61A Lecture 30

Wednesday, November 7

Functional Programming

Functional Programming

All functions are pure functions

Functional Programming

All functions are pure functions

No assignment and no mutable data types (except for re-define)

Functional Programming

All functions are pure functions

No assignment and no mutable data types (except for re-define)

Name-value bindings are permanent

Functional Programming

All functions are pure functions

No assignment and no mutable data types (except for re-define)

Name-value bindings are permanent

Advantages of functional programming:

Functional Programming

All functions are pure functions

No assignment and no mutable data types (except for re-define)

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 assignment and no mutable data types (except for re-define)

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 assignment and no mutable data types (except for re-define)

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 subexpression with the value of that subexpression.

Functional Programming

All functions are pure functions

No assignment and no mutable data types (except for re-define)

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 subexpression with the value of that subexpression.

But... Can we make basic loops efficient?

Functional Programming

All functions are pure functions

No assignment and no mutable data types (except for re-define)

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 subexpression with the value of that subexpression.

But... Can we make basic loops efficient?

Yes!

Iteration Versus Recursion in Python

In Python, recursive calls always create new active frames.

Iteration Versus Recursion in Python

In Python, recursive calls always create new active frames.

```
def exp(b, n):
    if n == 0:
        return 1
    return b * exp(b, n-1)
```

Iteration Versus Recursion in Python

In Python, recursive calls always create new active frames.

```
def exp(b, n):
    if n == 0:
        return 1
    return b * exp(b, n-1)
```

```
def exp(b, n):
    total = 1
    for _ in range(n):
        total = total * b
    return total
```

Iteration Versus Recursion in Python

In Python, recursive calls always create new active frames.

Time	Space

```
def exp(b, n):
    if n == 0:
        return 1
    return b * exp(b, n-1)
```

```
def exp(b, n):
    total = 1
    for _ in range(n):
        total = total * b
    return total
```

Iteration Versus Recursion in Python

In Python, recursive calls always create new active frames.

Time	Space
$\Theta(n)$	

```
def exp(b, n):
    if n == 0:
        return 1
    return b * exp(b, n-1)
```

```
def exp(b, n):
    total = 1
    for _ in range(n):
        total = total * b
    return total
```

Iteration Versus Recursion in Python

In Python, recursive calls always create new active frames.

Time	Space
$\Theta(n)$	$\Theta(n)$

```
def exp(b, n):
    if n == 0:
        return 1
    return b * exp(b, n-1)
```

```
def exp(b, n):
    total = 1
    for _ in range(n):
        total = total * b
    return total
```

Iteration Versus Recursion in Python

In Python, recursive calls always create new active frames.

```
def exp(b, n):
    if n == 0:
        return 1
    return b * exp(b, n-1)
```

```
def exp(b, n):
    total = 1
    for _ in range(n):
        total = total * b
    return total
```

Time	Space
$\Theta(n)$	$\Theta(n)$
$\Theta(n)$	

Iteration Versus Recursion in Python

In Python, recursive calls always create new active frames.

```
def exp(b, n):
    if n == 0:
        return 1
    return b * exp(b, n-1)
```

```
def exp(b, n):
    total = 1
    for _ in range(n):
        total = total * b
    return total
```

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

Tail Recursion

From the Revised^7 Report on the Algorithmic Language Scheme:

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."

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."

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

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."

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

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

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 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 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 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 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 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 expression in a lambda expression
- Sub-expressions 2 & 3 in a tail context **if** expression
- All non-predicate sub-expressions in a tail context **cond**
- The last sub-expression in a tail context **and** or **or**
- The last sub-expression in a tail context **begin**

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

Tail Calls

A procedure call that has not yet returned is *active.* Some procedure calls are *tail calls*. A Scheme interpreter should support an unbounded number of active tail calls.

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

- The last expression in a lambda expression
- Sub-expressions 2 & 3 in a tail context **if** expression
- All non-predicate sub-expressions in a tail context **cond**
- The last sub-expression in a tail context **and** or **or**
- The last sub-expression in a tail context **begin**

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

Tail Calls

A procedure call that has not yet returned is *active.* Some procedure calls are *tail calls*. A Scheme interpreter should support an unbounded number of active tail calls.

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

- The last 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

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

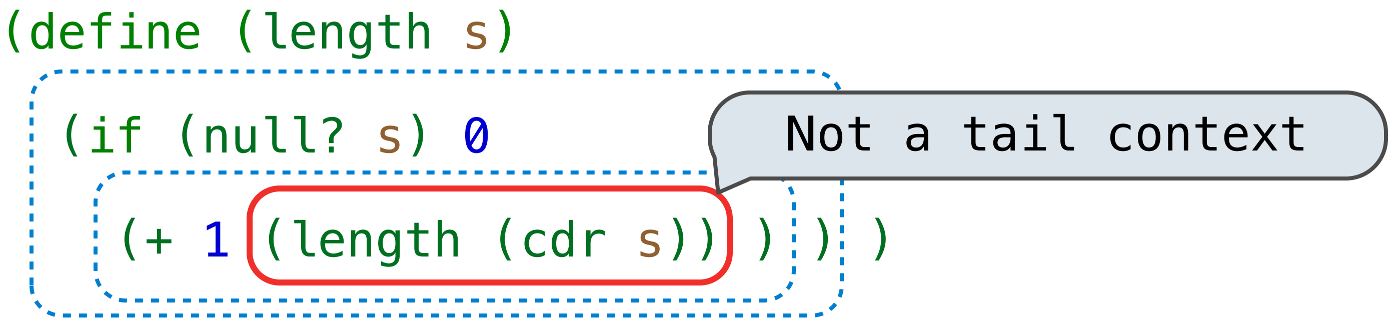
Example: Length of a List

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

Example: Length of a List

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

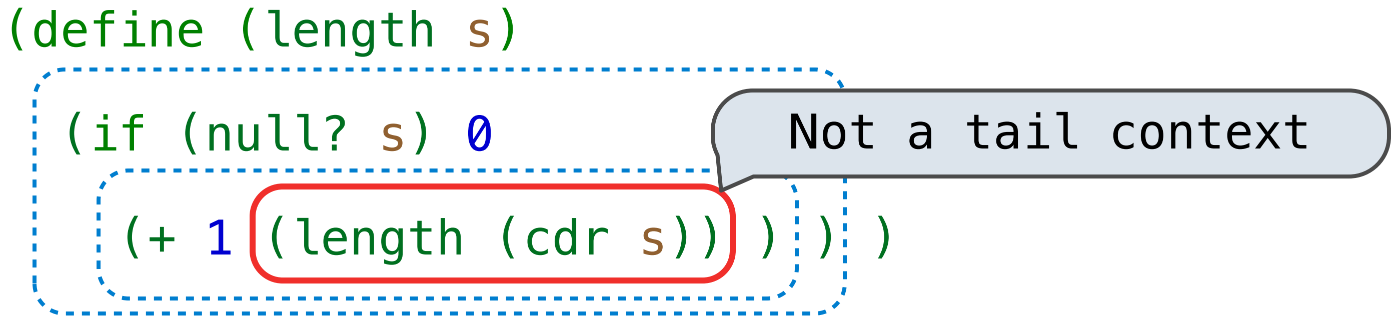
Example: Length of a List



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

Not a tail context

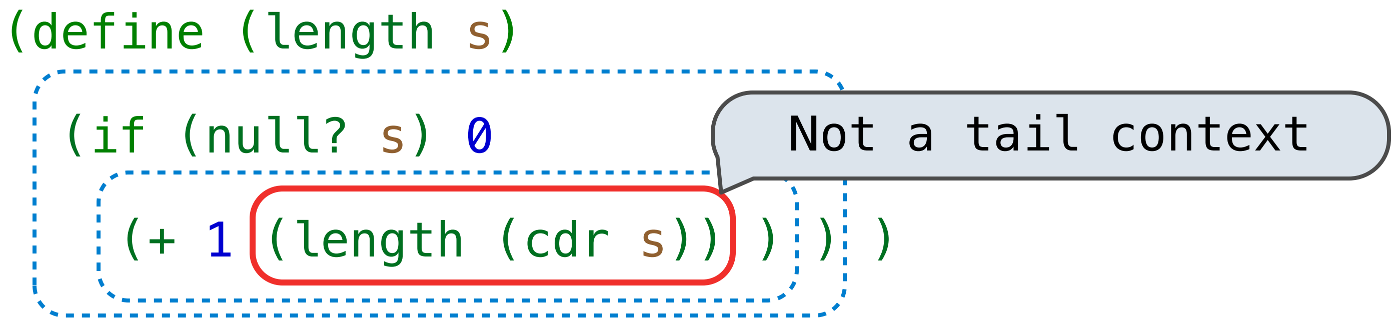
Example: Length of a List



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

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 re-written 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 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)) ) ) )
```

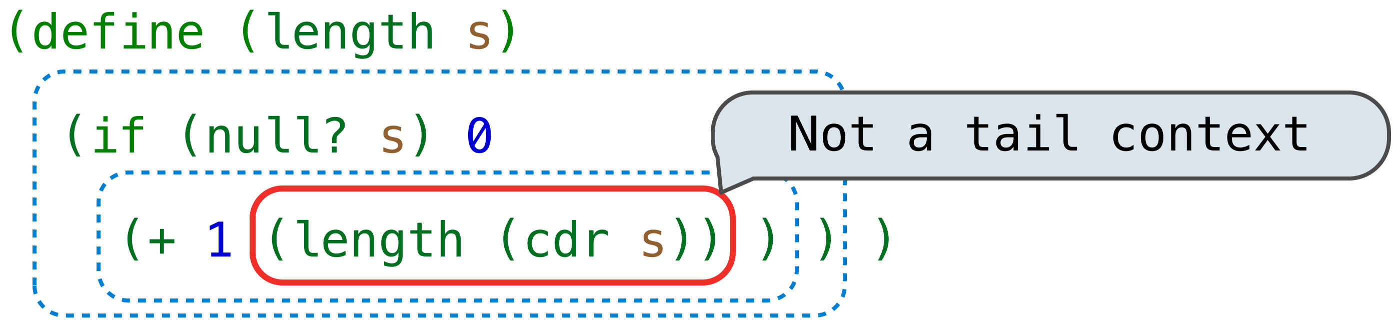
Not a tail context

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

Linear recursions can often be re-written 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)) ) ) )
```

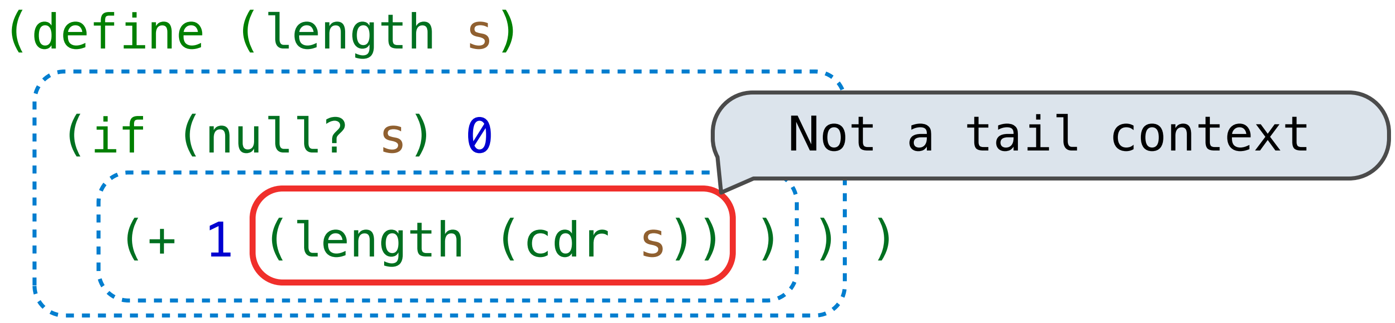
Not a tail context

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

Linear recursions can often be re-written 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)) ) ) )
```

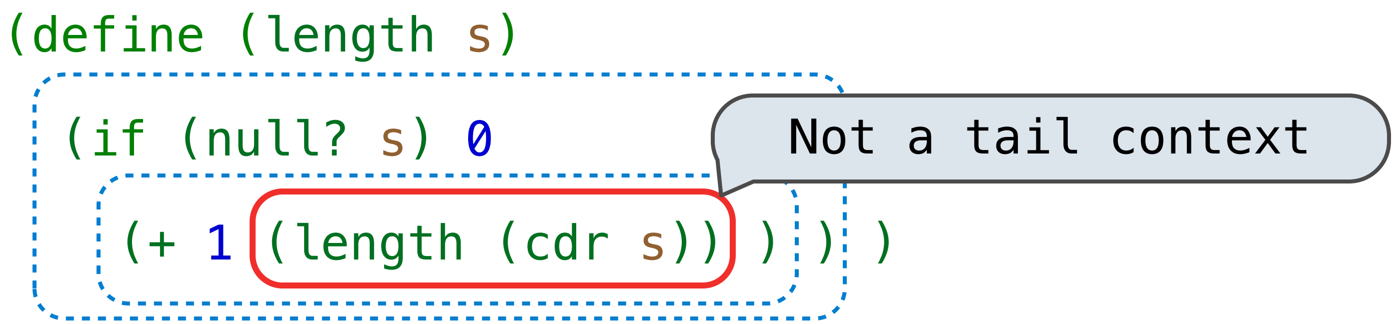
Not a tail context

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

Linear recursions can often be 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)) ) )
```

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 re-written to use tail calls.

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

Example: Length of a List

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

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 re-written to use tail calls.

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

Example: Length of a List

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

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 re-written to use tail calls.

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

Example: Length of a List

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

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 re-written to use tail calls.

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

Example: Length of a List

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

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 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) )
```

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.

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.

Demo

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>))

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 <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>.

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 <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>.

do_if_form

- 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.

- 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

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>.

do_if_form

- 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 <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>.

do_if_form

- Evaluate that sub-expression in place of the whole expression.

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 repeat.

Example: Reduce

Example: Reduce

```
(define (reduce fn s start)
```

Example: Reduce

```
(define (reduce fn s start)
```

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

Example: Reduce

```
(define (reduce fn s start)
```

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

Example: Reduce

```
(define (reduce fn s start)
```

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

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

Example: Reduce

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

```
(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 fn s start)
  (if (null? s) start
    (reduce fn
            (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 fn s start)
  (if (null? s) start
    (reduce fn
            (cdr s)
            (fn 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 fn s start)
  (if (null? s) start
    (reduce fn
            (cdr s)
            (fn 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

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

```scheme
(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 fn s start)
  (if (null? s) start
    (reduce fn
            (cdr s)
            (fn 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 fn s start)
  (if (null? s) start
    (reduce fn
            (cdr s)
            (fn 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 fn s start)
  (if (null? s) start
    (reduce fn
            (cdr s)
            (fn start (car s)) ) ) )
```

Recursive call is a tail call.

Other calls are not; constant space depends on fn.

```
(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 fn s)
```

Example: Map

```
(define (map fn s)
  (define (map-iter fn s m)
```

Example: Map

```
(define (map fn s)
  (define (map-iter fn s m)
    (if (null? s) m
```

Example: Map

```
(define (map fn s)
  (define (map-iter fn s m)
    (if (null? s) m
      (map-iter fn
```

Example: Map

```
(define (map fn s)
  (define (map-iter fn s m)
    (if (null? s) m
      (map-iter fn
                (cdr s)
```

Example: Map

```
(define (map fn s)
  (define (map-iter fn s m)
    (if (null? s) m
      (map-iter fn
                (cdr s)
                (cons (fn (car s)) m))))
```

Example: Map

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

Example: Map

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

Example: Map

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

(define (reverse s)
```

Example: Map

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

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

Example: Map

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

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

Example: Map

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

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

Example: Map

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

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

Example: Map

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

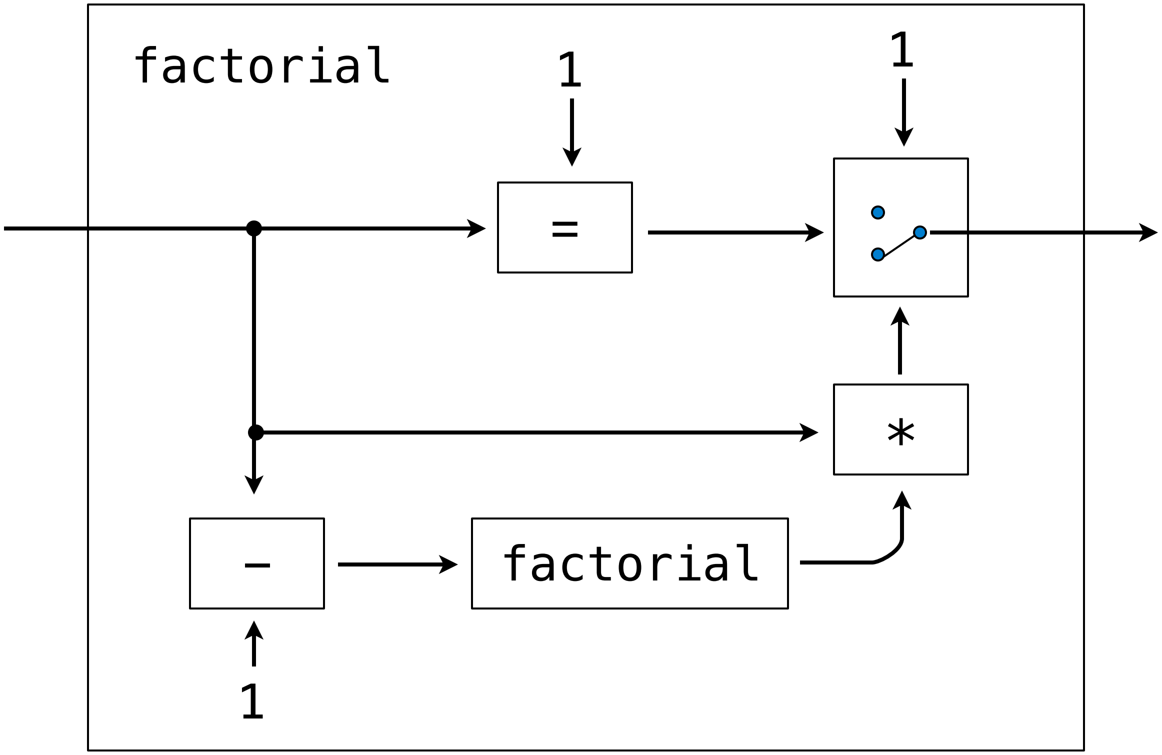
An Analogy: Programs Define Machines

Programs specify the logic of a computational device

factorial

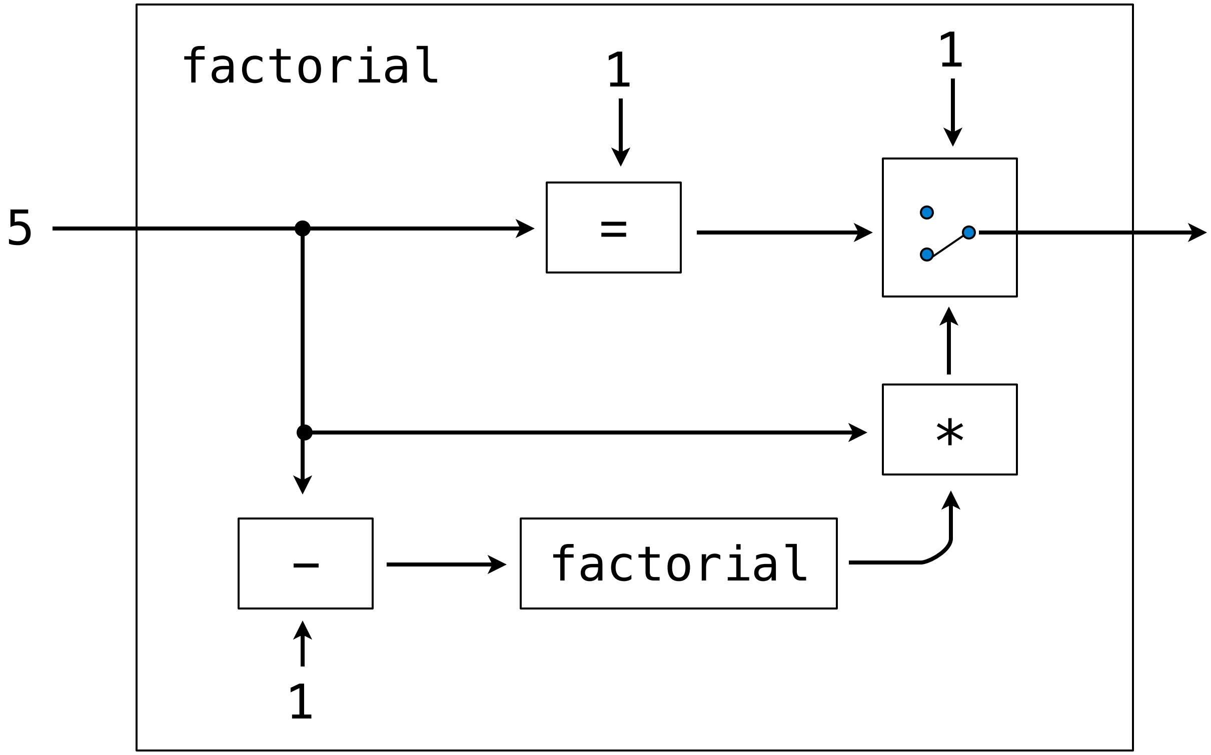
An Analogy: Programs Define Machines

`Programs specify the logic of a computational device`



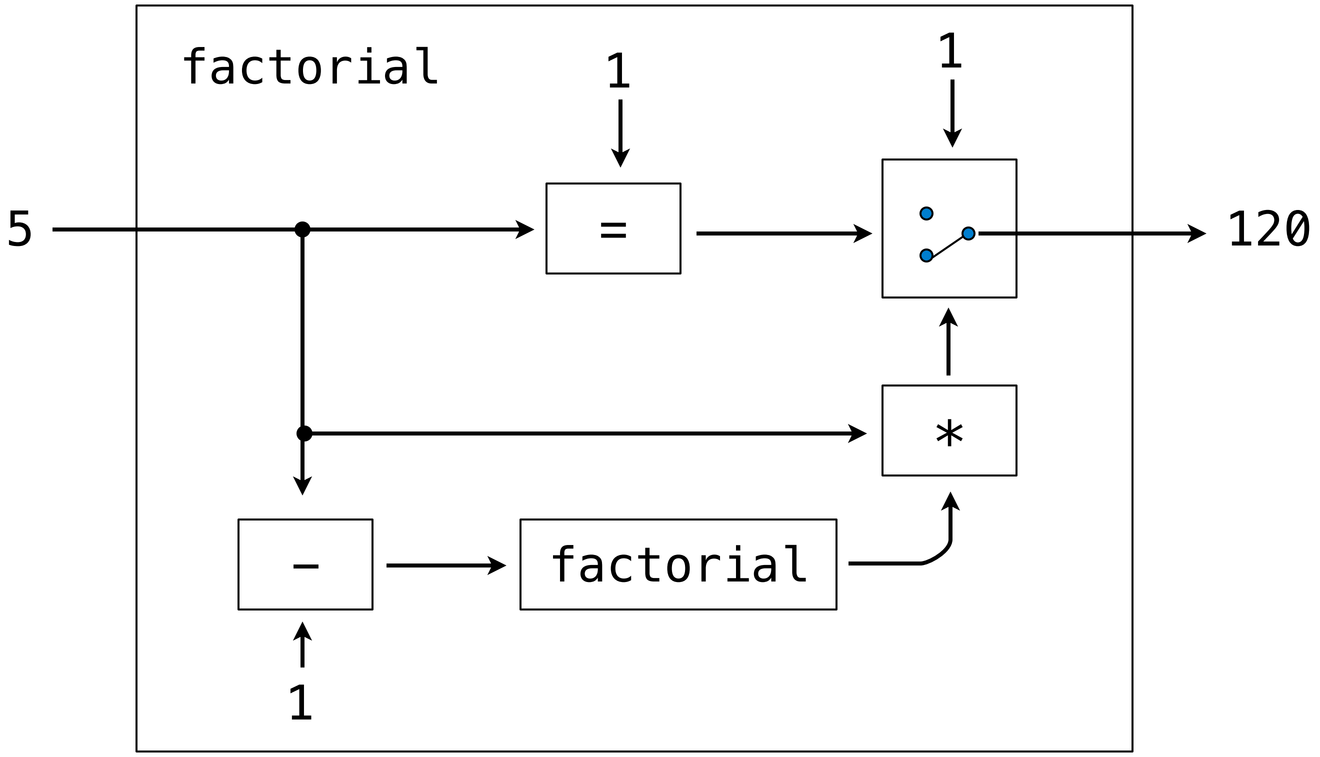
An Analogy: Programs Define Machines

`Programs specify the logic of a computational device`



An Analogy: Programs Define Machines

```
Programs specify the logic of a computational device
```



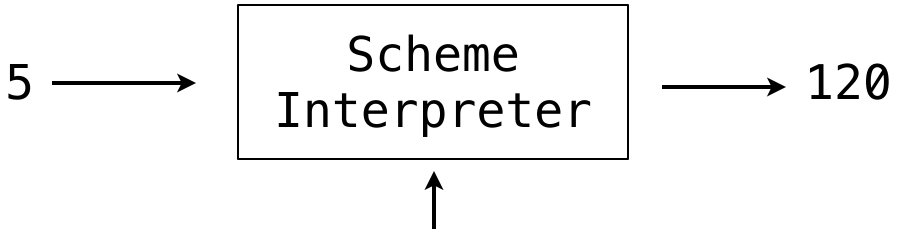
Interpreters are General Computing Machine

Interpreters are General Computing Machine

An interpreter can be parameterized to simulate any machine

Interpreters are General Computing Machine

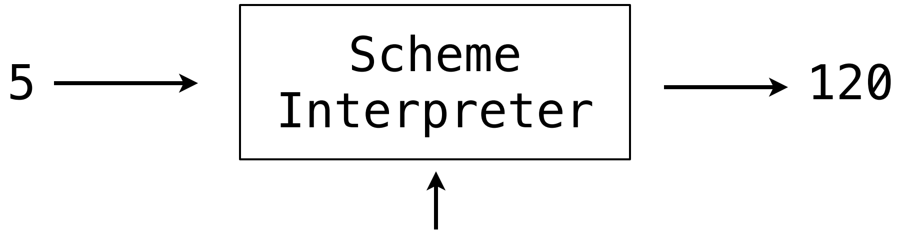
An interpreter can be parameterized to simulate any machine



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

Interpreters are General Computing Machine

An interpreter can be parameterized to simulate any machine



5

Scheme Interpreter

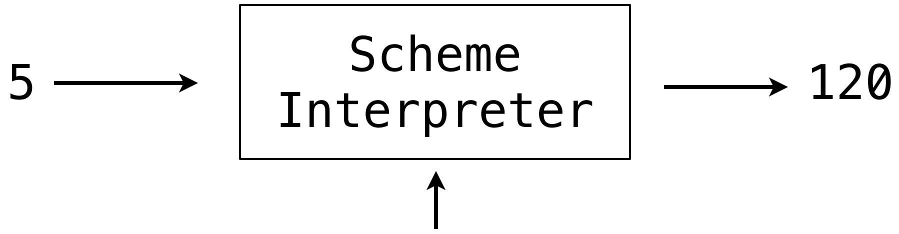
120

```
(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



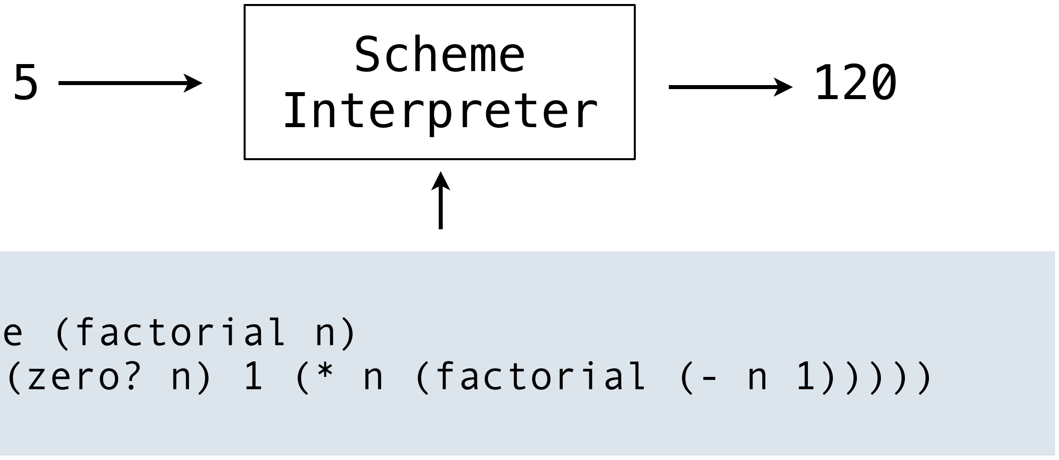
```
(define (factorial n)
  (if (zero? n) 1 (* n (factorial (- n 1)))))
```

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



5

Scheme Interpreter

120

```
(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.

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.

Demo