61A Lecture 26

Wednesday, November 6

Announcements

Announcements

- Project 1 composition revisions due Thursday 11/7 @ 11:59pm.

Announcements

- Project 1 composition revisions due Thursday 11/7 @ 11:59pm.
- Homework 8 due Tuesday 11/12 @ 11:59pm, and it's in Scheme!

Announcements

- Project 1 composition revisions due Thursday 11/7 @ 11:59pm.
- Homework 8 due Tuesday 11/12 @ 11:59pm, and it's in Scheme!
- Project 4 due Thursday 11/21 @ 11:59pm, and it's a Scheme interpreter!

Announcements

- Project 1 composition revisions due Thursday 11/7 @ 11:59pm.
- Homework 8 due Tuesday 11/12 @ 11:59pm, and it's in Scheme!
- Project 4 due Thursday 11/21 @ 11:59pm, and it's a Scheme interpreter!
- **New Policy:** An improved final exam score can make up for low midterm scores.

Announcements

- Project 1 composition revisions due Thursday 11/7 @ 11:59pm.
- Homework 8 due Tuesday 11/12 @ 11:59pm, and it's in Scheme!
- Project 4 due Thursday 11/21 @ 11:59pm, and it's a Scheme interpreter!
- **New Policy:** An improved final exam score can make up for low midterm scores.
  - If you scored less than 60/100 midterm points total, then you can earn some points back.

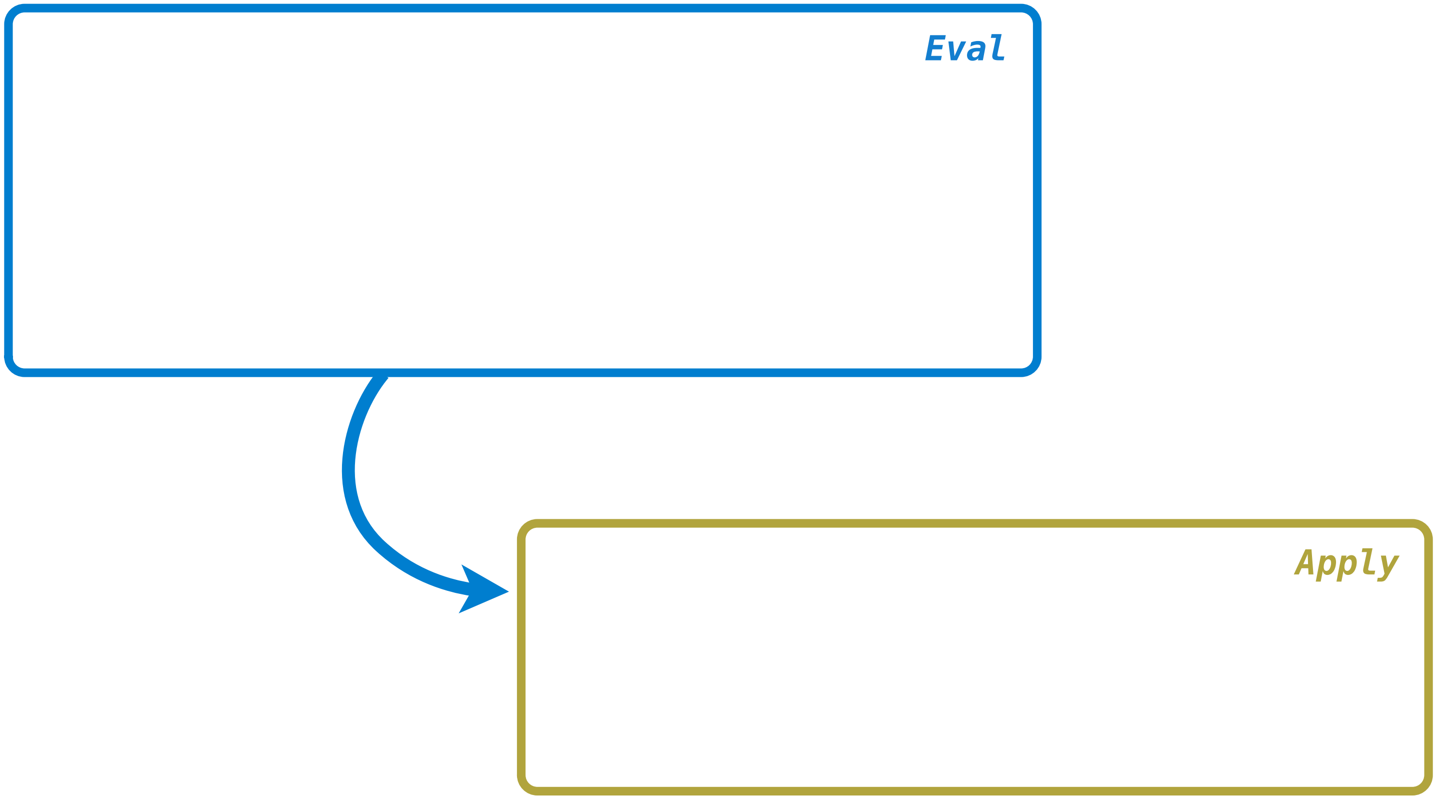
Announcements

- Project 1 composition revisions due Thursday 11/7 @ 11:59pm.
- Homework 8 due Tuesday 11/12 @ 11:59pm, and it's in Scheme!
- Project 4 due Thursday 11/21 @ 11:59pm, and it's a Scheme interpreter!
- **New Policy:** An improved final exam score can make up for low midterm scores.
  - If you scored less than 60/100 midterm points total, then you can earn some points back.
  - You don't need a perfect score on the final to do so.

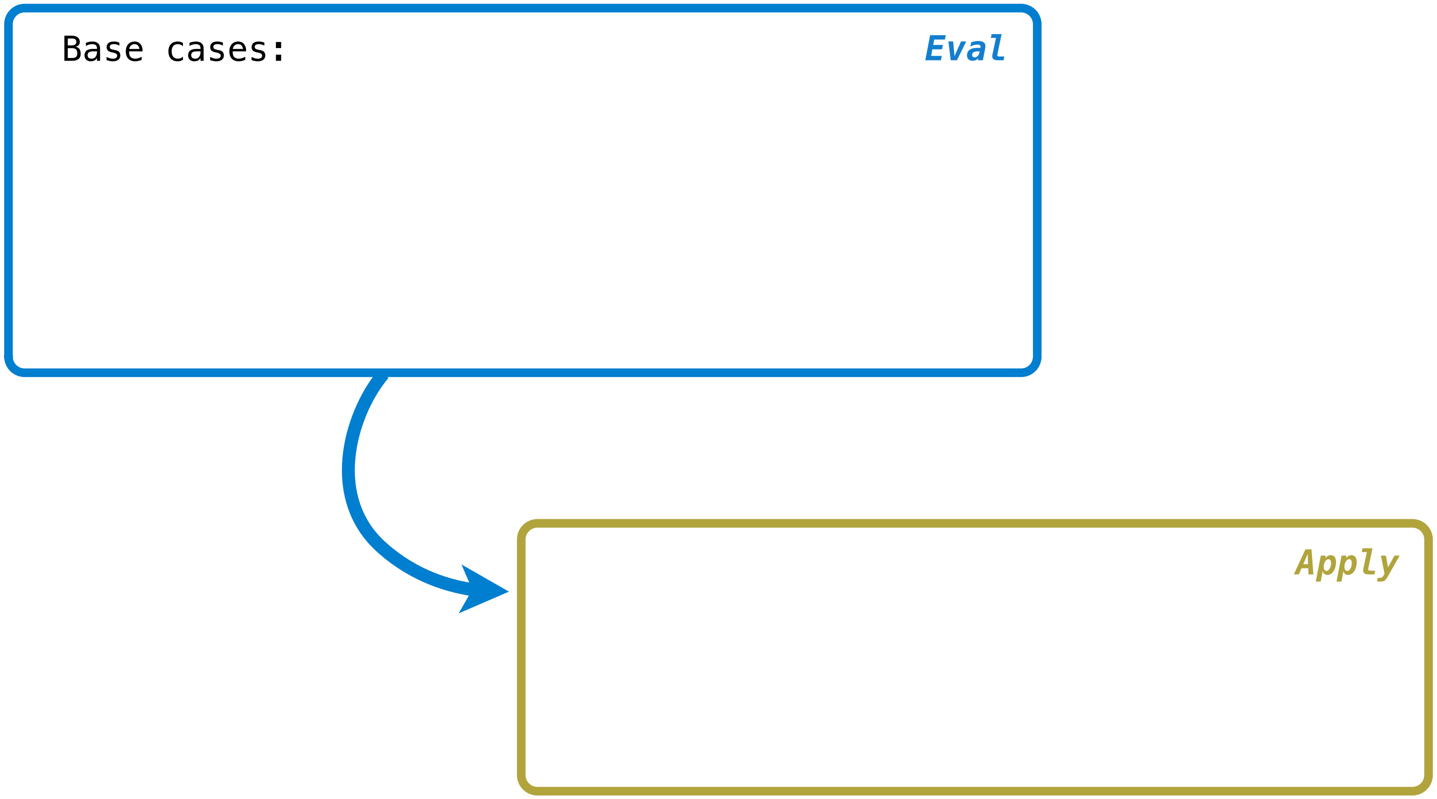
Interpreting Scheme

The Structure of an Interpreter

The Structure of an Interpreter



The Structure of an Interpreter

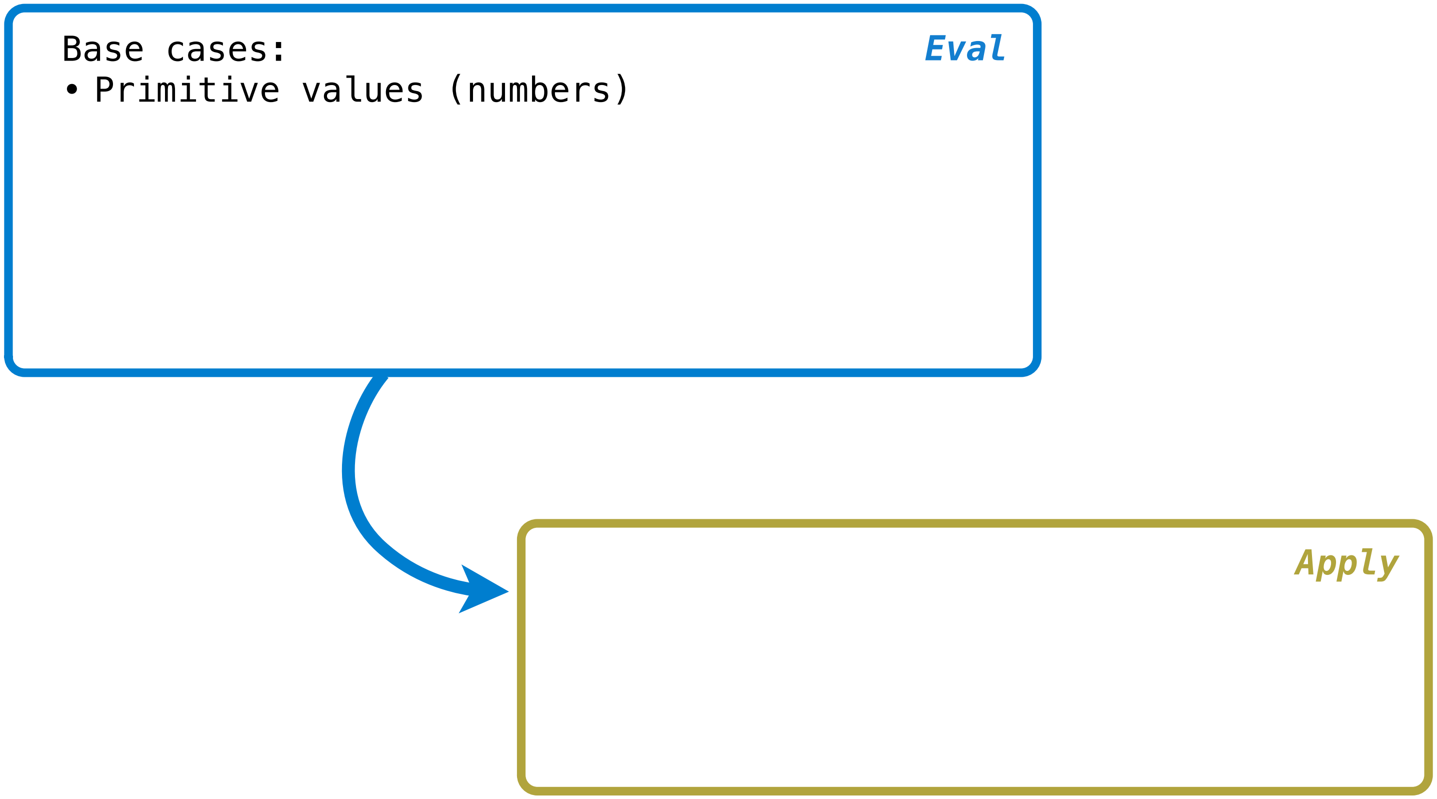


Base cases:

Eval

Apply

The Structure of an Interpreter



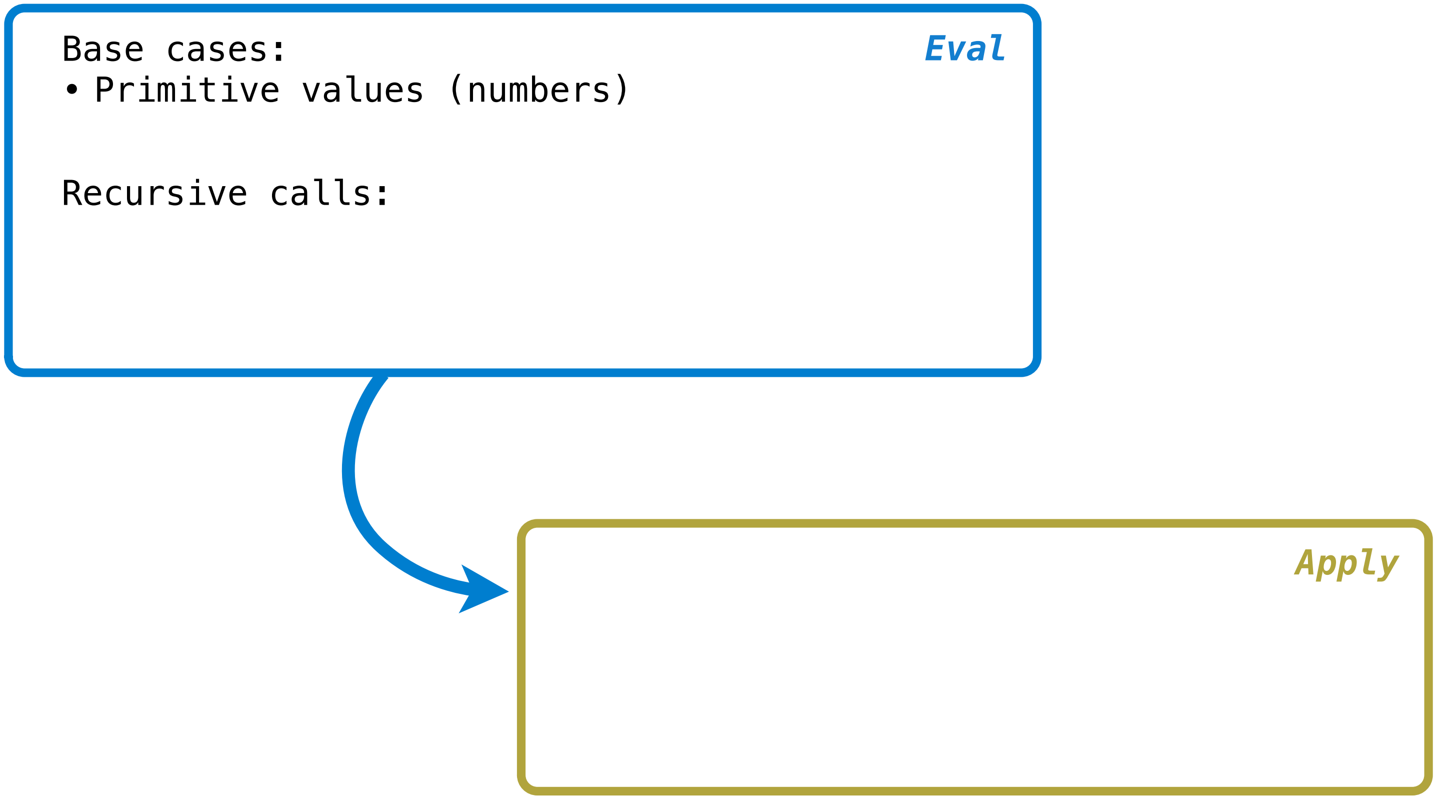
Eval

Base cases:

- Primitive values (numbers)

Apply

The Structure of an Interpreter



Eval

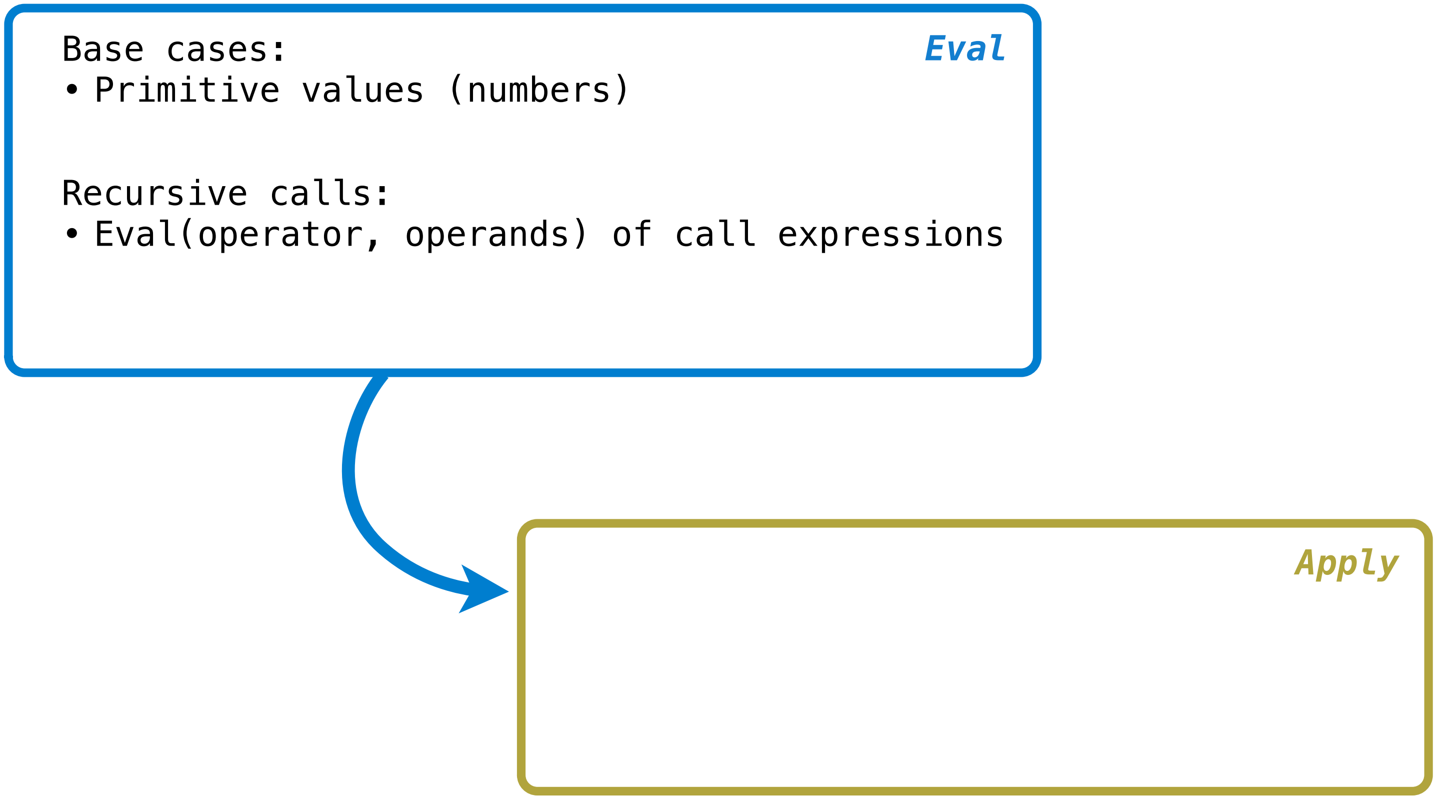
Base cases:

- Primitive values (numbers)

Recursive calls:

Apply

The Structure of an Interpreter



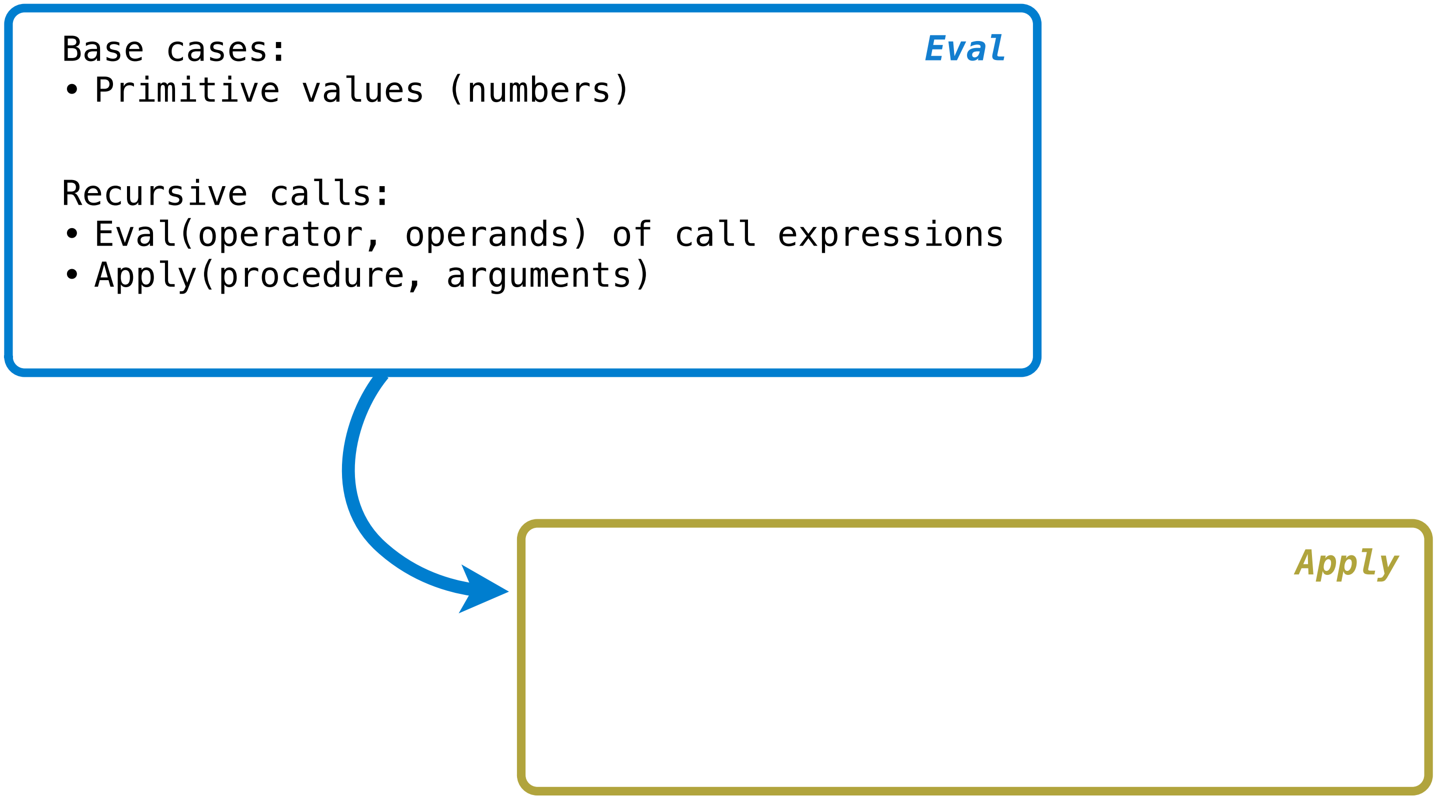
Eval

Base cases:
- Primitive values (numbers)

Recursive calls:
- Eval(operator, operands) of call expressions

Apply

The Structure of an Interpreter



Eval

Base cases:

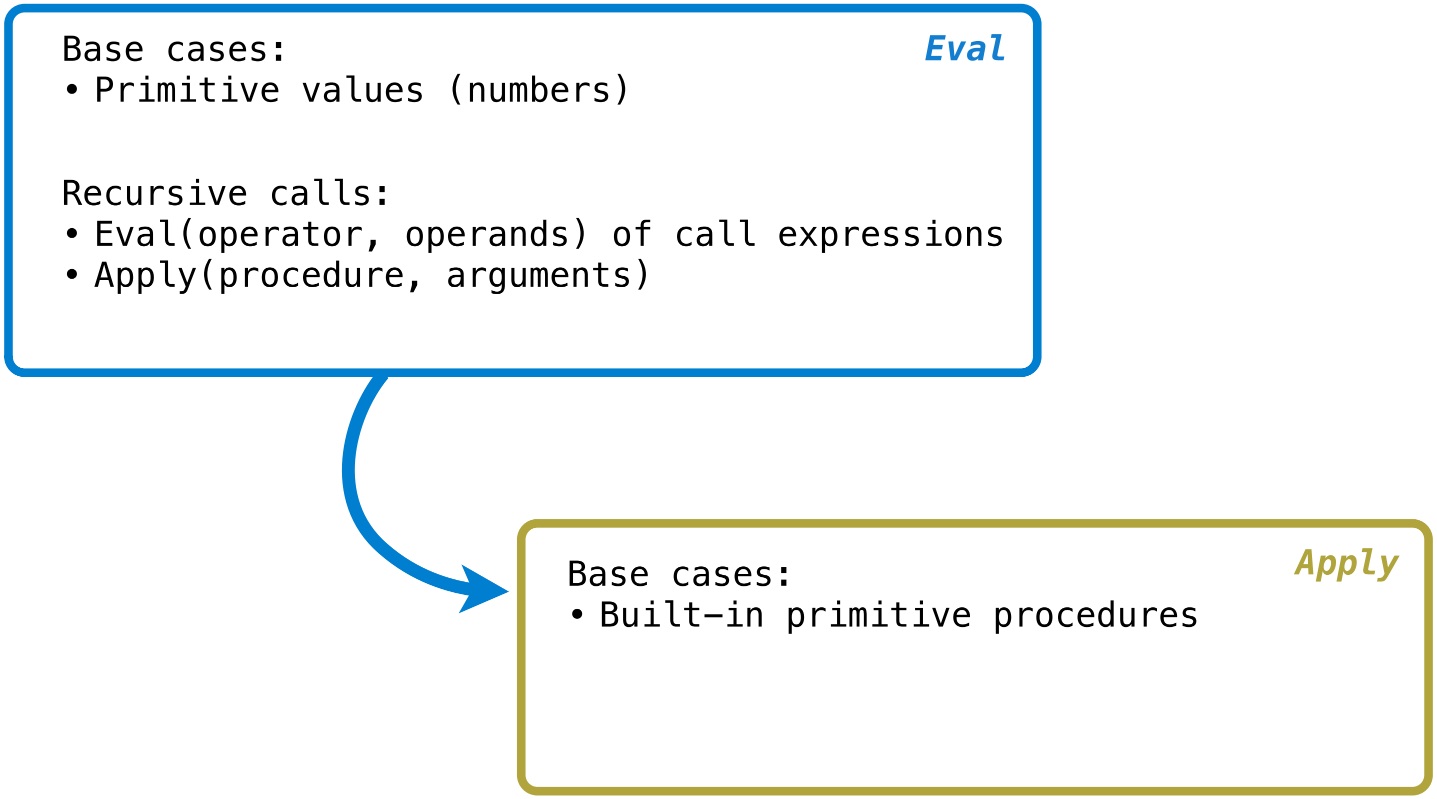
- Primitive values (numbers)

Recursive calls:

- Eval(operator, operands) of call expressions
- Apply(procedure, arguments)

Apply

The Structure of an Interpreter



Eval

Base cases:

- Primitive values (numbers)

Recursive calls:

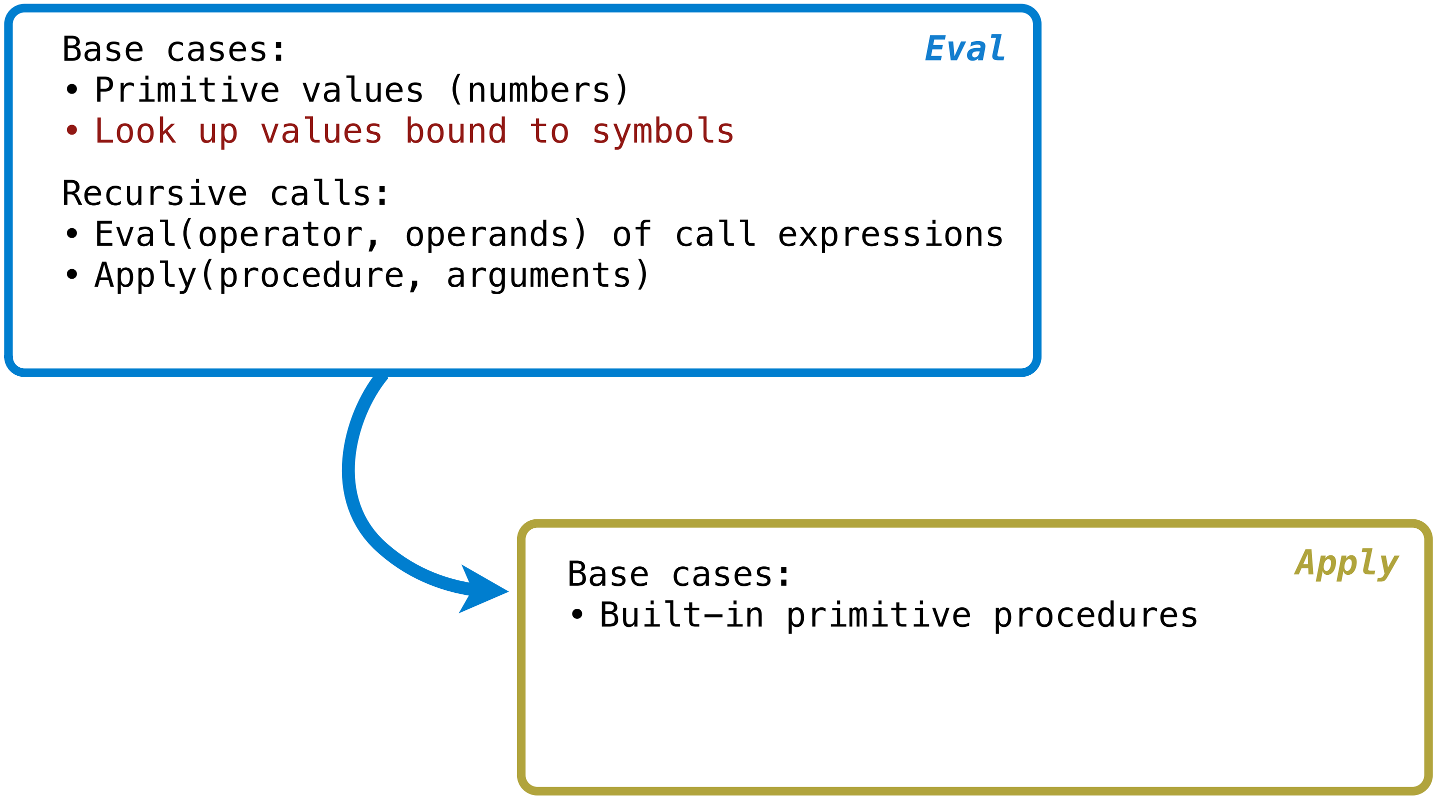
- Eval(operator, operands) of call expressions
- Apply(procedure, arguments)

Apply

Base cases:

- Built-in primitive procedures

The Structure of an Interpreter



Eval

Base cases:

- Primitive values (numbers)
- Look up values bound to symbols

Recursive calls:

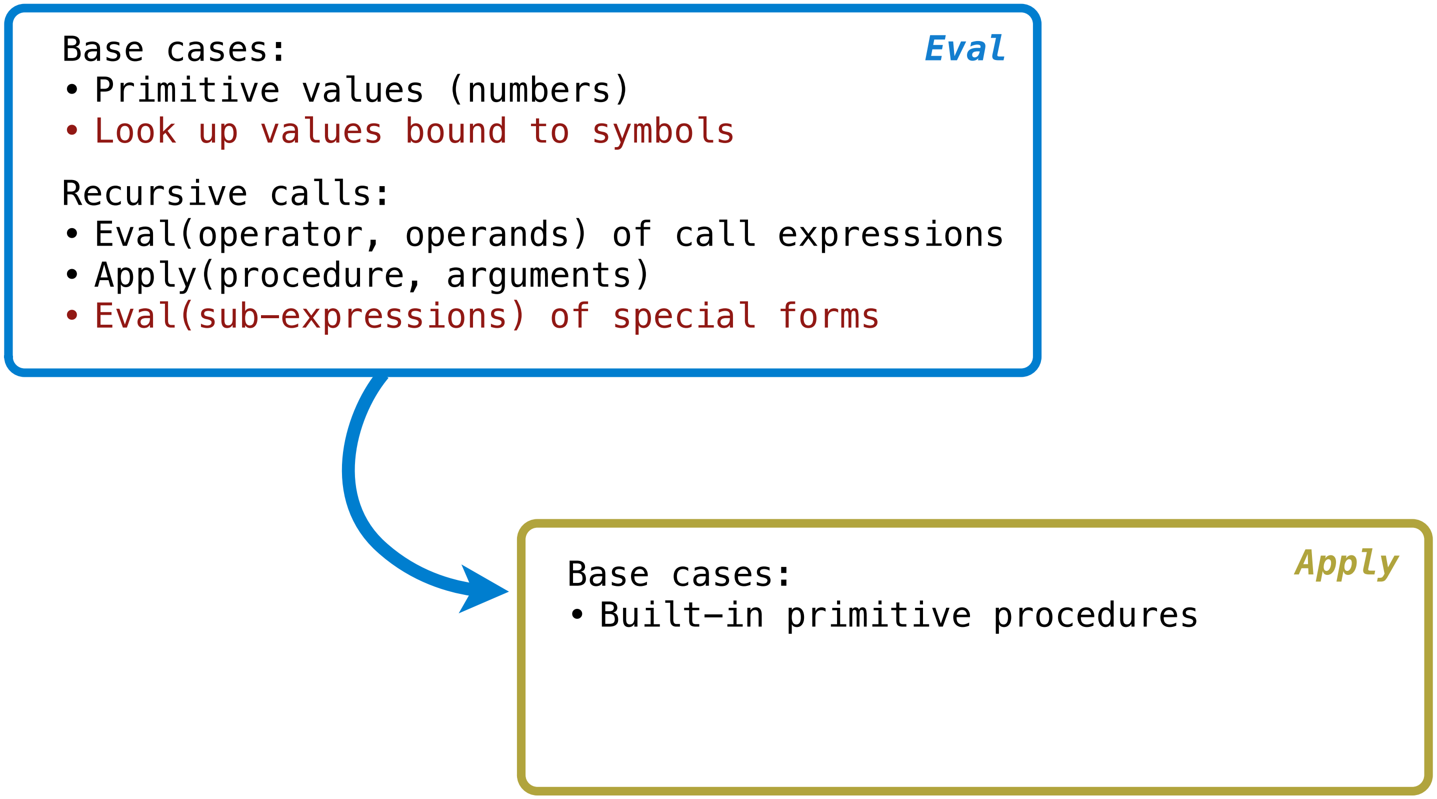
- Eval(operator, operands) of call expressions
- Apply(procedure, arguments)

Apply

Base cases:

- Built-in primitive procedures

The Structure of an Interpreter



Eval

Base cases:

- Primitive values (numbers)
- Look up values bound to symbols

Recursive calls:

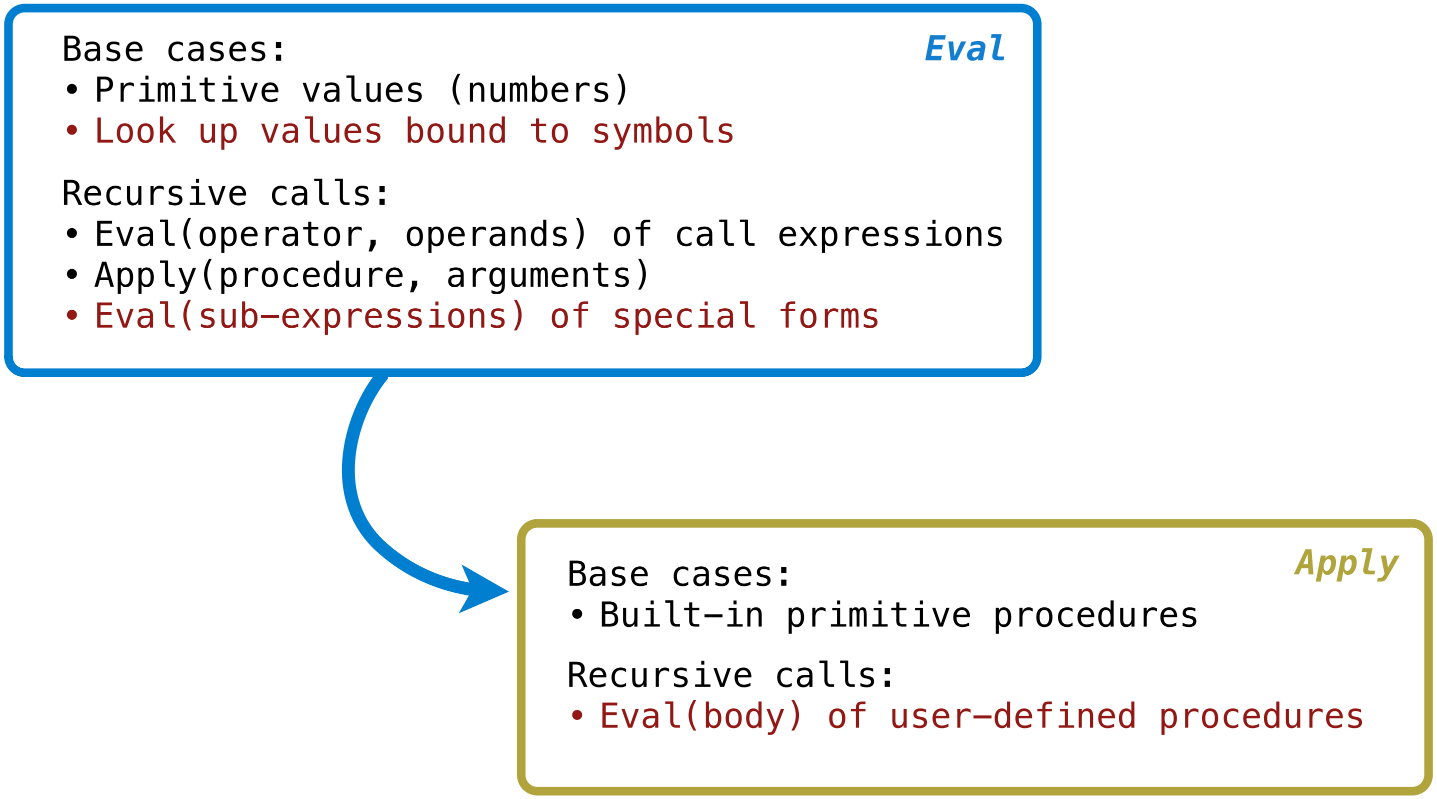
- Eval(operator, operands) of call expressions
- Apply(procedure, arguments)
- Eval(sub-expressions) of special forms

Apply

Base cases:

- Built-in primitive procedures

The Structure of an Interpreter



Eval

Base cases:
- Primitive values (numbers)
- Look up values bound to symbols

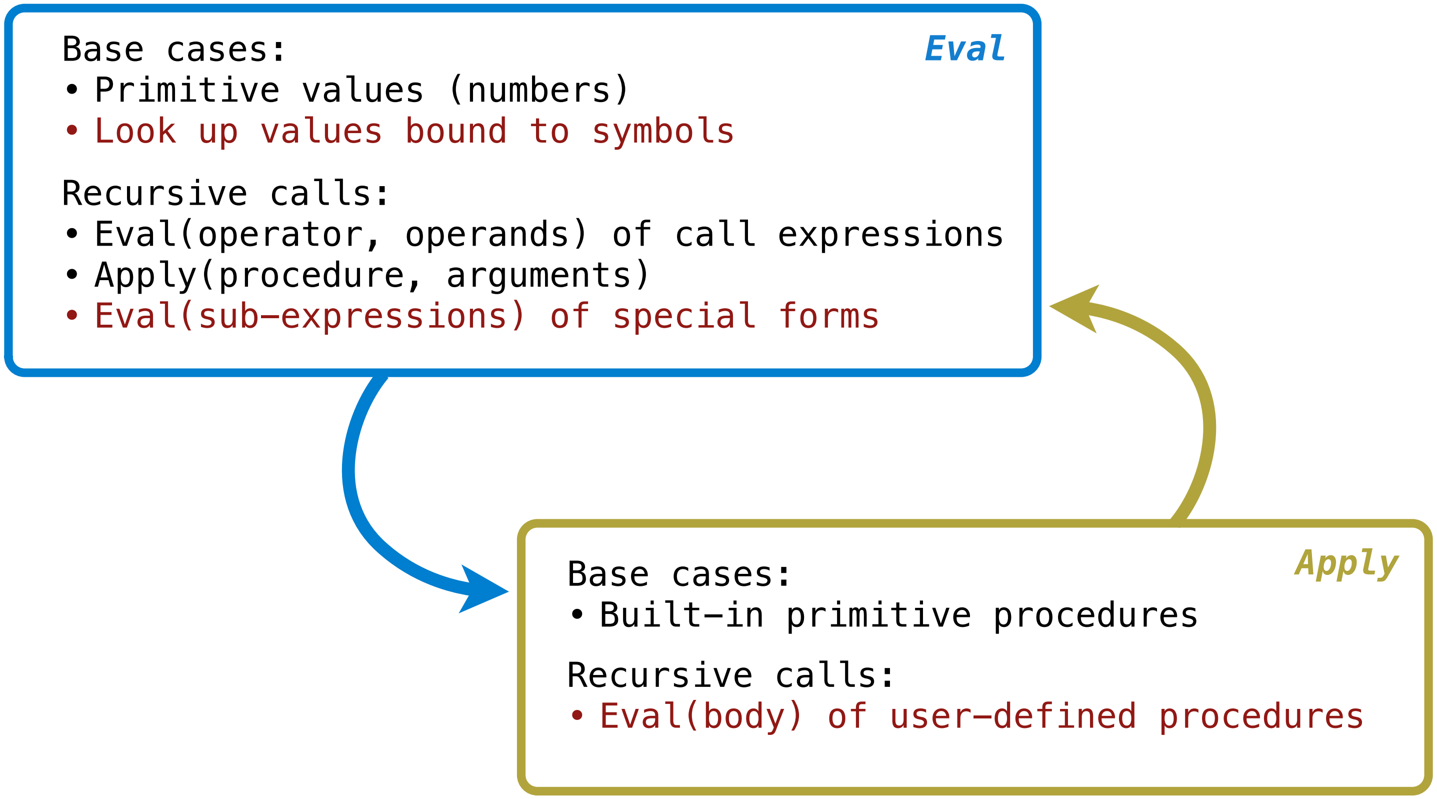
Recursive calls:
- Eval(operator, operands) of call expressions
- Apply(procedure, arguments)
- Eval(sub-expressions) of special forms

Apply

Base cases:
- Built-in primitive procedures

Recursive calls:
- Eval(body) of user-defined procedures

The Structure of an Interpreter



Eval

Base cases:
- Primitive values (numbers)
- Look up values bound to symbols

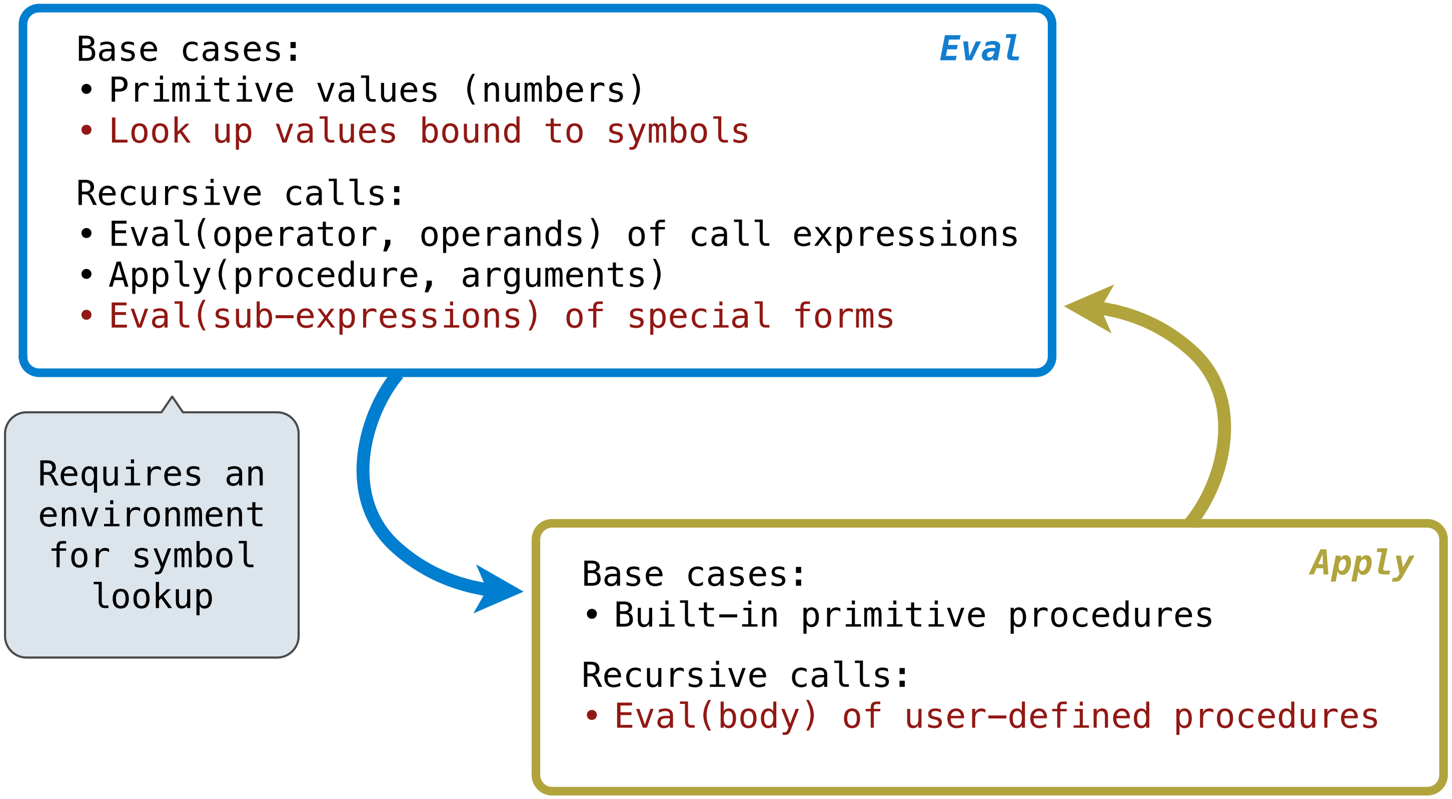
Recursive calls:
- Eval(operator, operands) of call expressions
- Apply(procedure, arguments)
- Eval(sub-expressions) of special forms

Apply

Base cases:
- Built-in primitive procedures

Recursive calls:
- Eval(body) of user-defined procedures

The Structure of an Interpreter



Eval

Base cases:

- Primitive values (numbers)
- Look up values bound to symbols

Recursive calls:

- Eval(operator, operands) of call expressions
- Apply(procedure, arguments)
- Eval(sub-expressions) of special forms

Requires an environment for symbol lookup

Apply

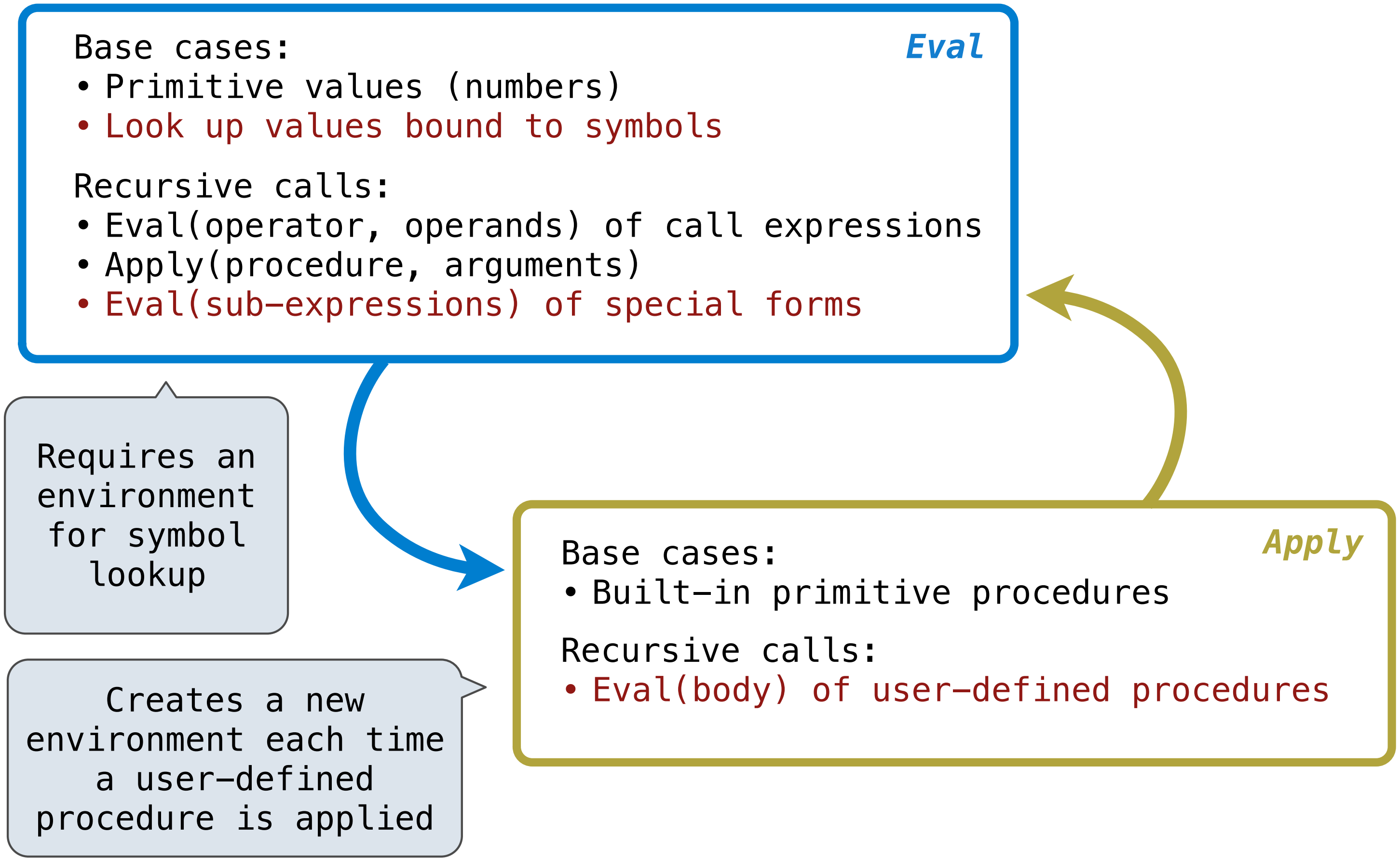
Base cases:

- Built-in primitive procedures

Recursive calls:

- Eval(body) of user-defined procedures

The Structure of an Interpreter



Eval

Base cases:

- Primitive values (numbers)
- Look up values bound to symbols

Recursive calls:

- Eval(operator, operands) of call expressions
- Apply(procedure, arguments)
- Eval(sub-expressions) of special forms

Requires an environment for symbol lookup

Apply

Base cases:

- Built-in primitive procedures

Recursive calls:

- Eval(body) of user-defined procedures

Creates a new environment each time a user-defined procedure is applied

Special Forms

Scheme Evaluation

Scheme Evaluation

The scheme_eval function dispatches on expression form:

Scheme Evaluation

The scheme_eval function dispatches on expression form:

- Symbols are bound to values in the current environment.

Scheme Evaluation

The scheme_eval function dispatches on expression form:

- Symbols are bound to values in the current environment.
- Self-evaluating expressions are returned.

Scheme Evaluation

The scheme_eval function dispatches on expression form:

- Symbols are bound to values in the current environment.
- Self-evaluating expressions are returned.
- All other legal expressions are represented as Scheme lists, called *combinations*.

Scheme Evaluation

The scheme_eval function dispatches on expression form:

- Symbols are bound to values in the current environment.
- Self-evaluating expressions are returned.
- All other legal expressions are represented as Scheme lists, called *combinations*.

```
(if <predicate> <consequent> <alternative>)
```

Scheme Evaluation

The scheme_eval function dispatches on expression form:

- Symbols are bound to values in the current environment.
- Self-evaluating expressions are returned.
- All other legal expressions are represented as Scheme lists, called *combinations*.

```
(if <predicate> <consequent> <alternative>)

(lambda (<formal-parameters>) <body>)
```

Scheme Evaluation

The scheme_eval function dispatches on expression form:

- Symbols are bound to values in the current environment.
- Self-evaluating expressions are returned.
- All other legal expressions are represented as Scheme lists, called *combinations*.

```
(if <predicate> <consequent> <alternative>)

(lambda (<formal-parameters>) <body>)

(define <name> <expression>)
```

Scheme Evaluation

The scheme_eval function dispatches on expression form:

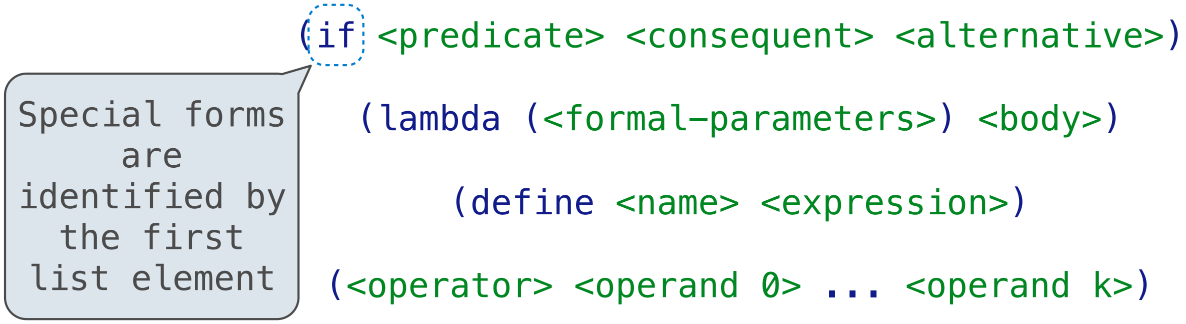
- Symbols are bound to values in the current environment.
- Self-evaluating expressions are returned.
- All other legal expressions are represented as Scheme lists, called *combinations*.

```
(if <predicate> <consequent> <alternative>)

(lambda (<formal-parameters>) <body>)

(define <name> <expression>)

(<operator> <operand 0> ... <operand k>)
```

Scheme Evaluation

The scheme_eval function dispatches on expression form:

- Symbols are bound to values in the current environment.
- Self-evaluating expressions are returned.
- All other legal expressions are represented as Scheme lists, called *combinations*.



```
(if <predicate> <consequent> <alternative>)

  (lambda (<formal-parameters>) <body>)

      (define <name> <expression>)

 (<operator> <operand 0> ... <operand k>)
```

Special forms are identified by the first list element

Scheme Evaluation

The scheme_eval function dispatches on expression form:

- Symbols are bound to values in the current environment.
- Self-evaluating expressions are returned.
- All other legal expressions are represented as Scheme lists, called *combinations*.

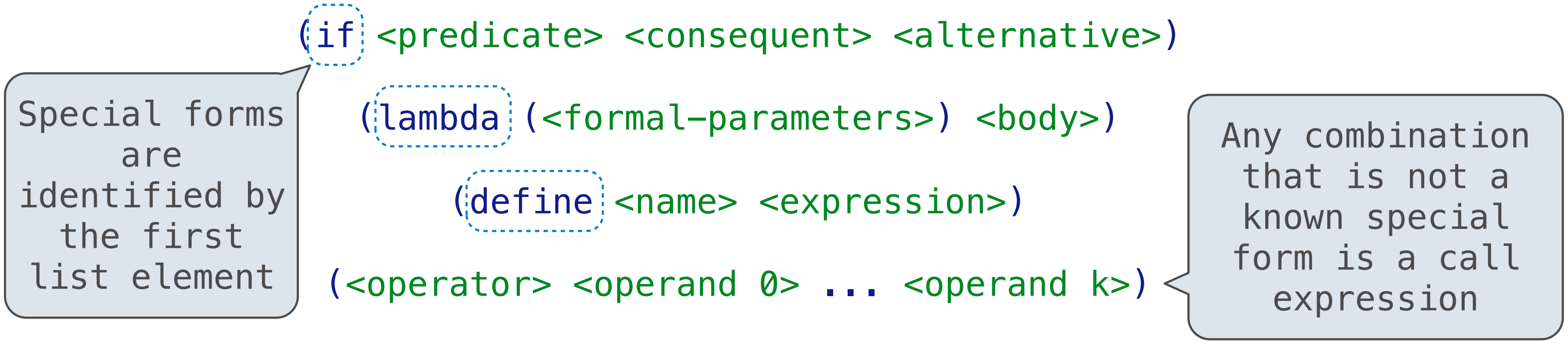
```
(if <predicate> <consequent> <alternative>)
(lambda (<formal-parameters>) <body>)
(define <name> <expression>)
(<operator> <operand 0> ... <operand k>)
```

Special forms are identified by the first list element

Scheme Evaluation

The scheme_eval function dispatches on expression form:

- Symbols are bound to values in the current environment.
- Self-evaluating expressions are returned.
- All other legal expressions are represented as Scheme lists, called *combinations*.



```
(if <predicate> <consequent> <alternative>)
(lambda (<formal-parameters>) <body>)
(define <name> <expression>)
(<operator> <operand 0> ... <operand k>)
```

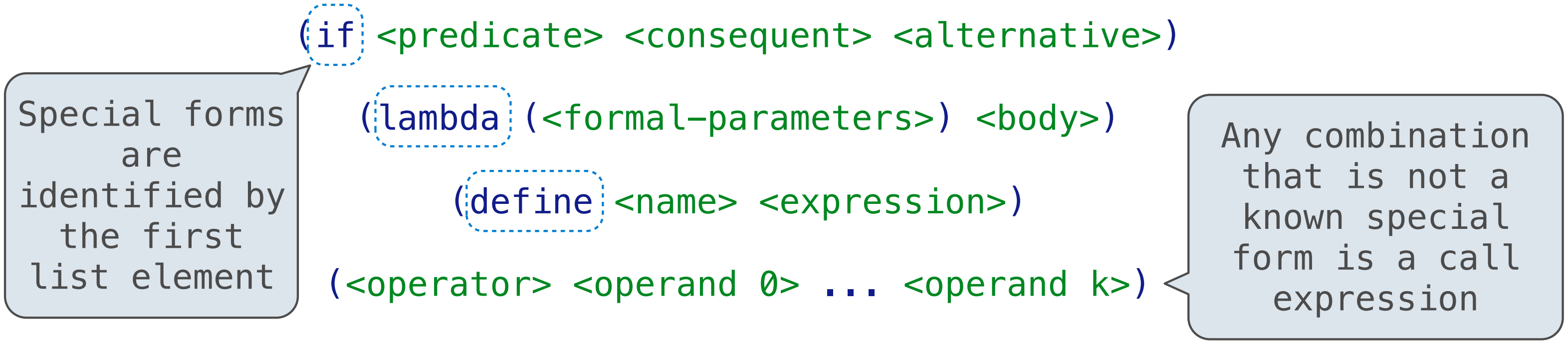
Special forms are identified by the first list element

Any combination that is not a known special form is a call expression

Scheme Evaluation

The scheme_eval function dispatches on expression form:

- Symbols are bound to values in the current environment.
- Self-evaluating expressions are returned.
- All other legal expressions are represented as Scheme lists, called *combinations*.



```
(if <predicate> <consequent> <alternative>)
(lambda (<formal-parameters>) <body>)
(define <name> <expression>)
(<operator> <operand 0> ... <operand k>)
```

Special forms are identified by the first list element

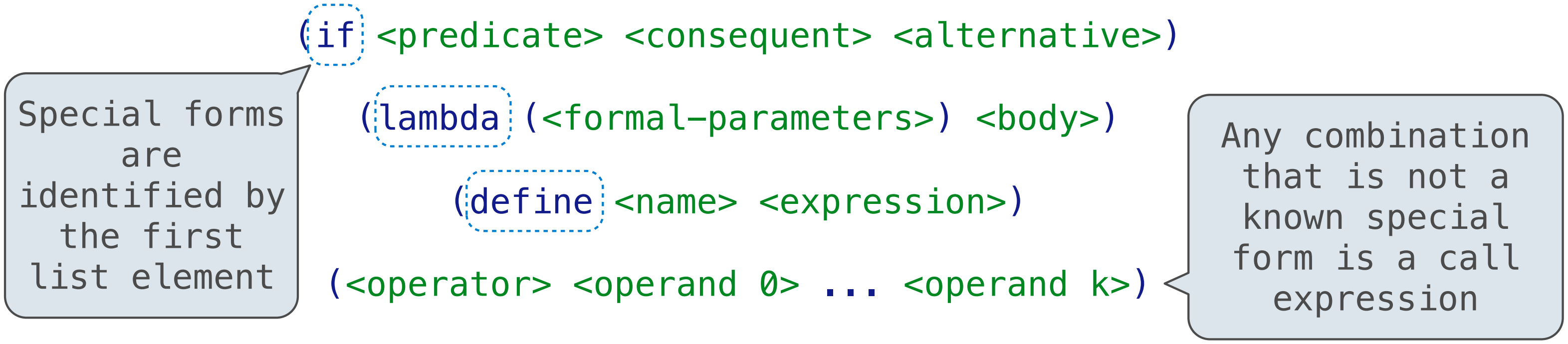
Any combination that is not a known special form is a call expression

```
(define (demo s) (if (null? s) '(3) (cons (car s) (demo (cdr s))) ))
```

Scheme Evaluation

The scheme_eval function dispatches on expression form:

- Symbols are bound to values in the current environment.
- Self-evaluating expressions are returned.
- All other legal expressions are represented as Scheme lists, called *combinations*.



```
(if <predicate> <consequent> <alternative>)

(lambda (<formal-parameters>) <body>)

(define <name> <expression>)

(<operator> <operand 0> ... <operand k>)
```

Special forms are identified by the first list element

Any combination that is not a known special form is a call expression

```
(define (demo s) (if (null? s) '(3) (cons (car s) (demo (cdr s))) ))

(demo (list 1 2))
```

Logical Forms

Logical Special Forms

Logical Special Forms

Logical forms may only evaluate some sub-expressions.

Logical Special Forms

Logical forms may only evaluate some sub-expressions.

- **If** expression: `(if <predicate> <consequent> <alternative>)`

Logical Special Forms

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>)

Logical Special Forms

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

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

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

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

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

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.

Logical Special Forms

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

Logical Special Forms

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

(Demo)

Quotation

Quotation

Quotation

The **quote** special form evaluates to the quoted expression, which is **not** evaluated.

Quotation

The **quote** special form evaluates to the quoted expression, which is **not** evaluated.

```
(quote <expression>)
```

Quotation

The **quote** special form evaluates to the quoted expression, which is **not** evaluated.

```
(quote <expression>)
```

```
(quote (+ 1 2))
```

evaluates to the three-element Scheme list

```
(+ 1 2)
```

Quotation

The **quote** special form evaluates to the quoted expression, which is **not** evaluated.

```
(quote <expression>)
```

```
(quote (+ 1 2))
```

evaluates to the three-element Scheme list

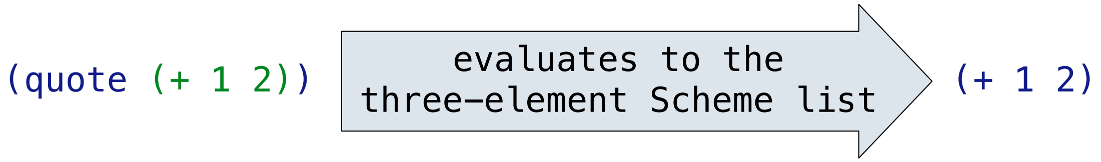
```
(+ 1 2)
```

The <expression> itself is the value of the whole quote expression.

Quotation

The **quote** special form evaluates to the quoted expression, which is **not** evaluated.

```
(quote <expression>)
```



(quote (+ 1 2))

evaluates to the
three-element Scheme list

(+ 1 2)

The <expression> itself is the value of the whole quote expression.

'<expression> is shorthand for (quote <expression>).

Quotation

The **quote** special form evaluates to the quoted expression, which is **not** evaluated.

(quote <expression>)   (quote (+ 1 2))

evaluates to the
three-element Scheme list

(+ 1 2)

The <expression> itself is the value of the whole quote expression.

'<expression> is shorthand for (quote <expression>).

(quote (1 2))   is equivalent to   '(1 2)

Quotation

The **quote** special form evaluates to the quoted expression, which is **not** evaluated.

(quote <expression>)

(quote (+ 1 2))

evaluates to the
three-element Scheme list

(+ 1 2)

The <expression> itself is the value of the whole quote expression.

'<expression> is shorthand for (quote <expression>).

(quote (1 2)) is equivalent to '(1 2)

The scheme_read parser converts shorthand to a combination.

Quotation

The **quote** special form evaluates to the quoted expression, which is **not** evaluated.

```
(quote <expression>)        (quote (+ 1 2))
```

evaluates to the three-element Scheme list

```
(+ 1 2)
```

The <expression> itself is the value of the whole quote expression.

'<expression> is shorthand for (quote <expression>).

```
(quote (1 2))     is equivalent to     '(1 2)
```

The scheme_read parser converts shorthand to a combination.

(Demo)

Lambda Expressions

Lambda Expressions

Lambda Expressions

Lambda expressions evaluate to user-defined procedures.

Lambda Expressions

Lambda expressions evaluate to user-defined procedures.

```
(lambda (<formal-parameters>) <body>)
```

Lambda Expressions

Lambda expressions evaluate to user-defined procedures.

```
(lambda (<formal-parameters>) <body>)

(lambda (x) (* x x))
```

Lambda Expressions

Lambda expressions evaluate to user-defined procedures.

```
(lambda (<formal-parameters>) <body>)

(lambda (x) (* x x))
```

```
class LambdaProcedure:
    def __init__(self, formals, body, env):
        self.formals = formals
        self.body = body
        self.env = env
```

Lambda Expressions

Lambda expressions evaluate to user-defined procedures.

```
(lambda (<formal-parameters>) <body>)

(lambda (x) (* x x))
```

```
class LambdaProcedure:
    def __init__(self, formals, body, env):
        self.formals = formals
        self.body = body
        self.env = env
```

A scheme list of symbols

Lambda Expressions

Lambda expressions evaluate to user-defined procedures.

```
(lambda (<formal-parameters>) <body>)

(lambda (x) (* x x))
```

```
class LambdaProcedure:
    def __init__(self, formals, body, env):
        self.formals = formals
        self.body = body
        self.env = env
```

A scheme list of symbols

A scheme expression

Lambda Expressions

Lambda expressions evaluate to user-defined procedures.

```
(lambda (<formal-parameters>) <body>)

(lambda (x) (* x x))
```

```
class LambdaProcedure:
    def __init__(self, formals, body, env):
        self.formals = formals                  A scheme list of symbols
        self.body = body                        A scheme expression
        self.env = env                          A Frame instance
```

Frames and Environments

Frames and Environments

A frame represents an environment by having a parent frame.

Frames and Environments

A frame represents an environment by having a parent frame.

Frames are Python instances with methods **lookup** and **define.**

Frames and Environments

A frame represents an environment by having a parent frame.

Frames are Python instances with methods **lookup** and **define.**

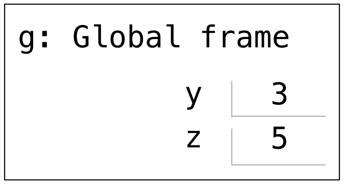
In Project 4, Frames do not hold return values.

Frames and Environments

A frame represents an environment by having a parent frame.

Frames are Python instances with methods **lookup** and **define.**

In Project 4, Frames do not hold return values.


g: Global frame

y 3

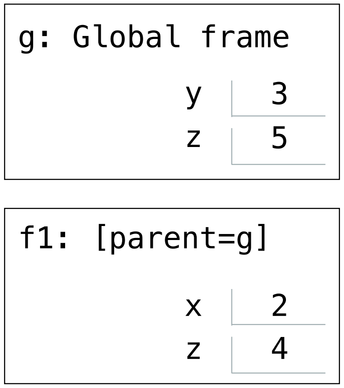
z 5

Frames and Environments

A frame represents an environment by having a parent frame.

Frames are Python instances with methods **lookup** and **define.**

In Project 4, Frames do not hold return values.



g: Global frame

y 3

z 5

f1: [parent=g]

x 2

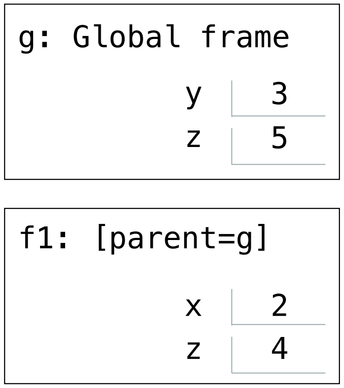
z 4

Frames and Environments

A frame represents an environment by having a parent frame.

Frames are Python instances with methods **lookup** and **define.**

In Project 4, Frames do not hold return values.



g: Global frame

y 3

z 5

f1: [parent=g]

x 2

z 4

(Demo)

Define Expressions

Define Expressions

Define Expressions

Define binds a symbol to a value in the first frame of the current environment.

Define Expressions

Define binds a symbol to a value in the first frame of the current environment.

```
(define <name> <expression>)
```

Define Expressions

Define binds a symbol to a value in the first frame of the current environment.

```
(define <name> <expression>)
```

1. Evaluate the <expression>.

Define Expressions

Define binds a symbol to a value in the first frame of the current environment.

```
(define <name> <expression>)
```

1. Evaluate the <expression>.

2. Bind <name> to its value in the current frame.

Define Expressions

Define binds a symbol to a value in the first frame of the current environment.

```
(define <name> <expression>)
```

1. Evaluate the <expression>.

2. Bind <name> to its value in the current frame.

```
(define x (+ 1 2))
```

Define Expressions

Define binds a symbol to a value in the first frame of the current environment.

```
(define <name> <expression>)
```

1. Evaluate the <expression>.
2. Bind <name> to its value in the current frame.

```
(define x (+ 1 2))
```

Procedure definition is shorthand of define with a lambda expression.

Define Expressions

Define binds a symbol to a value in the first frame of the current environment.

```
(define <name> <expression>)
```

1. Evaluate the <expression>.

2. Bind <name> to its value in the current frame.

```
(define x (+ 1 2))
```

Procedure definition is shorthand of define with a lambda expression.

```
(define (<name> <formal parameters>) <body>)
```

Define Expressions

Define binds a symbol to a value in the first frame of the current environment.

```
(define <name> <expression>)
```

1. Evaluate the <expression>.

2. Bind <name> to its value in the current frame.

```
(define x (+ 1 2))
```

Procedure definition is shorthand of define with a lambda expression.

```
(define (<name> <formal parameters>) <body>)

(define <name> (lambda (<formal parameters>) <body>))
```

Applying User-Defined Procedures

Applying User-Defined Procedures

To apply a user-defined procedure, create a new frame in which formal parameters are bound to argument values, whose parent is the **env** of the procedure.

Applying User-Defined Procedures

To apply a user-defined procedure, create a new frame in which formal parameters are bound to argument values, whose parent is the **env** of the procedure.

Evaluate the body of the procedure in the environment that starts with this new frame.

Applying User-Defined Procedures

To apply a user-defined procedure, create a new frame in which formal parameters are bound to argument values, whose parent is the **env** of the procedure.

Evaluate the body of the procedure in the environment that starts with this new frame.

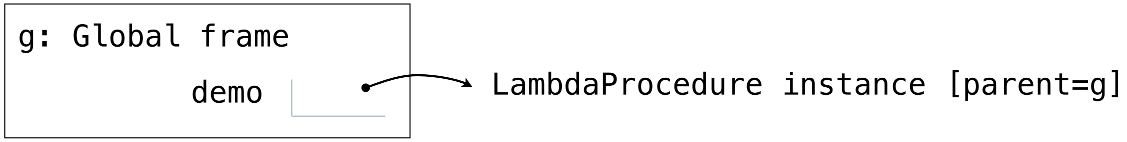
```
(define (demo s) (if (null? s) '(3) (cons (car s) (demo (cdr s)))))
```

Applying User-Defined Procedures

To apply a user-defined procedure, create a new frame in which formal parameters are bound to argument values, whose parent is the **env** of the procedure.

Evaluate the body of the procedure in the environment that starts with this new frame.

```
(define (demo s) (if (null? s) '(3) (cons (car s) (demo (cdr s)))))
```



g: Global frame

demo

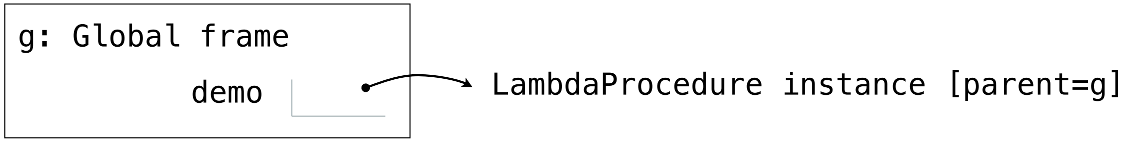
LambdaProcedure instance [parent=g]

Applying User-Defined Procedures

To apply a user-defined procedure, create a new frame in which formal parameters are bound to argument values, whose parent is the **env** of the procedure.

Evaluate the body of the procedure in the environment that starts with this new frame.

```
(define (demo s) (if (null? s) '(3) (cons (car s) (demo (cdr s)))))

                                  (demo (list 1 2))
```



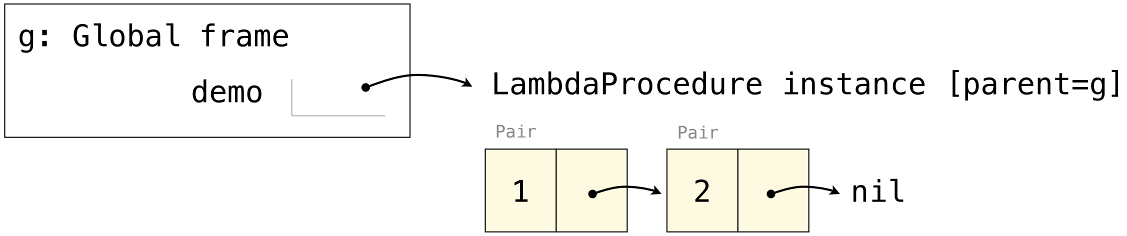
Applying User-Defined Procedures

To apply a user-defined procedure, create a new frame in which formal parameters are bound to argument values, whose parent is the **env** of the procedure.

Evaluate the body of the procedure in the environment that starts with this new frame.

```
(define (demo s) (if (null? s) '(3) (cons (car s) (demo (cdr s)))))
```

```
(demo (list 1 2))
```



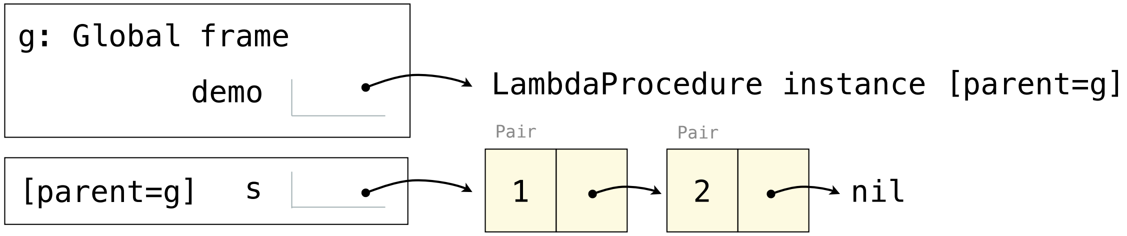
Applying User-Defined Procedures

To apply a user-defined procedure, create a new frame in which formal parameters are bound to argument values, whose parent is the **env** of the procedure.

Evaluate the body of the procedure in the environment that starts with this new frame.

```
(define (demo s) (if (null? s) '(3) (cons (car s) (demo (cdr s)))))
```

```
(demo (list 1 2))
```



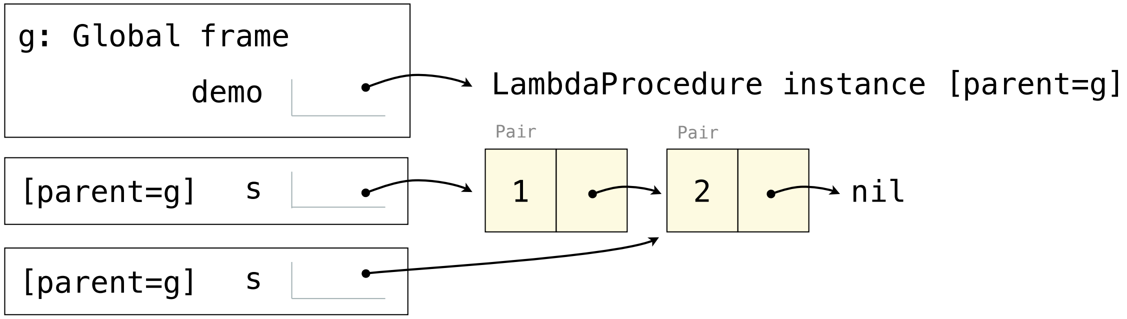
Applying User-Defined Procedures

To apply a user-defined procedure, create a new frame in which formal parameters are bound to argument values, whose parent is the **env** of the procedure.

Evaluate the body of the procedure in the environment that starts with this new frame.

```
(define (demo s) (if (null? s) '(3) (cons (car s) (demo (cdr s)))))
```

```
(demo (list 1 2))
```



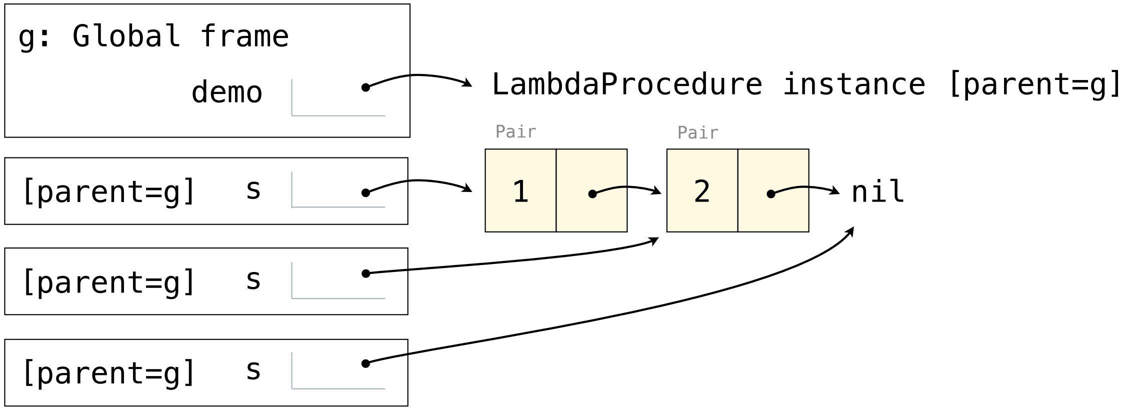
Applying User-Defined Procedures

To apply a user-defined procedure, create a new frame in which formal parameters are bound to argument values, whose parent is the **env** of the procedure.

Evaluate the body of the procedure in the environment that starts with this new frame.

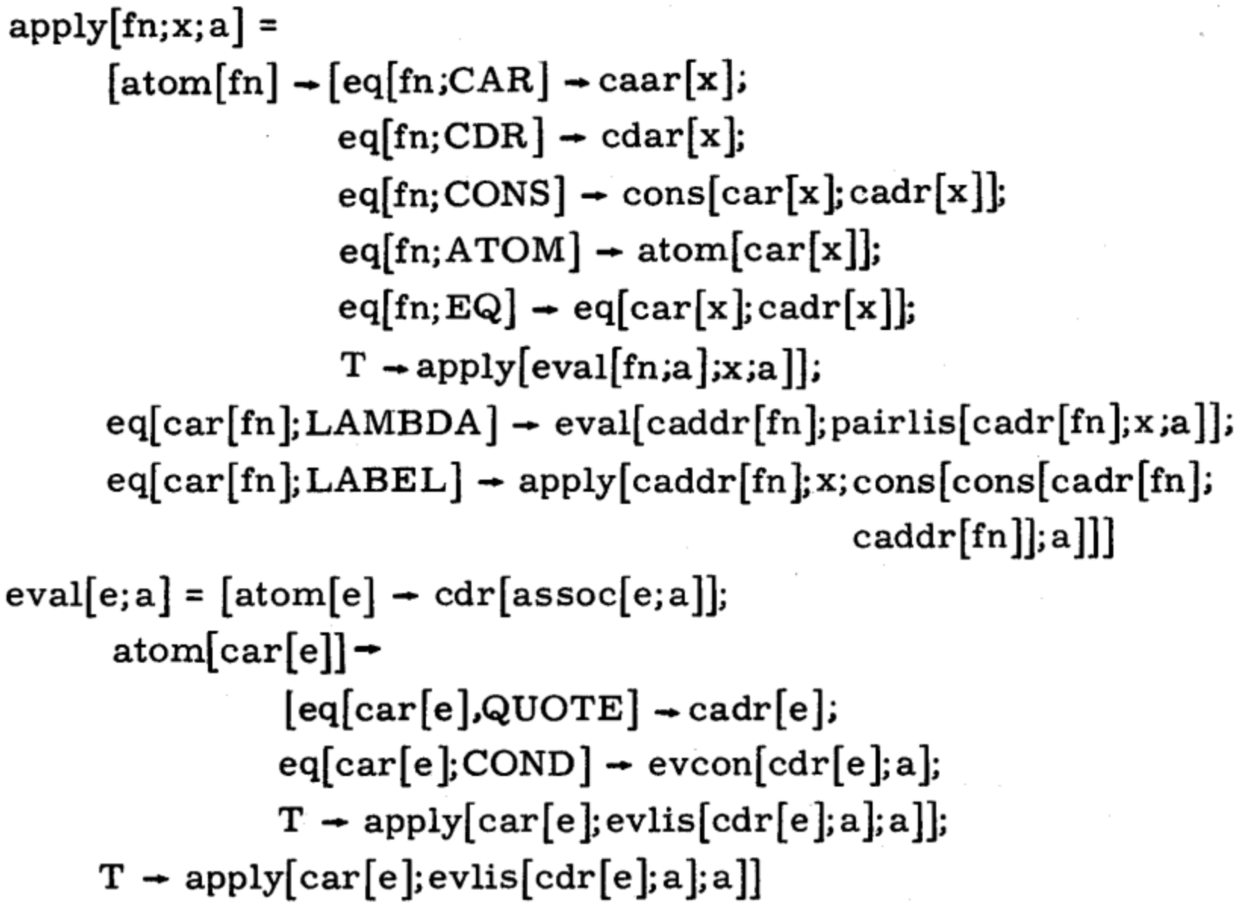
```
(define (demo s) (if (null? s) '(3) (cons (car s) (demo (cdr s)))))
```

```
(demo (list 1 2))
```



Eval/Apply in Lisp 1.5

Eval/Apply in Lisp 1.5



```
apply[fn;x;a] =
        [atom[fn] → [eq[fn;CAR] → caar[x];
                     eq[fn;CDR] → cdar[x];
                     eq[fn;CONS] → cons[car[x];cadr[x]];
                     eq[fn;ATOM] → atom[car[x]];
                     eq[fn;EQ] → eq[car[x];cadr[x]];
                     T → apply[eval[fn;a];x;a]];
        eq[car[fn];LAMBDA] → eval[caddr[fn];pairlis[cadr[fn];x;a]];
        eq[car[fn];LABEL] → apply[caddr[fn];x;cons[cons[cadr[fn];
                                                    caddr[fn]];a]]]
eval[e;a] = [atom[e] → cdr[assoc[e;a]];
        atom[car[e]]→
                  [eq[car[e],QUOTE] → cadr[e];
                  eq[car[e];COND] → evcon[cdr[e];a];
                  T → apply[car[e];evlis[cdr[e];a];a]];
        T → apply[car[e];evlis[cdr[e];a];a]]
```

Dynamic Scope

Dynamic Scope

Dynamic Scope

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

Dynamic Scope

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

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

Dynamic Scope

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

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

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

Dynamic Scope

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

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

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

```
(define f (lambda (x) (+ x y)))
```

Dynamic Scope

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

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

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

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

Dynamic Scope

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

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

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

```
(define f (lambda (x) (+ x y)))

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

(g 3 7)
```

Dynamic Scope

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

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

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

```
(define f (lambda (x) (+ x y)))

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

(g 3 7)
```

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

Dynamic Scope

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

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

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

```
(define f (lambda (x) (+ x y)))

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

(g 3 7)
```

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

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

Dynamic Scope

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

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

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

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

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

Error: unknown identifier: y

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

Dynamic Scope

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

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

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

```
(define f (lambda (x) (+ x y)))

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

(g 3 7)
```

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

Error: unknown identifier: y

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

13

Dynamic Scope

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

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

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

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