]
\[
\text{\hspace{1cm} (if (null? s) n}
\]
\[
\text{\hspace{1cm} (length-iter (cdr s) (+ 1 n))) ) )}
\]
\[
\text{(length-iter s 0)) )}
\]
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.

In the interpreter, recursive calls to `scheme_eval` for tail calls must instead be expressed iteratively.
Logical Special Forms, Revisited
Logical forms may only evaluate some sub-expressions.

- **If** expression: \(\text{if} \ <\text{predicate}> \ <\text{consequent}> \ <\text{alternative}>\)
- **And** and **or**: \(\text{and} \ <e_1> \ldots \ <e_n>\), \(\text{or} \ <e_1> \ldots \ <e_n>\)
- **Cond** expr'n: \(\text{cond} \ (<p_1> \ <e_1>) \ldots (<p_n> \ <e_n>) \ (\text{else} \ <e>)\)
Logical Special Forms, Revisited

Logical forms may only evaluate some sub-expressions.

- **If** expression: \((\text{if} \ <\text{predicate}> \ <\text{consequent}> \ <\text{alternative}>))\)
- **And** and **or**: \((\text{and} \ <\text{e}_1> \ldots \ <\text{e}_n>), \quad (\text{or} \ <\text{e}_1> \ldots \ <\text{e}_n>))\)
- **Cond** expr'n: \((\text{cond} \ (<\text{p}_1> \ <\text{e}_1>) \ldots \ (<\text{p}_n> \ <\text{e}_n>) \ (\text{else} \ <\text{e}>) ))\)

The value of an **if** expression is the value of a sub-expression.
Logical Special Forms, Revisited

Logical forms may only evaluate some sub-expressions.

- **If** expression: `(if <predicate> <consequent> <alternative>)`
- **And** and **or**: `(and <e₁> ... <eₙ>), (or <e₁> ... <eₙ>)`
- **Cond** expr'n: `(cond (<p₁> <e₁>) ... (<pₙ> <eₙ>) (else <e>))`

The value of an **if** expression is the value of a sub-expression.

- Evaluate the predicate.
Logical Special Forms, Revisited

Logical forms may only evaluate some sub-expressions.

- **If** expression: \((\text{if } \langle\text{predicate}\rangle \ \langle\text{consequent}\rangle \ \langle\text{alternative}\rangle)\)
- **And** and **or**: \((\text{and } \langle e_1 \rangle \ldots \langle e_n \rangle), \quad \text{(or } \langle e_1 \rangle \ldots \langle e_n \rangle)\)
- **Cond** expr’n: \((\text{cond } (\langle p_1 \rangle \ \langle e_1 \rangle) \ldots (\langle p_n \rangle \ \langle e_n \rangle) \ (\text{else } \langle e \rangle))\)

The value of an **if** expression is the value of a sub-expression.

- Evaluate the predicate.
- Choose a sub-expression: \(\langle\text{consequent}\rangle\) or \(\langle\text{alternative}\rangle\)
Logical forms may only evaluate some sub-expressions.

- **If** expression: \( (\text{if} \ <\text{predicate}> \ <\text{consequent}> \ <\text{alternative}>) \)

- **And** and **or**: \( (\text{and} \ <e_1> \ldots <e_n>), \quad (\text{or} \ <e_1> \ldots <e_n>) \)

- **Cond** expr'n: \( (\text{cond} \ (<p_1> <e_1>) \ldots (<p_n> <e_n>) \ <\text{else} > <e>) ) \)

The value of an **if** expression is the value of a sub-expression.

- Evaluate the predicate.
- Choose a sub-expression: \( <\text{consequent}> \) or \( <\text{alternative}> \)
- Evaluate that sub-expression in place of the whole expression.
Logical Special Forms, Revisited

Logical forms may only evaluate some sub-expressions.

- **If** expression: \( (\text{if} \ <\text{predicate}> \ <\text{consequent}> \ <\text{alternative}> ) \)

- **And** and **or**: \( (\text{and} \ <\text{e}_1> \ldots \ <\text{e}_n> ), \quad (\text{or} \ <\text{e}_1> \ldots \ <\text{e}_n> ) \)

- **Cond** expr’n: \( (\text{cond} \ (<\text{p}_1> \ <\text{e}_1>) \ldots \ (<\text{p}_n> \ <\text{e}_n>) \ (\text{else} \ <\text{e}> ) ) \)

The value of an **if** expression is the value of a sub-expression.

- Evaluate the predicate.

- Choose a sub-expression: \( <\text{consequent}> \) or \( <\text{alternative}> \)

- Evaluate that sub-expression in place of the whole expression.
Logical forms may only evaluate some sub-expressions.

- **If** expression: `(if <predicate> <consequent> <alternative>)`
- **And** and **or**: `(and <e₁> ... <eₙ>)`, `(or <e₁> ... <eₙ>)`
- **Cond** expr'n: `(cond (<p₁> <e₁>) ... (<pₙ> <eₙ>) (else <e>))`

The value of an *if* expression is the value of a sub-expression.

- Evaluate the predicate.
- Choose a sub-expression: `<consequent>` or `<alternative>`
- Evaluate that sub-expression in place of the whole expression.
Logical Special Forms, Revisited

Logical forms may only evaluate some sub-expressions.

- **If** expression: `(if <predicate> <consequent> <alternative>)`
- **And** and **or**: `(and <e₁> ... <eₙ>), (or <e₁> ... <eₙ>)`
- **Cond** expr'n: `(cond (<p₁> <e₁>) ... (<pₙ> <eₙ>) (else <e>))`

The value of an **if** expression is the value of a sub-expression.

- Evaluate the predicate.
- Choose a sub-expression: `<consequent>` or `<alternative>`
- Evaluate that sub-expression in place of the whole expression.

Evaluation of the tail context does not require a recursive call.
Logical Special Forms, Revisited

Logical forms may only evaluate some sub-expressions.

- **If** expression: `(if <predicate> <consequent> <alternative>)`
- **And** and **or**: `(and <e₁> ... <eₙ>), (or <e₁> ... <eₙ>)`
- **Cond** expr'n: `(cond (<p₁> <e₁>) ... (<pₙ> <eₙ>) (else <e>))`

The value of an **if** expression is the value of a sub-expression.

- Evaluate the predicate.
- Choose a sub-expression: `<consequent>` or `<alternative>`
- Evaluate that sub-expression in place of the whole expression.

Evaluation of the tail context does not require a recursive call.

E.g., replace `(if false 1 (+ 2 3))` with `(+ 2 3)` and iterate.
Example: Reduce
Example: Reduce

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

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

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

```scheme
(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

```scheme
(define (reduce procedure s start)
  (if (null? 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

\[
\text{(define (reduce procedure s start)}
\]
\[
\text{\quad (if (null? s) start)}
\]
\[
\text{\quad (reduce procedure}
\]
\[
\text{(reduce \*)}
\]
\[
\text{\quad (reduce (lambda (x y) (cons y x)) '(3 4 5) '(2))}
\]
\[
\quad 120
\]
\[
\quad (5 4 3 2)
\]
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

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

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

120

(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

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

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

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

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

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

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

(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.

Other calls are not; constant space depends on `procedure`.

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

(define (map procedure s))
Example: Map

(define (map procedure s)
  (define (map-iter procedure s m))
Example: Map

(define (map procedure s)
  (define (map-iter procedure s m)
    (if (null? s) m

Example: Map

(define (map procedure s)
  (define (map-iter procedure s m)
    (if (null? s) m
      (map-iter procedure

Example: Map

(define (map procedure s)
  (define (map-iter procedure s m)
    (if (null? s) m
      (map-iter procedure (cdr s))
    )
  )
)
Example: Map

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

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

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

(define (reverse s)
  (define (reverse-iter s r)
```
Example: Map

```
(define (map procedure s)
  (define (map-iter procedure s m)
    (if (null? s) m
     (map-iter procedure
       (map-iter procedure (car s)) m)))
    (reverse (map-iter procedure s nil)))

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

Example: Map

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

(define (reverse s)
  (define (reverse-iter s r)
    (if (null? s) r
      (reverse-iter (cdr s)
Example: Map

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

(define (reverse s)
  (define (reverse-iter s r)
    (if (null? s) r
        (reverse-iter (cdr s)
          (reverse-iter (car s) r)))))
```
Example: Map

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

\[ \text{factorial} = \text{factorial} - 1 \]

\[ \text{factorial} < 1 \]
Programs specify the logic of a computational device.

$$5 \rightarrow \text{factorial} \rightarrow 1 \rightarrow \text{switch} \rightarrow 1 \leftarrow \star \leftarrow \text{factorial} \leftarrow \uparrow 1 \uparrow$$
An Analogy: Programs Define Machines

Programs specify the logic of a computational device.
Interpreters are General Computing Machines
Interpreters are General Computing Machines

An interpreter can be parameterized to simulate any machine
An interpreter can be parameterized to simulate any machine

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

(define (factorial n)
  (if (zero? n) 1 (* n (factorial (- n 1)))))
Interpreters are General Computing Machines

An interpreter can be parameterized to simulate any machine

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

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

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
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.

Internally, it is just a set of manipulation rules.
Interpretation in Python
**eval**: Evaluates an expression in the current environment and returns the result. Doing so may affect the environment.
**eval**: Evaluates an expression in the current environment and returns the result. Doing so may affect the environment.

**exec**: Executes a statement in the current environment. Doing so may affect the environment.
Interpretation in Python

**eval**: Evaluates an expression in the current environment and returns the result. Doing so may affect the environment.

**exec**: Executes a statement in the current environment. Doing so may affect the environment.

```python
eval('2 + 2')
```
**Interpretation in Python**

**eval**: Evaluates an expression in the current environment and returns the result. Doing so may affect the environment.

**exec**: Executes a statement in the current environment. Doing so may affect the environment.

```
eval('2 + 2')
```

```
exec('def square(x): return x * x')
```
**Interpretation in Python**

**eval**: Evaluates an expression in the current environment and returns the result. Doing so may affect the environment.

**exec**: Executes a statement in the current environment. Doing so may affect the environment.

```
eval('2 + 2')
```

```
exec('def square(x): return x * x')
```

**os.system('python <file>')**: Directs the operating system to invoke a new instance of the Python interpreter.