

# Interpreters

# Translation

## Problem:

Computers can only understand one language, binary (0s and 1s)

Humans can't really write a program using only 0s and 1s (not quickly anyways)

## Solution:

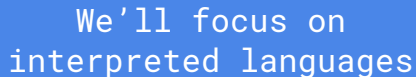
Programming languages

Languages like Python, Java, C, etc are *translated* to 0s and 1s

This translation step comes in a couple forms:

Compiled (pre-translated) - translate all at once and run later

Interpreted (translated on-the-fly) - translate while the program is running



We'll focus on  
interpreted languages

# Interpreters

An **interpreter** does 3 things:

**Reads** input from user in a specific programming language

Translates input to be computer readable and **evaluates** the result

**Prints** the result for the user

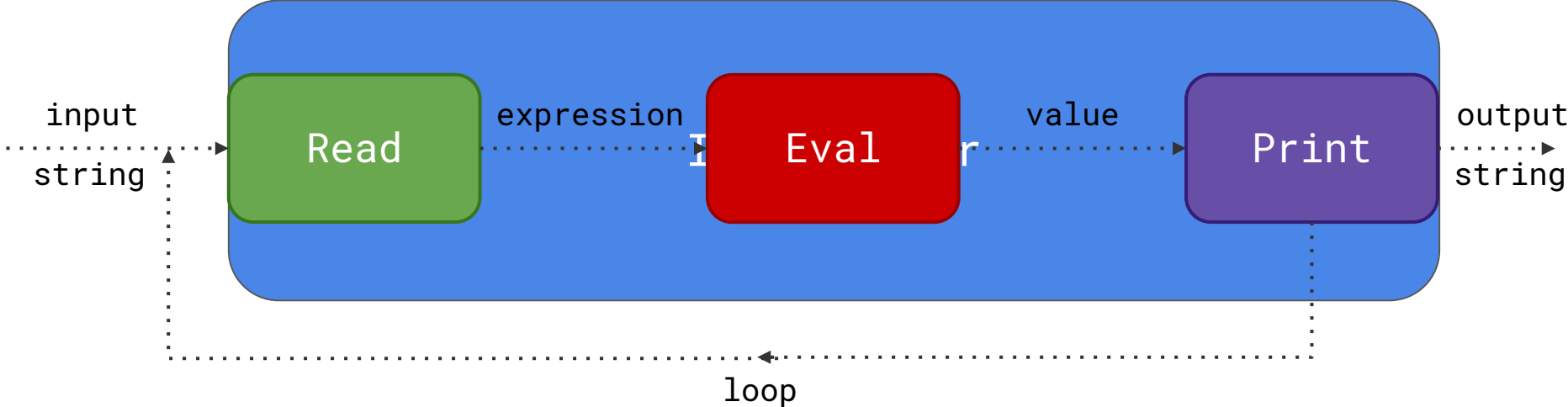
There are two languages involved:

**Implemented language:** this is the language the user types in

**Implementation language:** this is the language interpreter is implemented in

**Implemented Language is translated into the Implementation Language**

# Read-Eval-Print Loop (REPL)



```
while True:  
    exp = read()  
    val = eval(exp)  
    print(val)
```

Read

# Reading Input

**Lexical Analysis (Lexer):** Turning the input into a collection of *tokens*

- A token: single input of the input string, e.g. literals, names, keywords, delimiters

**Syntactic Analysis (Parser):** Turning tokens into a representation of the expression in the implementing language

- The exact “representation” depends on the type of expression
- Types of Scheme Expressions: self-evaluating expressions, symbols, call expressions, special form expressions.



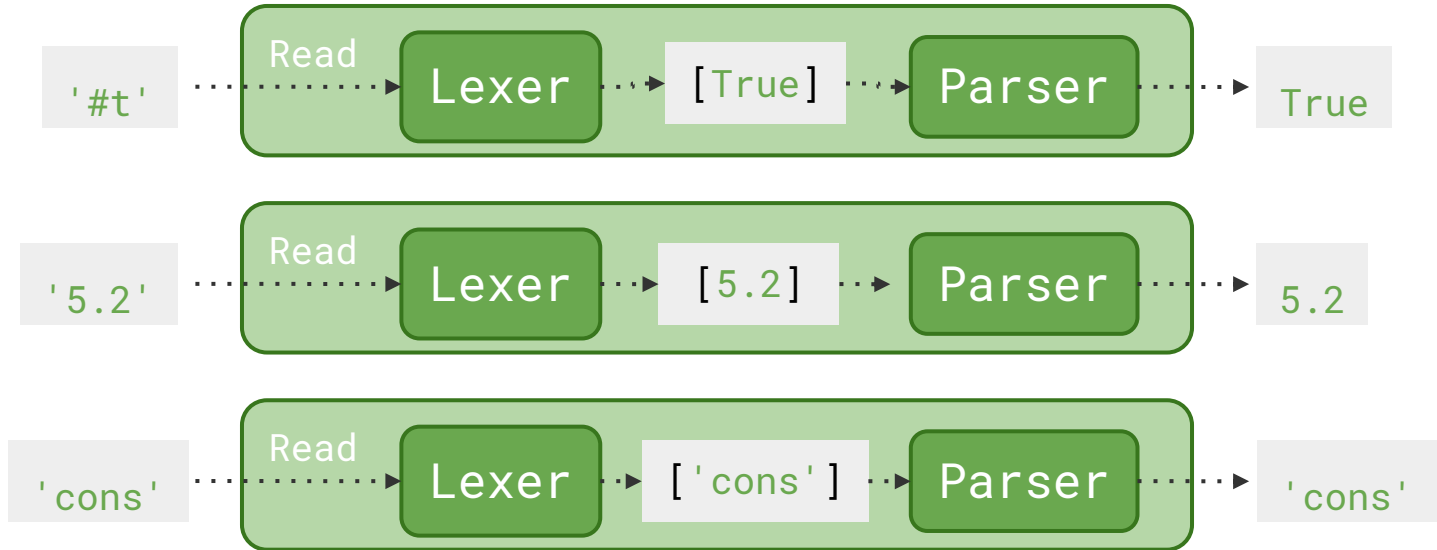
# Representing Scheme Primitive Expressions

## Self-Evaluating expressions (*booleans and numbers*)

Use Python booleans and Python numbers

## Symbols

Use Python strings



# Representing Combinations

(<operator> <operand1> <operand2> ...)

**Combinations** are just Scheme lists containing an operator and operands.

```
scm> (define expr '(+ 2 3)) ; Create the expression (+ 2 3)
expr
scm> (eval expr)           ; Evaluate the expression
5
scm> (car expr)            ; Get the operator
+
scm> (cdr expr)            ; Get the operands
(2 3)
```

```
>>> expr = ['+', 2, 3] # Representation of (+ 2 3)
>>> expr[0]           # Get the operator
'+'
>>> expr[1:]          # Get the operands
[2, 3]
```

Works, but isn't  
an exact  
representation of  
Scheme lists.



# Python Pairs

To accurately represent Scheme combinations as linked lists, let's write a `Pair` class in Python!

```
class Pair:
    def __init__(self, first, second):
        self.first = first
        self.second = second

    def __repr__(self):
        return 'Pair({0}, {1})'.format(
            self.first, self.second)
```

```
class nil:
    def __repr__(self):
        return 'nil'
```

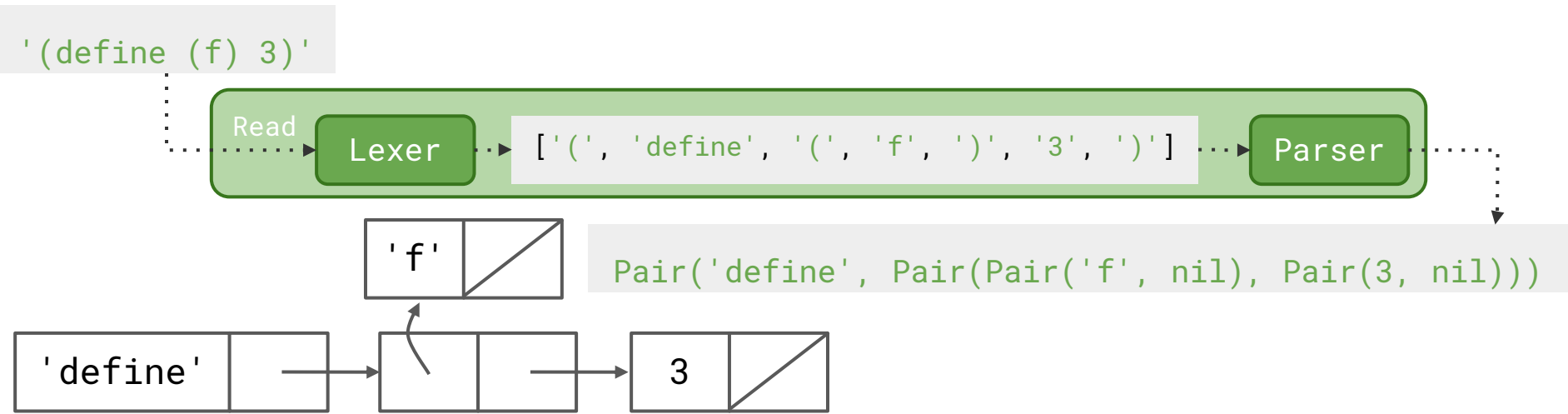
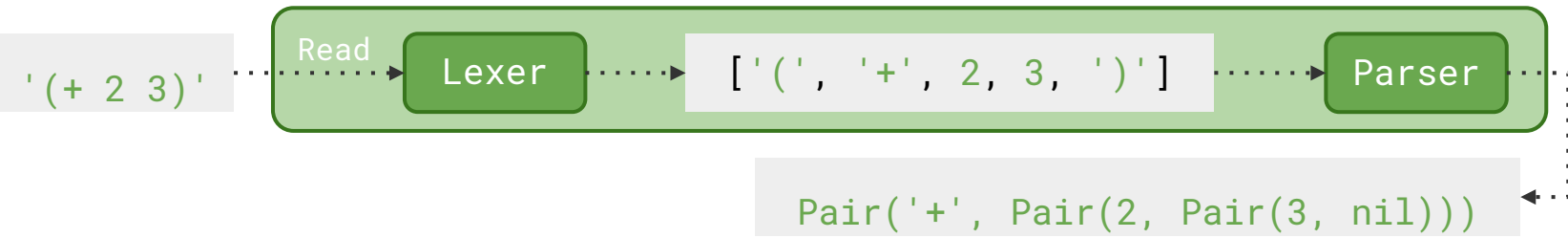
```
nil = nil()
```



There is one instance of `nil`. No other instances can be created.

```
>>> expr = Pair('+', Pair(2, Pair(3, nil))) # Represent (+ 2 3)
>>> expr.first                               # Get the operator
'+'
>>> expr.second                               # Get the operands
Pair(2, Pair(3, nil))
```

# Reading Combinations



# Special Case: quote

Recall that the quote special form can be invoked in two ways:

```
(quote <expr>)
```

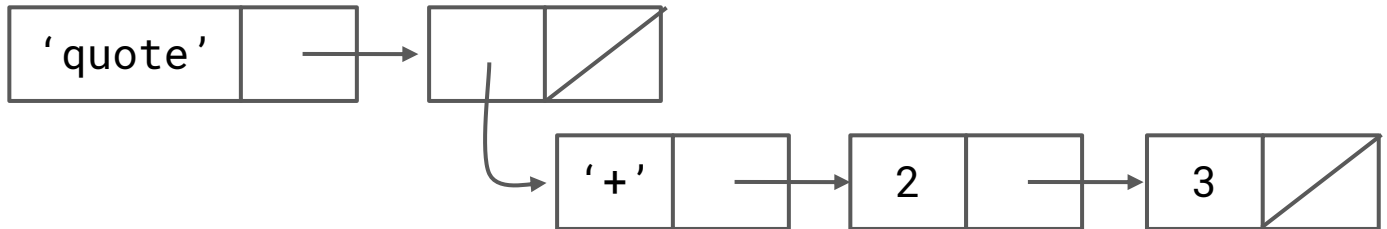
```
scm> 'hello
hello
scm> (quote hello)
hello
```

```
'<expr>
```

```
scm> '(1 2 3 4)
(1 2 3 4)
scm> (quote (1 2 3 4))
(1 2 3 4)
```

The special `'` syntax gets converted by the reader into a `quote` expression, which is a list with 2 elements:

```
Pair('quote', Pair(<expr>, nil))
```



```
'(+ 2 3)
```

```
Pair('quote', Pair(Pair('+', Pair(2, Pair(3, nil))), nil))
```

# Check your understanding

How would each of the Scheme expressions below be represented in Python when read by our interpreter? If it would be a Pair object, write out the constructor call for that Pair and draw out the corresponding box-and-pointer diagram.

ex) `(+ 2 3)`

`Pair('+', Pair(2, Pair(3, nil)))`

1) `4.67`

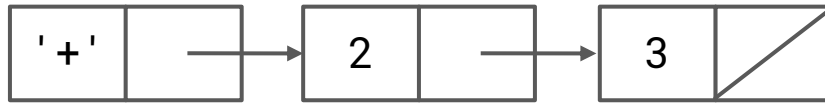
2) `#t`

3) `list`

4) `(cons 2 3)`

5) `(if (< x 0) 1 (+ x 1))`

6) `'hello`



## Check your understanding (soln)

1) `4.67`  
`4.67`

1) `#t`  
`True`

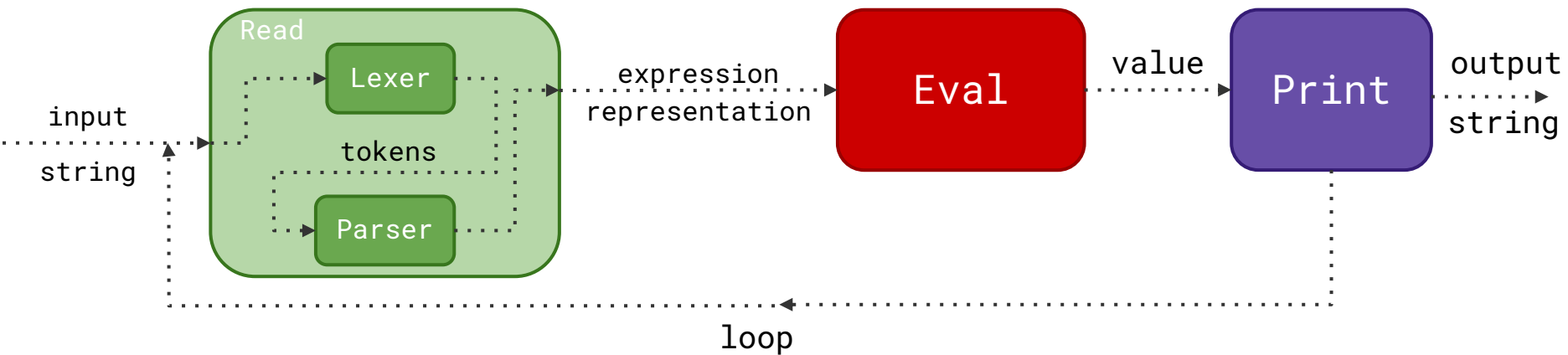
1) `list`  
`'list'`

1) `(cons 2 3)` ; lexer/parser does not care about this: 3 should be a pair  
`Pair('cons', Pair(2, Pair(3, nil)))`

1) `(if (< x 0) 1 (+ x 1))`  
`Pair('if', Pair(Pair('<', Pair('x', Pair(0, nil))), Pair(1, Pair(Pair('+', Pair('x', Pair(1, nil))), nil))))`

1) `'hello`  
`Pair('quote', Pair('hello', nil))`

# Read-Eval-Print Loop (REPL)

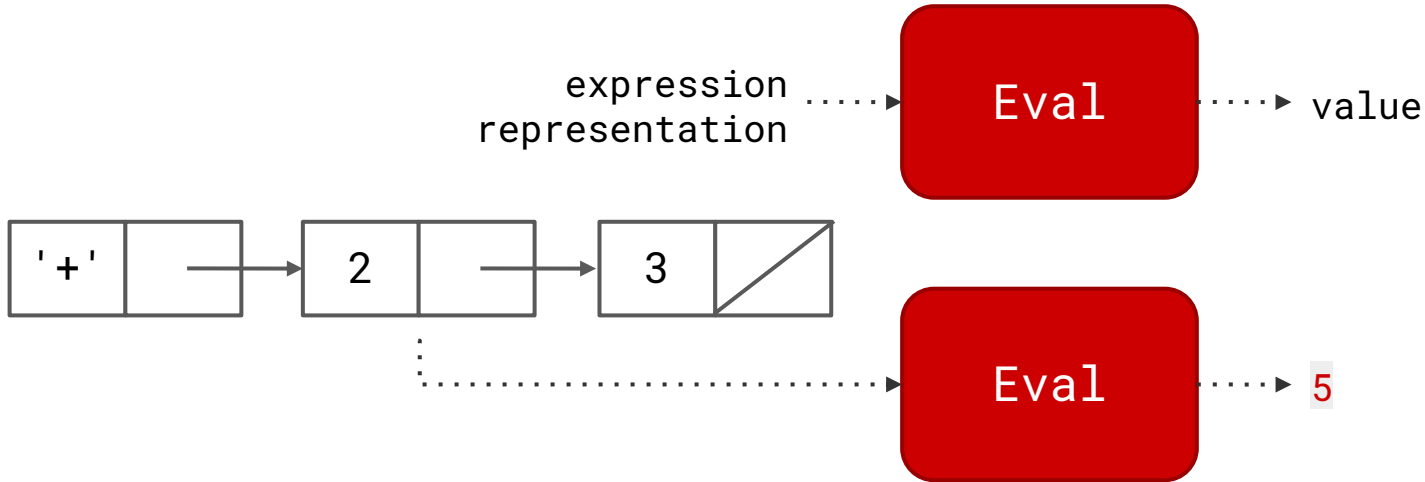


Eval

# Evaluating Expression

Rules for evaluating an expression depends on the expression's type.

Types of Scheme expressions: self-evaluating expressions, symbols, call expressions, special form expressions



Eval takes in one argument besides the expression itself: the current environment.



# Frames and Environments

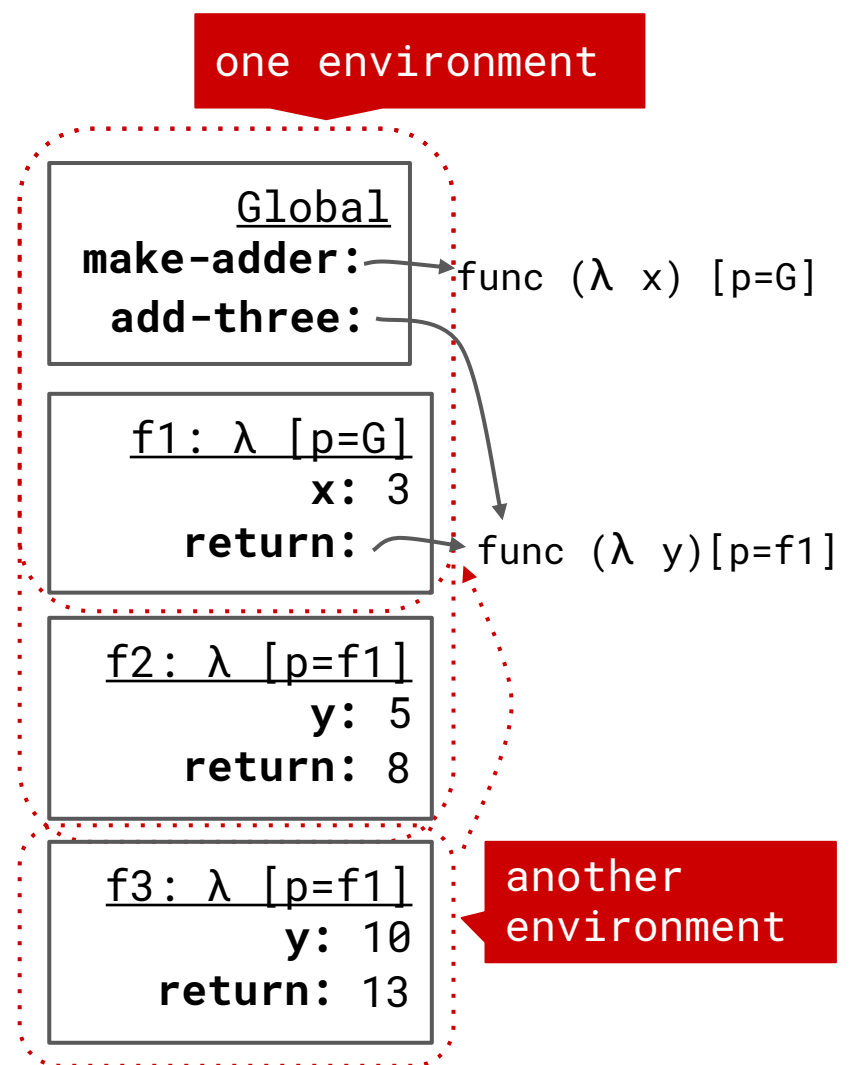
When evaluating expressions, the current **environment** consists of the current frame, its parent frame, and all its ancestor frames until the Global Frame.

```
(define (make-adder x)
  (lambda (y) (+ x y)))
```

```
(define add-three (make-adder 3))
```

```
(add-three 5)
```

```
(add-three 10)
```



# Frames in our interpreter

Frames are represented in our interpreter as instances of the **Frame** class

Each **Frame** instance has two instance attributes:

- **bindings**: a dictionary that binds Scheme symbols (Python strings) to Scheme values
- **parent**: the parent frame, another **Frame** instance

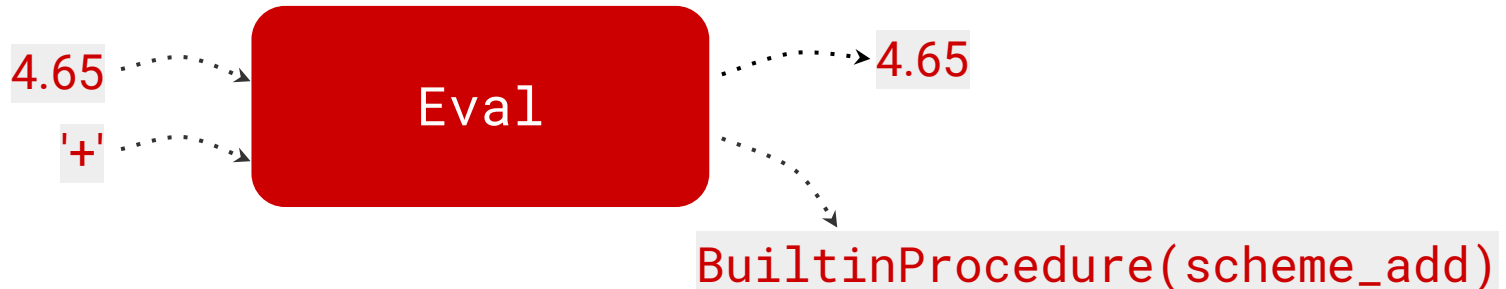
The evaluator needs to know the current environment, given as a single **Frame** instance, in order to look up names in expressions.

# Evaluating primitive expressions

**Self-evaluating expressions:** These expressions evaluate to themselves.

**Symbols:**

- 1) Look in the current frame for the symbol. If it is found, return the value bound to it.
- 2) If it is not found in the current frame, look in the parent frame. If it is not found in the parent frame, look in its parent frame, and so on.
- 3) If the global frame is reached and the name is not found, raise a `SchemeError`.



# Evaluating Combinations

`(<operator> <operand1> <operand2> ...)`

The operator of a combination tells us whether it is a special form expression or a call expression.

If the operator is a symbol and is found in the dictionary of special forms, the combination is a special form.

- Each special form has special rules for evaluation.

Otherwise, the combination is a call expression.

First two steps are recursive calls to eval.

**Step 1.** Evaluate the operator to get a procedure.

**Step 2.** Evaluate all of the operands from left to right.

**Step 3.** Apply the procedure to the values of the operands.

How does apply work?

# Types of Procedures

A **built-in procedure** is a procedure that is predefined in our Scheme interpreter, e.g. `+`, `list`, `modulo`, etc.

- Each built-in procedure has a corresponding Python function that performs the appropriate operation.
- In our interpreter -- instances of the `BuiltinProcedure` class

A **user-defined procedure** is a procedure defined by the user, either with a lambda expression or a define expression.

- Each user-defined procedure has
  1. a list of formal parameters
  2. a body (which is a Scheme list)
  3. a parent frame.
- In our interpreter -- instances of the `LambdaProcedure` class

# Built-In Procedures

```
scm> (+ 4 (* 2 3))  
10
```

## Applying built-in procedures:

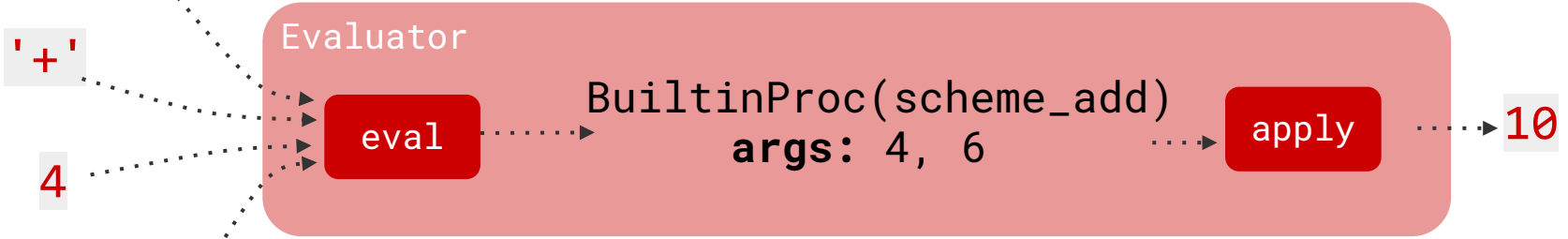
- Call the Python function that implements the built-in procedure on the arguments.

**operator:**  
expr.first

**operands:**  
expr.second



```
Pair('+', Pair(4, Pair(Pair('*',  
Pair(2, Pair(3, nil))), nil)))
```



```
Pair('*', Pair(2, Pair(3, nil)))
```

# User-Defined Procedures

```
scm> (define (square x) (* x x))
square
scm> (square 5)
25
```

**operator:**  
expr.first

**operands:**  
expr.second



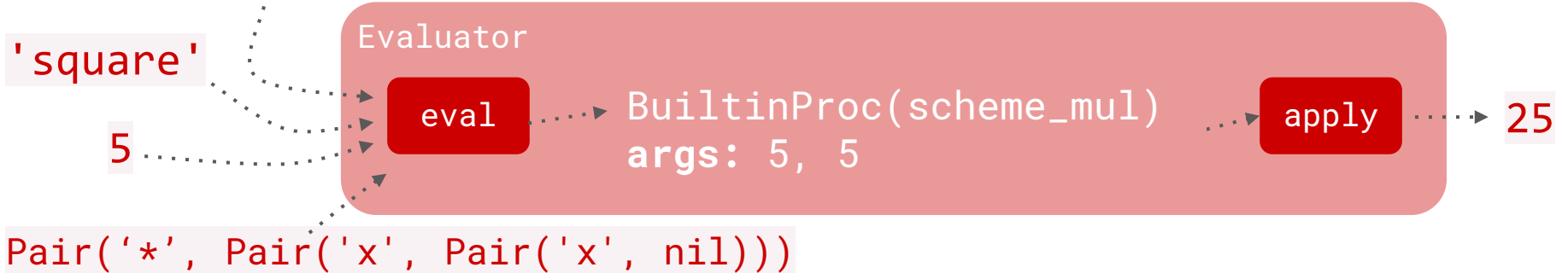
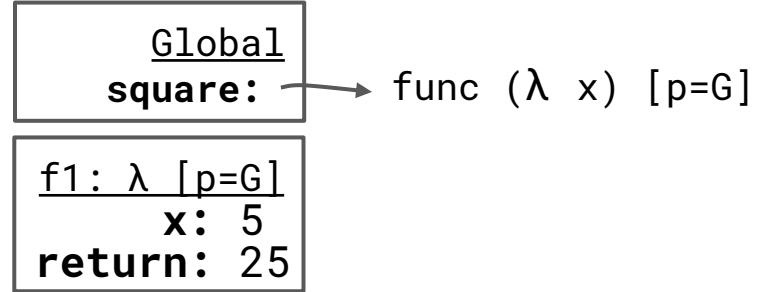
Pair('square', Pair(5, nil))

## Applying user-defined procedures:

**Step 1.** Open a new frame whose parent is the parent frame of the procedure being applied.

**Step 2.** Bind the formal parameters of the procedure to the arguments in the new frame.

**Step 3.** Evaluate the body of the procedure in the new frame.



# The evaluator

The evaluator consists of two *mutually-recursive* components:

## Evaluator

### Eval

#### Base Cases:

- Self-evaluating expressions
- Look up values bound to symbols

#### Recursive Cases:

- **Eval(operator)**, **Eval(o)** for each operand **o**
- **Apply(proc, args)**
- **Eval(expr)** for expression **expr** in body of special form

### Apply

#### Base Cases:

- Built-in procedures

#### Recursive Cases:

- **Eval(body)** of user defined procedures





# Counting eval/apply calls: built-in procedures

How many calls to eval and apply are made in order to evaluate this expression?

```
(+ 2 (* 4 1) 5)
```

- **eval**(Pair('+', Pair(2, Pair(Pair('\*', Pair(4, Pair(1, nil))), Pair(5, nil)))))
  - **eval**('+')
  - **eval**(2)
  - **eval**(Pair('\*', Pair(4, Pair(1, nil))))
    - **eval**('\*')
    - **eval**(4)
    - **eval**(1)
    - **apply**(BuiltinProc(scheme\_mul), [4, 1])
  - **eval**(5)
  - **apply**(BuiltinProc(scheme\_add), [2, 4, 5])

```
# calls:  
eval: 8  
apply: 2
```

# Counting eval/apply calls: user-defined procedures

How many calls to eval and apply are made in order to evaluate the second expression? (Assume the define expression has already been evaluated.)

```
(define (f x) (+ x 1))  
(* (f 3) 2)
```

- **eval**(Pair('\*',  
 Pair(Pair('f', Pair(3, nil))  
 Pair(2, nil))))
  - **eval**('\*')
  - **eval**(Pair('f', Pair(3, nil)))
    - **eval**('f')
    - **eval**(3)
    - **apply**(λ, 3)
      - **eval**(Pair('+', Pair('x', Pair(1, nil))))
        - **eval**('+')
        - **eval**('x')
        - **eval**(1)
        - **apply**(BuiltinProc(scheme\_add), [3, 1])
  - **eval**(2)
  - **apply**(BuiltinProc(scheme\_mul), [4, 2])

```
# calls:  
eval: 10  
apply: 3
```