WELCOME TO INTERPRETERS TWO

July 28, 2016

A Lecture by
Neil Agarwal

the plan

review.

special forms.

environments.

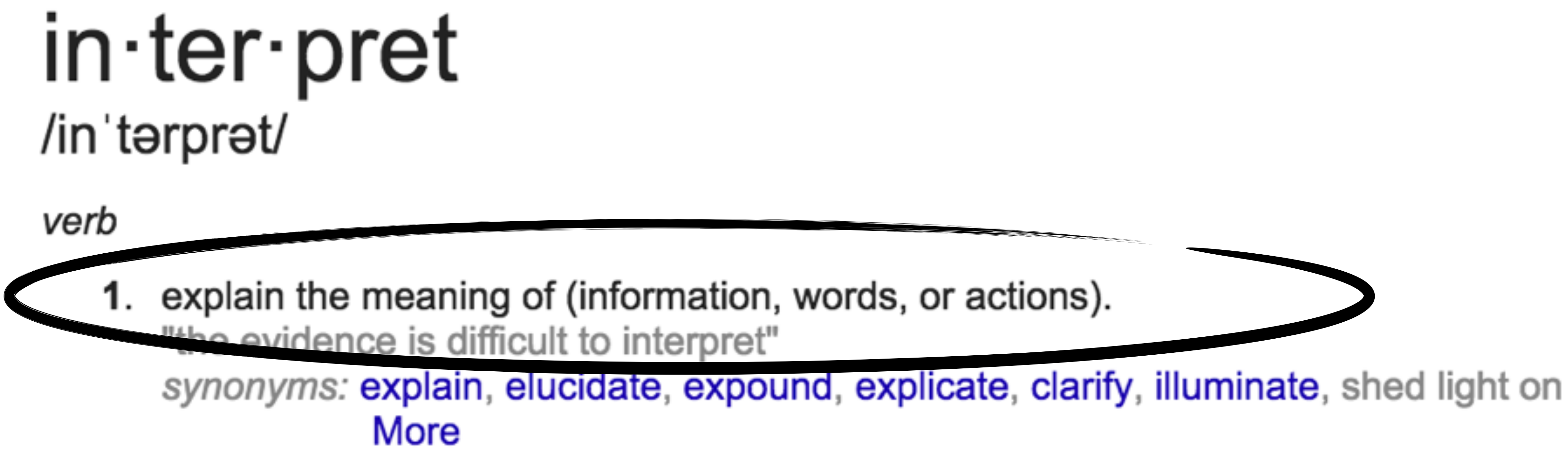
sounds

good?

wait.

but

why?



in·ter·pret

/inˈtərprət/

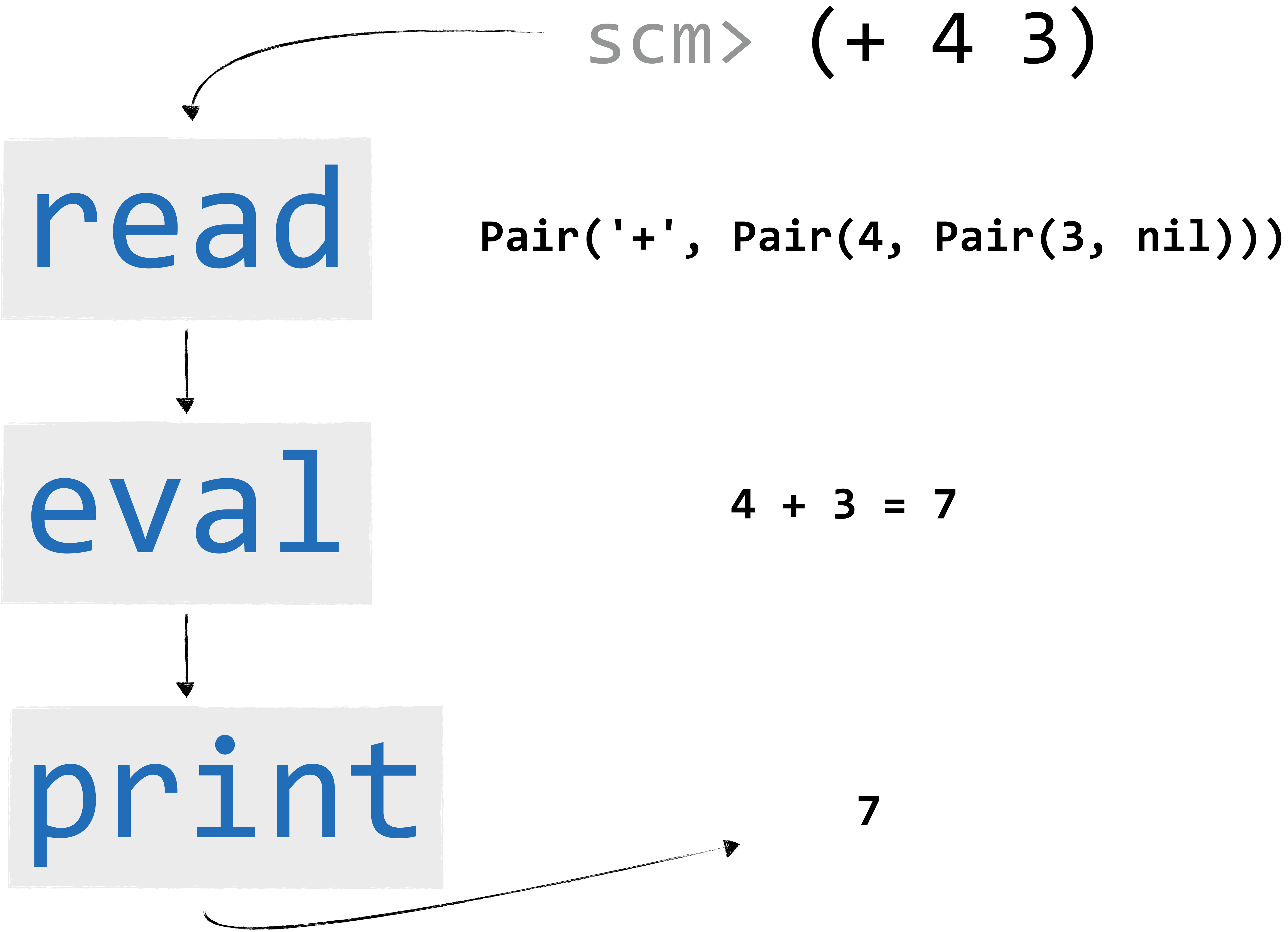
verb

1. explain the meaning of (information, words, or actions).
"the evidence is difficult to interpret"
synonyms: explain, elucidate, expound, explicate, clarify, illuminate, shed light on
More

quick note

code in today's lecture is intended to act as a transition from yesterday's lecture to the project - this code will NOT work for the project

review



scm> (+ 4 3)

read

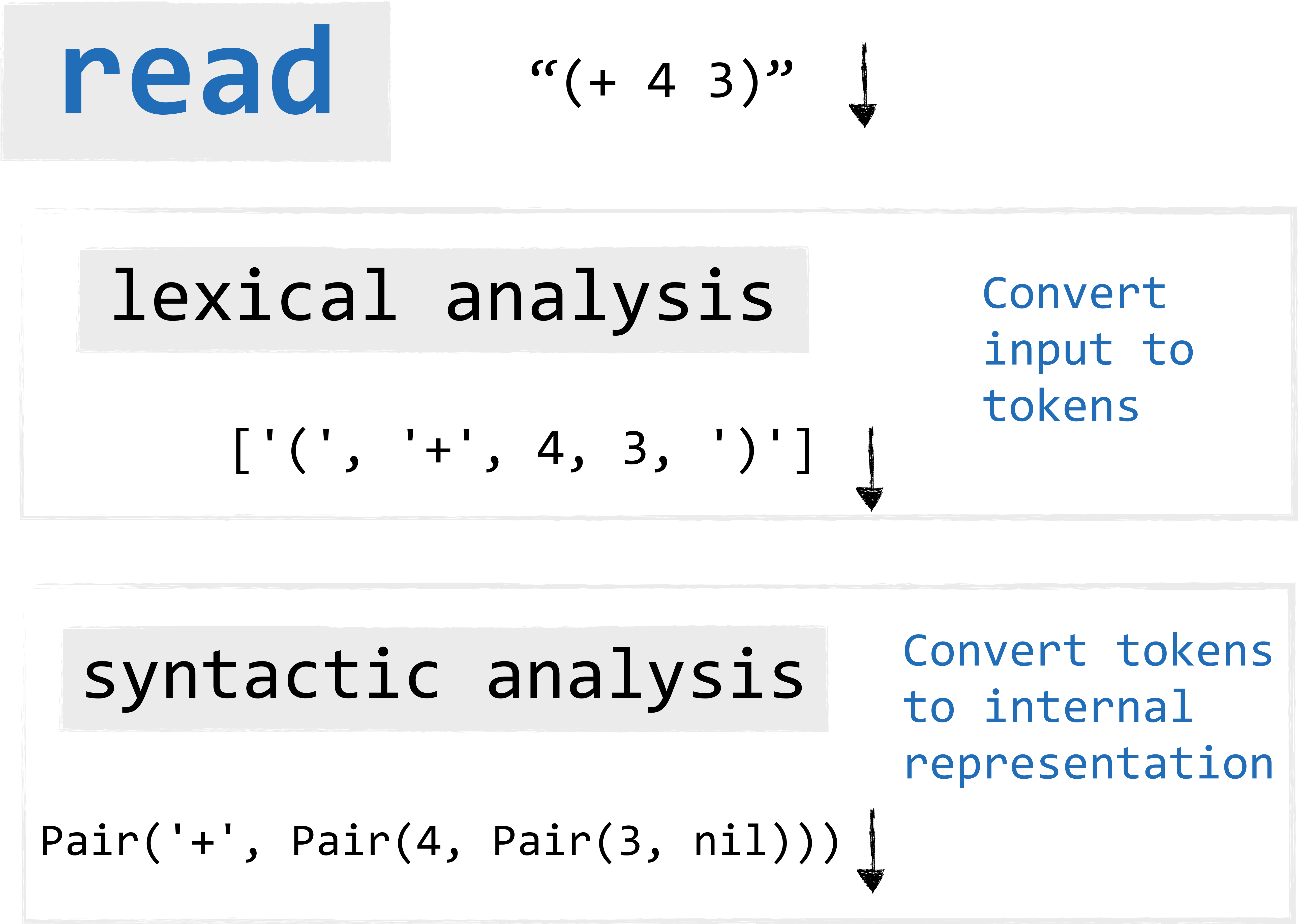
Pair('+', Pair(4, Pair(3, nil)))

eval

4 + 3 = 7

print

7



read

“(+ 4 3)”

lexical analysis

Convert input to tokens

```
['(', '+', 4, 3, ')']
```

syntactic analysis

Convert tokens to internal representation

```
Pair('+', Pair(4, Pair(3, nil)))
```

SYNTACTIC ANALYSIS: Parsing Scheme

```
def read_exp(tokens):
    token = tokens.pop(0)
    if token == '(':
        exp = read_tail(tokens)
        if exp is nil:
            raise error
        return exp
    elif token == ')':
        raise error
    else:
        return token
```

```
def read_tail(tokens):
    if tokens[0] == ')':
        tokens.pop(0)
        return nil
    first = read_exp(tokens)
    rest = read_tail(tokens)
    return Pair(first, rest)
```

```
>>> tokens = ['(', '+', 4, 3, ')']
>>> read_exp(tokens)
Pair('+', Pair(4, Pair(3, nil)))
```

SYNTACTIC ANALYSIS: Parsing Scheme

```
def read_exp(exp):
  """Returns the first calculator expression."""
  ...

def read_tail(tokens):
  """Reads up to the first mismatched close parenthesis."""
  ...
```

```
['(', '+', 4, 3, ')']
```

SYNTACTIC ANALYSIS: Parsing Scheme

```
def read_exp(exp):
  """Returns the first calculator expression."""
  ...

def read_tail(tokens):
  """Reads up to the first mismatched close parenthesis."""
  ...
```

- ['(', '+', 4, 3, ')']

Resulting expression:

SYNTACTIC ANALYSIS: Parsing Scheme

```
def read_exp(exp):
  """Returns the first calculator expression."""
  ...

def read_tail(tokens):
  """Reads up to the first mismatched close parenthesis."""
  ...
```

▶ ['+', 4, 3, ')']

Resulting expression: Pair(

SYNTACTIC ANALYSIS: Parsing Scheme

```
def read_exp(exp):
  """Returns the first calculator expression."""
  ...

def read_tail(tokens):
  """Reads up to the first mismatched close parenthesis."""
  ...
```

▶ `['+', 4, 3, ')']`

Resulting expression: Pair(

SYNTACTIC ANALYSIS: Parsing Scheme

```
def read_exp(exp):
 """Returns the first calculator expression."""
 ...

def read_tail(tokens):
 """Reads up to the first mismatched close parenthesis."""
 ...
```

▶ [4, 3, ')']

Resulting expression: Pair(‘+’

SYNTACTIC ANALYSIS: Parsing Scheme

```
def read_exp(exp):
 """Returns the first calculator expression."""
 ...

def read_tail(tokens):
 """Reads up to the first mismatched close parenthesis."""
 ...
```

▶ [4, 3, ')']

Resulting expression: Pair(‘+’

SYNTACTIC ANALYSIS: Parsing Scheme

```
def read_exp(exp):
 """Returns the first calculator expression."""
 ...

def read_tail(tokens):
 """Reads up to the first mismatched close parenthesis."""
 ...
```

▶ [3, ')']

Resulting expression: Pair(‘+’, Pair(4

SYNTACTIC ANALYSIS: Parsing Scheme

```
def read_exp(exp):
  """Returns the first calculator expression."""
  ...

def read_tail(tokens):
  """Reads up to the first mismatched close parenthesis."""
  ...
```

▶ [3, ')']

Resulting expression: Pair(‘+’, Pair(4

SYNTACTIC ANALYSIS: Parsing Scheme

```
def read_exp(exp):
 """Returns the first calculator expression."""
 ...

def read_tail(tokens):
 """Reads up to the first mismatched close parenthesis."""
 ...
```

▶ [')']

Resulting expression: Pair(‘+’, Pair(4, Pair(3

SYNTACTIC ANALYSIS: Parsing Scheme

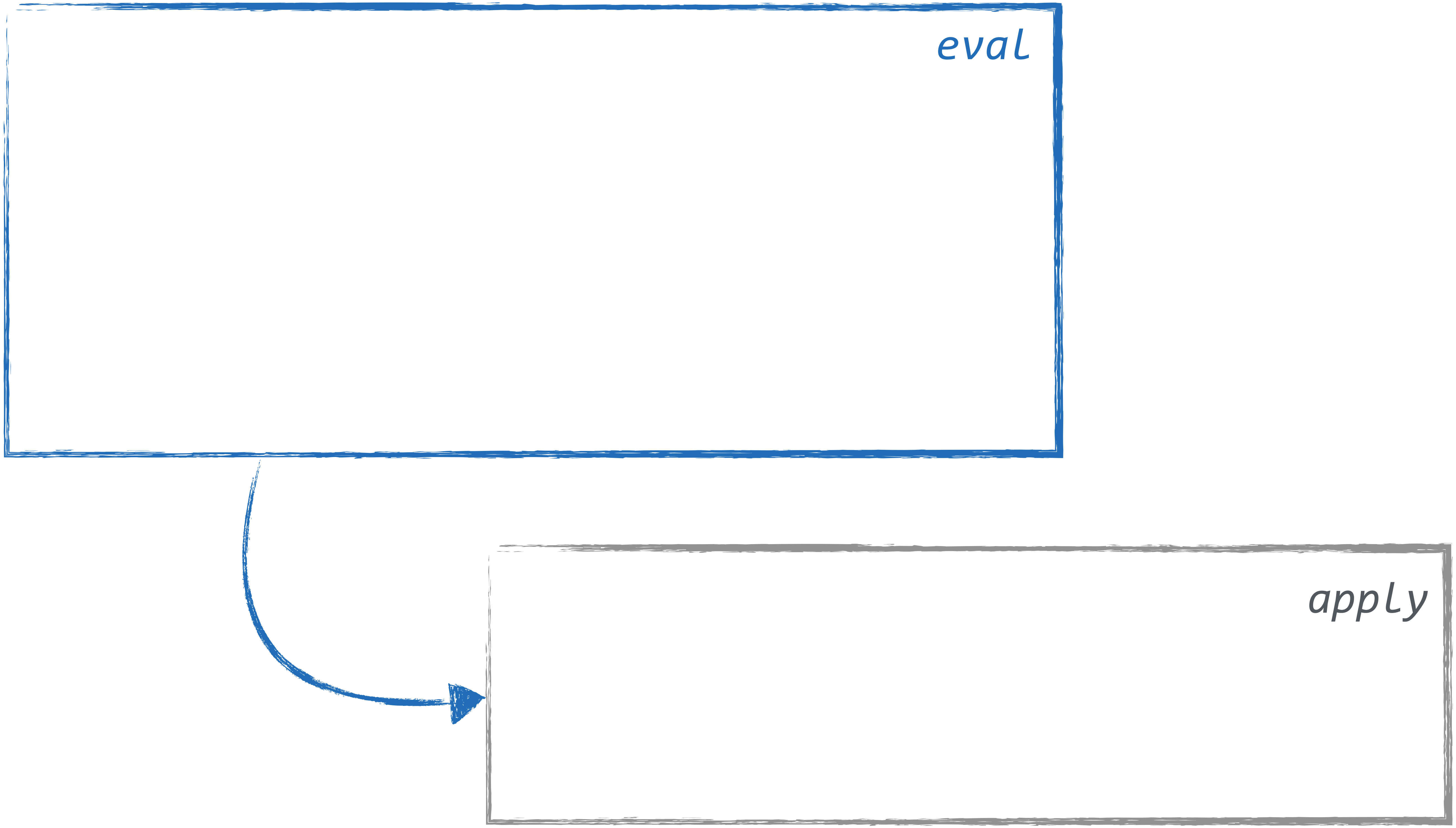
```
def read_exp(exp):
 """Returns the first calculator expression."""
 ...

def read_tail(tokens):
 """Reads up to the first mismatched close parenthesis."""
 ...
```

[]

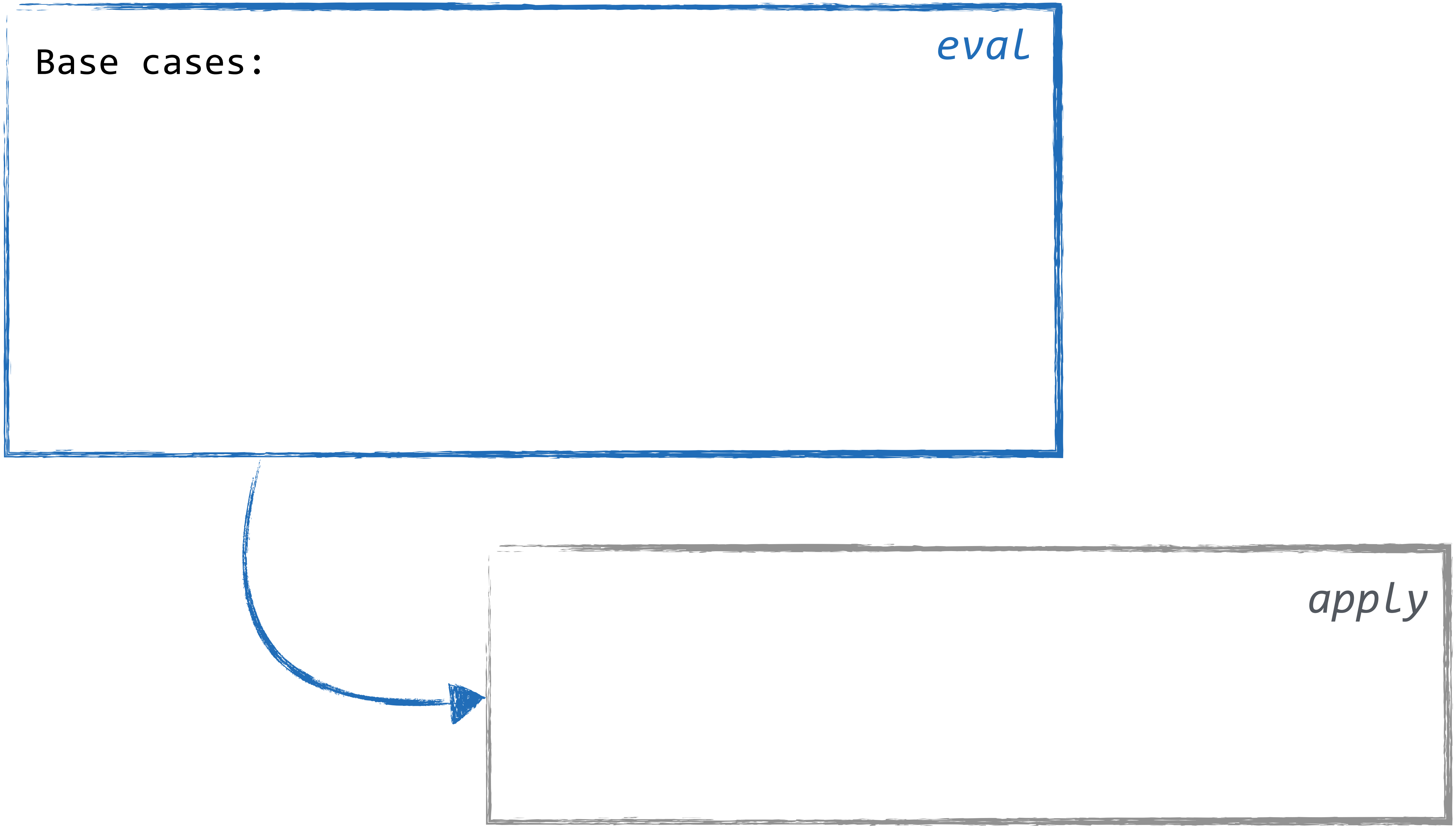
Resulting expression: Pair(‘+’, Pair(4, Pair(3, nil)))

eval and apply

eval and apply



eval and apply

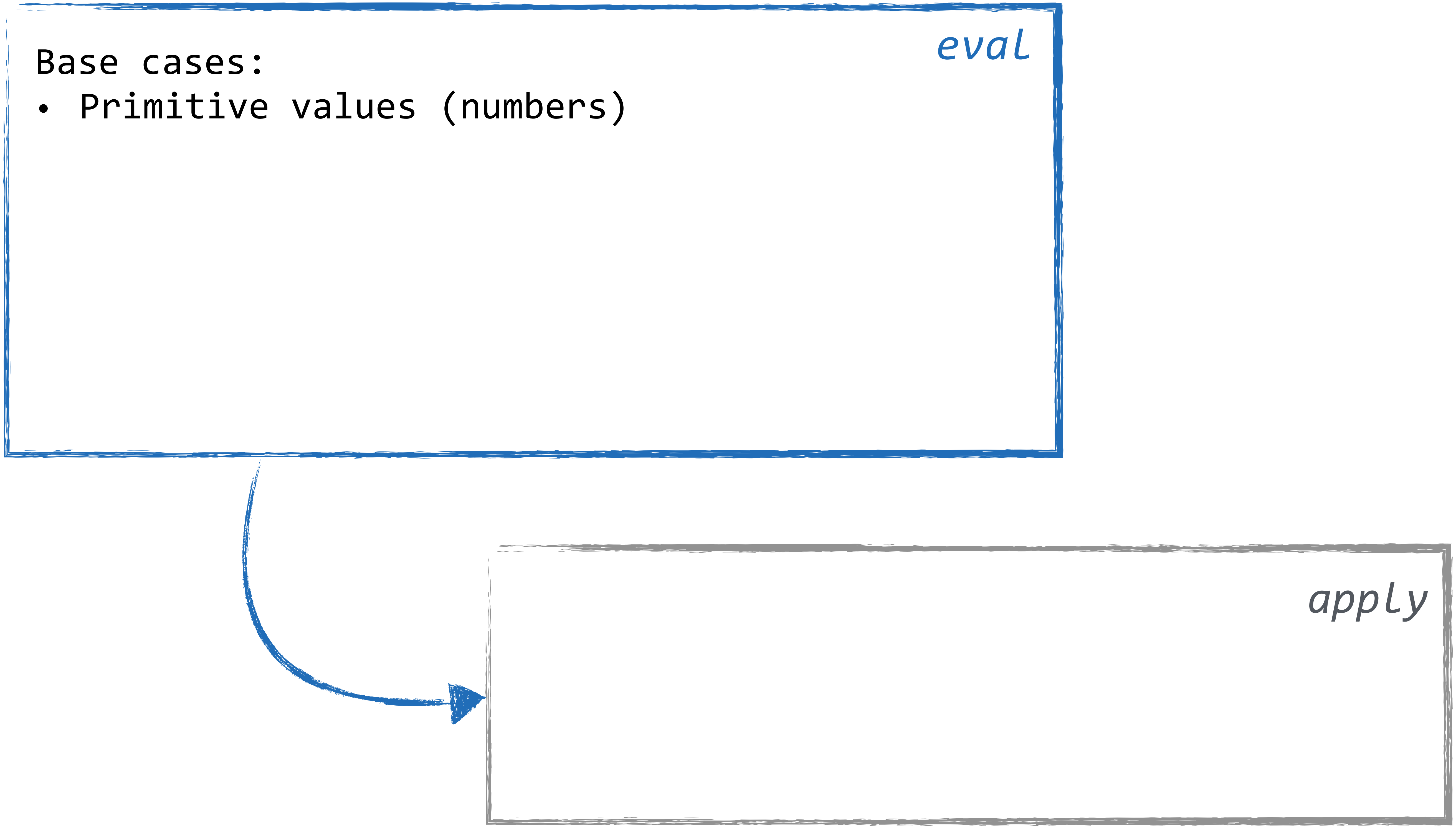


eval

Base cases:

apply

eval and apply



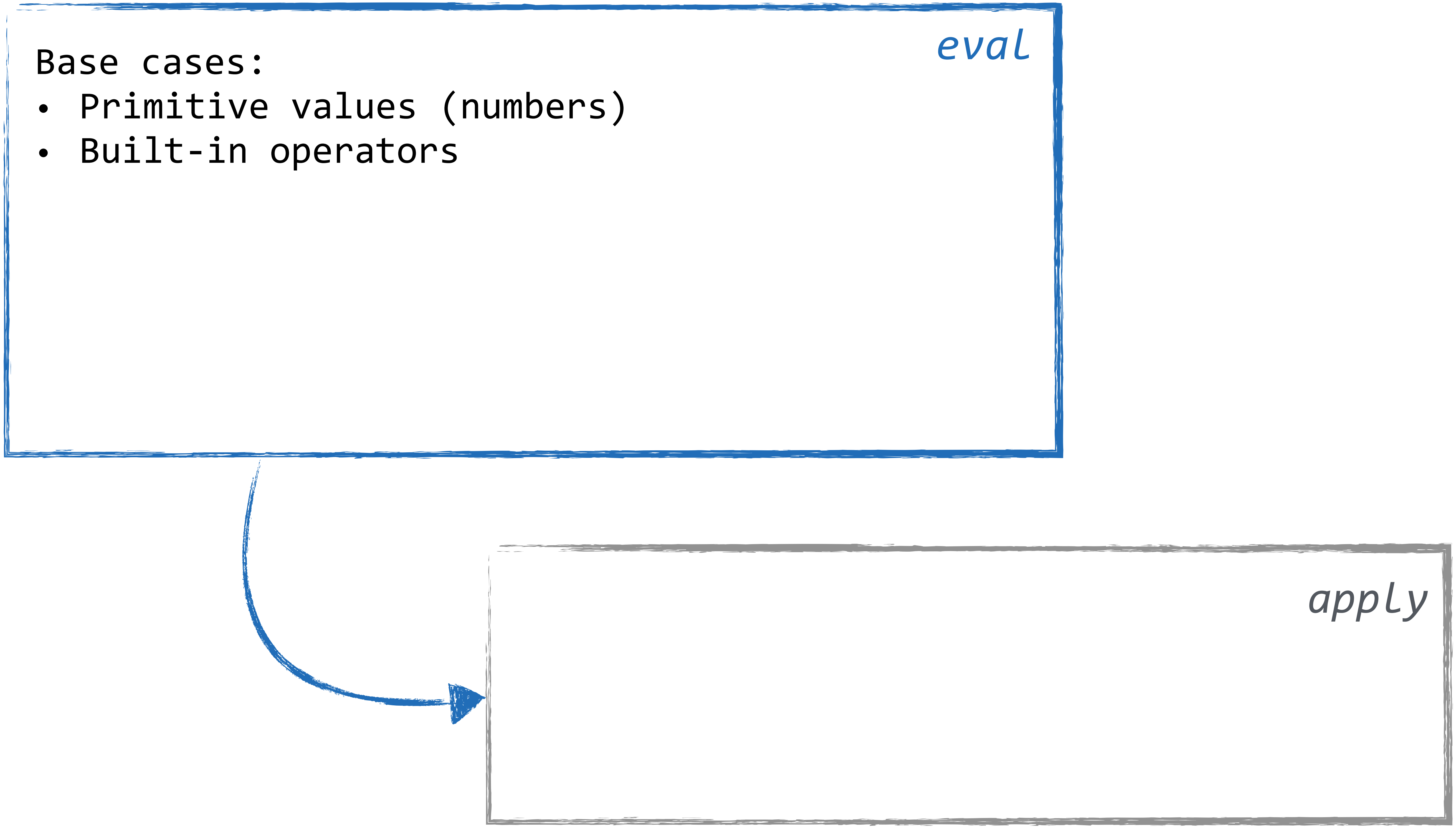
eval

Base cases:

- Primitive values (numbers)

apply

eval and apply



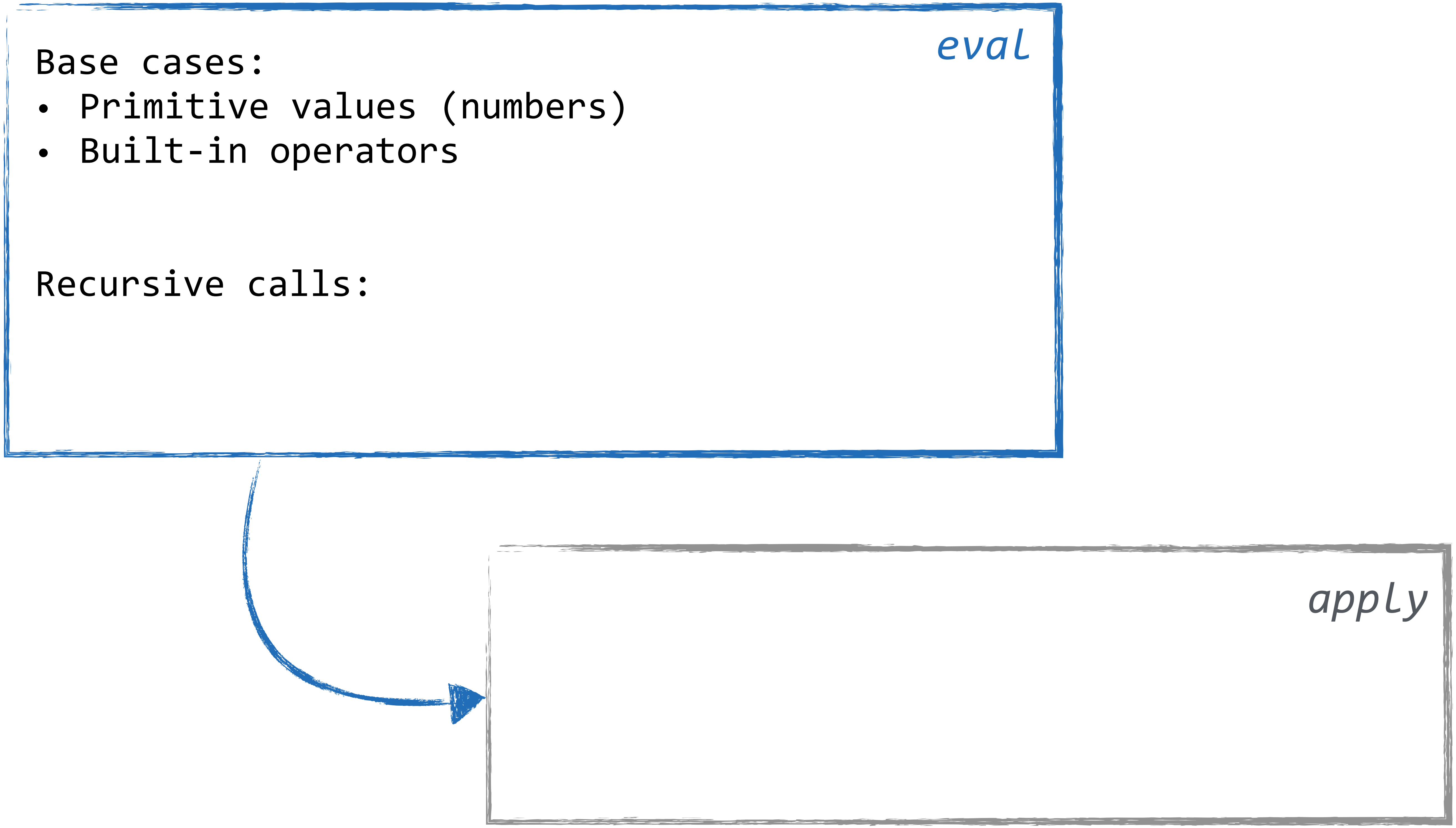
eval

Base cases:

- Primitive values (numbers)
- Built-in operators

apply

eval and apply



eval

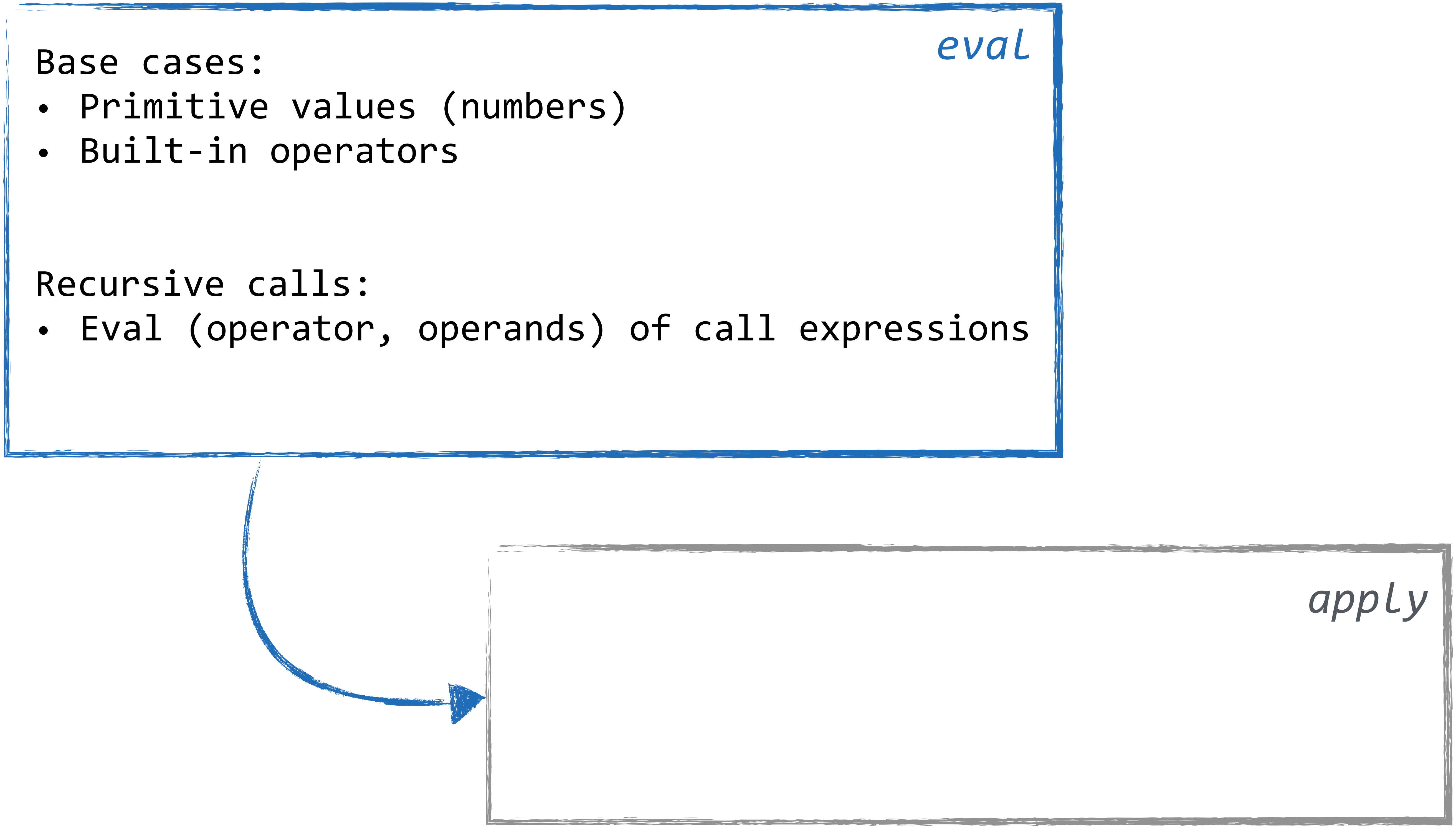
Base cases:

- Primitive values (numbers)
- Built-in operators

Recursive calls:

apply

eval and apply



eval

Base cases:

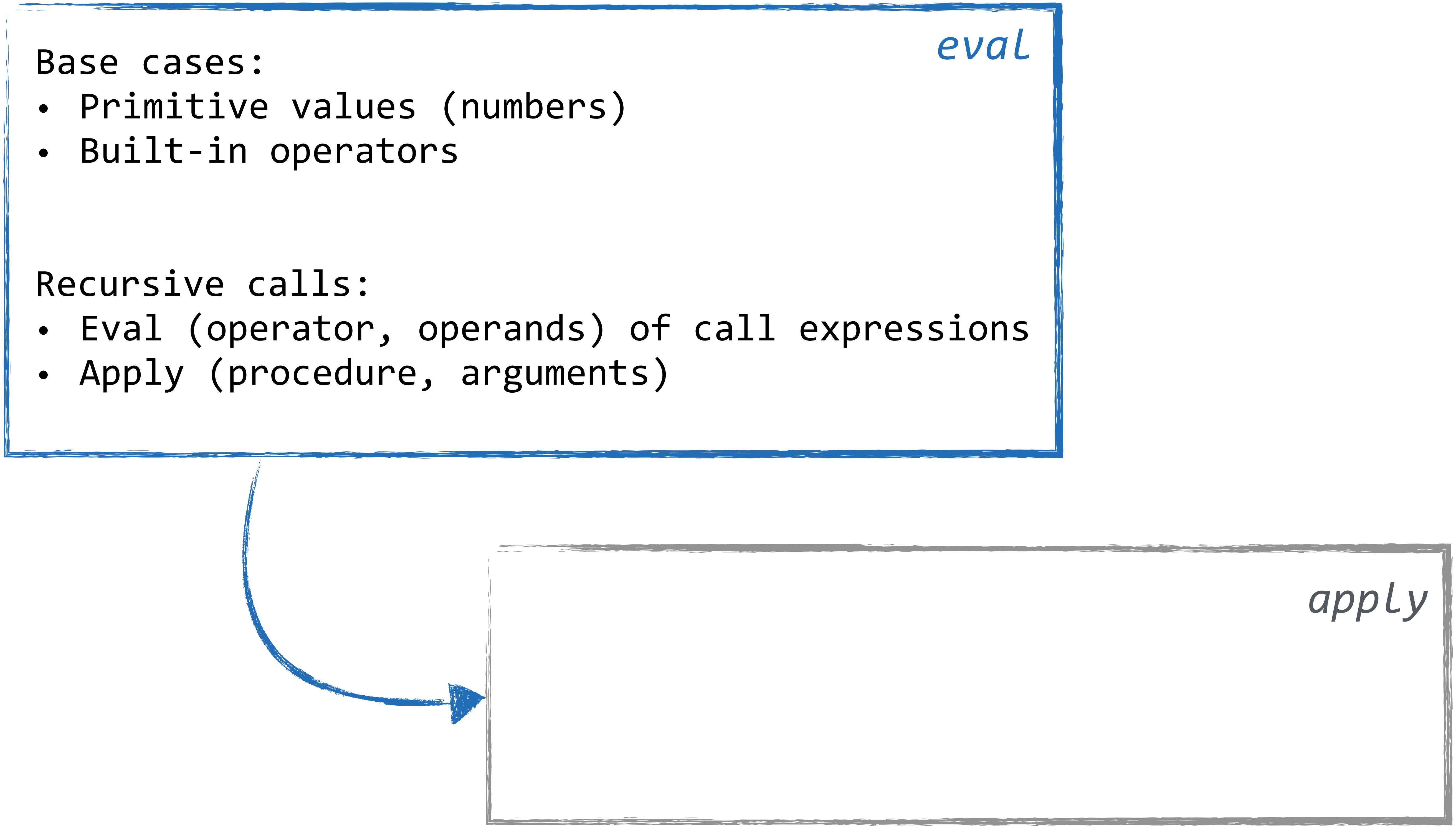
- Primitive values (numbers)
- Built-in operators

Recursive calls:

- Eval (operator, operands) of call expressions

apply

eval and apply



eval

Base cases:

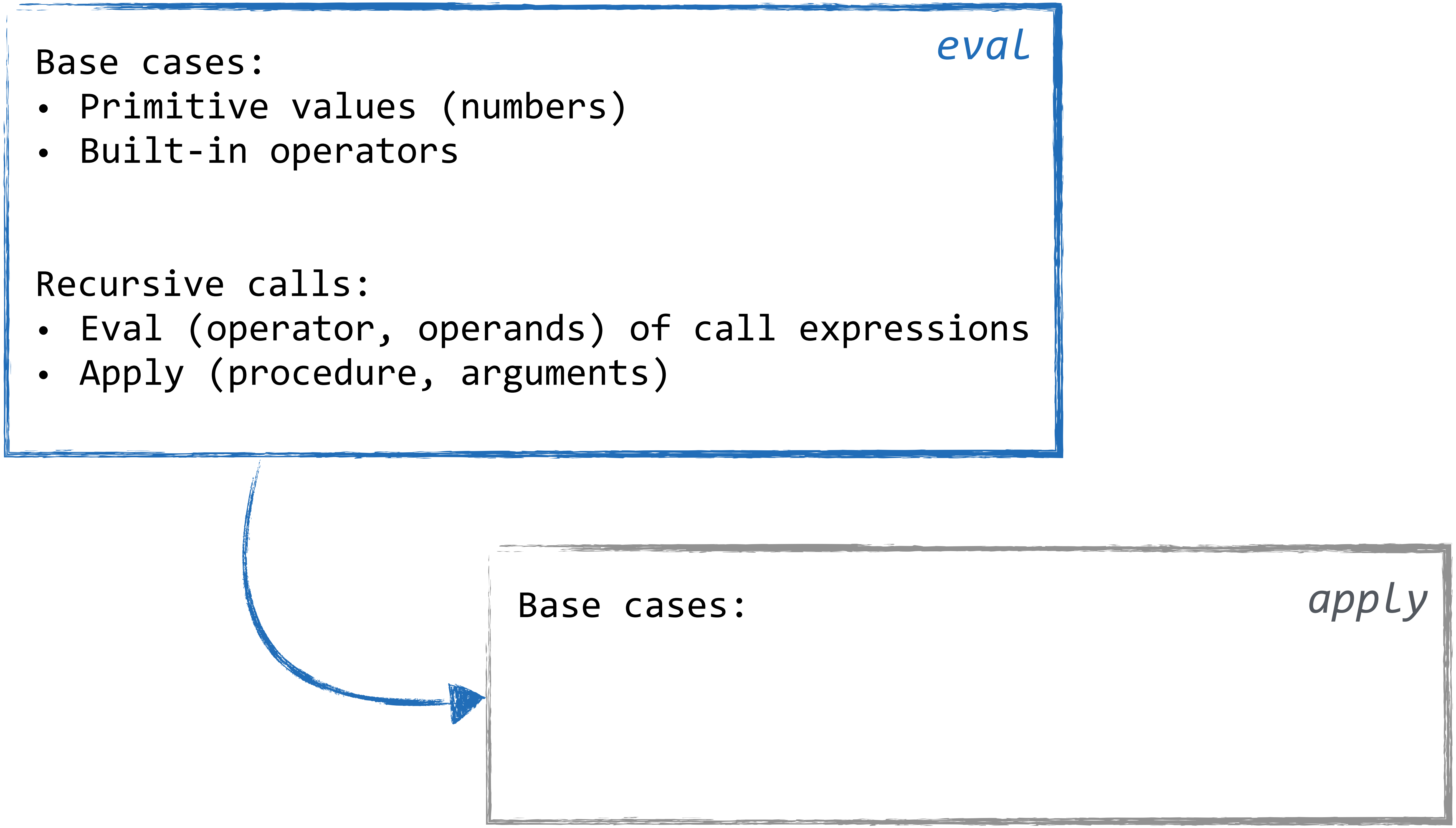
- Primitive values (numbers)
- Built-in operators

Recursive calls:

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

apply

eval and apply



eval

Base cases:

- Primitive values (numbers)
- Built-in operators

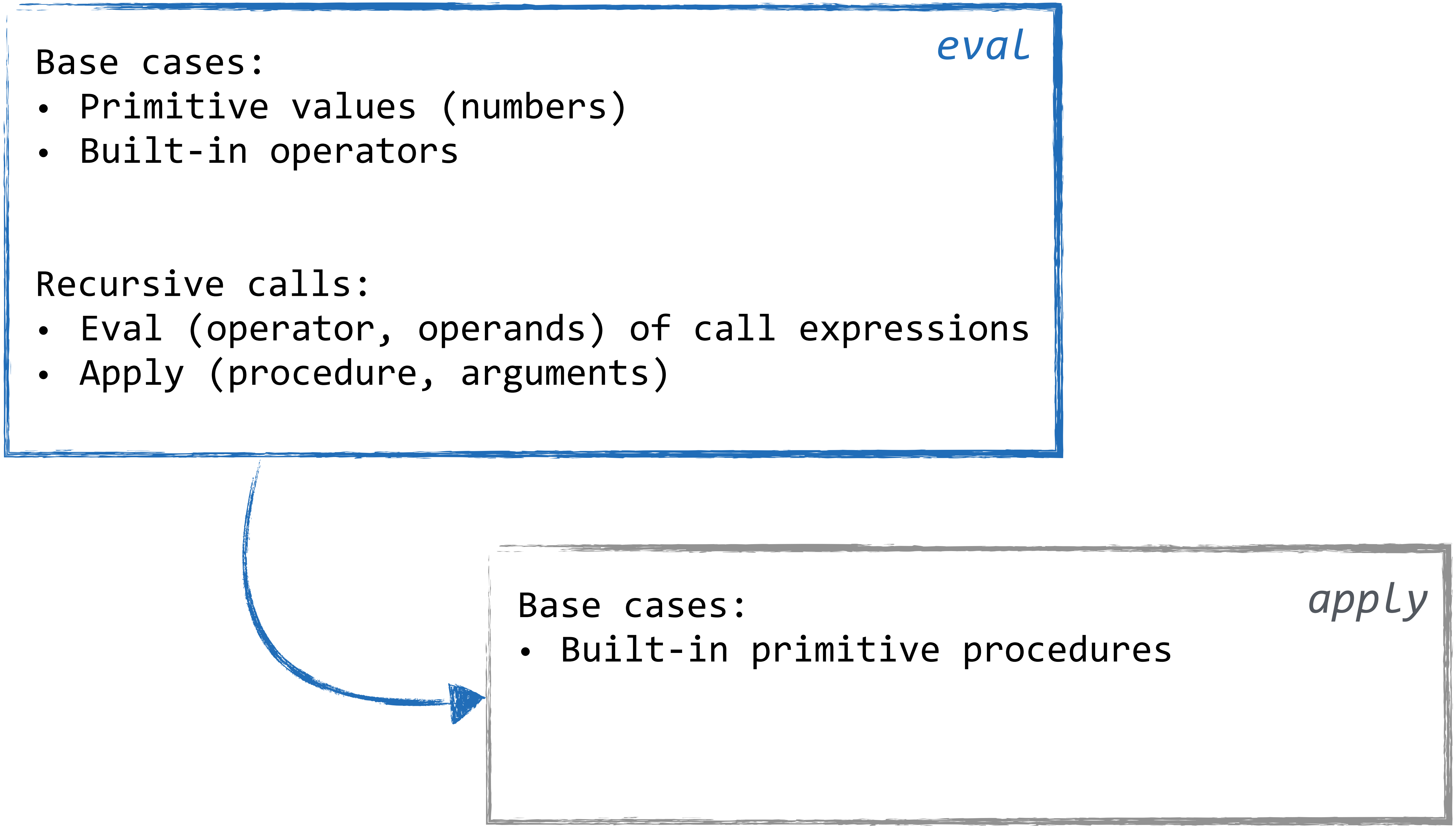
Recursive calls:

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

apply

Base cases:

eval and apply



eval

Base cases:

- Primitive values (numbers)
- Built-in operators

Recursive calls:

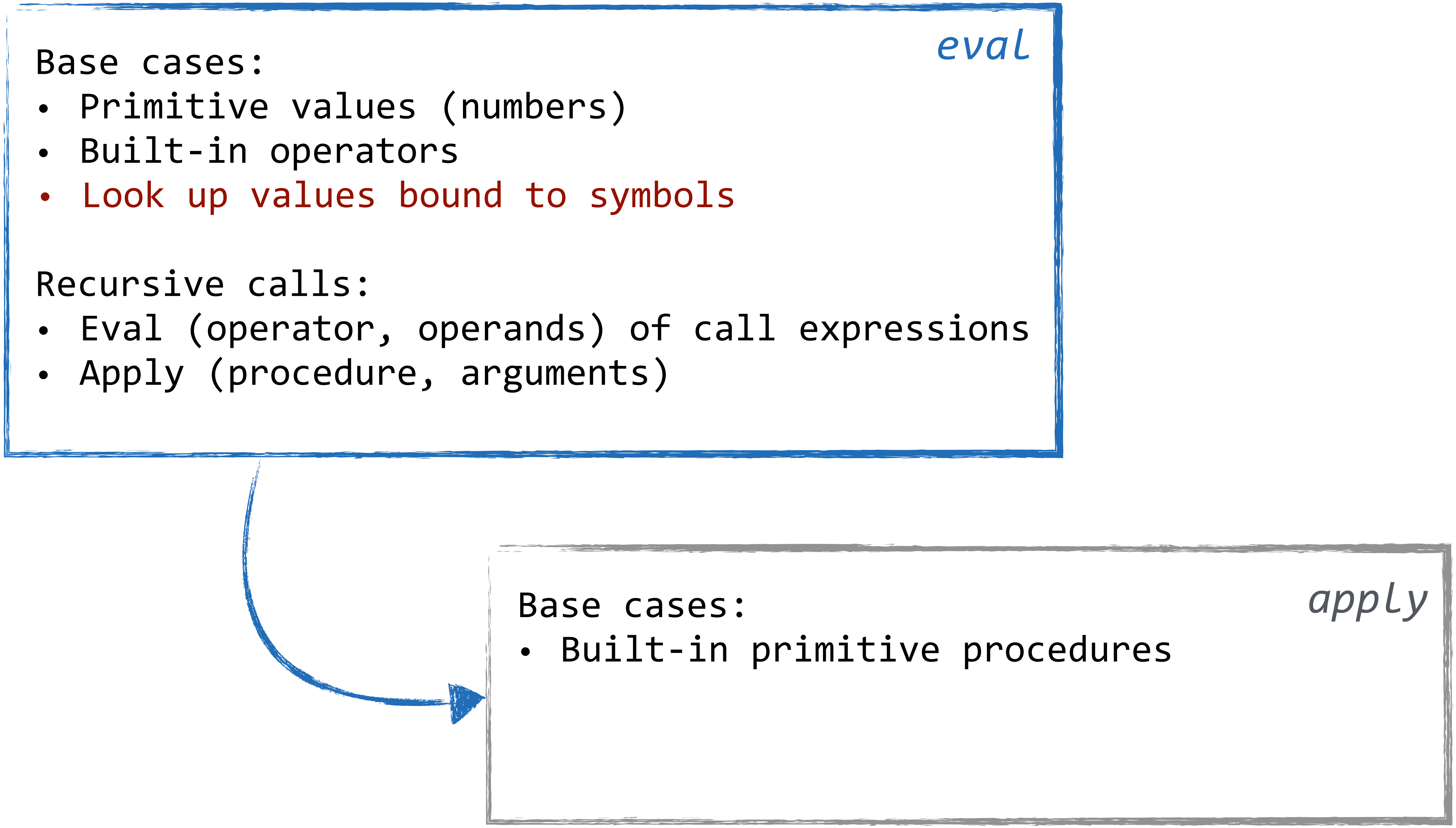
- Eval (operator, operands) of call expressions
- Apply (procedure, arguments)

apply

Base cases:

- Built-in primitive procedures

eval and apply



eval

Base cases:

- Primitive values (numbers)
- Built-in operators
- Look up values bound to symbols

Recursive calls:

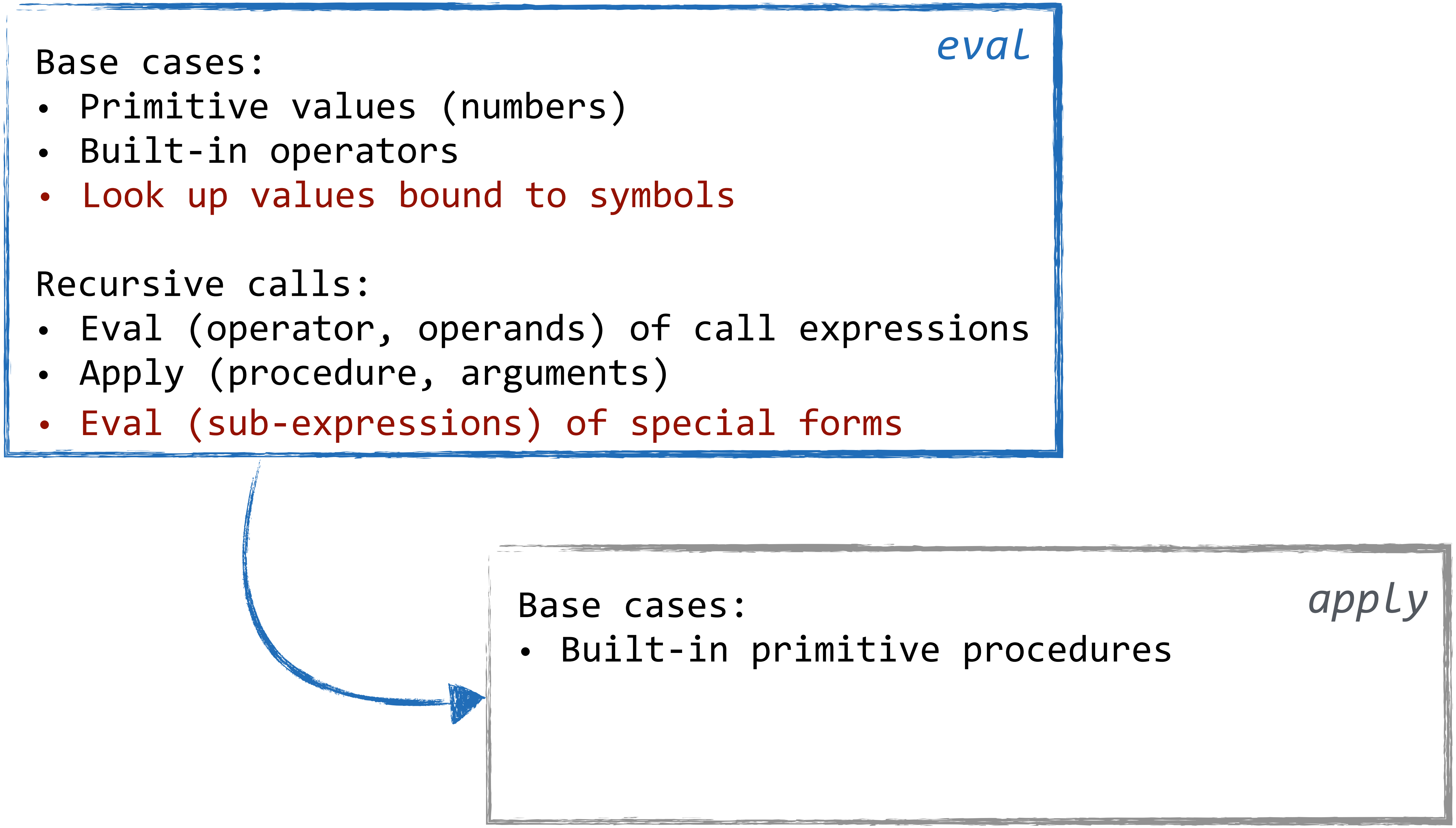
- Eval (operator, operands) of call expressions
- Apply (procedure, arguments)

apply

Base cases:

- Built-in primitive procedures

eval and apply



eval

Base cases:

- Primitive values (numbers)
- Built-in operators
- Look up values bound to symbols

Recursive calls:

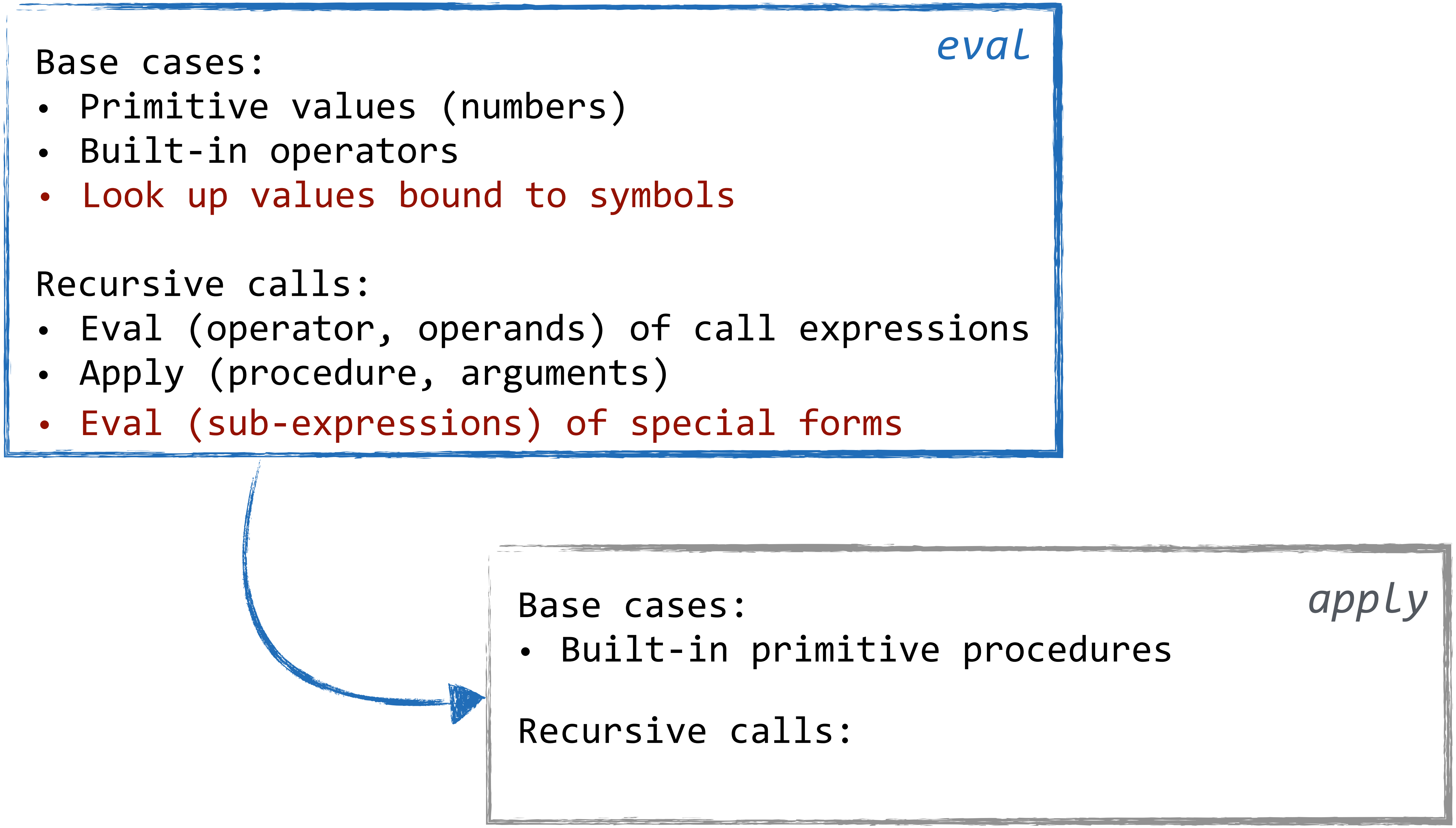
- Eval (operator, operands) of call expressions
- Apply (procedure, arguments)
- Eval (sub-expressions) of special forms

apply

Base cases:

- Built-in primitive procedures

eval and apply



eval

Base cases:

- Primitive values (numbers)
- Built-in operators
- Look up values bound to symbols

Recursive calls:

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

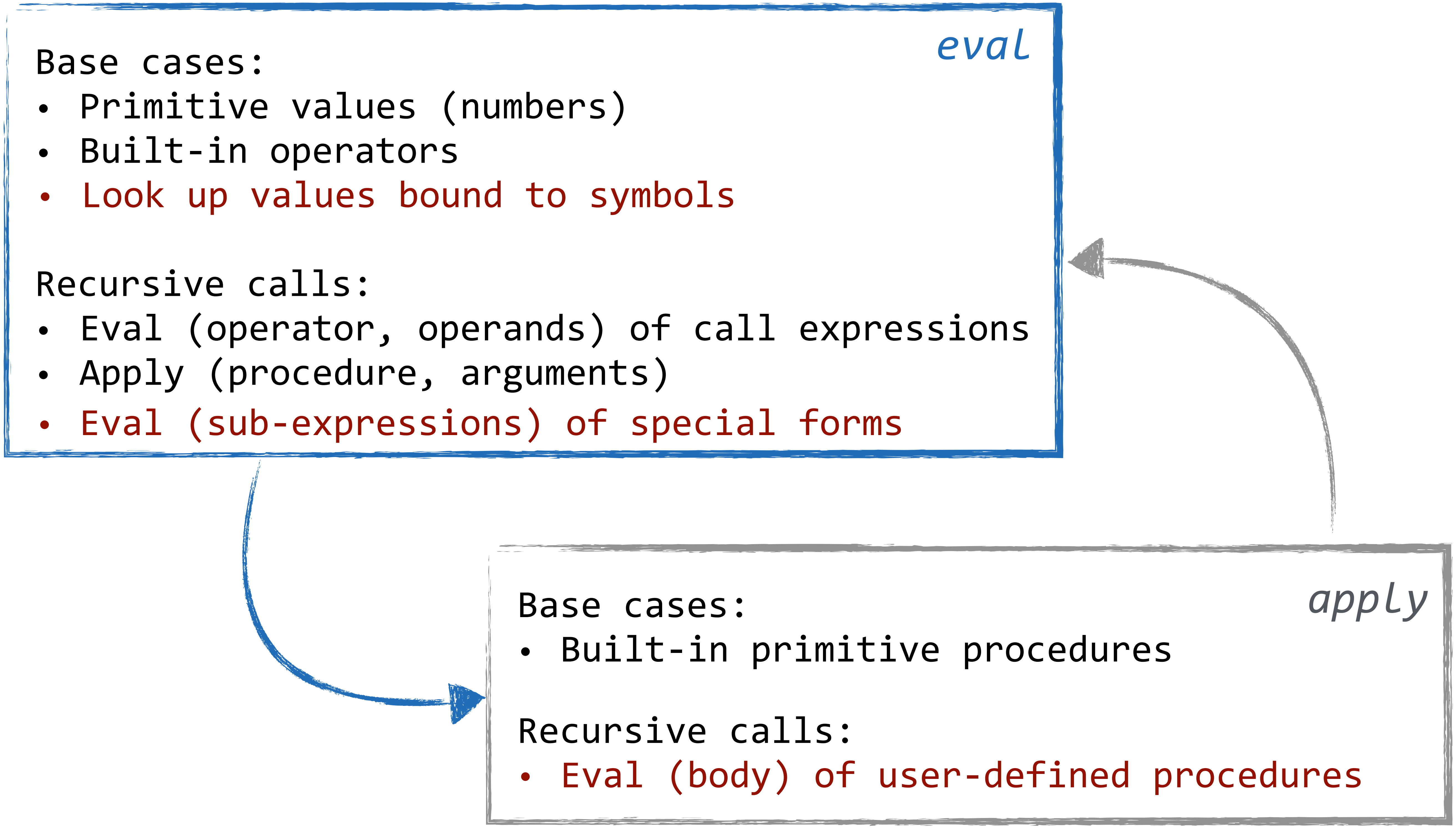
apply

Base cases:

- Built-in primitive procedures

Recursive calls:

eval and apply



eval

Base cases:
- Primitive values (numbers)
- Built-in operators
- 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

eval & apply

```
def calc_eval(exp):
    if isinstance(exp, Pair):
        first, rest = exp.first, exp.second
        op = calc_eval(first)
        args = list(rest.map(calc_eval))
        return calc_apply(op, args)

    elif exp in OPERATORS:
        return OPERATORS[exp]

    else:
        return exp
```

call expressions

built-in procedure

primitives

```
def calc_apply(op, args):
    return op(*args)
```

eval & apply

```
def calc_eval(exp):
    if isinstance(exp, Pair):
        first, rest = exp.first, exp.second
        op = calc_eval(first)
        args = list(rest.map(calc_eval))
        return calc_apply(op, args)
    elif exp in OPERATORS:
        return OPERATORS[exp]
    else:
        return exp
```

```
def calc_apply(op, args):
    return op(*args)
```

```
Pair('+', Pair(4, Pair(3, nil)))
```

eval & apply

```
def calc_eval(exp):
    if isinstance(exp, Pair):
        first, rest = exp.first, exp.second
        op = calc_eval(first)
        args = list(rest.map(calc_eval))
        return calc_apply(op, args)
    elif exp in OPERATORS:
        return OPERATORS[exp]
    else:
        return exp
```

```
def calc_apply(op, args):
    return op(*args)
```

▶ `Pair('+', Pair(4, Pair(3, nil)))`

eval & apply

```
def calc_eval(exp):
    if isinstance(exp, Pair):
        first, rest = exp.first, exp.second
        op = calc_eval(first)
        args = list(rest.map(calc_eval))
        return calc_apply(op, args)
    elif exp in OPERATORS:
        return OPERATORS[exp]
    else:
        return exp
```

```
def calc_apply(op, args):
    return op(*args)
```

▶ `Pair('+', Pair(4, Pair(3, nil)))`

eval & apply

```
def calc_eval(exp):
    if isinstance(exp, Pair):
 ▶      first, rest = exp.first, exp.second
        op = calc_eval(first)
        args = list(rest.map(calc_eval))
        return calc_apply(op, args)
    elif exp in OPERATORS:
        return OPERATORS[exp]
    else:
        return exp
```

```
def calc_apply(op, args):
    return op(*args)
```

▶ `Pair('+', Pair(4, Pair(3, nil)))`

first: '+'

rest: Pair(4, Pair(3, nil))

eval & apply

```
def calc_eval(exp):
    if isinstance(exp, Pair):
        first, rest = exp.first, exp.second
 ▶      op = calc_eval(first)
        args = list(rest.map(calc_eval))
        return calc_apply(op, args)
    elif exp in OPERATORS:
        return OPERATORS[exp]
    else:
        return exp
```

```
def calc_apply(op, args):
    return op(*args)
```

▶ Pair('+', Pair(4, Pair(3, nil)))

first: ‘+’

rest: Pair(4, Pair(3, nil))

▶ ‘+’

eval & apply

```
def calc_eval(exp):
    if isinstance(exp, Pair):
        first, rest = exp.first, exp.second
        op = calc_eval(first)
        args = list(rest.map(calc_eval))
        return calc_apply(op, args)
    elif exp in OPERATORS:
        return OPERATORS[exp]
    else:
        return exp
```

```
def calc_apply(op, args):
    return op(*args)
```

- Pair('+', Pair(4, Pair(3, nil)))

  first: ‘+’

  rest: Pair(4, Pair(3, nil))

- ‘+’

eval & apply

```
def calc_eval(exp):
    if isinstance(exp, Pair):
        first, rest = exp.first, exp.second
        op = calc_eval(first)
        args = list(rest.map(calc_eval))
        return calc_apply(op, args)
    elif exp in OPERATORS:
      ▶ return OPERATORS[exp]
    else:
        return exp
```

```
def calc_apply(op, args):
    return op(*args)
```

▶ Pair('+', Pair(4, Pair(3, nil)))

first: ‘+’

rest: Pair(4, Pair(3, nil))

▶ ‘+’ <function calc_add at ……… >

eval & apply

```
def calc_eval(exp):
    if isinstance(exp, Pair):
        first, rest = exp.first, exp.second
▶       op = calc_eval(first)
        args = list(rest.map(calc_eval))
        return calc_apply(op, args)
    elif exp in OPERATORS:
        return OPERATORS[exp]
    else:
        return exp
```

```
def calc_apply(op, args):
    return op(*args)
```

▶ Pair('+', Pair(4, Pair(3, nil)))

first: ‘+’ *op*:

rest: Pair(4, Pair(3, nil))

▶ ‘+’ <function calc_add at ……… >

eval & apply

```
def calc_eval(exp):
    if isinstance(exp, Pair):
        first, rest = exp.first, exp.second
        op = calc_eval(first)
  ▶     args = list(rest.map(calc_eval))
        return calc_apply(op, args)
    elif exp in OPERATORS:
        return OPERATORS[exp]
    else:
        return exp
```

```
def calc_apply(op, args):
    return op(*args)
```

▶ Pair('+', Pair(4, Pair(3, nil)))

first: ‘+’ *op*:

rest: Pair(4, Pair(3, nil))

▶ ‘+’ <function calc_add at ……… >

▶ 4 ▶ 3

eval & apply

```
def calc_eval(exp):
   if isinstance(exp, Pair):
      first, rest = exp.first, exp.second
      op = calc_eval(first)
      args = list(rest.map(calc_eval))
      return calc_apply(op, args)
   elif exp in OPERATORS:
      return OPERATORS[exp]
   else:
      return exp
```

```
def calc_apply(op, args):
   return op(*args)
```

▶ Pair('+', Pair(4, Pair(3, nil)))

first: '+'   *op*:

rest: Pair(4, Pair(3, nil))

▶ '+' <function calc_add at ……… >

▶ 4   ▶ 3

eval & apply

```
def calc_eval(exp):
    if isinstance(exp, Pair):
        first, rest = exp.first, exp.second
        op = calc_eval(first)
        args = list(rest.map(calc_eval))
        return calc_apply(op, args)
    elif exp in OPERATORS:
        return OPERATORS[exp]
    else:
      ▶ return exp
```

```
def calc_apply(op, args):
    return op(*args)
```

▶ Pair('+', Pair(4, Pair(3, nil)))

first: ‘+’ *op*:

rest: Pair(4, Pair(3, nil))

▶ ‘+’ <function calc_add at ……… >

▶ 4 ▶ 3 [4,

eval & apply

```
def calc_eval(exp):
    if isinstance(exp, Pair):
        first, rest = exp.first, exp.second
        op = calc_eval(first)
        args = list(rest.map(calc_eval))
        return calc_apply(op, args)
    elif exp in OPERATORS:
        return OPERATORS[exp]
    else:
      ▶ return exp
```

```
def calc_apply(op, args):
    return op(*args)
```

▶ Pair('+', Pair(4, Pair(3, nil)))

first: ‘+’ *op*:

rest: Pair(4, Pair(3, nil))

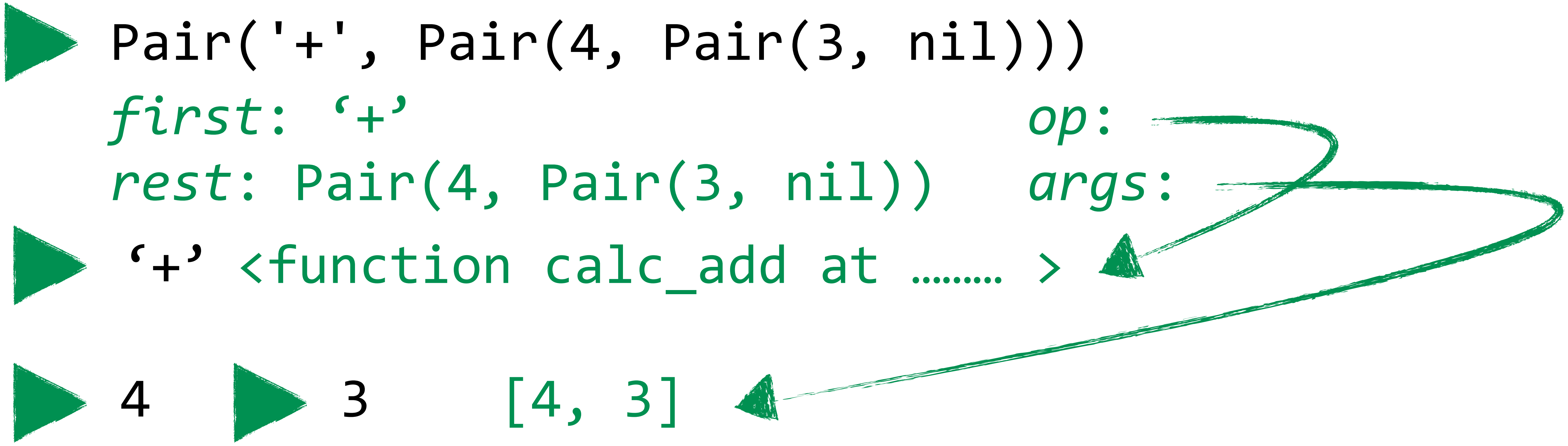
▶ ‘+’ <function calc_add at ……… >

▶ 4 ▶ 3 [4, 3]

eval & apply

```
def calc_eval(exp):
    if isinstance(exp, Pair):
        first, rest = exp.first, exp.second
        op = calc_eval(first)
  ▶     args = list(rest.map(calc_eval))
        return calc_apply(op, args)
    elif exp in OPERATORS:
        return OPERATORS[exp]
    else:
        return exp
```

```
def calc_apply(op, args):
    return op(*args)
```



▶ Pair('+', Pair(4, Pair(3, nil)))

first: '+'

rest: Pair(4, Pair(3, nil))

op:

args:

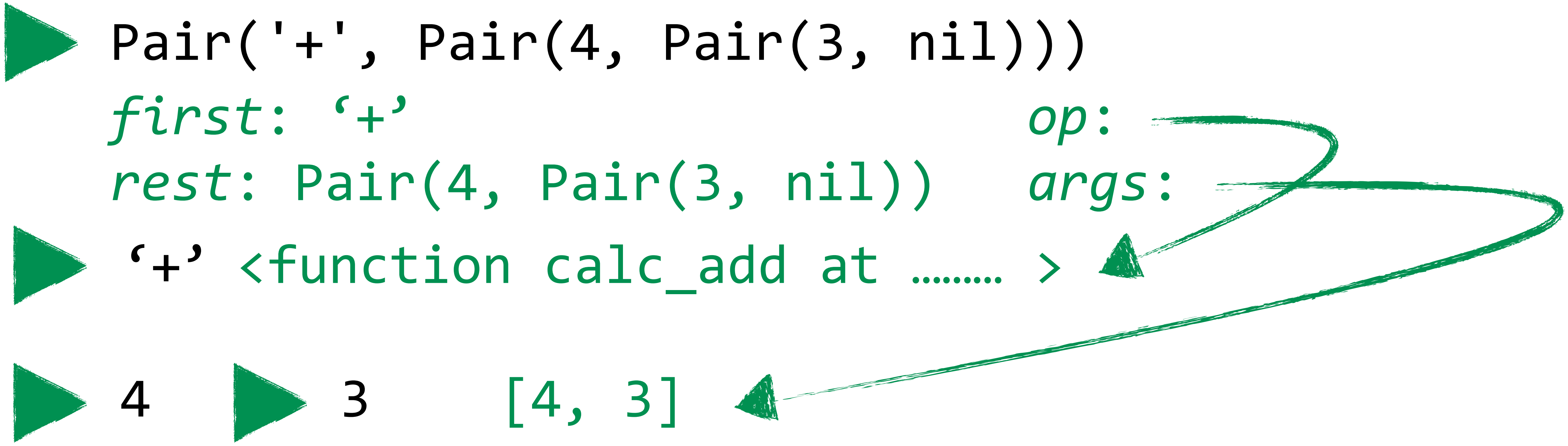
▶ '+' <function calc_add at .... >

▶ 4 ▶ 3 [4, 3]

eval & apply

```
def calc_eval(exp):
    if isinstance(exp, Pair):
        first, rest = exp.first, exp.second
        op = calc_eval(first)
        args = list(rest.map(calc_eval))
        return calc_apply(op, args)
    elif exp in OPERATORS:
        return OPERATORS[exp]
    else:
        return exp
```

```
def calc_apply(op, args):
    return op(*args)
```



▶ Pair('+', Pair(4, Pair(3, nil)))

first: ‘+’

rest: Pair(4, Pair(3, nil))

op:

args:

▶ ‘+’ <function calc_add at ……… >

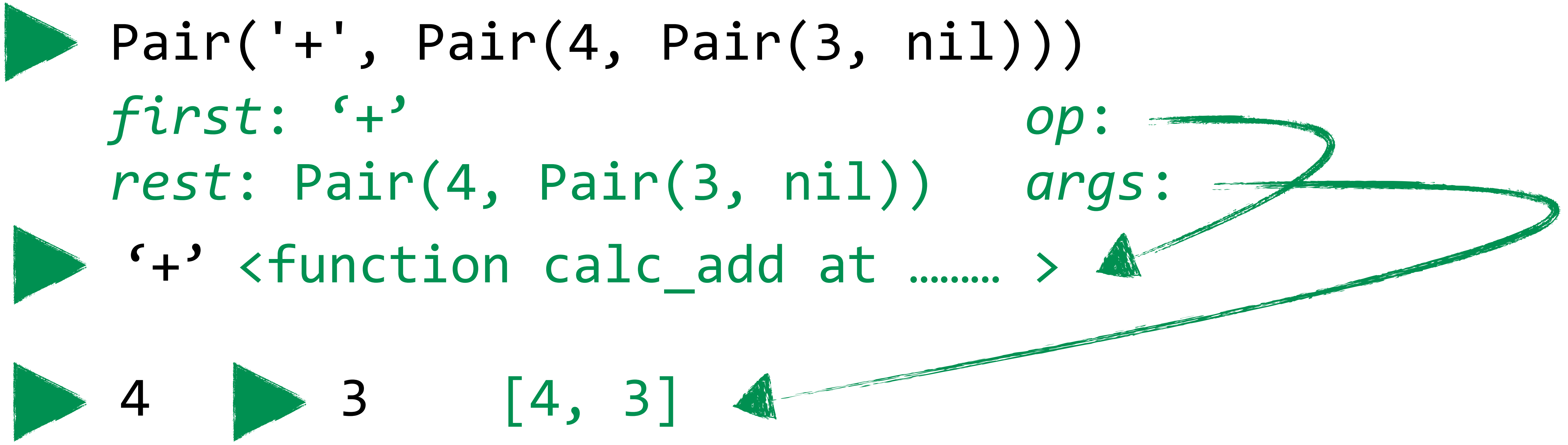
▶ 4 ▶ 3 [4, 3]

eval & apply

```
def calc_eval(exp):
    if isinstance(exp, Pair):
        first, rest = exp.first, exp.second
        op = calc_eval(first)
        args = list(rest.map(calc_eval))
        return calc_apply(op, args)
    elif exp in OPERATORS:
        return OPERATORS[exp]
    else:
        return exp
```

```
def calc_apply(op, args):
    return op(*args)
```

▶ op, args



▶ Pair('+', Pair(4, Pair(3, nil)))

first: ‘+’ *op*:

rest: Pair(4, Pair(3, nil)) *args*:

▶ ‘+’ <function calc_add at ……… >

▶ 4 ▶ 3 [4, 3]

eval & apply

```
def calc_eval(exp):
   if isinstance(exp, Pair):
      first, rest = exp.first, exp.second
      op = calc_eval(first)
      args = list(rest.map(calc_eval))
      return calc_apply(op, args)
   elif exp in OPERATORS:
      return OPERATORS[exp]
   else:
      return exp
```

```
def calc_apply(op, args):
▶ return op(*args)
```

▶ op, args

7

▶ Pair('+', Pair(4, Pair(3, nil)))

first: ‘+’

rest: Pair(4, Pair(3, nil))

op:

args:

▶ ‘+’ <function calc_add at ……… >

▶ 4 ▶ 3 [4, 3]

eval & apply

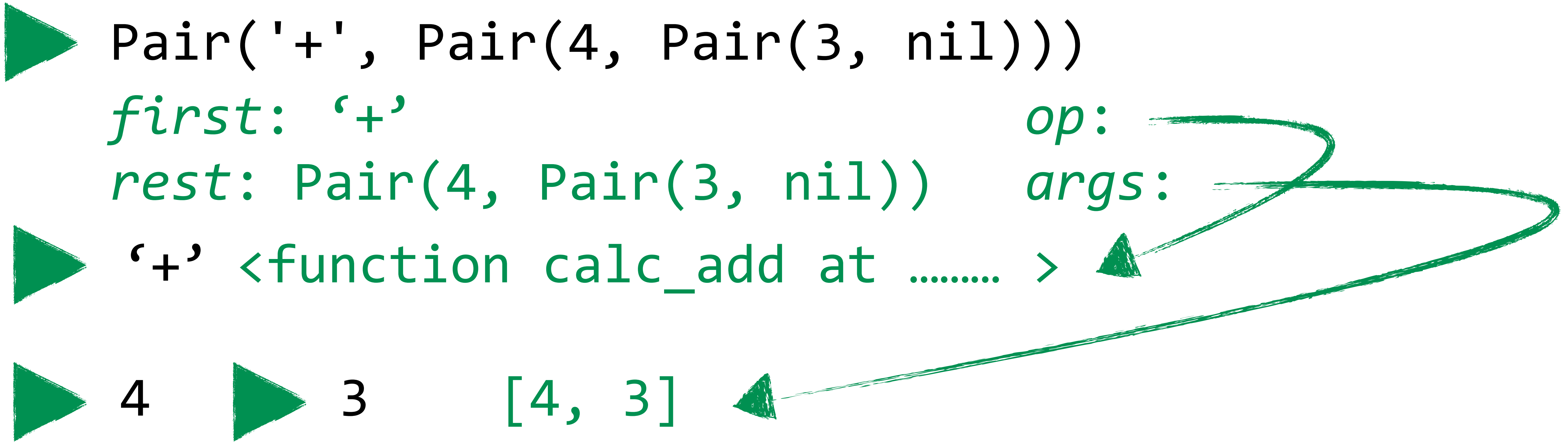
```
def calc_eval(exp):
   if isinstance(exp, Pair):
      first, rest = exp.first, exp.second
      op = calc_eval(first)
      args = list(rest.map(calc_eval))
      return calc_apply(op, args)
   elif exp in OPERATORS:
      return OPERATORS[exp]
   else:
      return exp
```

```
def calc_apply(op, args):
▶  return op(*args)
```

▶ op, args

7

ANSWER



▶ Pair('+', Pair(4, Pair(3, nil)))

first: ‘+’

rest: Pair(4, Pair(3, nil))

op:

args:

▶ ‘+’ <function calc_add at ……… >

▶ 4 ▶ 3 [4, 3]

special forms

special forms

special form: an expression that does NOT follow normal evaluation procedure

special forms

special form: an expression that does NOT follow normal evaluation procedure

```
(define x (+ 3 4))

      exp.first: "define"
      exp.second: (x (+ 3 4))
```

special forms

special form: an expression that does NOT follow normal evaluation procedure

```
(define x (+ 3 4))

    exp.first: "define"
    exp.second: (x (+ 3 4))
```

NORMALLY, we MAP calc_eval on EACH argument in exp.second

BUT, here, we should NOT call calc_eval on x

special forms

```
SPECIAL_FORMS = {
    'define': do_define_form,
   ...
}
```

```
def calc_eval(exp):
  if isinstance(exp, Pair):
    first, rest = exp.first, exp.second
    if first in SPECIAL_FORMS:
      return SPECIAL_FORMS[first](rest)
    else:
      op = calc_eval(first)
      args = list(rest.map(calc_eval))
      return calc_apply(op, args)
  elif exp in OPERATORS:
    return OPERATORS[exp]
  else:
    return exp
```

special forms

call expressions

built-in procedures

primitives

special forms: Defining variables

```
SPECIAL_FORMS = {
    'define': do_define_form,
}
```

```
(define x (+ 3 4))
```

special forms: Defining variables

```
SPECIAL_FORMS = {
    'define': do_define_form,
}
```

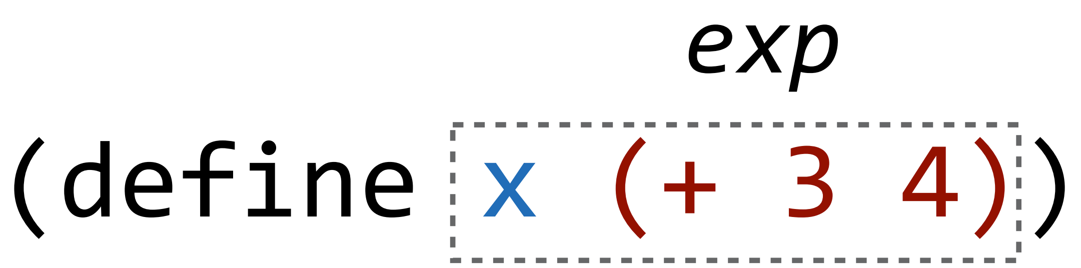
exp

```
(define x (+ 3 4))
```

```
           exp: (x (+ 3 4))
     exp.first: x
exp.second.first: (+ 3 4)
```

special forms: Defining variables

```
SPECIAL_FORMS = {
    'define': do_define_form,
}
```



exp

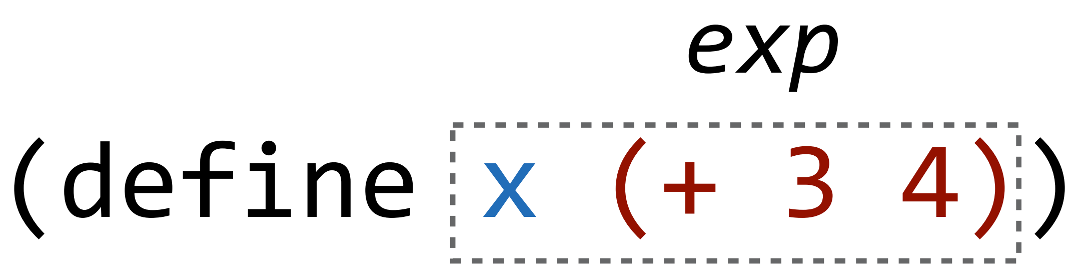
(define x (+ 3 4))

```
           exp: (x (+ 3 4))
     exp.first: x
exp.second.first: (+ 3 4)
```

```
def do_define_form(exp):
    target = exp.first
    if target is a symbol:
      1.Evaluate exp.second.first
      2.Bind target to value
```

special forms: Defining variables

```
SPECIAL_FORMS = {
    'define': do_define_form,
}
```



exp

(define x (+ 3 4))

```
exp: (x (+ 3 4))
exp.first: x
exp.second.first: (+ 3 4)
```

```
def do_define_form(exp):
    target = exp.first
    if target is a symbol:
      1.Evaluate exp.second.first
      2.Bind target to value
```

BUT we **DO NOT** evaluate the target

special forms: Defining procedures

```
SPECIAL_FORMS = {
    'define': do_define_form,
}
```

```
(define (square x)
        (* x x)    )
```

special forms: Defining procedures

```
SPECIAL_FORMS = {
    'define': do_define_form,
}
```

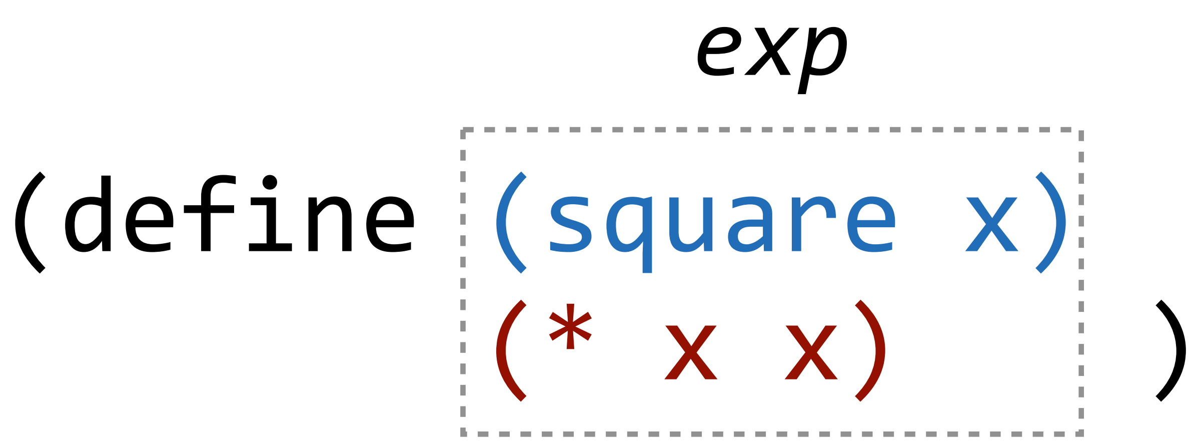
exp

```
(define (square x)
        (* x x)     )
```

```
           exp: ((square x) (* x x))
     exp.first: (square x)
exp.second.first: (* x x)
```

special forms: Defining procedures

```
SPECIAL_FORMS = {
    'define': do_define_form,
}
```



exp

```
(define (square x)
        (* x x)     )
```

```
            exp: ((square x) (* x x))
      exp.first: (square x)
exp.second.first: (* x x)
```

```
def do_define_form(exp):
    target = exp.first
    if target is a symbol:
        . . .
```

special forms: Defining procedures

```
SPECIAL_FORMS = {
    'define': do_define_form,
}
```

exp

```
(define (square x)
        (* x x)   )
```

```
                 exp: ((square x) (* x x))
           exp.first: (square x)
    exp.second.first: (* x x)
```

```
def do_define_form(exp):
    target = exp.first
    if target is a symbol:
        . . .
    elif isinstance(target, Pair):
        1.Create a procedure object using target.second
          (formal parameters) and exp.second.first (body)
        2.Bind target.first (name) to procedure object
```

special forms: Defining procedures

```
SPECIAL_FORMS = {
    'define': do_define_form,
}
```

exp

```
(define (square x)
        (* x x)     )
```

```
            exp: ((square x) (* x x))
      exp.first: (square x)
exp.second.first: (* x x)
```

```
def do_define_form(exp):
    target = exp.first
    if target is a symbol:
        . . .
```

BUT we **DO NOT** evaluate anything

```
    elif isinstance(target, Pair):
        1.Create a procedure object using target.second
          (formal parameters) and exp.second.first (body)
        2.Bind target.first (name) to procedure object
```

special forms: Defining procedures

```
SPECIAL_FORMS = {
    'define': do_define_form,
}
```

exp

```
(define (square x)
        (* x x)    )
```

restricted to one body expression

```
exp: ((square x) (* x x))
exp.first: (square x)
exp.second.first: (* x x)
```

```
def do_define_form(exp):
    target = exp.first
    if target is a symbol:
      . . .
```

BUT we **DO NOT** evaluate anything

```
  elif isinstance(target, Pair):
      1.Create a procedure object using target.second
        (formal parameters) and exp.second.first (body)
      2.Bind target.first (name) to procedure object
```

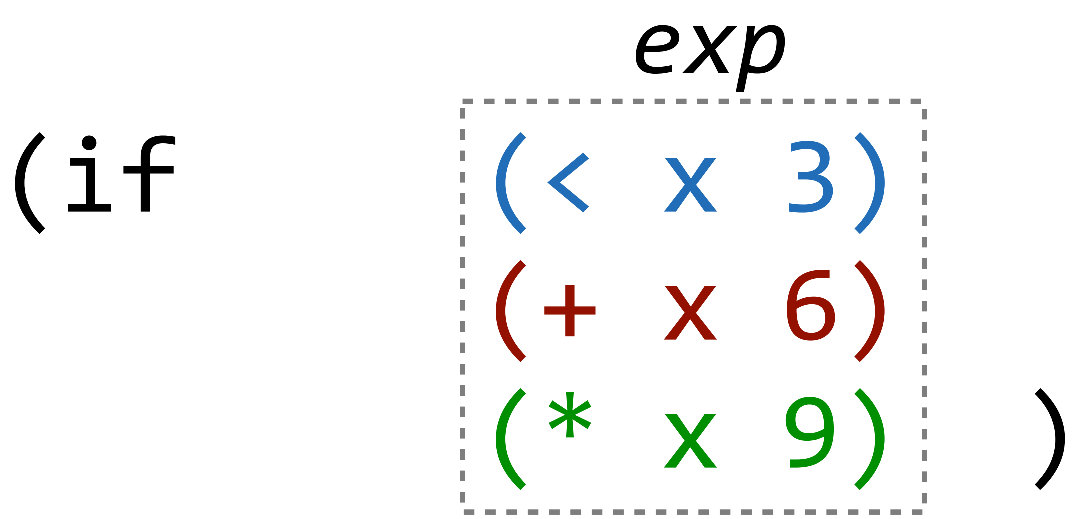
special forms: if expressions

```
SPECIAL_FORMS = {
    'define': do_define_form,
    'if': do_if_form,
}
```

```
(if     (< x 3)
        (+ x 6)
        (* x 9)   )
```

special forms: if expressions

```
SPECIAL_FORMS = {
    'define': do_define_form,
    'if': do_if_form,
}
```

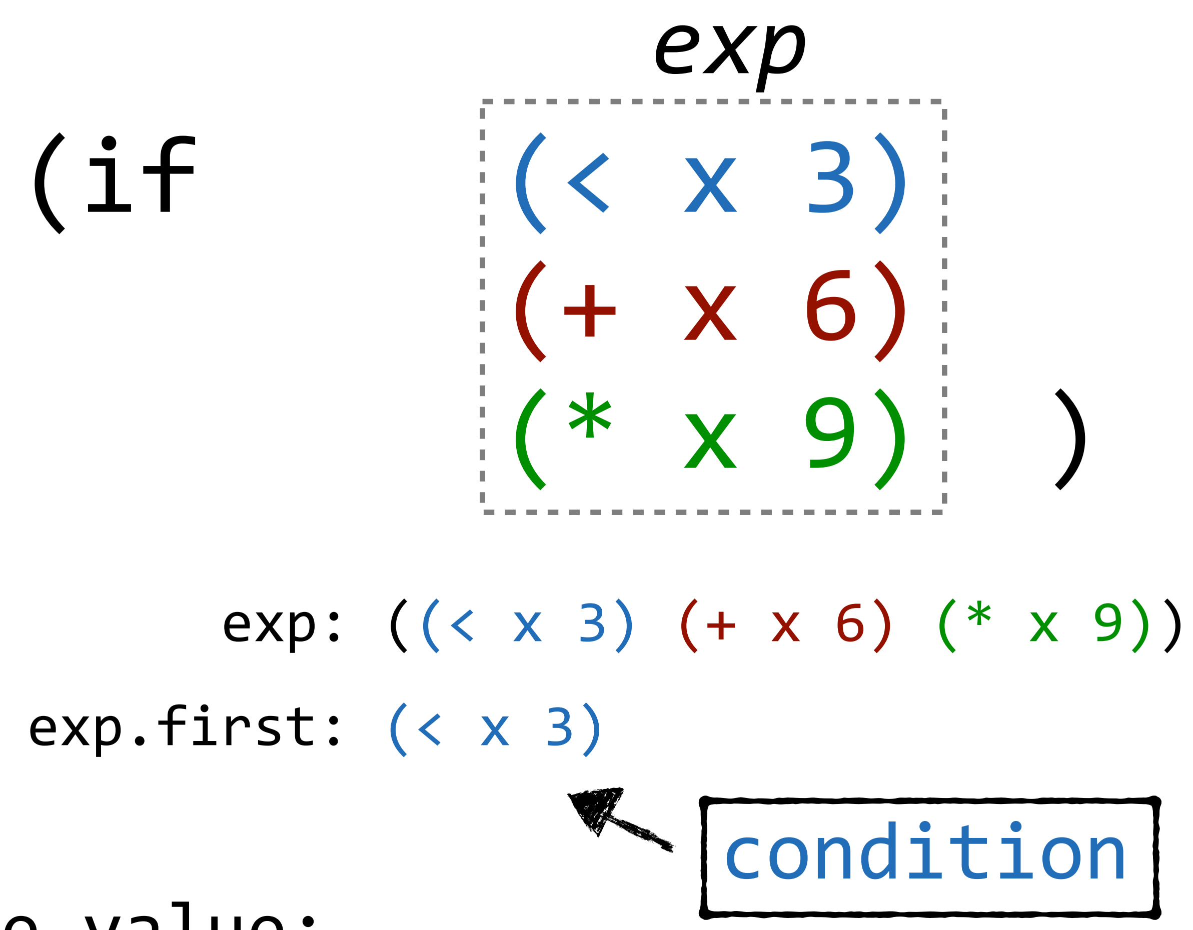


```
     exp: ((< x 3) (+ x 6) (* x 9))
exp.first: (< x 3)
```

condition

special forms: if expressions

```
SPECIAL_FORMS = {
    'define': do_define_form,
    'if': do_if_form,
}
```



exp

```
(if (< x 3)
    (+ x 6)
    (* x 9) )
```

```
     exp: ((< x 3) (+ x 6) (* x 9))
exp.first: (< x 3)
```

condition

```
def do_if_form(exp):
    if exp.first evaluates to true value:
        evaluate and return true clause
    elif exp has a false clause:
        evaluate and return false clause
```

special forms: if expressions

```
SPECIAL_FORMS = {
    'define': do_define_form,
    'if': do_if_form,
}
```

exp

```
(if     (< x 3)
        (+ x 6)
        (* x 9)   )
```

```
     exp: ((< x 3) (+ x 6) (* x 9))
exp.first: (< x 3)
```

condition

```
def do_if_form(exp):
    if exp.first evaluates to true value:
        evaluate and return true clause
    elif exp has a false clause:
        evaluate and return false clause
```

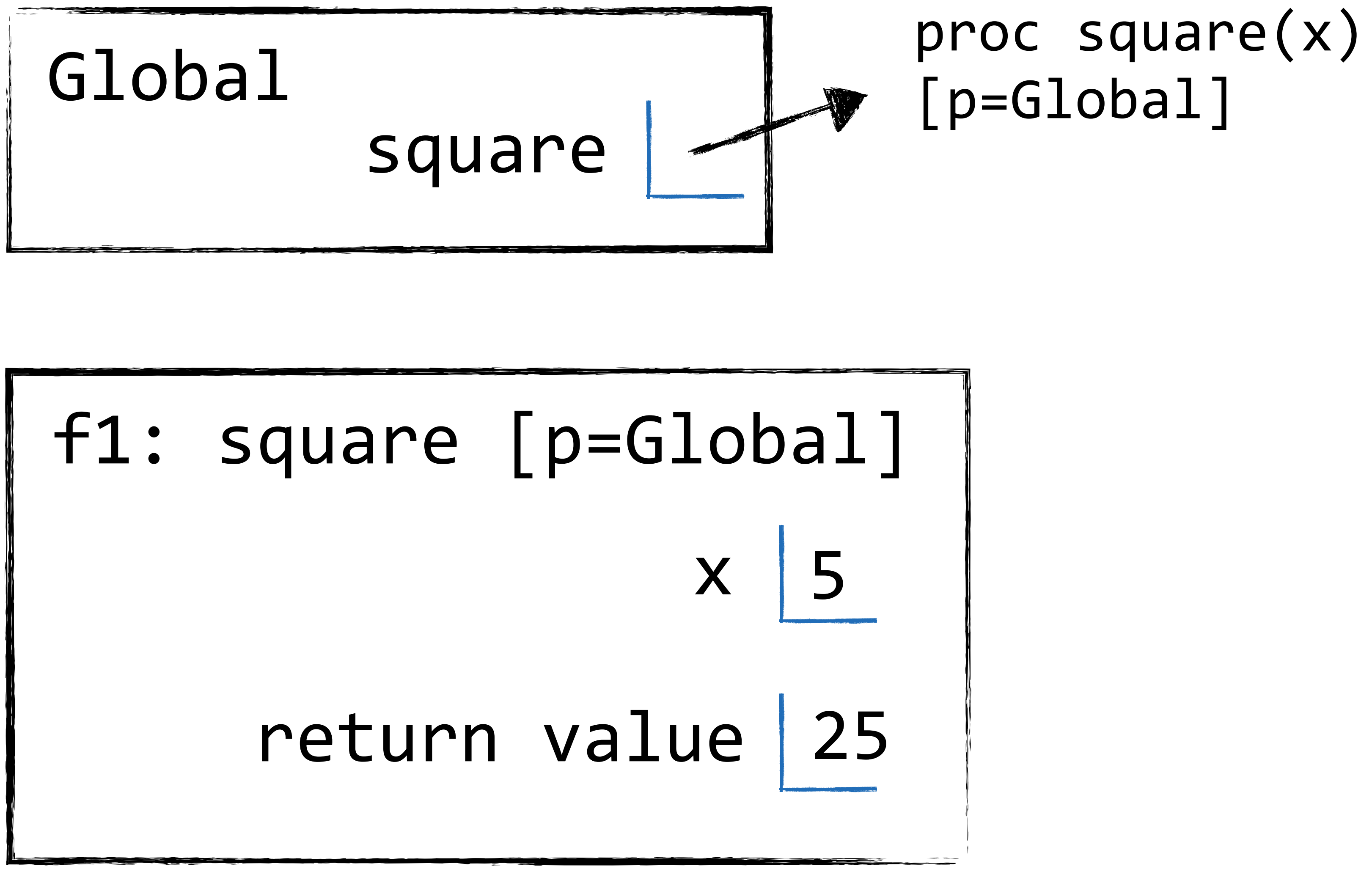
DO NOT evaluate both the true clause and the false clause; **only evaluate ONE** of them

environments

environments & frames.

```
(define (square x)
    (* x x))
```

```
(square 5)
```



Global

square

proc square(x)
[p=Global]

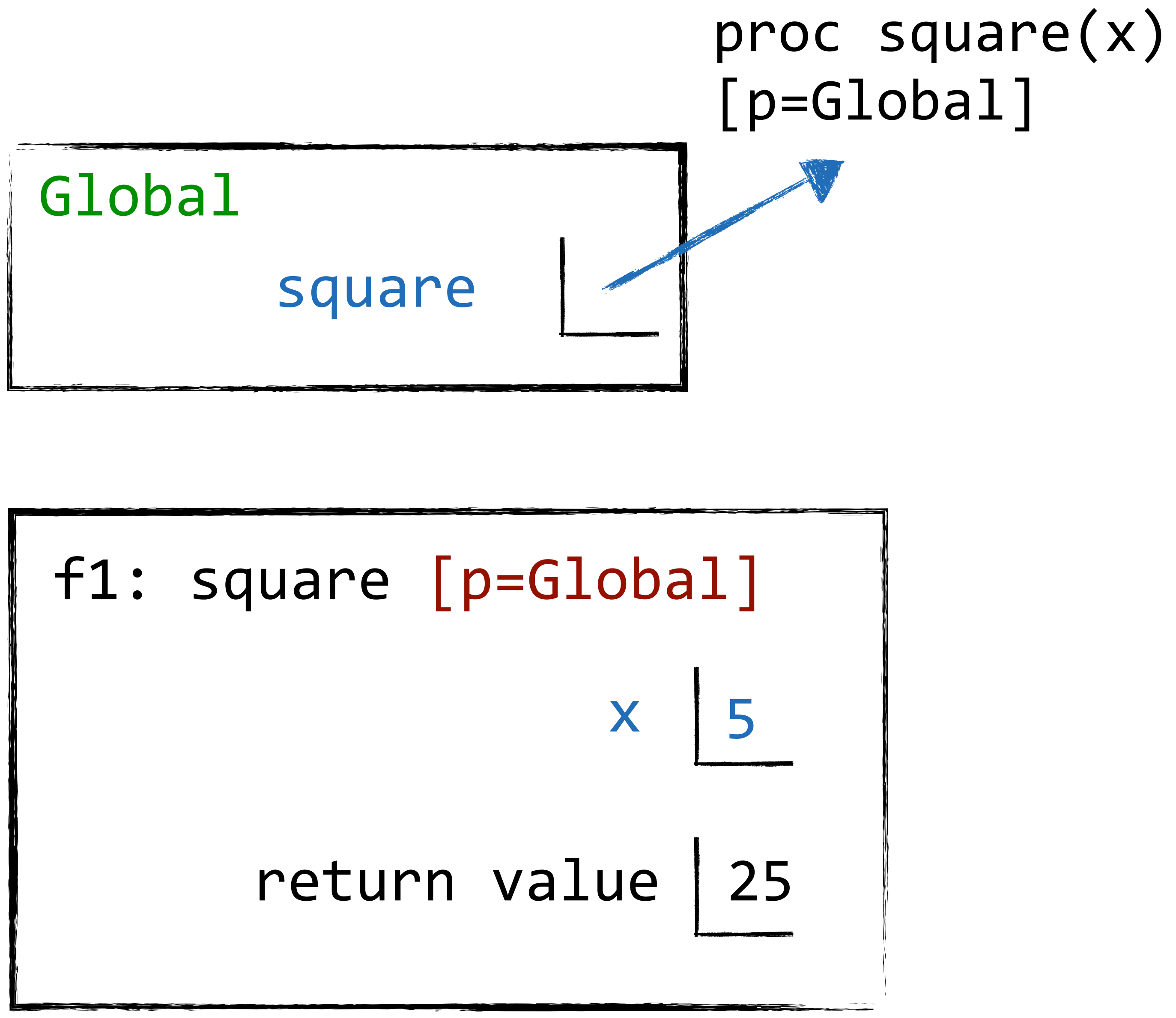
f1: square [p=Global]

x 5

return value 25

our interpreter NEEDS to KEEP TRACK of frames

environments & frames.



proc square(x)
[p=Global]

Global

square

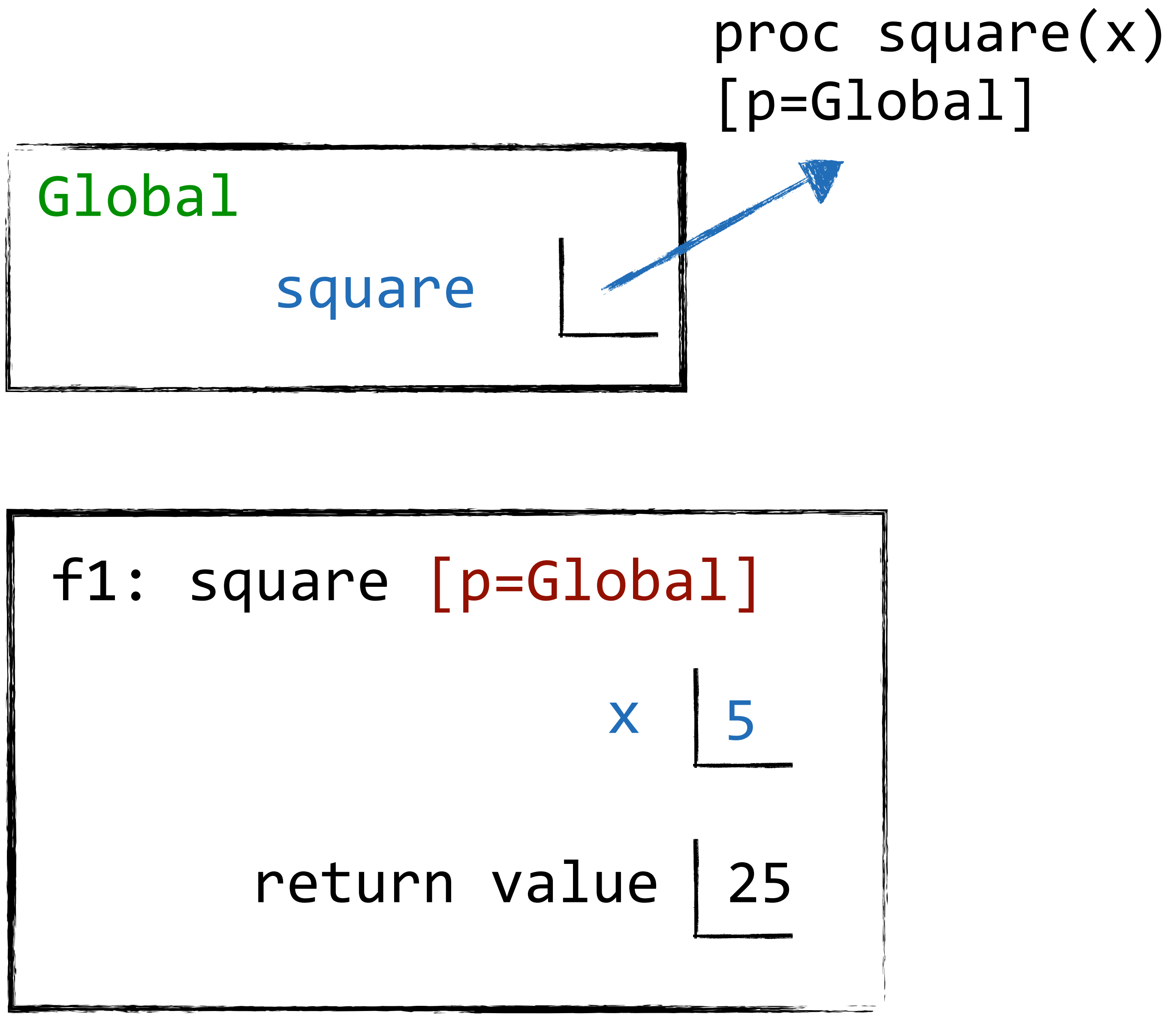
f1: square [p=Global]

x 5

return value 25

environments & frames.

scheme.py



proc square(x)
[p=Global]

Global

square

f1: square [p=Global]

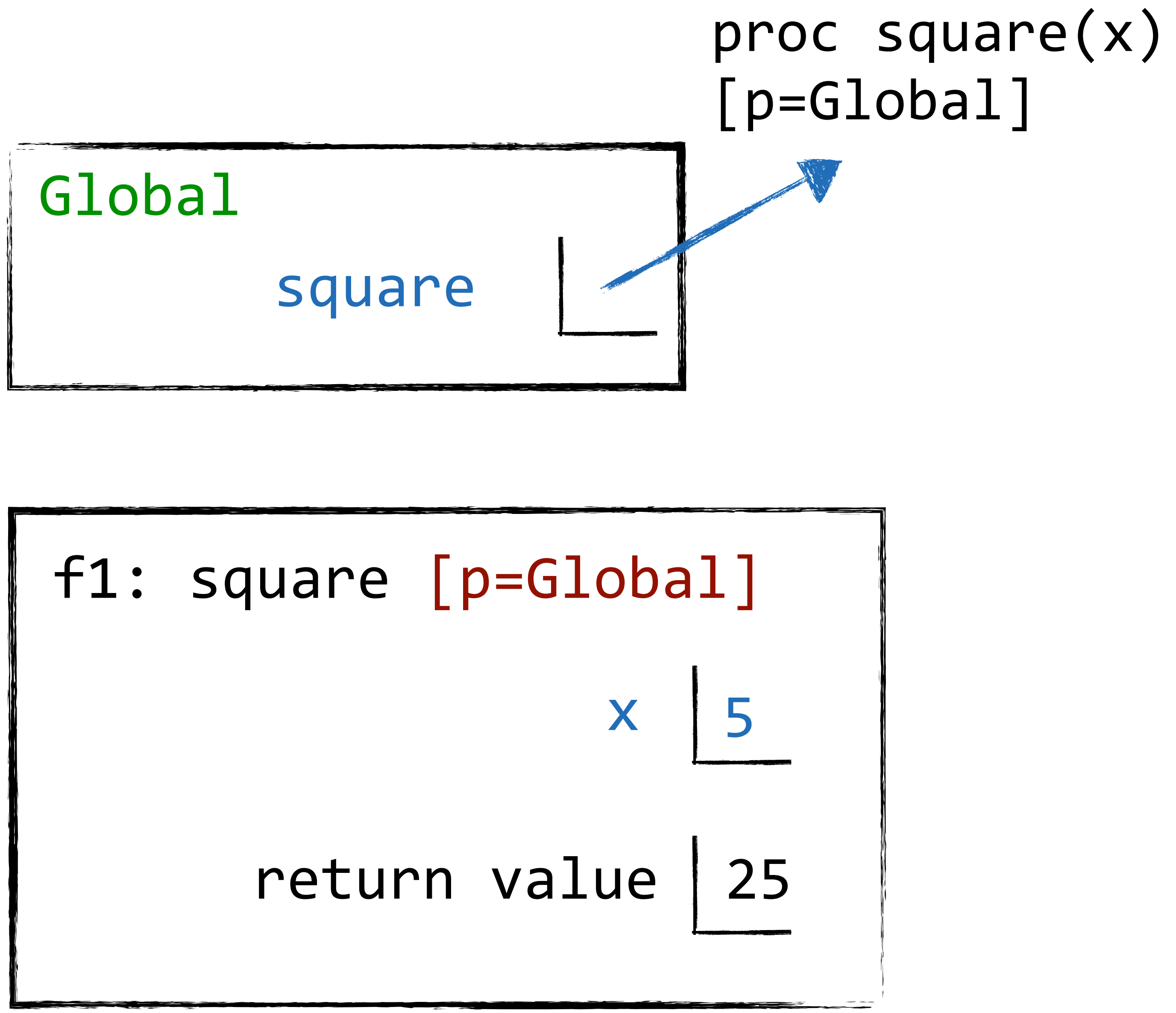
x 5

return value 25

environments & frames.

scheme.py

```
class Frame:
    def __init__(self, parent):
        self.bindings = {}
        self.parent = parent
```



proc square(x)
[p=Global]

Global

square

f1: square [p=Global]

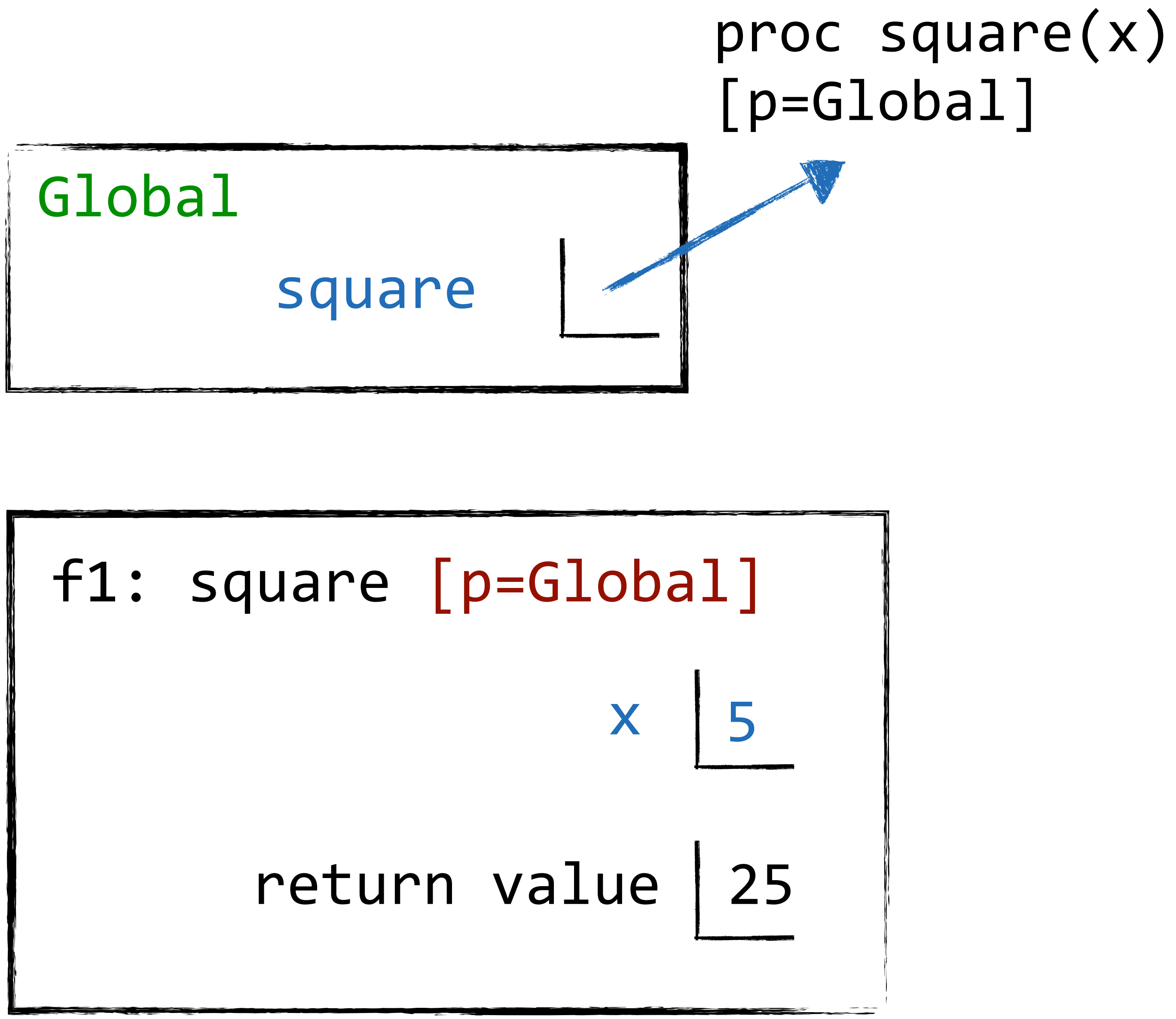
x 5

return value 25

environments & frames.

scheme.py

```
class Frame:
    def __init__(self, parent):
        self.bindings = {}
        self.parent = parent
```



proc square(x)
[p=Global]

Global

square

f1: square [p=Global]

x 5

return value 25

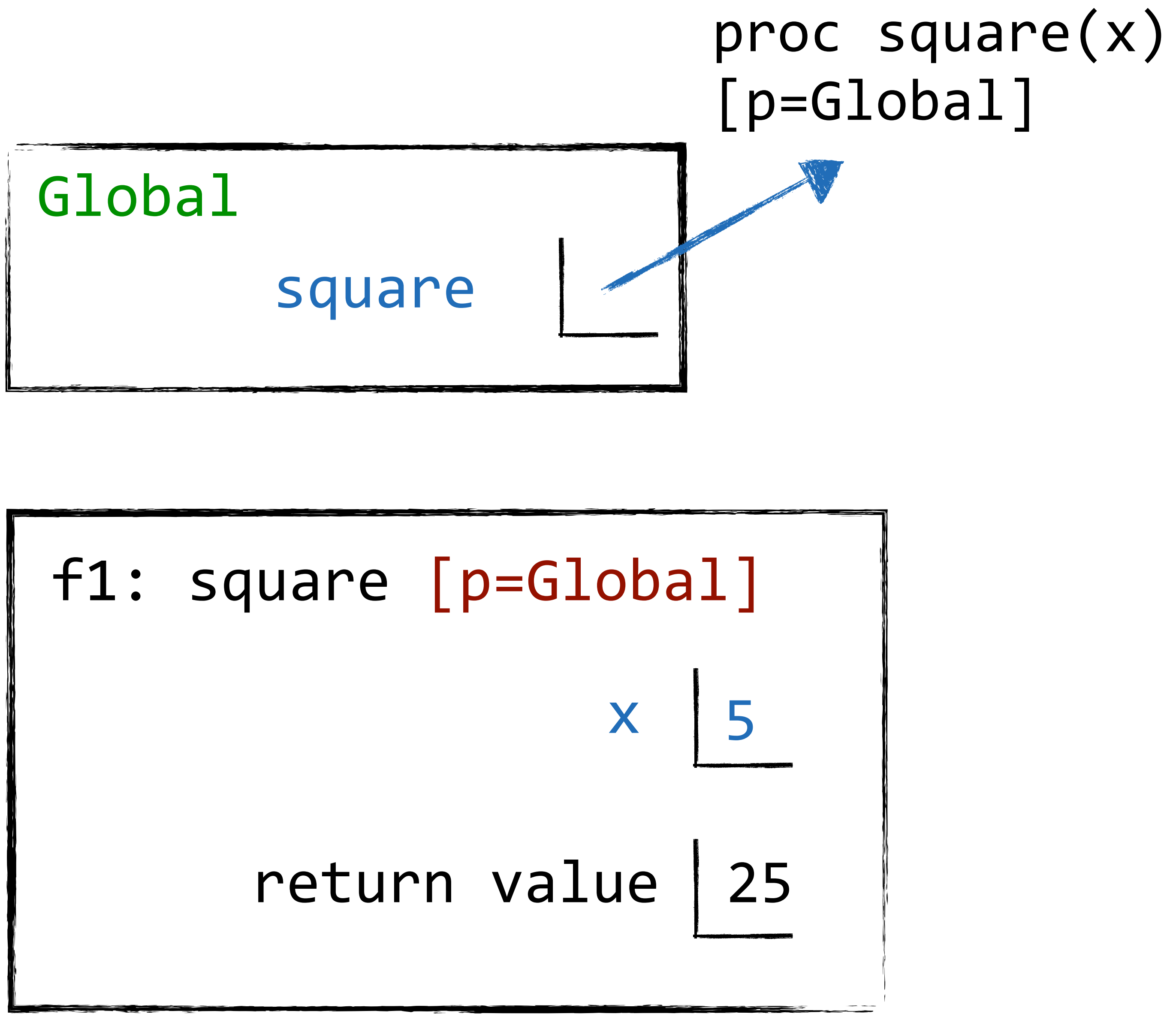
bindings: a dictionary mapping symbols to values

parent: the parent frame (an instance of Frame or None)

environments & frames.

scheme.py

```
class Frame:
    def __init__(self, parent):
        self.bindings = {}
        self.parent = parent


def create_global_frame():
    return Frame(None)
```



bindings: a dictionary mapping symbols to values

parent: the parent frame (an instance of Frame or None)

environments & frames:

binding & lookup

scheme.py

```
class Frame:
    def __init__(self, parent):
        self.bindings = {}
        self.parent = parent

    def define(self, symbol, value):
        self.bindings[symbol] = value
```

environments & frames:

binding & lookup

scheme.py

```
class Frame:
    def __init__(self, parent):
        self.bindings = {}
        self.parent = parent

    def define(self, symbol, value):
        self.bindings[symbol] = value

    def lookup(self, symbol):
        if symbol in self.bindings:
            return self.bindings[symbol]
        elif self.parent is None:
            raise NameError(symbol + “ is not defined”)
        else:
            return self.parent.lookup(symbol)
```

environments & frames:

binding & lookup

scheme.py

```
class Frame:
    def __init__(self, parent):
        self.bindings = {}
        self.parent = parent

    def define(self, symbol, value):
        self.bindings[symbol] = value

    def lookup(self, symbol):
        if symbol in self.bindings:
            return self.bindings[symbol]
        elif self.parent is None:
            raise NameError(symbol + “ is not defined”)
        else:
            return self.parent.lookup(symbol)
```

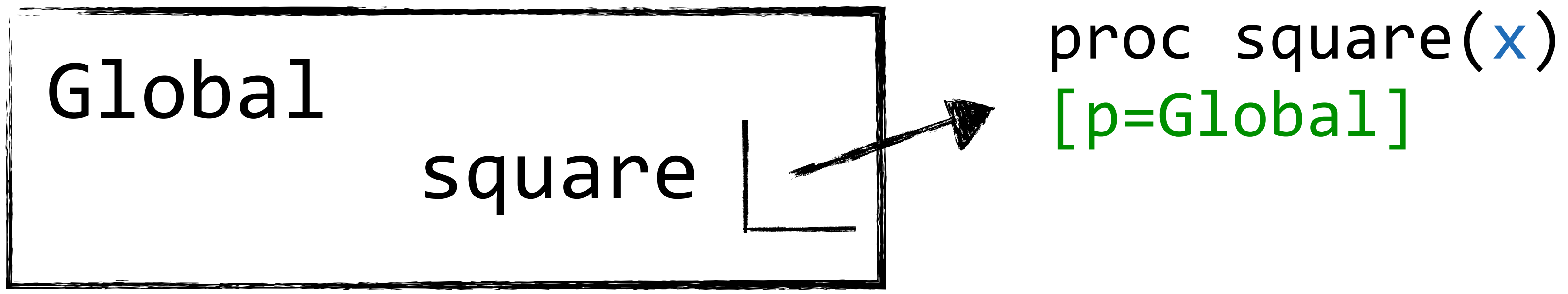
pass in a frame

```
def calc_eval(exp, env):
    if exp is a symbol:
        return env.lookup(exp)
    ...
```

variable lookups!!

environments & frames:

procedure objects



Global

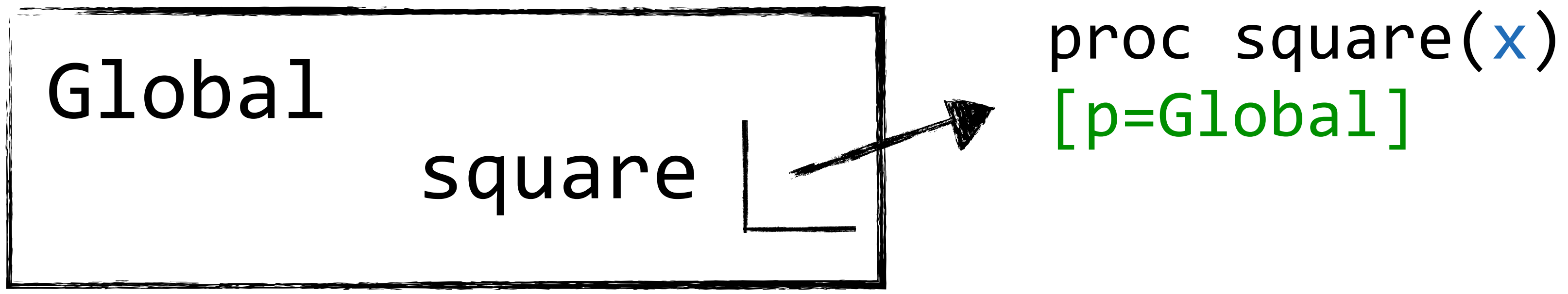
square

proc square(x)
[p=Global]

```
(define (square x)
     (* x x))
```

environments & frames:

procedure objects



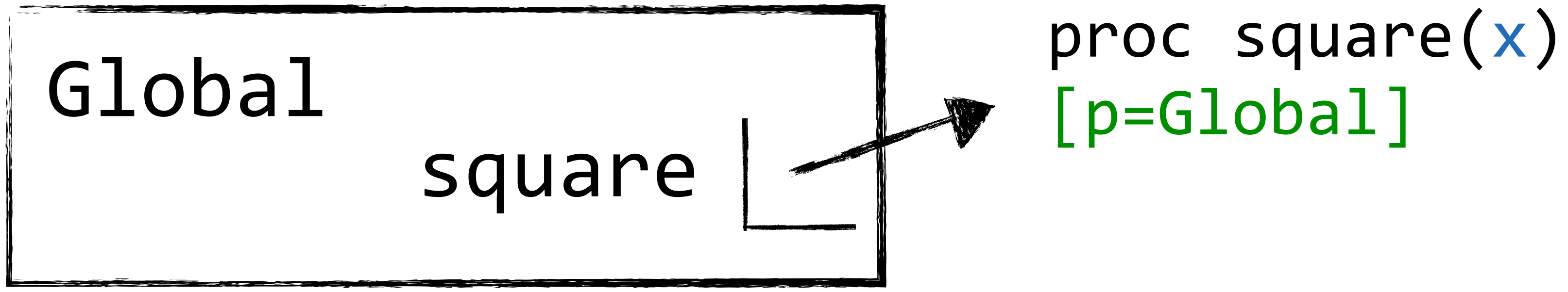
```
(define (square x)
      (* x x))
```

scheme.py

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

environments & frames:

procedure objects



Global

square

proc square(x)
[p=Global]

```
(define (square x)
    (* x x))
```

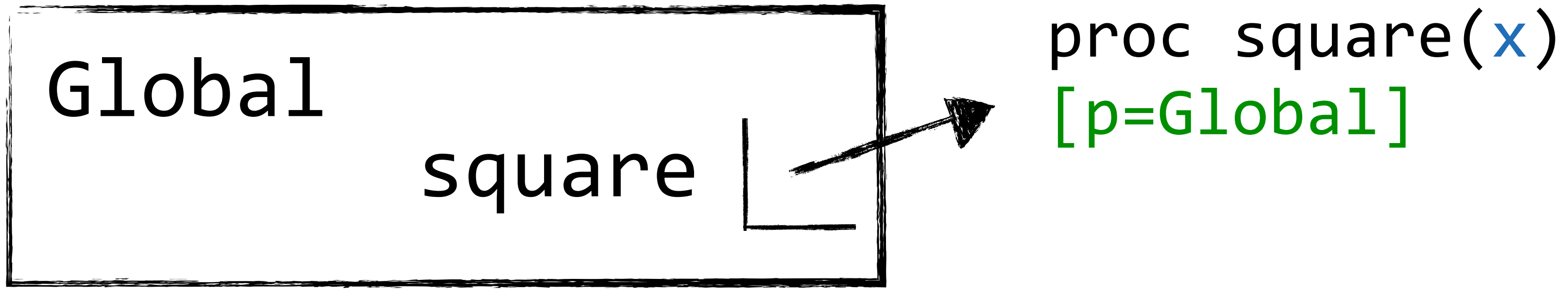
formals: parameters that the procedure takes

scheme.py

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

environments & frames:

procedure objects



```
(define (square x)
    (* x x))
```

formals: parameters that the procedure takes

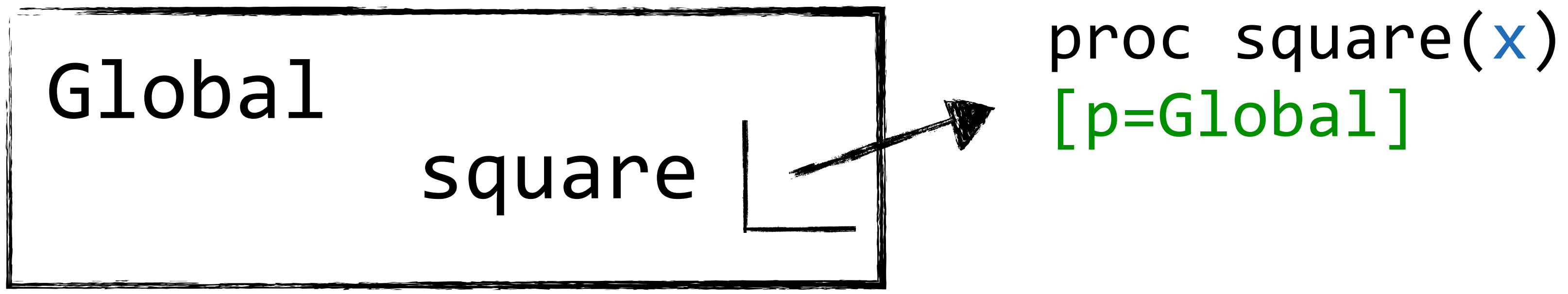
body: the body of the procedure

scheme.py

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

environments & frames:

procedure objects



```
(define (square x)
    (* x x))
```

scheme.py

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

formals: parameters that the procedure takes

body: the body of the procedure

env: the Frame in which this procedure is defined

eval & apply

handling lambda procedures

instance of the Frame class

```
def calc_eval(exp, env):
    ...
    else:
        op = calc_eval(first, env)
        args = map calc_eval on
                   each element of rest
        return calc_apply(op, args)
```

instance of the LambdaProcedure class

```
def calc_apply(op, args):
    return op(*args)
```

square is NOT handled here

calc_apply needs to handle lambda procedures

eval & apply

handling lambda procedures

```
def calc_apply(procedure, args):
     if isinstance(procedure, LambdaProcedure):
          new_env = procedure.make_call_frame(args)
          return calc_eval(procedure.body, new_env)
     else:
       return procedure(*args)
```

eval & apply

handling lambda procedures

```
def calc_apply(procedure, args):
     if isinstance(procedure, LambdaProcedure):
          new_env = procedure.make_call_frame(args)
          return calc_eval(procedure.body, new_env)
     else:
       return procedure(*args)
```

restricted to one body expression

eval & apply

handling lambda procedures

```
def calc_apply(procedure, args):
     if isinstance(procedure, LambdaProcedure):
          new_env = procedure.make_call_frame(args)
          return calc_eval(procedure.body, new_env)
     else:
        return procedure(*args)
```

restricted to one body expression

Rules for call expressions:

1. Create a new frame
2. Bind formal parameters
3. Execute the body

eval & apply

handling lambda procedures

```
def calc_apply(procedure, args):
    if isinstance(procedure, LambdaProcedure):
        new_env = procedure.make_call_frame(args)
        return calc_eval(procedure.body, new_env)
    else:
      return procedure(*args)
```

restricted to one body expression

Rules for call expressions:

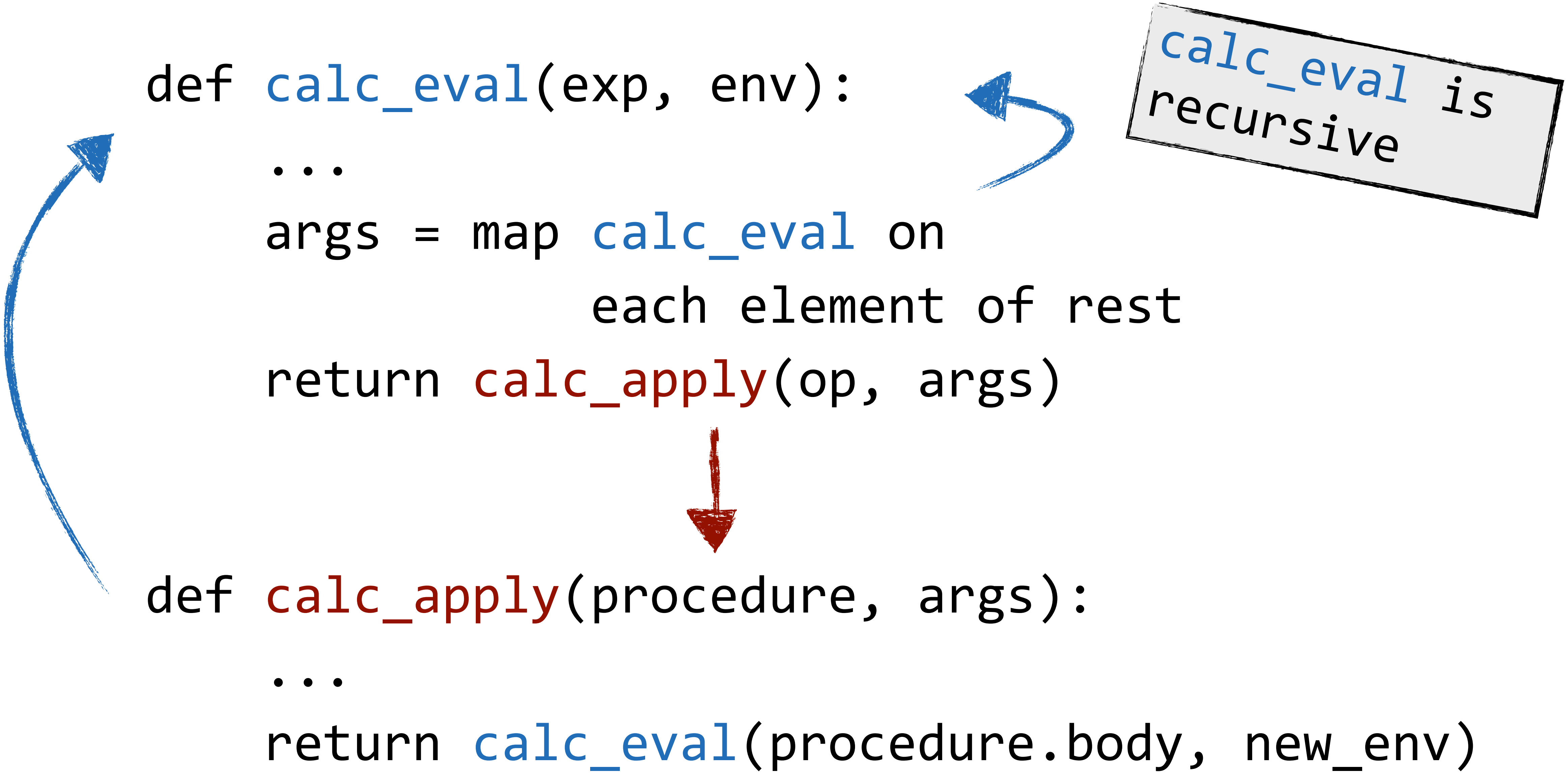
1. Create a new frame
2. Bind formal parameters
3. Execute the body

```
new_env = procedure.make_call_frame(args)
```

```
return calc_eval(procedure.body, new_env)
```

eval & apply

handling lambda procedures



```
def calc_eval(exp, env):
    ...
    args = map calc_eval on
                each element of rest
    return calc_apply(op, args)


def calc_apply(procedure, args):
    ...
    return calc_eval(procedure.body, new_env)
```

calc_eval is recursive

calc_eval and **calc_apply** are **mutually recursive**

scope

visibility of variables

where in your program
can you access it or see it

two types of scoping

1. lexical
2. dynamic

two types of scoping:

Lexical scope:
parent frame is the frame where procedure was DEFINED

Dynamic scope:
parent frame is the frame where procedure was CALLED

two types of scoping:

Lexical scope: parent frame is the frame where procedure was DEFINED

Dynamic scope: parent frame is the frame where procedure was CALLED

```
(define lime
    (lambda (x) (* x y)))
```

two types of scoping:

Lexical scope: parent frame is the frame where procedure was DEFINED

Dynamic scope: parent frame is the frame where procedure was CALLED

```
(define lime
    (lambda (x) (* x y)))
```

```
(define f
    (mu (x) (* x y)))
```

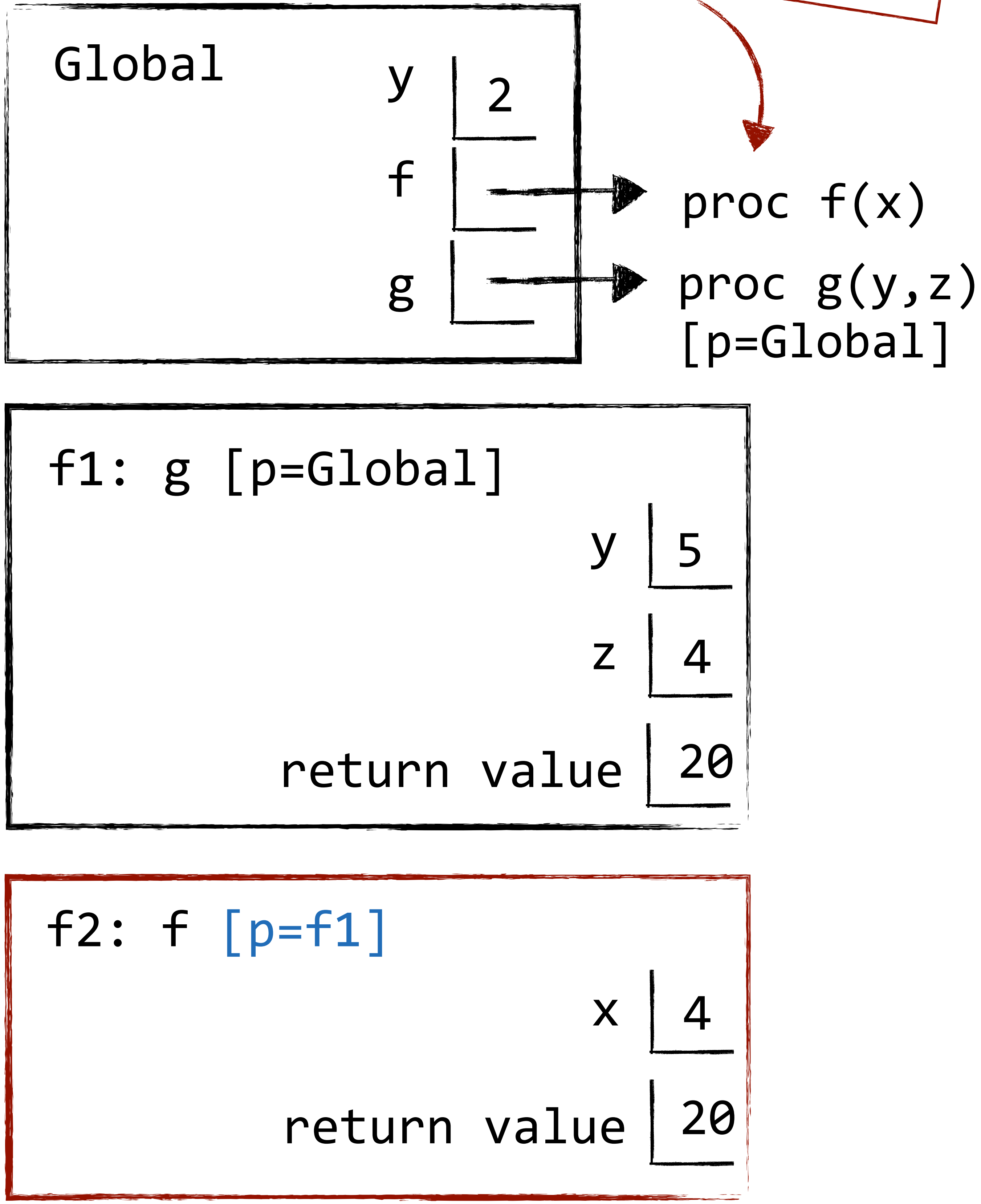
dynamic scoping:

procedure that uses dynamic scope

```
(define y 2)

(define f
    (mu (x) (* x y)))

(define g
    (lambda (y z) (f z)))

(g 5 4)
```

Parent frame is frame in which f was called

parent frame can be different



Global

y 2

f

g

proc f(x)

proc g(y,z) [p=Global]

f1: g [p=Global]

y 5

z 4

return value 20

f2: f [p=f1]

x 4

return value 20

dynamic scoping: MuProcedure

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

```
(define f
    (mu (x) (* x y)))
```

Dynamic scope: parent frame is the frame where procedure was CALLED

dynamic scoping: MuProcedure

```python
class MuProcedure:
    def __init__(self, formals, body):
        self.formals = formals
        self.body = body
```

```scheme
(define f
    (mu (x) (* x y)))
```

Dynamic scope: parent frame is the frame where procedure was CALLED

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

LambdaProcedure for reference

summary

- extended functionality of our interpreter to include special forms
- developed an understanding of environments, using Scheme as an example
- uncovered dynamic scoping via the MuProcedure

the end.

questions?

talk to me after class or email me at neilagarwal@berkeley.edu