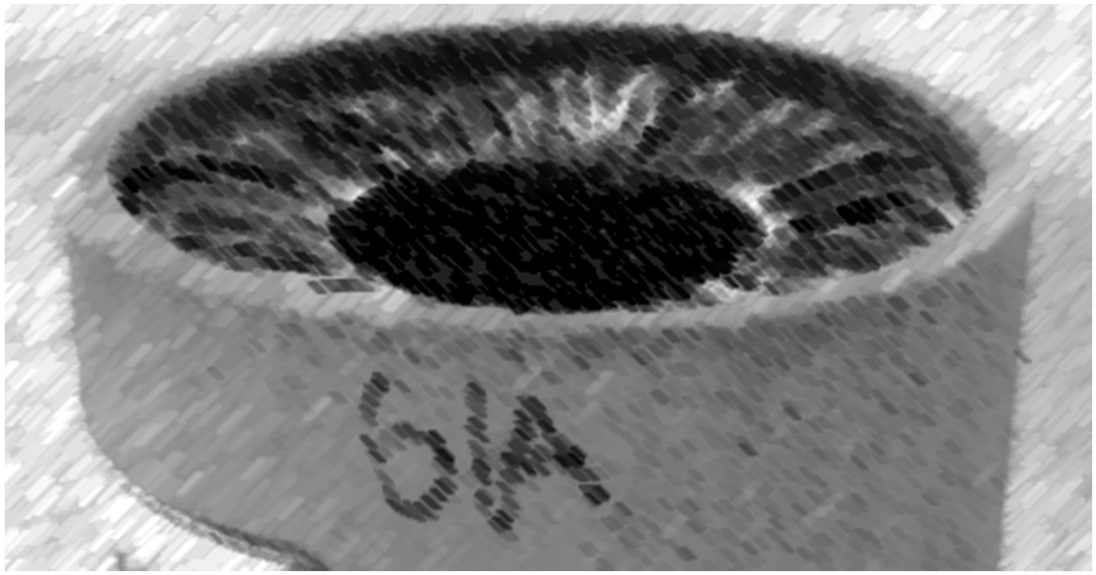


CS61A Lecture 34

Amir Kamil

UC Berkeley

April 10, 2013

Announcements

- HW10 deadline extended to 11:59pm Thursday
- Scheme project out

Read-Eval-Print Loop

Read-Eval-Print Loop

The user interface to many programming languages is an interactive loop, which

Read-Eval-Print Loop

The user interface to many programming languages is an interactive loop, which

- Reads an expression from the user,

Read-Eval-Print Loop

The user interface to many programming languages is an interactive loop, which

- Reads an expression from the user,
- Parses the input to build an expression tree,

Read-Eval-Print Loop

The user interface to many programming languages is an interactive loop, which

- Reads an expression from the user,
- Parses the input to build an expression tree,
- Evaluates the expression tree,

Read-Eval-Print Loop

The user interface to many programming languages is an interactive loop, which

- Reads an expression from the user,
- Parses the input to build an expression tree,
- Evaluates the expression tree,
- Prints the resulting value of the expression

Read-Eval-Print Loop

The user interface to many programming languages is an interactive loop, which

- Reads an expression from the user,
- Parses the input to build an expression tree,
- Evaluates the expression tree,
- Prints the resulting value of the expression

The REPL handles errors by printing informative messages for the user, rather than crashing

Read-Eval-Print Loop

The user interface to many programming languages is an interactive loop, which

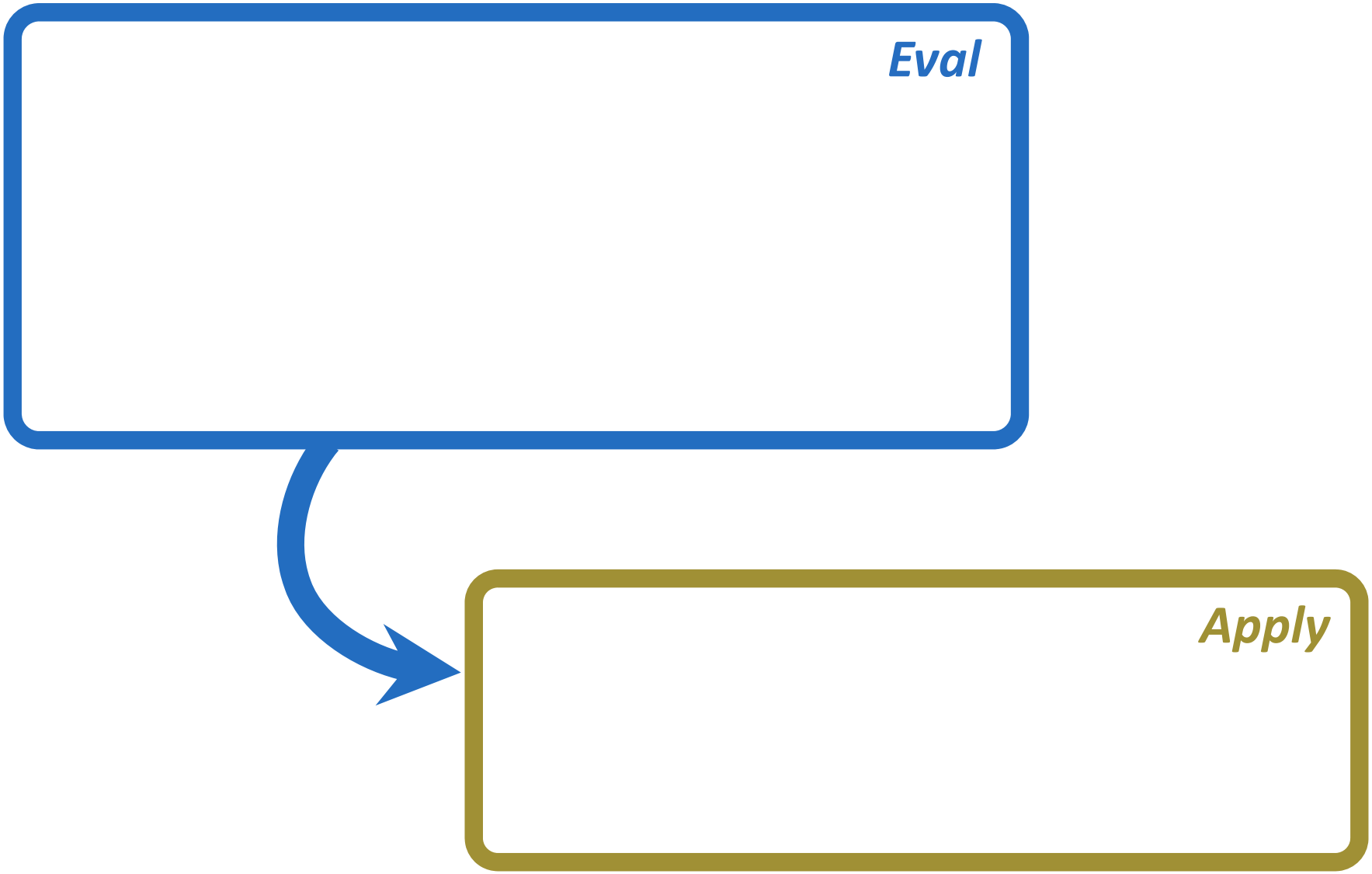
- Reads an expression from the user,
- Parses the input to build an expression tree,
- Evaluates the expression tree,
- Prints the resulting value of the expression

The REPL handles errors by printing informative messages for the user, rather than crashing

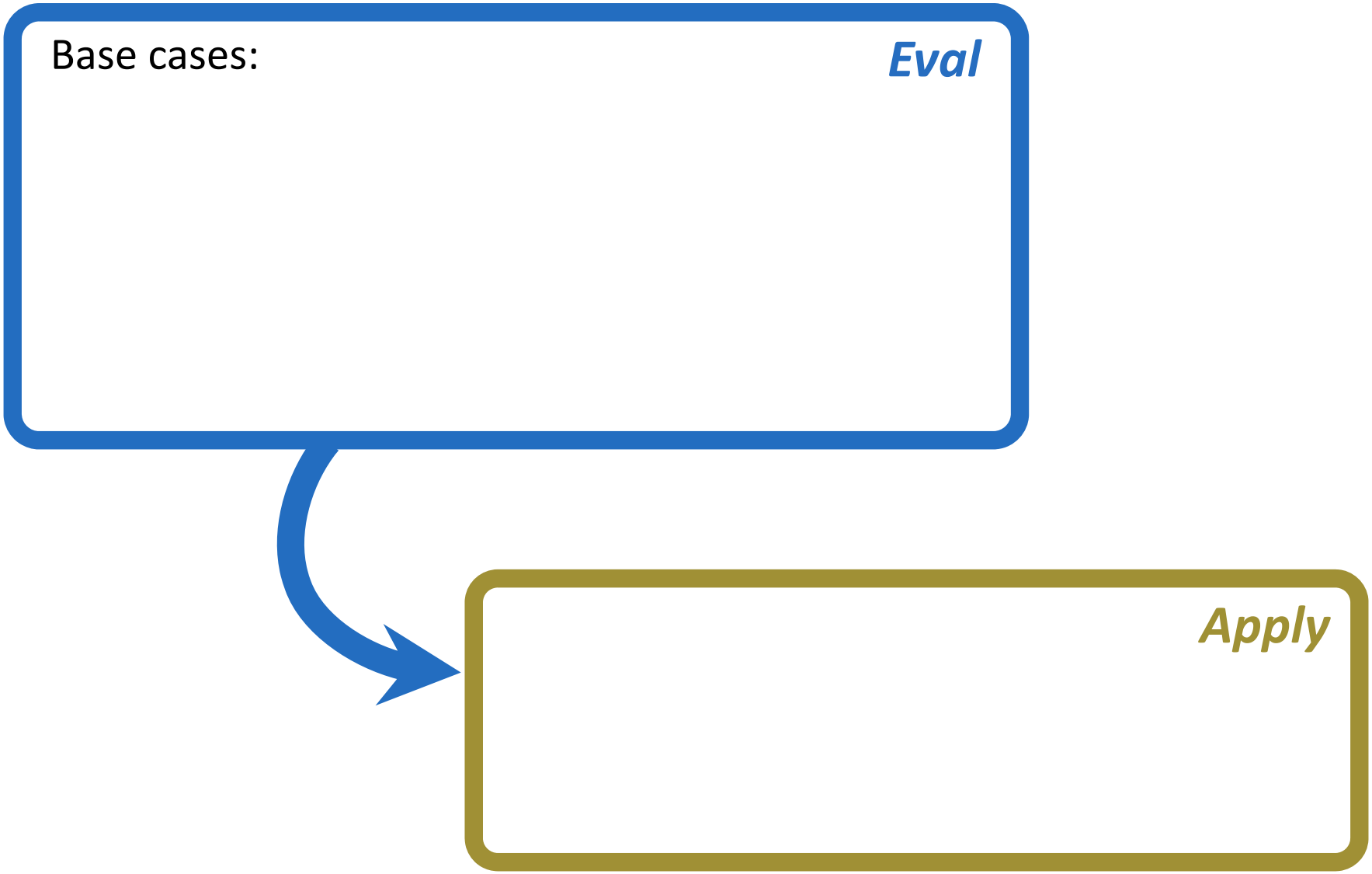
A well-designed REPL should not crash on any input!

The Structure of an Evaluator

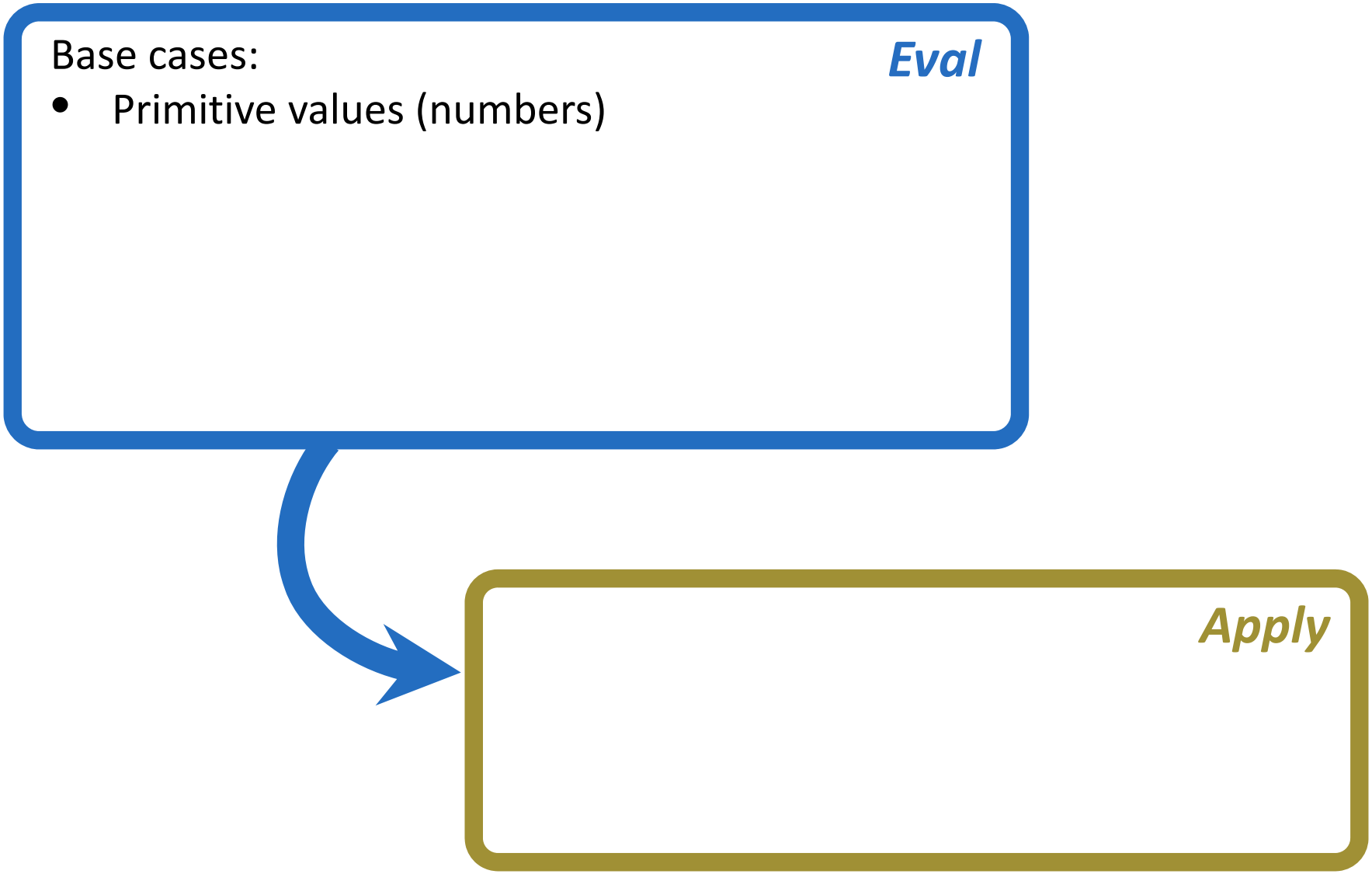
The Structure of an Evaluator



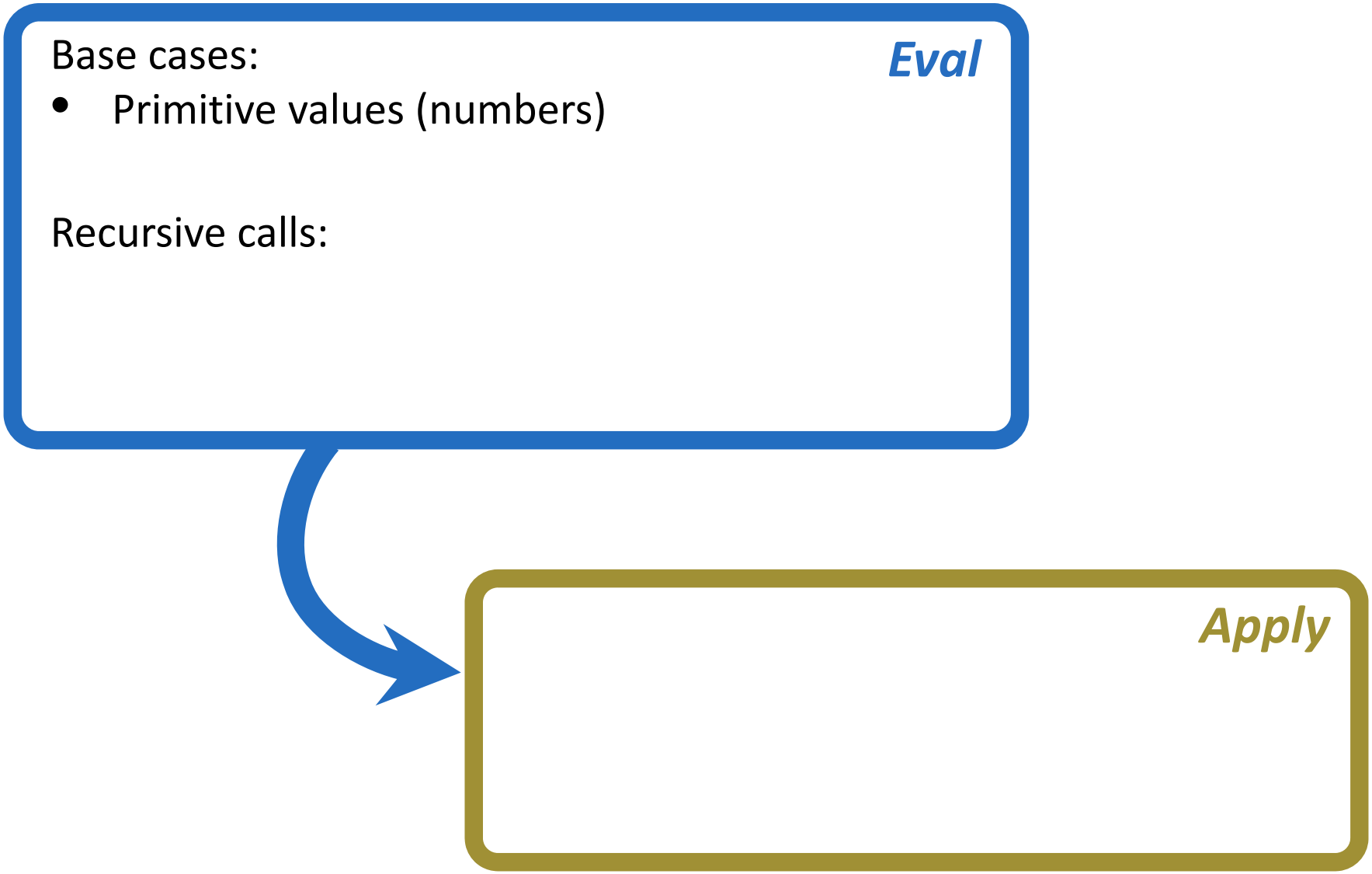
The Structure of an Evaluator



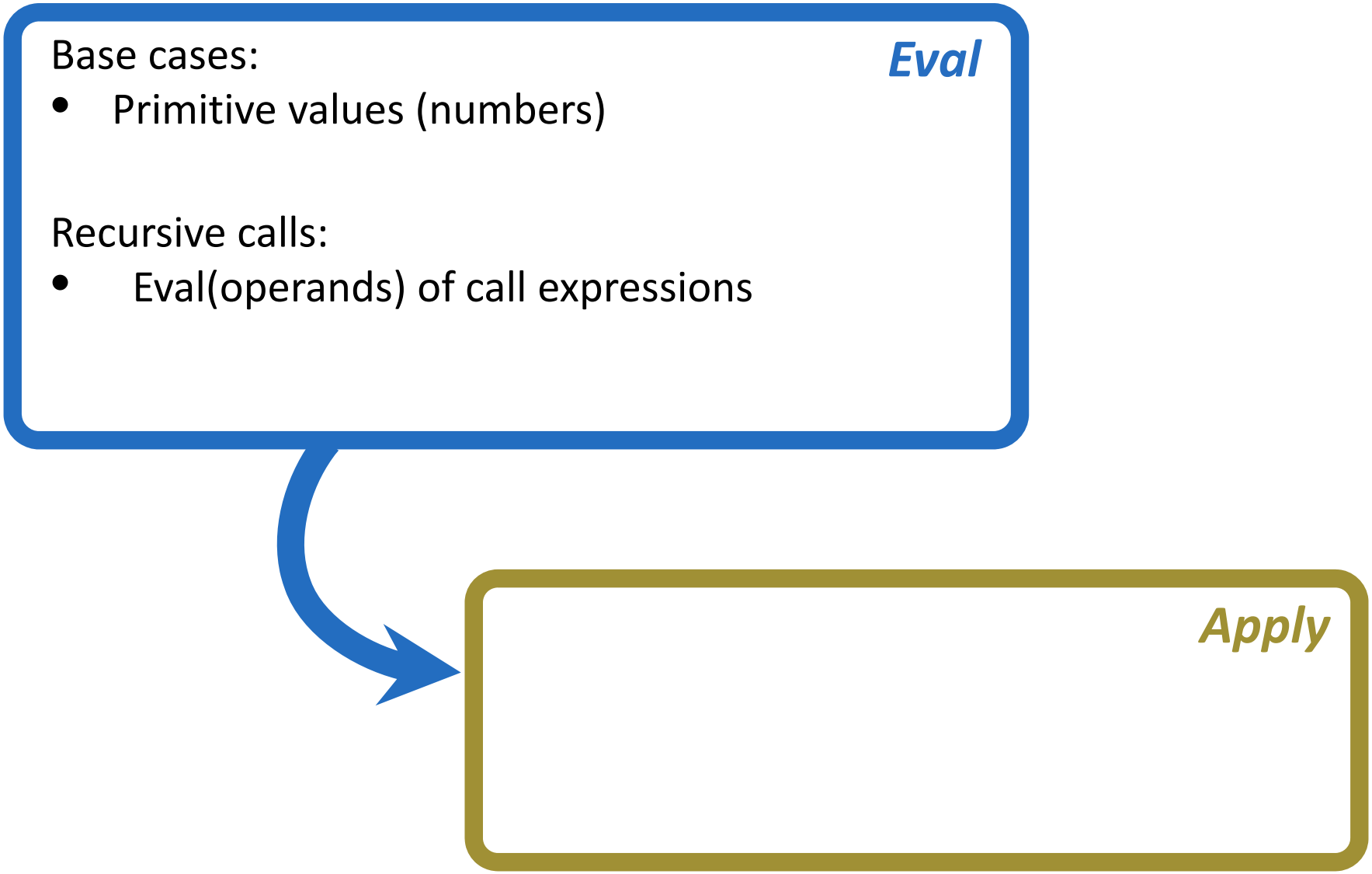
The Structure of an Evaluator



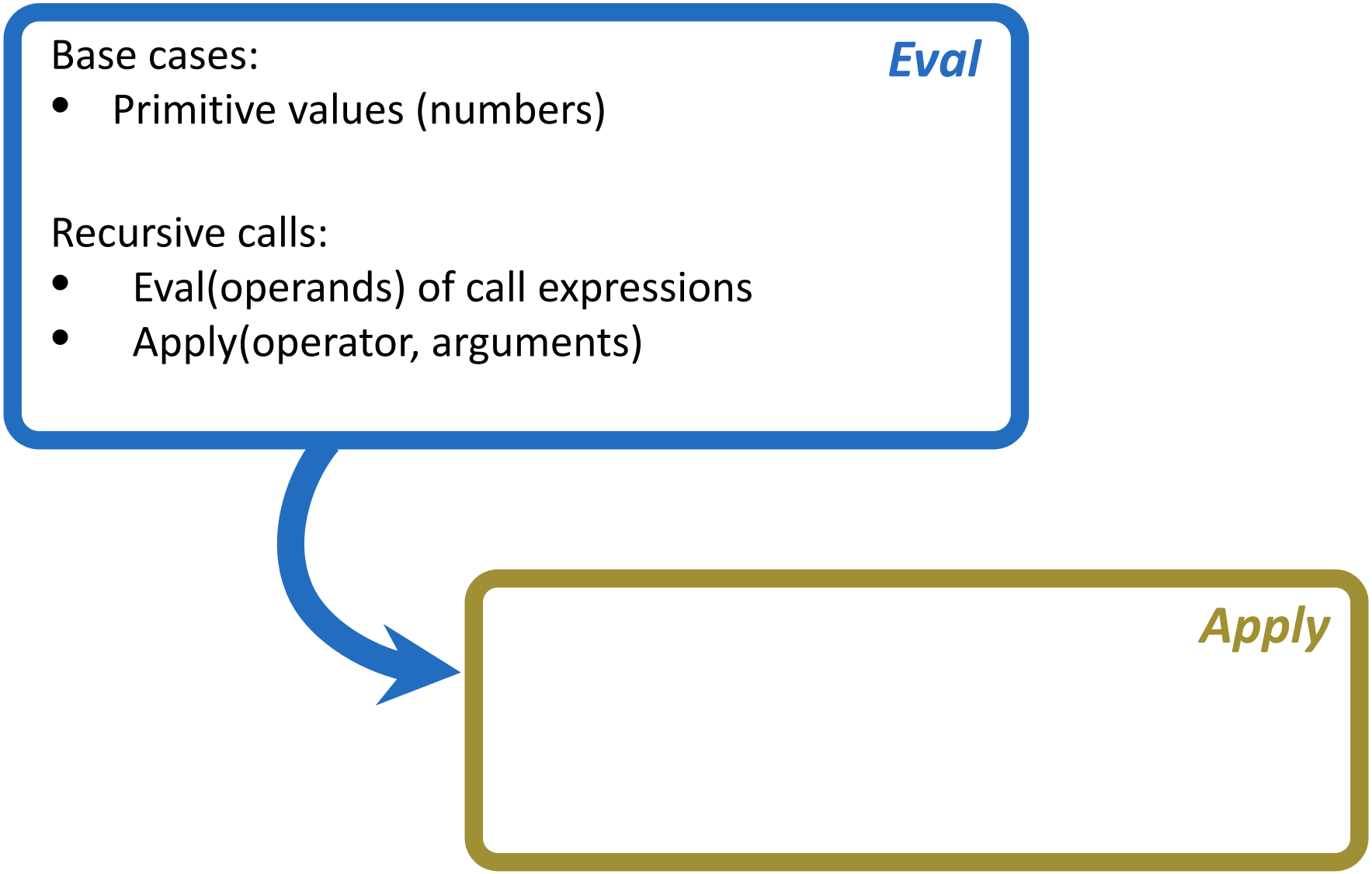
The Structure of an Evaluator



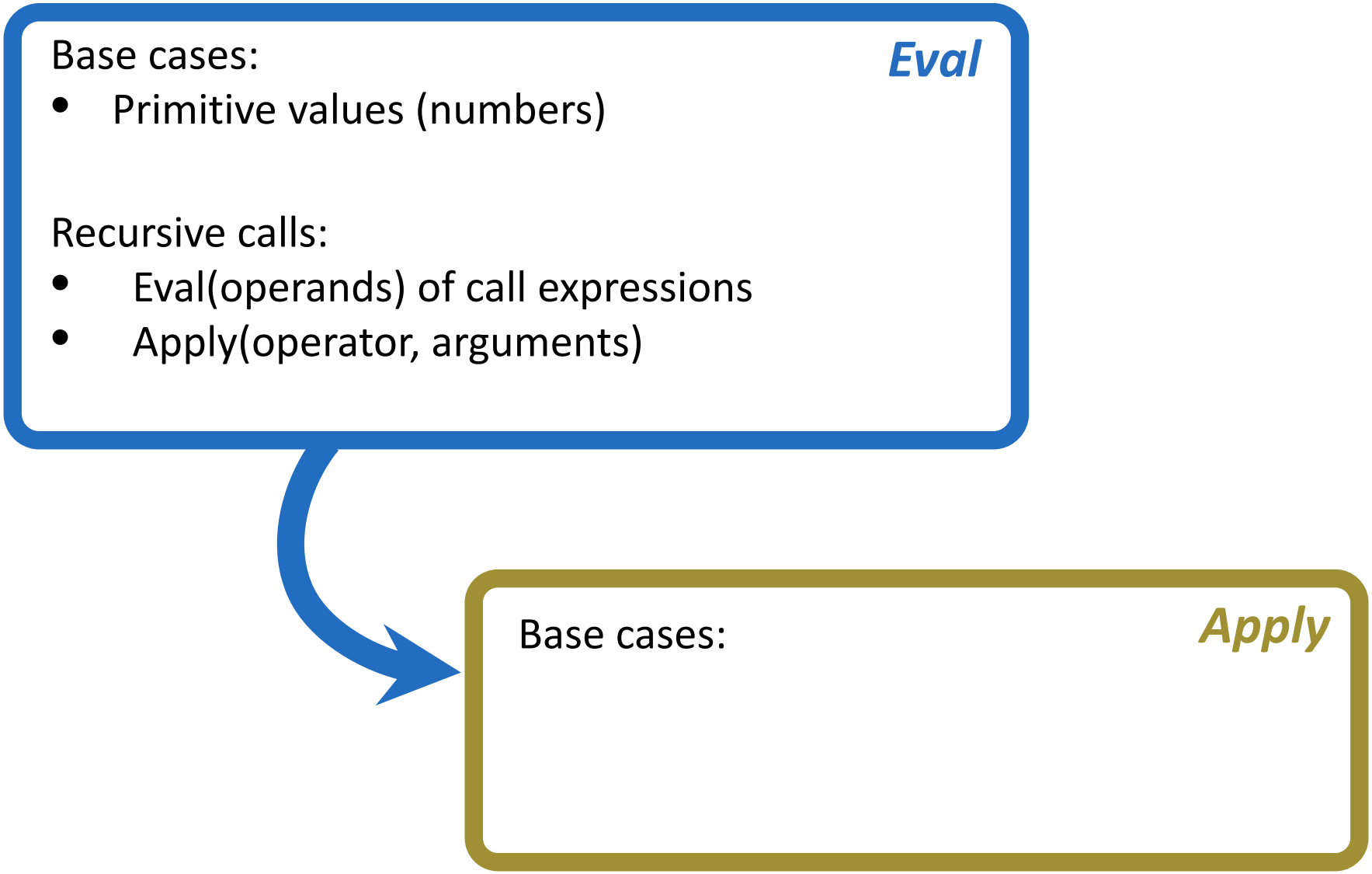
The Structure of an Evaluator



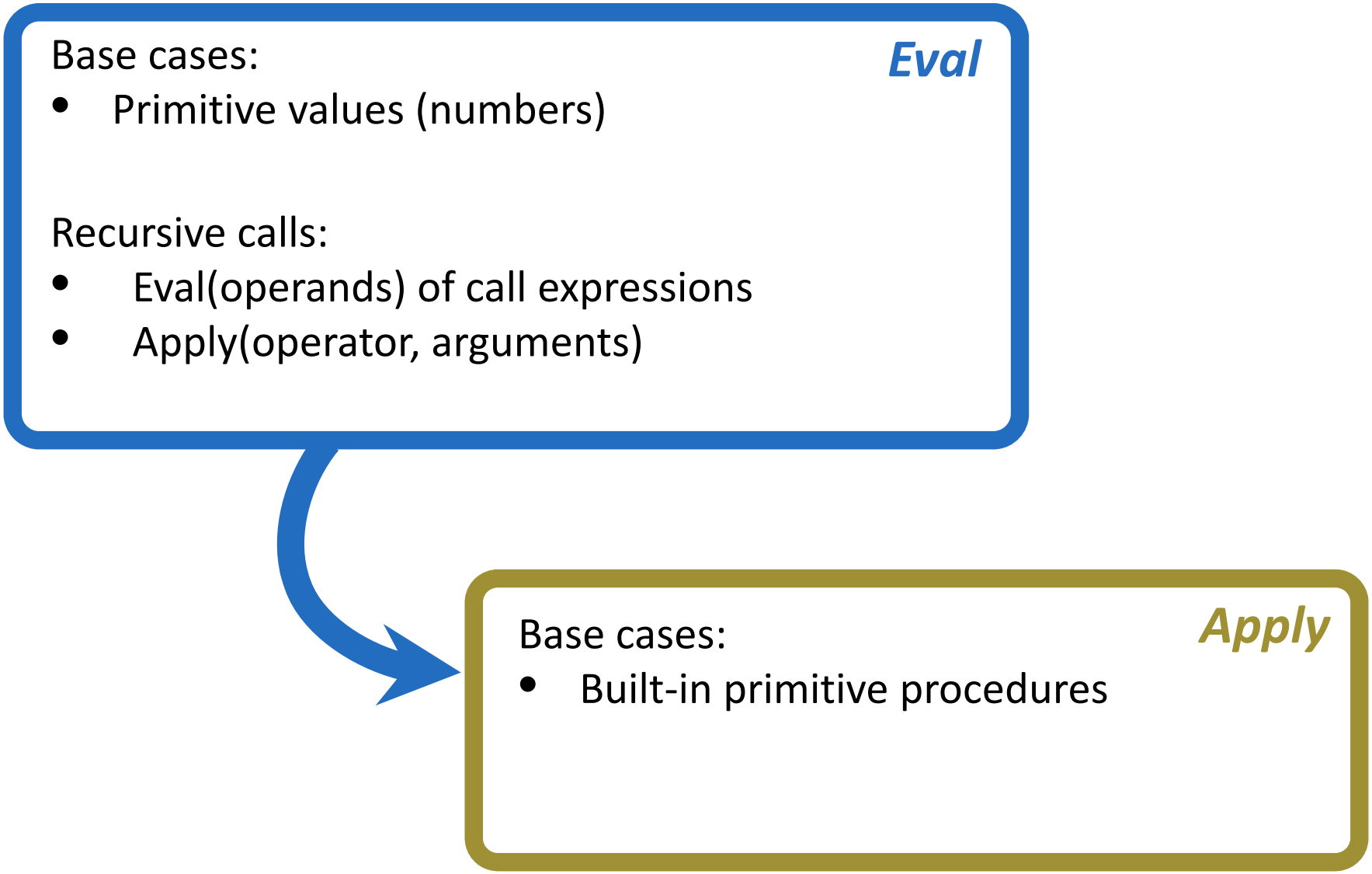
The Structure of an Evaluator



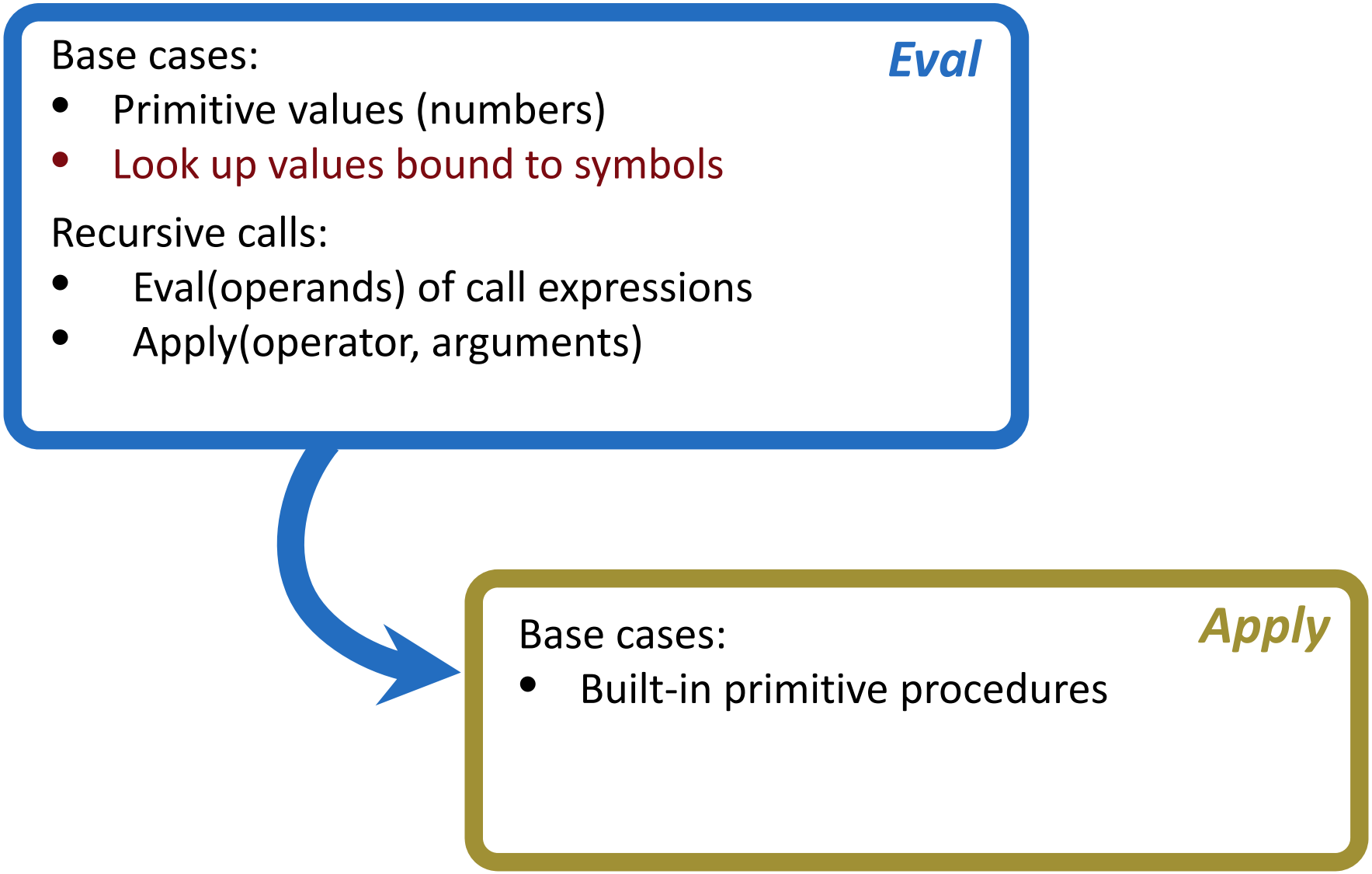
The Structure of an Evaluator



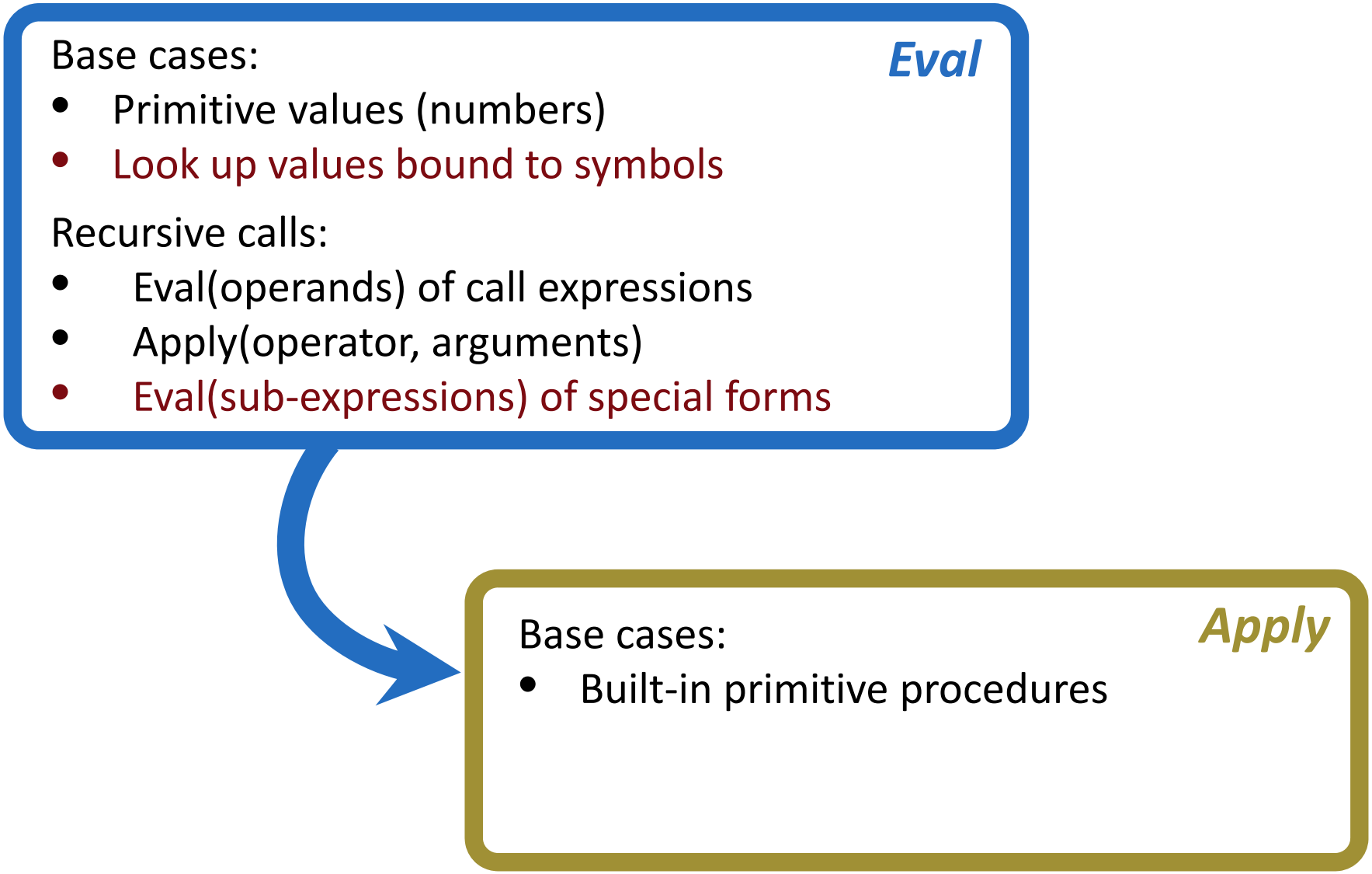
The Structure of an Evaluator



The Structure of an Evaluator



The Structure of an Evaluator



The Structure of an Evaluator

Eval

Base cases:

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

Recursive calls:

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

Apply

Base cases:

- Built-in primitive procedures

Recursive calls:

The Structure of an Evaluator

Eval

Base cases:

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

Recursive calls:

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

Apply

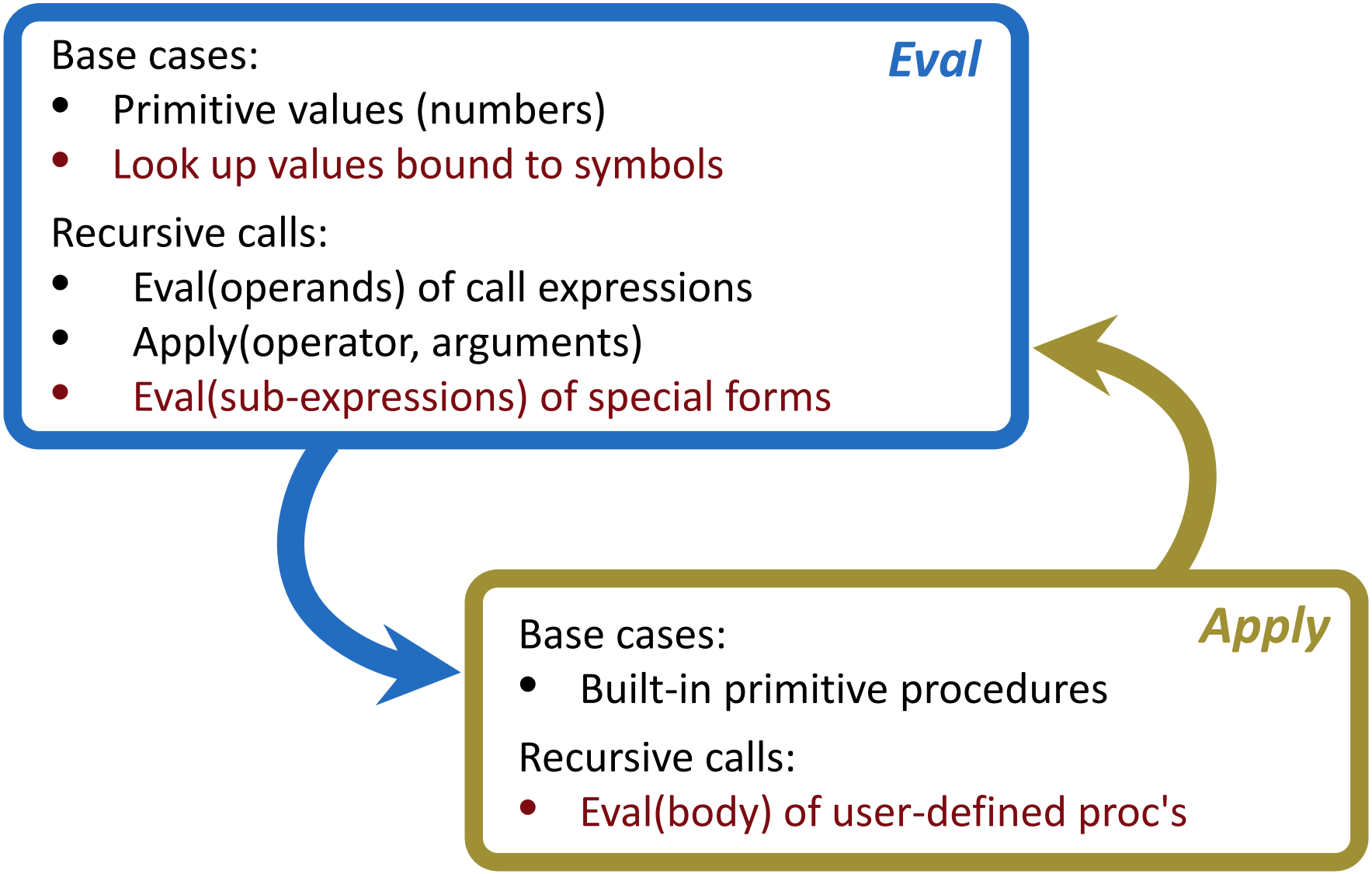
Base cases:

- Built-in primitive procedures

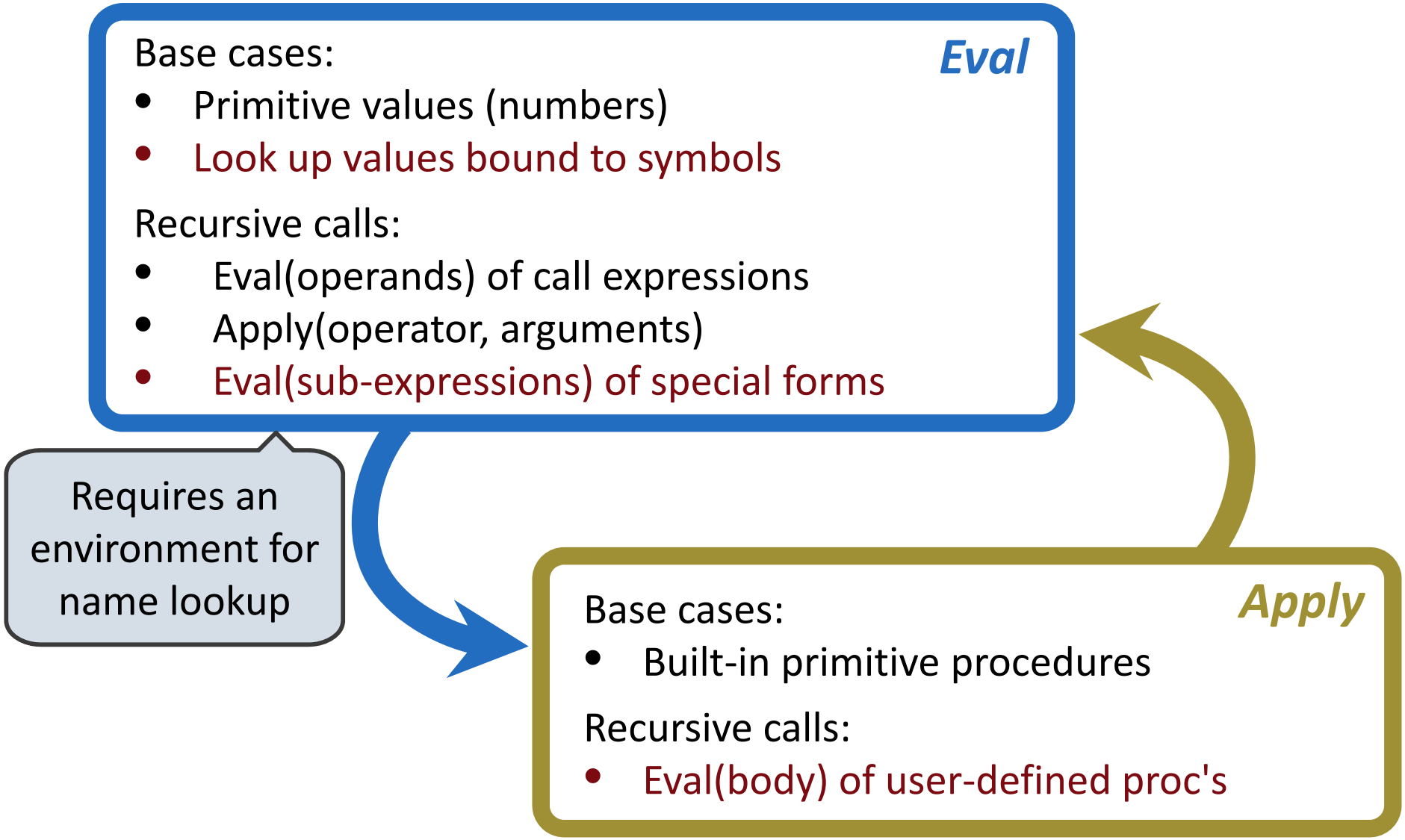
Recursive calls:

- Eval(body) of user-defined proc's

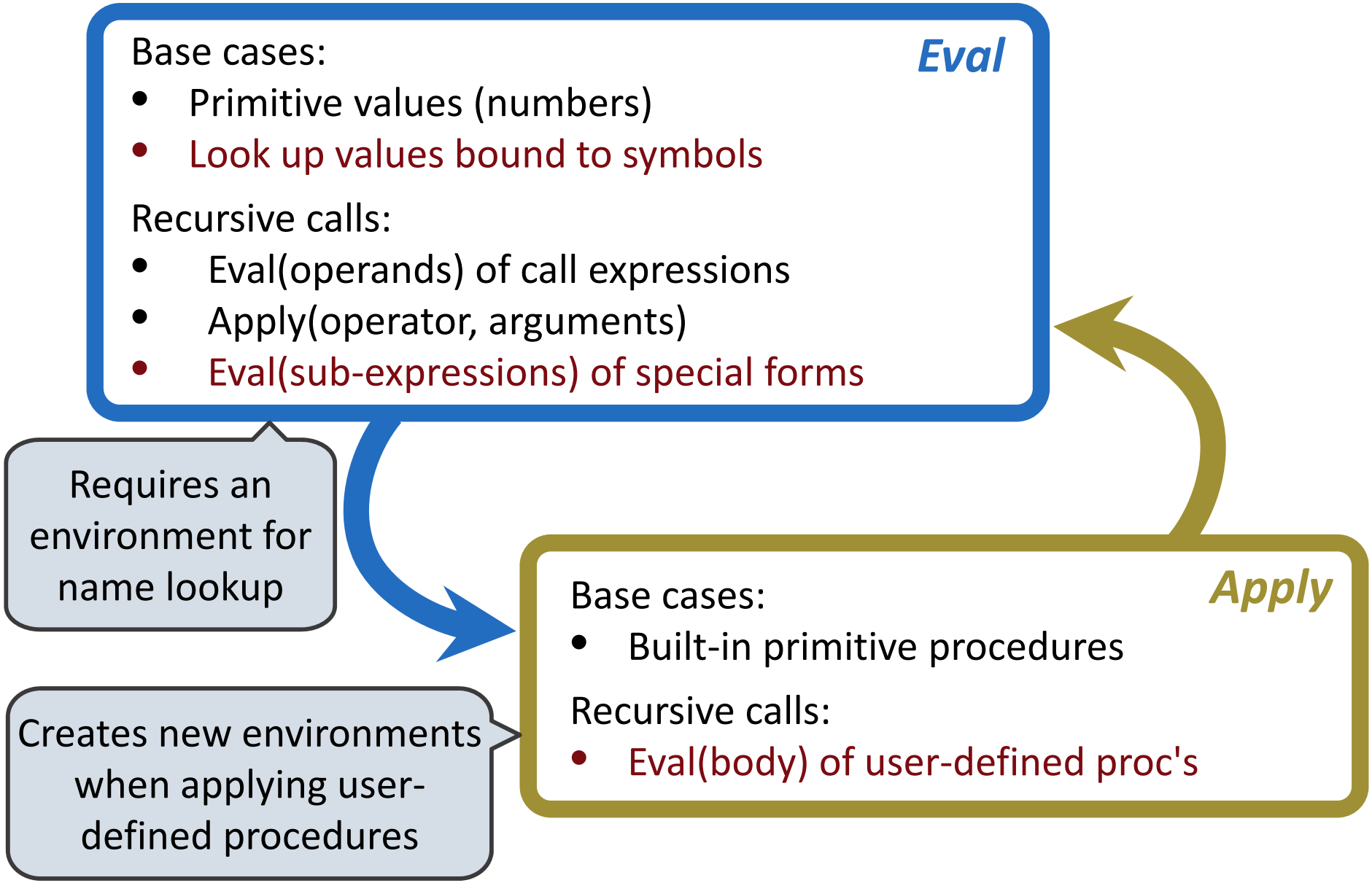
The Structure of an Evaluator



The Structure of an Evaluator



The Structure of an Evaluator



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 primitives are called atoms in Scheme

Scheme Evaluation

The **`scheme_eval`** function dispatches on expression form:

- Symbols are bound to values in the current environment
- Self-evaluating primitives are called atoms in Scheme
- All other legal expressions are represented as Scheme lists

Scheme Evaluation

The **`scheme_eval`** function dispatches on expression form:

- Symbols are bound to values in the current environment
- Self-evaluating primitives are called atoms in Scheme
- All other legal expressions are represented as Scheme lists

```
(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 primitives are called atoms in Scheme
- All other legal expressions are represented as Scheme lists

```
(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 primitives are called atoms in Scheme
- All other legal expressions are represented as Scheme lists

```
(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 primitives are called atoms in Scheme
- All other legal expressions are represented as Scheme lists

```
(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 primitives are called atoms in Scheme
- All other legal expressions are represented as Scheme lists

```
(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

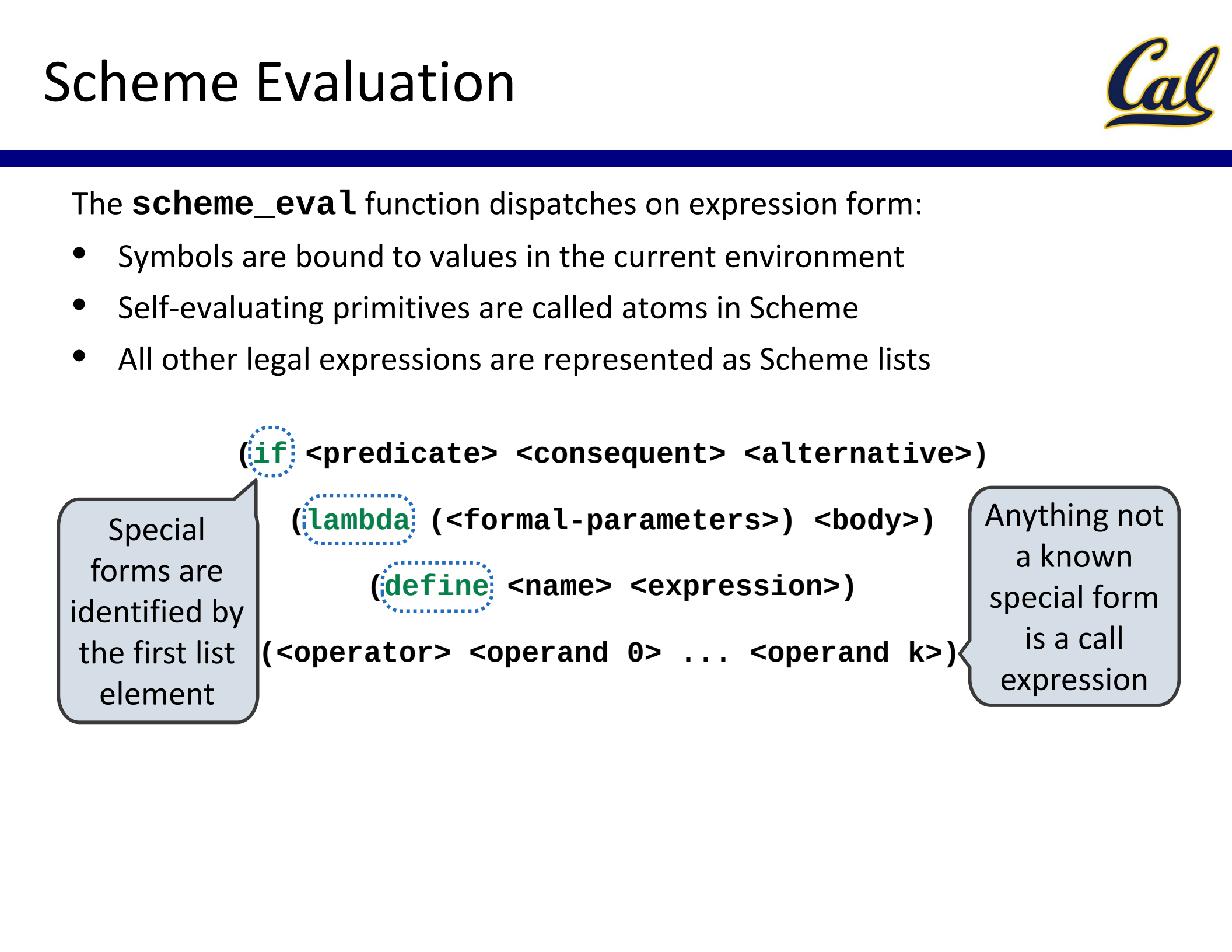
The **`scheme_eval`** function dispatches on expression form:

- Symbols are bound to values in the current environment
- Self-evaluating primitives are called atoms in Scheme
- All other legal expressions are represented as Scheme lists

```
(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 primitives are called atoms in Scheme
- All other legal expressions are represented as Scheme lists

```
(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

Anything 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 primitives are called atoms in Scheme
- All other legal expressions are represented as Scheme lists

```
(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

Anything not a known special form is a call expression

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

Scheme Evaluation

The **`scheme_eval`** function dispatches on expression form:

- Symbols are bound to values in the current environment
- Self-evaluating primitives are called atoms in Scheme
- All other legal expressions are represented as Scheme lists

```
(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

Anything not a known special form is a call expression

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

(f (list 1 2))
```

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 <e1> ... <en>)`**, **`(or <e1> ... <en>)`**

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>))`**

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.

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 <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

Logical forms may only evaluate some sub-expressions.

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

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

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

Logical Special Forms

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

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.

scheme_eval

Quotation

Quotation

The **quote** special form evaluates to the quoted expression

Quotation

The **quote** special form evaluates to the quoted expression

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

Quotation

The **quote** special form evaluates to the quoted expression

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

Evaluates to the **<expression>** itself, not its value!

Quotation

The **quote** special form evaluates to the quoted expression

(quote <expression>)

Evaluates to the **<expression>** itself, not its value!

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

Quotation

The **quote** special form evaluates to the quoted expression

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

Evaluates to the **<expression>** itself, not its value!

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

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

Quotation

The **quote** special form evaluates to the quoted expression

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

Evaluates to the **<expression>** itself, not its value!

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

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

```
'(1 2)
```

Quotation

The **quote** special form evaluates to the quoted expression

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

Evaluates to the **<expression>** itself, not its value!

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

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

```
'(1 2)
```

The **scheme_read** parser converts shorthand to a combination

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

A scheme list of symbols

A scheme expression

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`**

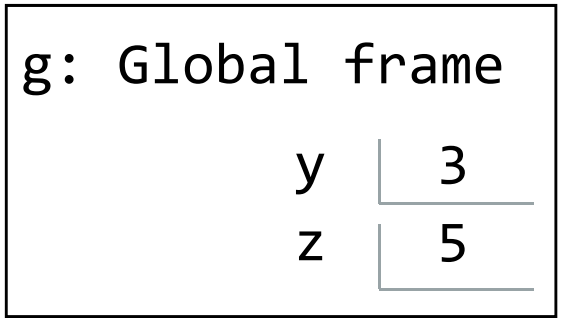
In Project 4, **`Frames`** do not hold return values

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

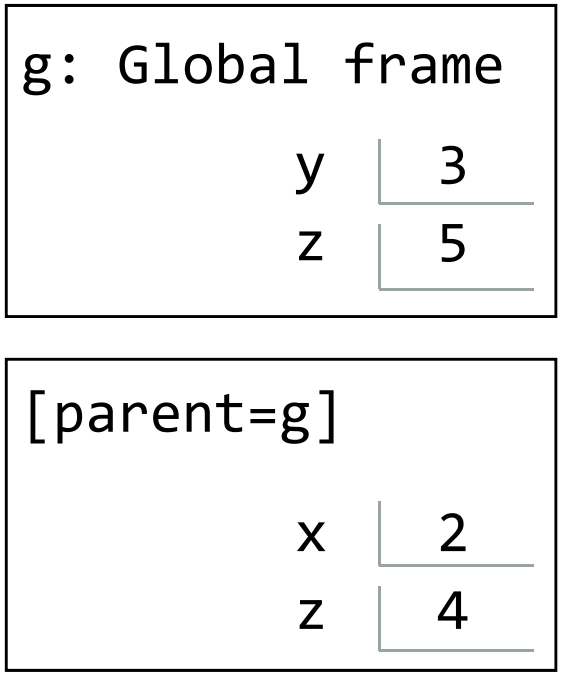


Frames and Environments

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



Define Expressions

Define Expressions

Define expressions bind a symbol to a value in the first frame of the current environment

Define Expressions

Define expressions bind a symbol to a value in the first frame of the current environment

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

Define Expressions

Define expressions bind a symbol to a value in the first frame of the current environment

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

Evaluate the **<expression>**

Define Expressions

Define expressions bind a symbol to a value in the first frame of the current environment

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

Evaluate the **`<expression>`**

Bind **`<name>`** to the result (**`define`** method of the current **`Frame`**)

Define Expressions

Define expressions bind a symbol to a value in the first frame of the current environment

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

Evaluate the **<expression>**

Bind **<name>** to the result (**define** method of the current **Frame**)

```
(define x 2)
```

Define Expressions

Define expressions bind a symbol to a value in the first frame of the current environment

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

Evaluate the **<expression>**

Bind **<name>** to the result (**define** method of the current **Frame**)

```
(define x 2)
```

Procedure definition is a combination of define and lambda

Define Expressions

Define expressions bind a symbol to a value in the first frame of the current environment

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

Evaluate the **<expression>**

Bind **<name>** to the result (**define** method of the current **Frame**)

```
(define x 2)
```

Procedure definition is a combination of define and lambda

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

Define Expressions

Define expressions bind a symbol to a value in the first frame of the current environment

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

Evaluate the **<expression>**

Bind **<name>** to the result (**define** method of the current **Frame**)

```
(define x 2)
```

Procedure definition is a combination of define and lambda

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

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

Applying User-Defined Procedures

Applying User-Defined Procedures

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

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

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

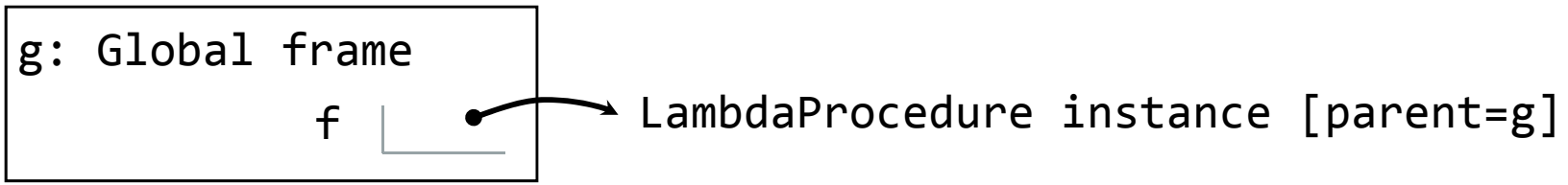
```
(define (f s) (if (null? s) '(3) (cons (car s) (f (cdr s)))))
```

Applying User-Defined Procedures

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



Applying User-Defined Procedures

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

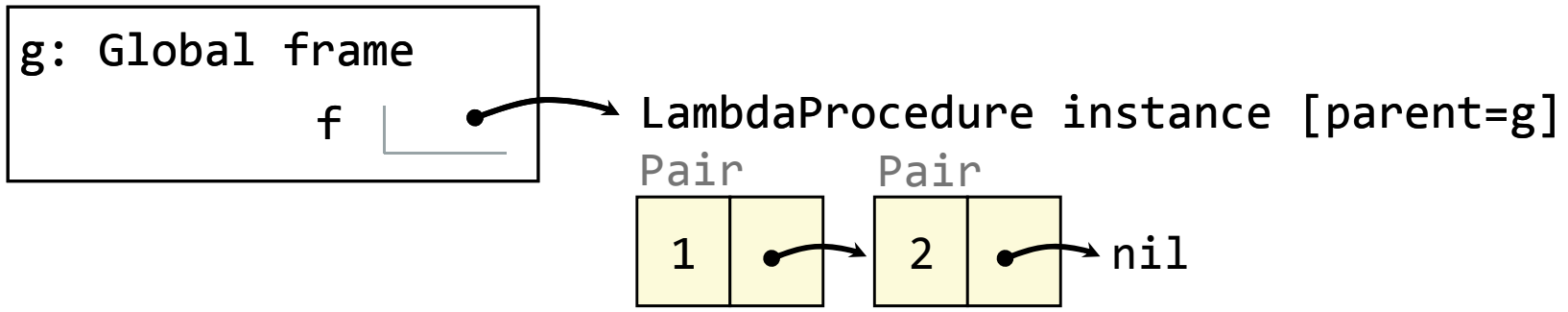
```
(define (f s) (if (null? s) '(3) (cons (car s) (f (cdr s)))))

(f (list 1 2))
```

Applying User-Defined Procedures

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 (f s) (if (null? s) '(3) (cons (car s) (f (cdr s)))))

(f (list 1 2))
```

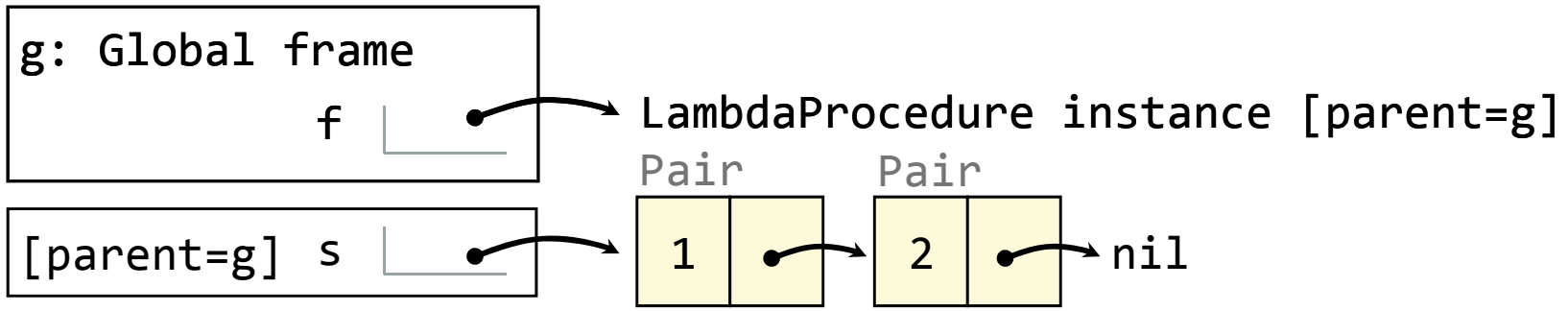


Applying User-Defined Procedures

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 (f s) (if (null? s) '(3) (cons (car s) (f (cdr s)))))

(f (list 1 2))
```

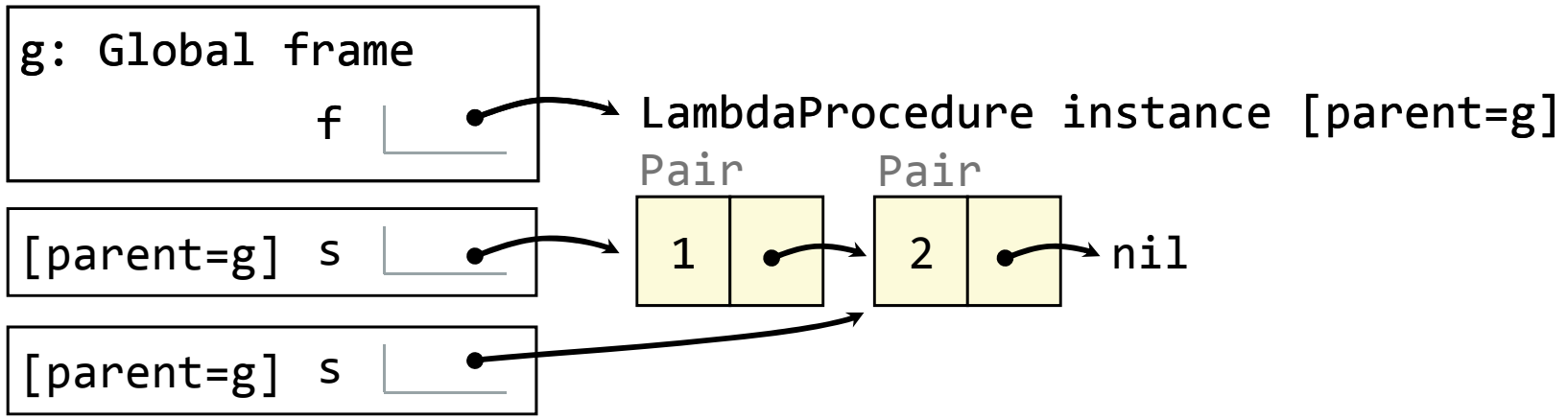


Applying User-Defined Procedures

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 (f s) (if (null? s) '(3) (cons (car s) (f (cdr s)))))

(f (list 1 2))
```

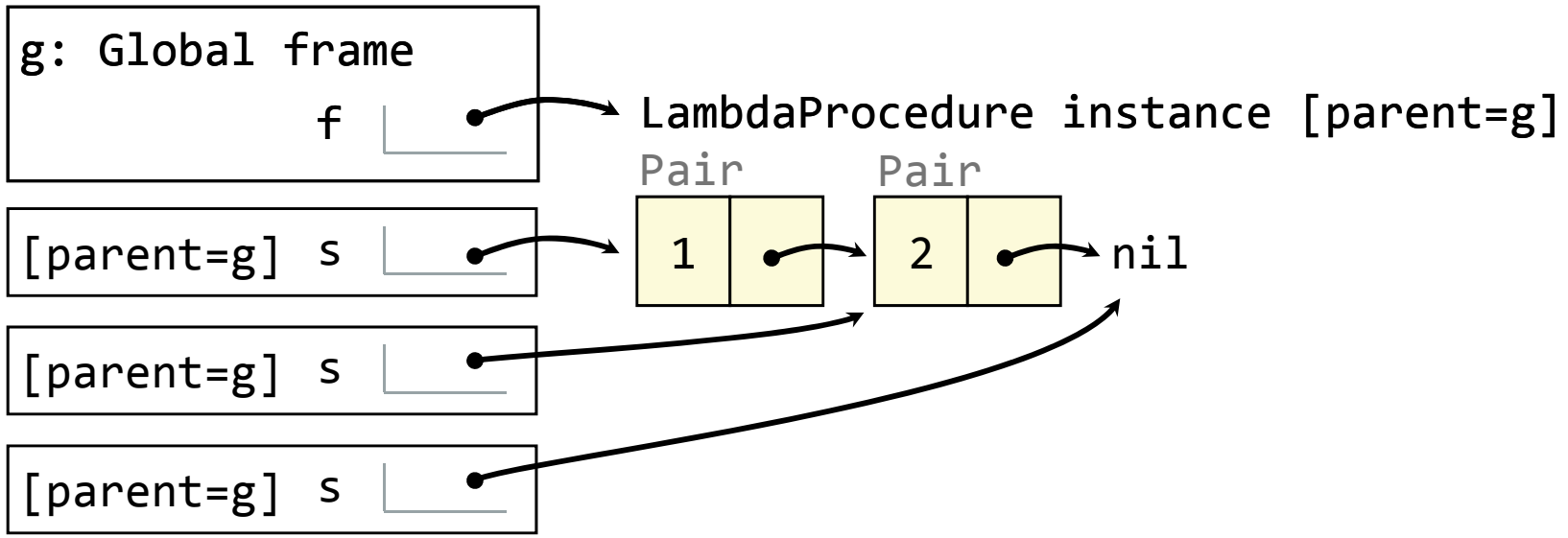


Applying User-Defined Procedures

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 (f s) (if (null? s) '(3) (cons (car s) (f (cdr s)))))

(f (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

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