

# Interpreters

---

# Announcements

Evaluation  
(review)

# Calculator Syntax

The Calculator language has primitive expressions and call expressions. (That's it!)

A primitive expression is a number: 2 -4 5.6

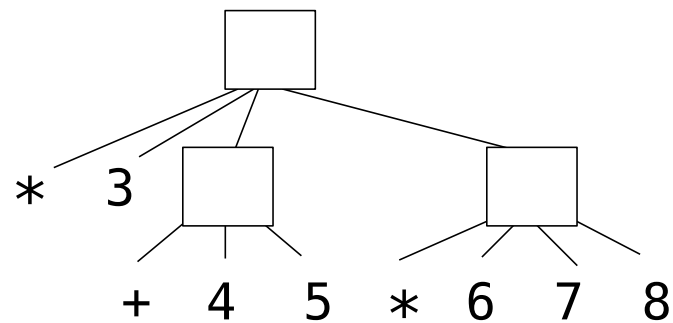
A call expression is a combination that begins with an operator (+, -, \*, /) followed by 0 or more expressions: (+ 1 2 3) (/ 3 (+ 4 5))

Expressions are represented as Scheme lists (Pair instances) that encode tree structures.

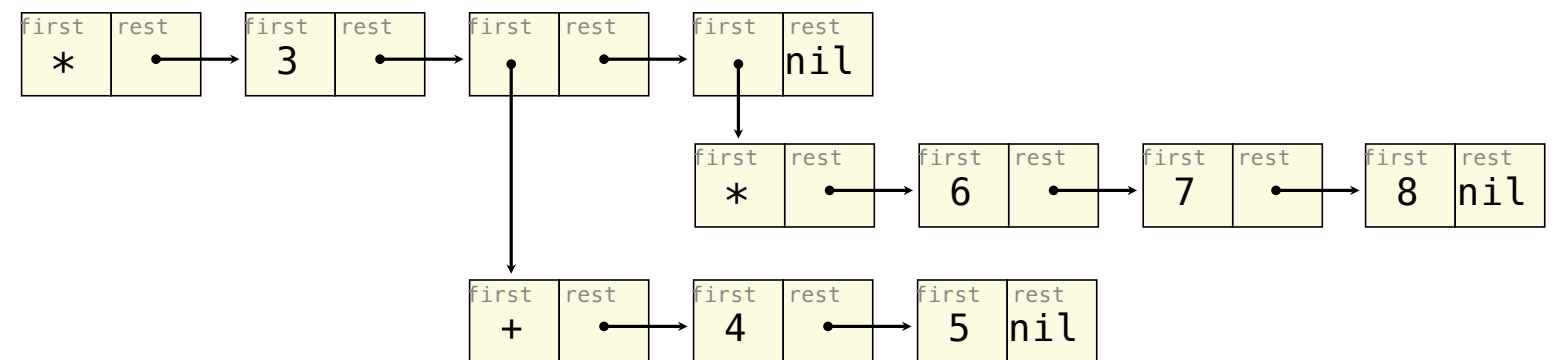
## Expression

```
(* 3  
  (+ 4 5)  
  (* 6 7 8))
```

## Expression Tree



## Representation as Pairs



(Demo with `scheme_reader`)

# The Eval Function

---

The eval function computes the value of an expression, which is always a number

It is a generic function that dispatches on the type of the expression (primitive or call)

## Implementation

---

```
def calc_eval(exp):  
    if isinstance(exp, (int, float)):  
        return exp  
    elif isinstance(exp, Pair):  
        arguments = exp.rest.map(calc_eval)  
        return calc_apply(exp.first, arguments)  
    else:  
        raise TypeError
```

Recursive call  
returns a number  
for each operand

'+', '-',  
'\*', '/'

A Scheme list  
of numbers

## Language Semantics

---

*A number evaluates...*

*to itself*

*A call expression evaluates...*

*to its argument values*

*combined by an operator*

# Applying Built-in Operators

---

The apply function applies some operation to a (Scheme) list of argument values

In calculator, all operations are named by built-in operators: +, -, \*, /

## Implementation

---

```
def calc_apply(operator, args):
    if operator == '+':
        return reduce(add, args, 0)
    elif operator == '-':
        ...
    elif operator == '*':
        ...
    elif operator == '/':
        ...
    else:
        raise TypeError
```

## Language Semantics

---

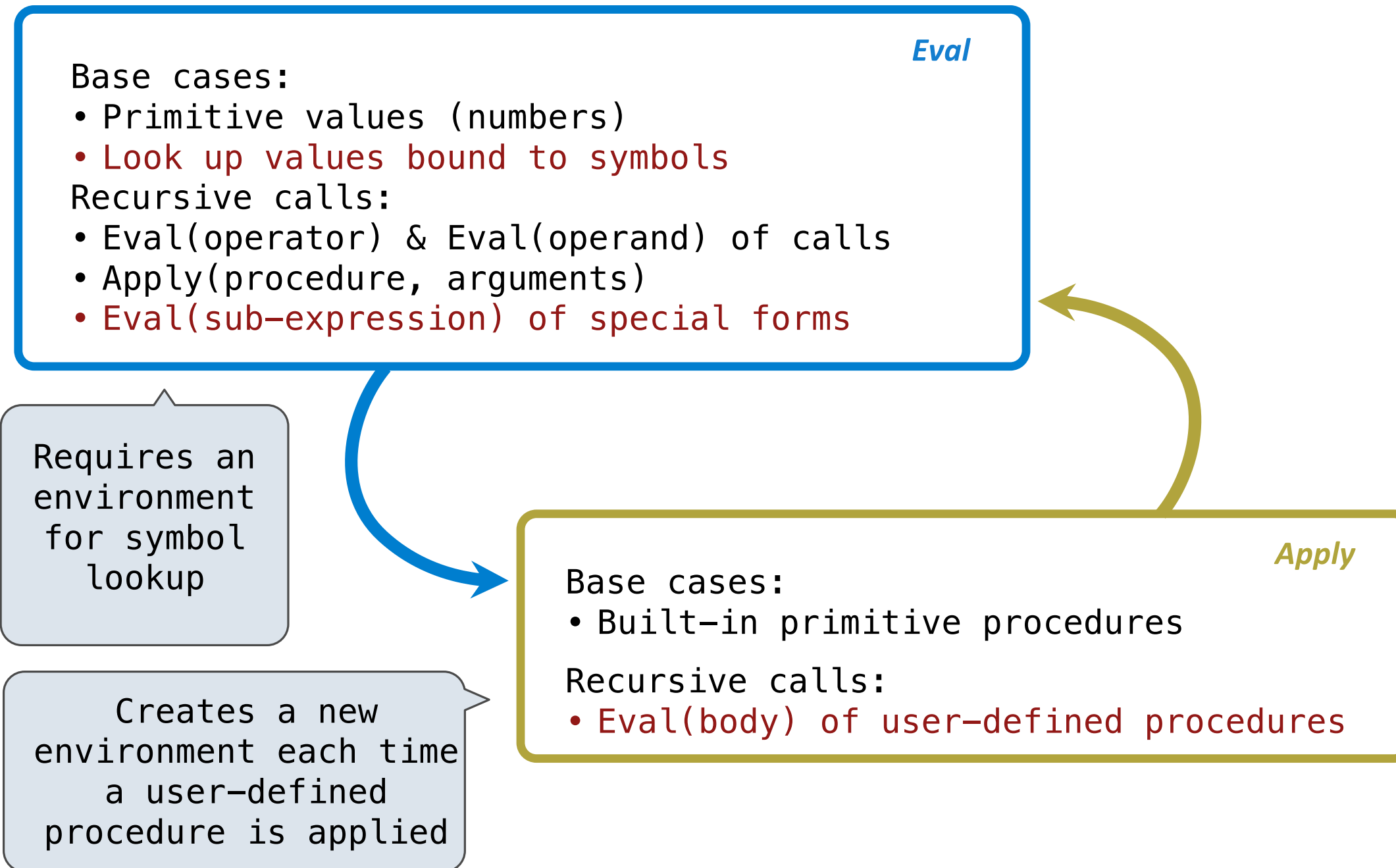
```
+:
    Sum of the arguments
-:
...
...
...
```

(Demo with trace)

# Interpreting Scheme

# The Structure of an Interpreter

---



# Project 4

## Pairs in Project 4: Scheme

---

<https://cs61a.org/proj/scheme/>

**Tokenization/Parsing:** Converts text into Python representation of Scheme expressions:

- Numbers are represented as numbers
- Symbols are represented as strings
- Lists are represented as instances of the `Pair` class

**Evaluation:** Converts Scheme expressions to values while executing side effects:

- `scheme_eval(expr, env)` returns the value of an expression in an environment
- `scheme_apply(procedure, args)` applies a procedure to its arguments
- The Python function `scheme_apply` returns the return value of the procedure it applies

## Discussion Question: The Symbol of a Define Expression

---

Return the symbol of a define expression. There are two formats for define expressions:

`(define x (+ 2 3))` or `(define (f x) (+ x 3))`

```
def symbol(exp):
```

```
    """Given a define expression exp, return the symbol defined.
```

```
    >>> def_x = read_line("(define x (+ 2 3))")
```

```
    >>> def_f = read_line("(define (f x) (+ x 3))")
```

```
    >>> symbol(def_x)
```

```
    'x'
```

```
    >>> symbol(def_f)
```

```
    'f'
```

```
    """
```

```
    assert exp.first == 'define' and exp.rest is not nil and exp.rest.rest is not nil
```

```
    signature = exp.rest.first
```

```
    if scheme_symbolp(signature):
```

```
        return signature
```

```
    else:
```

```
        return signature.first
```

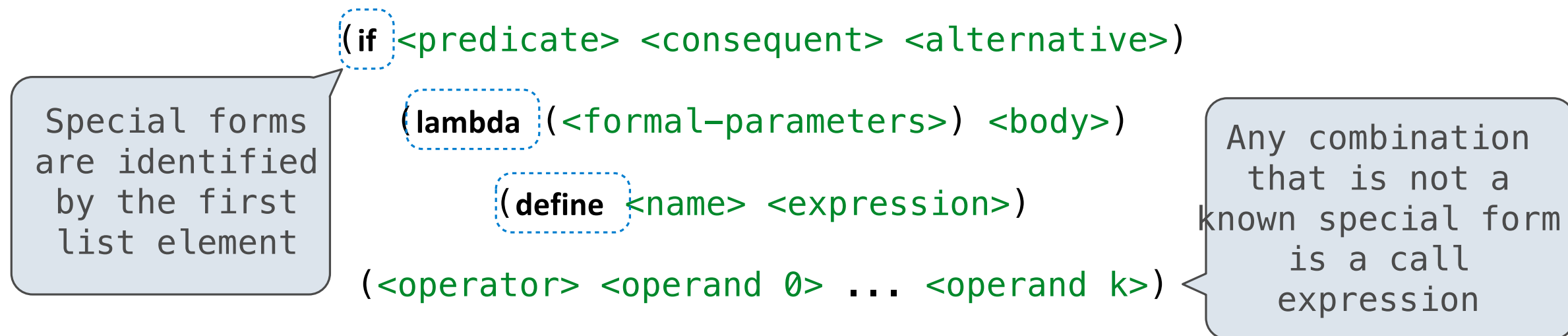
# Special Forms

# Scheme Evaluation

---

The `scheme_eval` function choose behavior based on expression form:

- Symbols are looked up in the current environment
- Self-evaluating expressions are returned as values
- All other legal expressions are represented as Scheme lists, called combinations



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

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

# 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 list of expressions
```

```
        self.env = env ..... A Frame instance
```

# 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
```

## Let's implement a special form for multiply in calculator!!

---

If any value is ever 0, stop evaluating and return 0

---