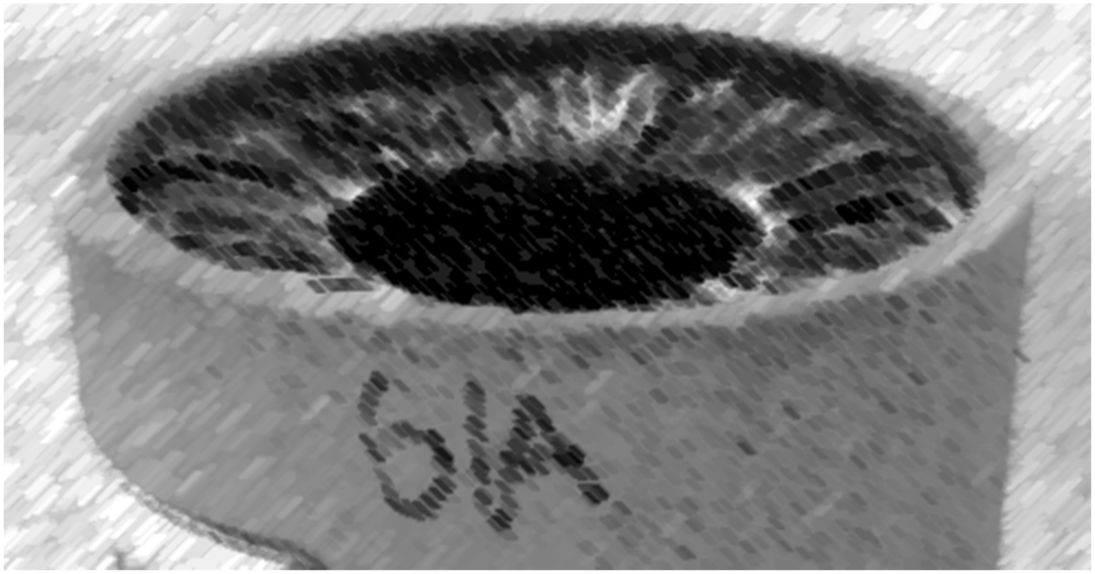


CS61A Lecture 35

Amir Kamil

UC Berkeley

April 12, 2013

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 *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*

Special form to create dynamically scoped procedures

mu

```
(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

Time	Space

Iteration Versus Recursion in Python

In Python, recursive calls always create new active frames

Time	Space

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

Iteration Versus Recursion in Python

In Python, recursive calls always create new active frames

	Time	Space

```
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

	Time	Space
Recursive factorial	$\Theta(n)$	
Iterative factorial		

```
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

	Time	Space
`def factorial(n): if n == 0: return 1; return n * factorial(n - 1)`	$\Theta(n)$	$\Theta(n)$

```
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

	Time	Space
`def factorial(n): if n == 0: return 1; return n * factorial(n - 1)`	$\Theta(n)$	$\Theta(n)$
`def factorial(n): total = 1; while n > 0: n, total = n - 1, total * n; return total`	$\Theta(n)$	

```
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
```

	Time	Space
Recursive	$\Theta(n)$	$\Theta(n)$
Iterative	$\Theta(n)$	$\Theta(1)$

Iteration and Recursion

Iteration and Recursion

Reminder: Iteration is a special case of recursion

Iteration and Recursion

Reminder: Iteration is a special case of recursion

Idea: The state of iteration can be passed as parameters

Iteration and Recursion

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
```

Iteration and Recursion

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
```

Local names become...

Iteration and Recursion

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
```

Local names become...

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

Iteration and Recursion

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
```

Local names become...

Parameters in a recursive function

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

Iteration and Recursion

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
```

Local names become...

Parameters in a recursive function

```
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 *Revised⁷ 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 *Revised⁷ 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 *Revised⁷ 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

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.

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

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

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**

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**

```
(define (factorial n total)
  (if (= n 0) total
      (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 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)) ) )
```

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**

```
(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

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

Not a tail context

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.

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 recursions can often be rewritten to use tail calls.

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 recursions can often be rewritten to use tail calls.

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

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 recursions can often be rewritten to use tail calls.

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

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 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

```
(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 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)) ) )
```

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 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)) ) ) )
```

Not a tail context

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)) ) ) )
```

Not a tail context

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)) ) ) )
```

Not a tail context

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)) ) ) )
```

Not a tail context

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) )
```

Recursive call is a tail call

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 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>))`

Logical Special Forms, Revisited

Logical forms may only evaluate some sub-expressions.

- **If** expression: **(if <predicate> <consequent> <alternative>)**
- **And** and **or**: **(and <e_1> ... <e_n>)**, **(or <e_1> ... <e_n>)**
- **Cond** expr'n: **(cond (<p_1> <e_1>) ... (<p_n> <e_n>) (else <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 <e_1> ... <e_n>)**, **(or <e_1> ... <e_n>)**
- **Cond** expr'n: **(cond (<p_1> <e_1>) ... (<p_n> <e_n>) (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: `(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>`

Logical Special Forms, Revisited

Logical forms may only evaluate some sub-expressions.

- **If** expression: **(if <predicate> <consequent> <alternative>)**
- **And** and **or**: **(and <e$_1$> ... <e$_n$>), (or <e$_1$> ... <e$_n$>)**
- **Cond** expr'n: **(cond (<p$_1$> <e$_1$>) ... (<p$_n$> <e$_n$>) (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 <e_1> ... <e_n>)**, **(or <e_1> ... <e_n>)**
- **Cond** expr'n: **(cond (<p_1> <e_1>) ... (<p_n> <e_n>) (else <e>))**

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

do_if_form

- 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 <e_1> ... <e_n>)**, **(or <e_1> ... <e_n>)**
- **Cond** expr'n: **(cond (<p_1> <e_1>) ... (<p_n> <e_n>) (else <e>))**

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

do_if_form

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

scheme_eval

Logical Special Forms, Revisited

Logical forms may only evaluate some sub-expressions.

- **If** expression: **(if <predicate> <consequent> <alternative>)**
- **And** and **or**: **(and <e$_1$> ... <e$_n$>)**, **(or <e$_1$> ... <e$_n$>)**
- **Cond** expr'n: **(cond (<p$_1$> <e$_1$>) ... (<p$_n$> <e$_n$>) (else <e>))**

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

do_if_form

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

scheme_eval

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.

do_if_form

scheme_eval

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

```
(define (reduce procedure s start)
```

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

Example: Reduce

```
(define (reduce procedure s start)
```

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

Example: Reduce

```
(define (reduce procedure s start)
```

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

(reduce (lambda (x y) (cons y x)) '(3 4 5) '(2))
```

Example: Reduce

```
(define (reduce procedure s start)
```

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

(reduce (lambda (x y) (cons y x)) '(3 4 5) '(2))       (5 4 3 2)
```

Example: Reduce

```
(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

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

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

(reduce (lambda (x y) (cons y x)) '(3 4 5) '(2))            (5 4 3 2)
```

Example: Reduce

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

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

(reduce (lambda (x y) (cons y x)) '(3 4 5) '(2))      (5 4 3 2)
```

Example: Reduce

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

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

(reduce (lambda (x y) (cons y x)) '(3 4 5) '(2))   (5 4 3 2)
```

Example: Reduce

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

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

(reduce (lambda (x y) (cons y x)) '(3 4 5) '(2))     (5 4 3 2)
```

Example: Reduce

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

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

(reduce (lambda (x y) (cons y x)) '(3 4 5) '(2))      (5 4 3 2)
```

Example: Reduce

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

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

(reduce (lambda (x y) (cons y x)) '(3 4 5) '(2))         (5 4 3 2)
```

Example: Reduce

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

Recursive call is a tail call.

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

(reduce (lambda (x y) (cons y x)) '(3 4 5) '(2))     (5 4 3 2)
```

Example: Reduce

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

Recursive call is a tail call.

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))))
```

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)))
```

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)
```

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
                  (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
```

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)
```

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))))
```

Example: Map

```scheme
(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

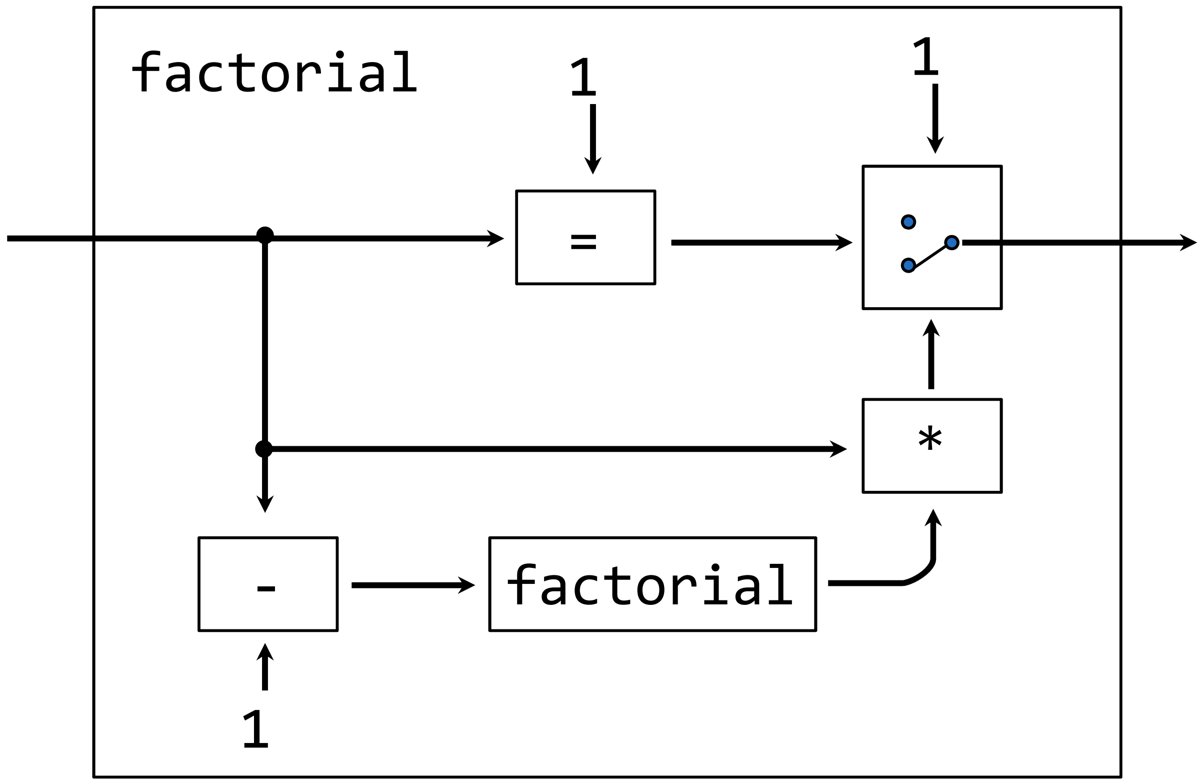
An Analogy: Programs Define Machines

Programs specify the logic of a computational device

```
factorial
```

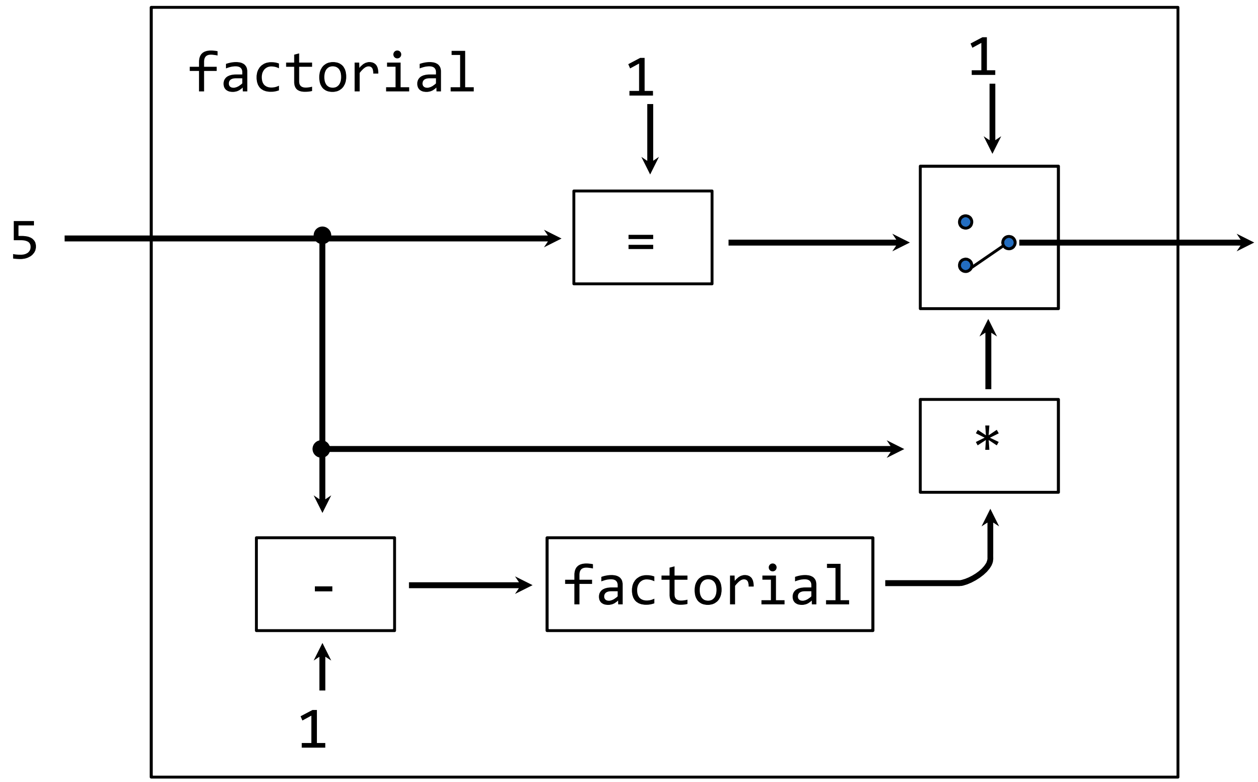
An Analogy: Programs Define Machines

Programs specify the logic of a computational device



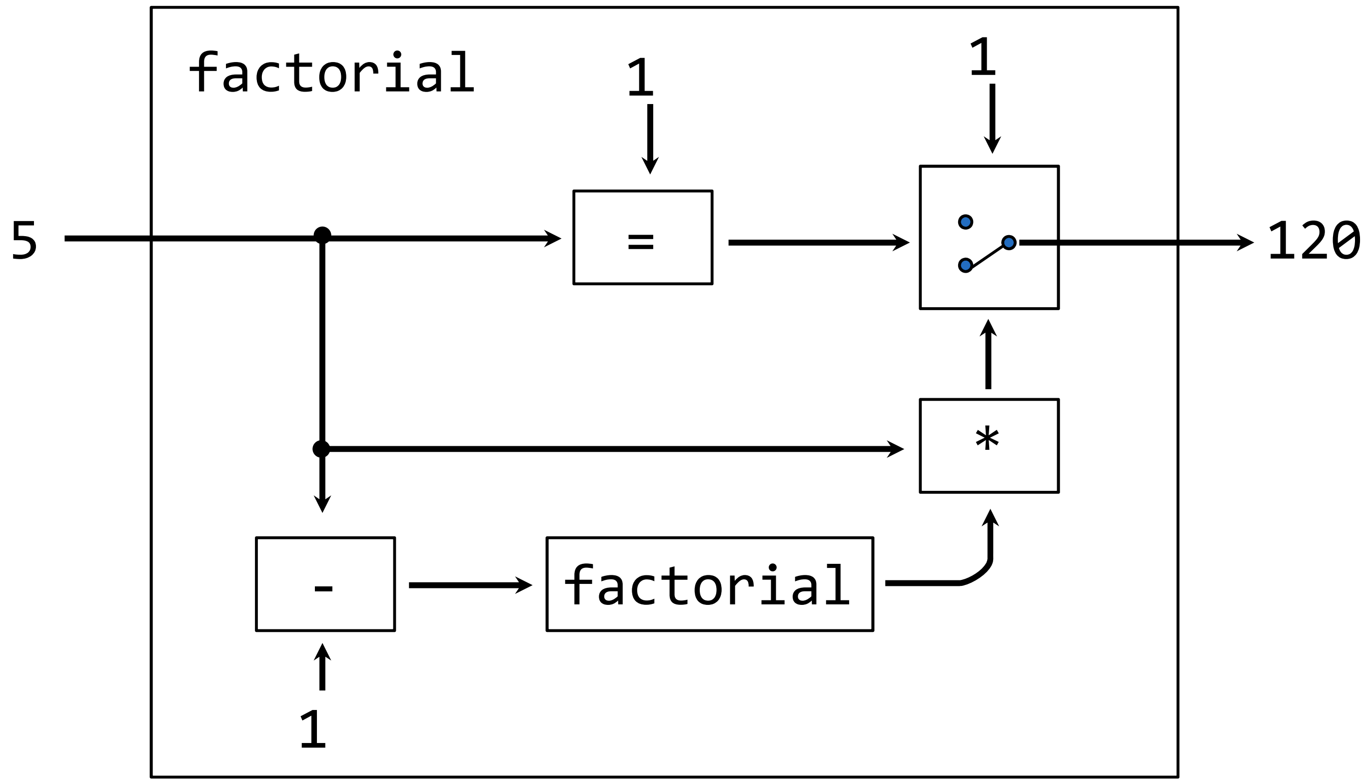
An Analogy: Programs Define Machines

Programs specify the logic of a computational device



An Analogy: Programs Define Machines

Programs specify the logic of a computational device



Interpreters are General Computing Machines

Interpreters are General Computing Machines

An interpreter can be parameterized to simulate any machine

Interpreters are General Computing Machines

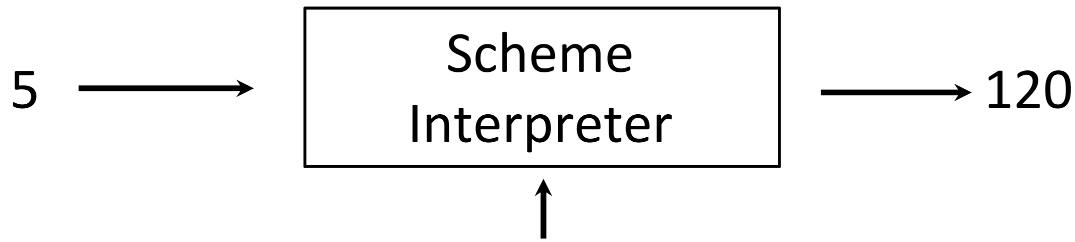
An interpreter can be parameterized to simulate any machine

5 → Scheme Interpreter → 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

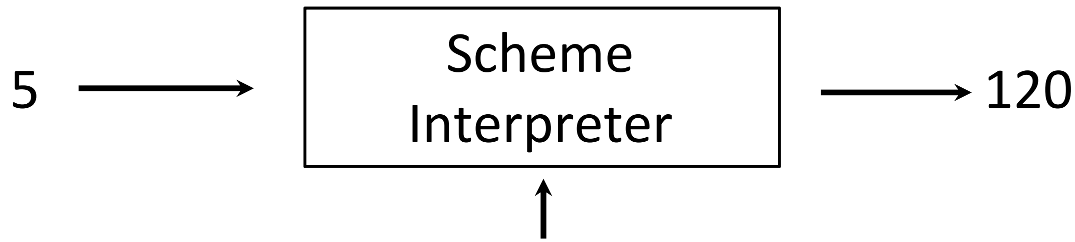



```
(define (factorial n)
  (if (zero? n) 1 (* n (factorial (- n 1)))))
```

Our Scheme interpreter is a universal machine

Interpreters are General Computing Machines

An interpreter can be parameterized to simulate any machine



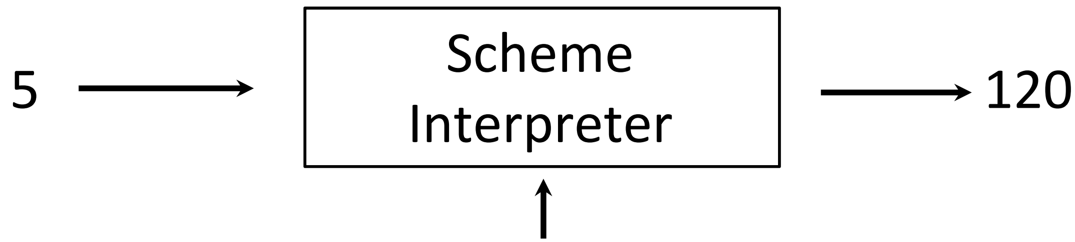

```
(define (factorial n)
  (if (zero? n) 1 (* n (factorial (- n 1)))))
```

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

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

Internally, it is just a set of manipulation rules

Interpretation in Python

Interpretation in Python

`eval`: Evaluates an expression in the current environment and returns the result. 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.

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')
```

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.