Interpreters

Hi! I'm Bryce (he / him)

About Me:

- Incoming EECS masters student from the Bay Area
  - Recently graduated in CS + Data Science
- 5th semester on 61A staff (3rd time TA, 1st time Head TA)

Technical Interests:

- Current: Computer / Network Security
- Past: California education research, building web applications

Announcements

- Homework 5 and Lab 10 are due tomorrow (7/27)
- Ants project is due Friday (7/28), 1 EC for submitting by tomorrow (7/27)
  - Please submit to the correct autograder!
- Homework 4 Recovery is released and due by Monday (7/31)
- Please complete the Midsemester Feedback form if you still haven’t!
  - Form was linked on Lab 9 and is still open for submissions

Preface

Historically, interpreters have been a difficult topic for students

- We’ve been in your shoes before!

This lecture is meant to introduce what interpreters are

- You are **not** expected to understand everything after this lecture
- Will be reinforced in multiple lab/discussion sections (Discussion 9 + Lab 11) and your Scheme project
- Please ask questions as we go!

For security reasons, we can’t release the .py files for this lecture

- However, you’ll have coded your own version of today’s lecture after Lab 11 + Project 4

Programming Languages

Levels of Languages

High-level Language

(Python, Scheme, SQL, Java)

↓

Assembly Language

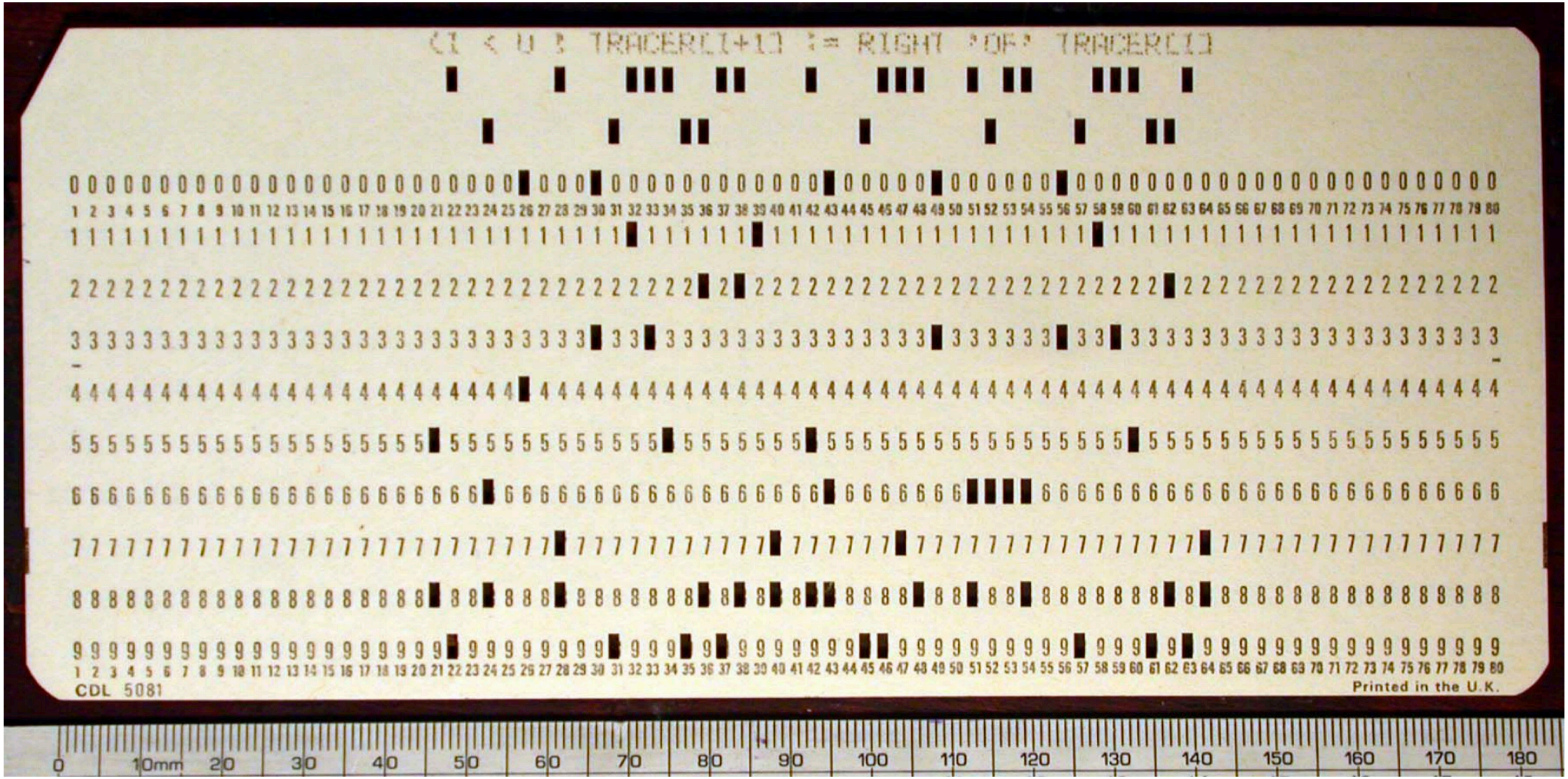
(RISC-V Assembly, x86 Assembly)

↓

Machine Language

(RISC-V Instruction Set, x86 Instruction Set)

Punchcard



Programming Languages

A computer typically executes programs written in many different programming languages

Machine languages: statements are interpreted by the hardware itself

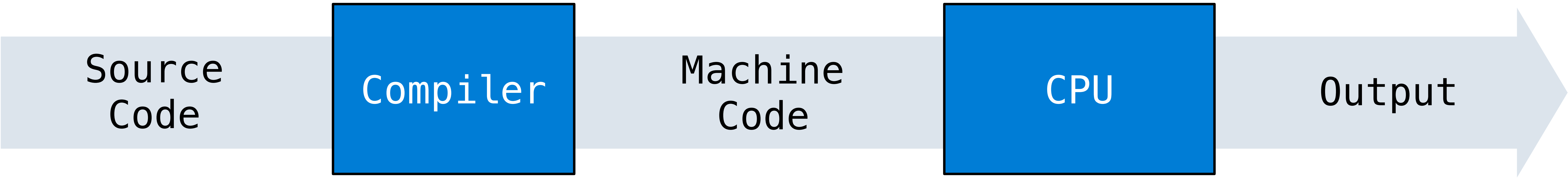
- A fixed set of instructions invoke operations implemented by the circuitry of the central processing unit (CPU)
- Operations refer to specific hardware memory addresses; no abstraction mechanisms

High-level languages: statements & expressions are interpreted by another program or compiled (translated) into another language

- Provide means of abstraction such as naming, function definition, and objects
- Abstract away system details to be independent of hardware and operating system

Compilers

Compilers: translate source code into machine code so that the machine code can be distributed and run repeatedly




```
int a = (b + c) - (d + e);
```
→

```
add t1, s1, s2
add t2, s3, s4
sub s0, t1, t2
```
→

```
0000 0001 0010 0100 1000 0011 0011 0011
0000 0001 0100 1001 1000 0011 1011 0011
0100 0000 0111 0011 0000 0100 0011 0011
```

C Code

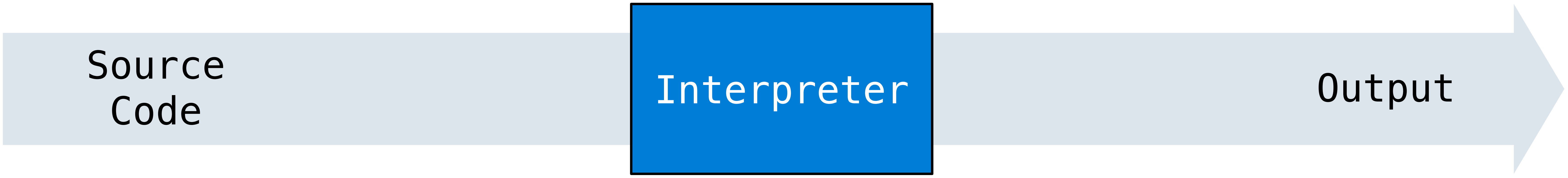
Assembly Language (RISC-V)

Machine Language

Interpreters

- In 61A, we focus on **interpreters**
- Compilers are explored in future courses (61C, 162, 164, etc.)

Interpreters: run source code directly producing an output/value, without first compiling it into machine code

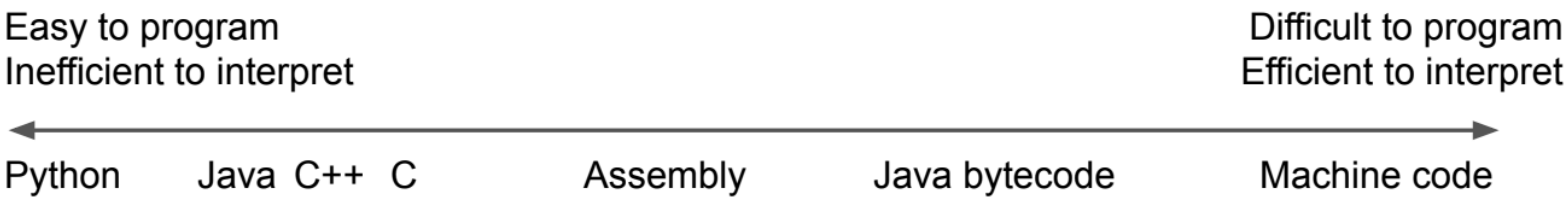


a = (b + c) - (d + e)

Evaluated!

Python Code

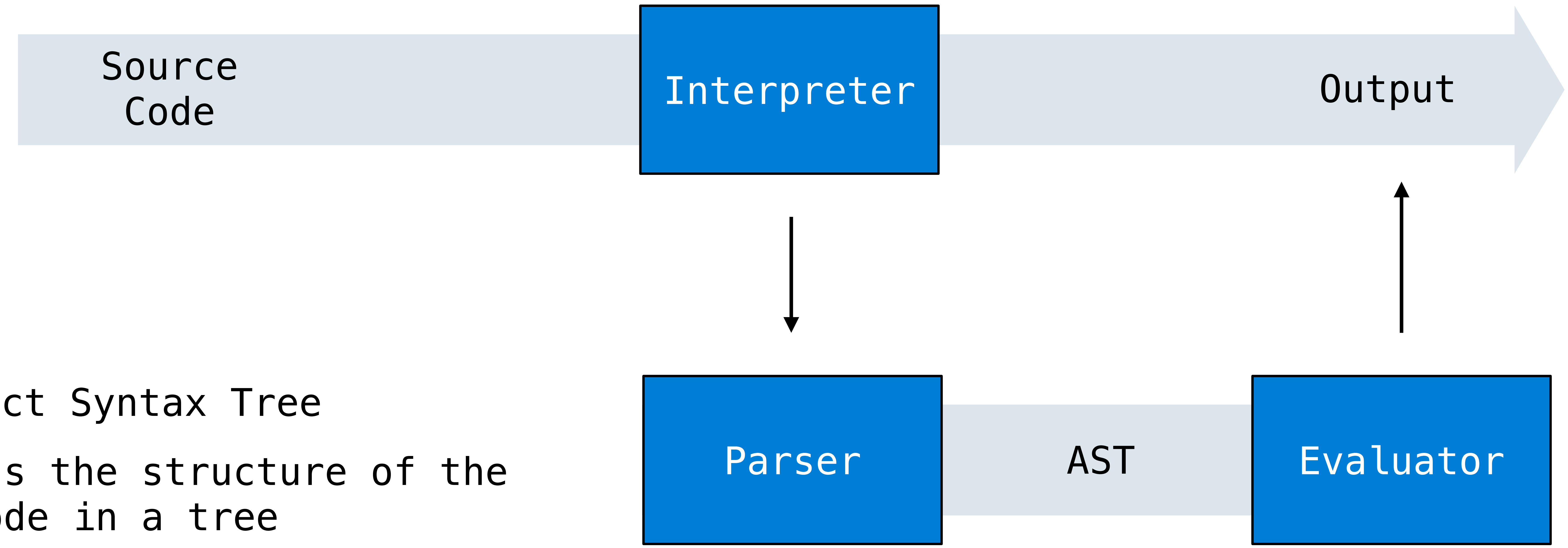
Tradeoffs:



Understanding Source Code

In order to interpret source code, a **parser** must be written to understand that source code

In the context of interpreters:



AST - Abstract Syntax Tree

- Represents the structure of the source code in a tree

Parsing

Reading Scheme Lists

All call expressions in Scheme are represented by a Scheme list

A Scheme list is written as elements in parentheses:

(<element_0> <element_1> ... <element_n>)

A Scheme list

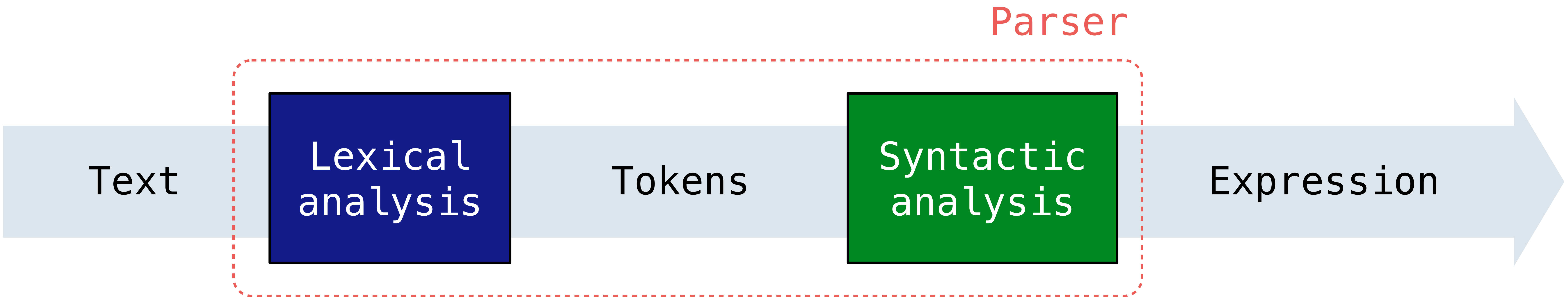
Each <element> can be a combination or primitive

- Combination - another Scheme list
- Primitive - simplest instance in Scheme (number, boolean, etc.)

```
(+ (* 3 (+ (* 2 4) (+ 3 5))) (+ (- 10 7) 6))
```

Parsing

A Parser takes in text and returns an expression that represents the text in a tree-like structure




```
'(+ 1'
'   (- 2 3)'
'   (* 4 5.6))'
```

```
'(', '+', 1
'(', '-', 2, 3, ')'
'(', '*', 4, 5.6, ')', ')'
```

```
Pair('+', Pair(1, ...))
```

printed as

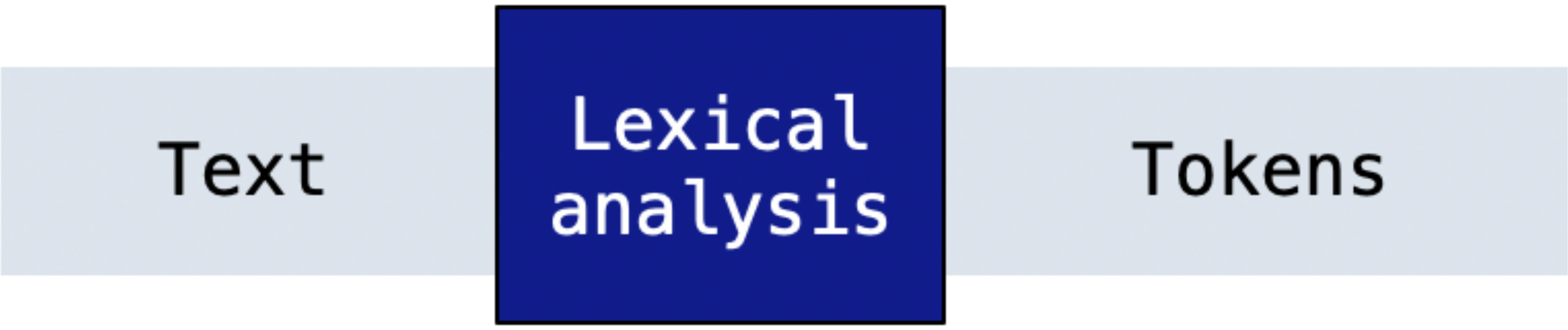
```
(+ 1 (- 2 3) (* 4 5.6))
```

Let's break this down!

Lexical Analysis

Lexical analysis converts input text into a list of tokens

- Each token represents **the smallest unit of information**




```
'(+ 1'
'   (- 2 3)'
'   (* 4 5.6))'
```

```
'(', '+', 1
'(', '-', 2, 3, ')'
'(', '*', 4, 5.6, ')', ')'
```

- Iterative process
- Processes one line at a time
- Checks for malformed tokens
- Determines types of tokens

Syntactic Analysis

Syntactic analysis identifies the hierarchical structure of an expression

- Formal way of representing the tokens generated from lexical analysis
- Symbols can be “nested”


```
'(', '+', 1
'(', '-', 2, 3, ')'
'(', '*', 4, 5.6, ')', ')'
```

```
Pair('+', Pair(1, ...))
```

printed as

```
(+ 1 (- 2 3) (* 4 5.6))
```

What exactly is a Pair?

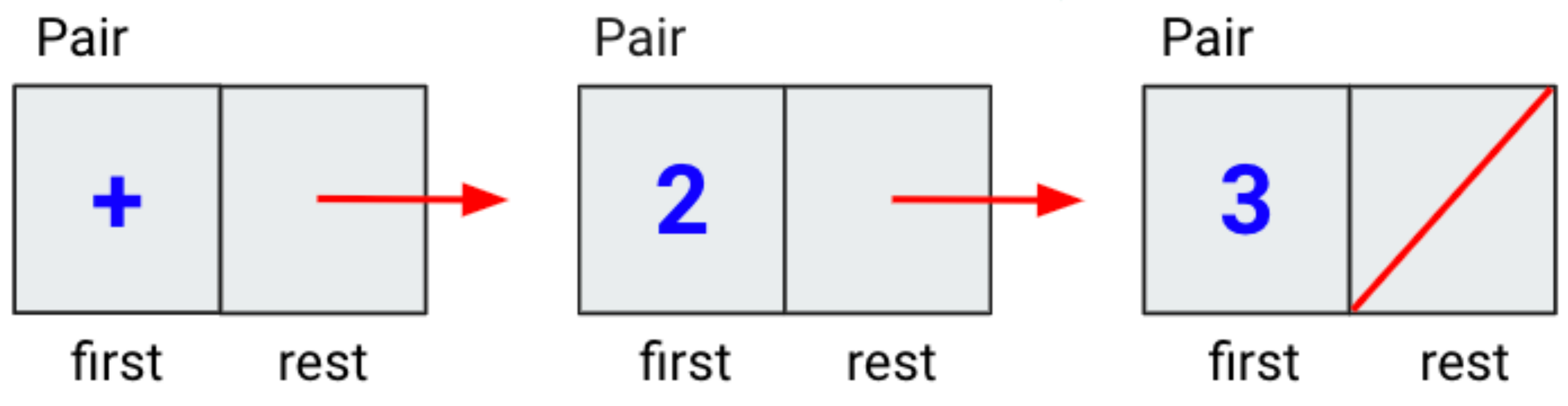
Pair Abstraction

A Pair is similar to a linked list!

Syntactic Analysis

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

(+ 2 3)

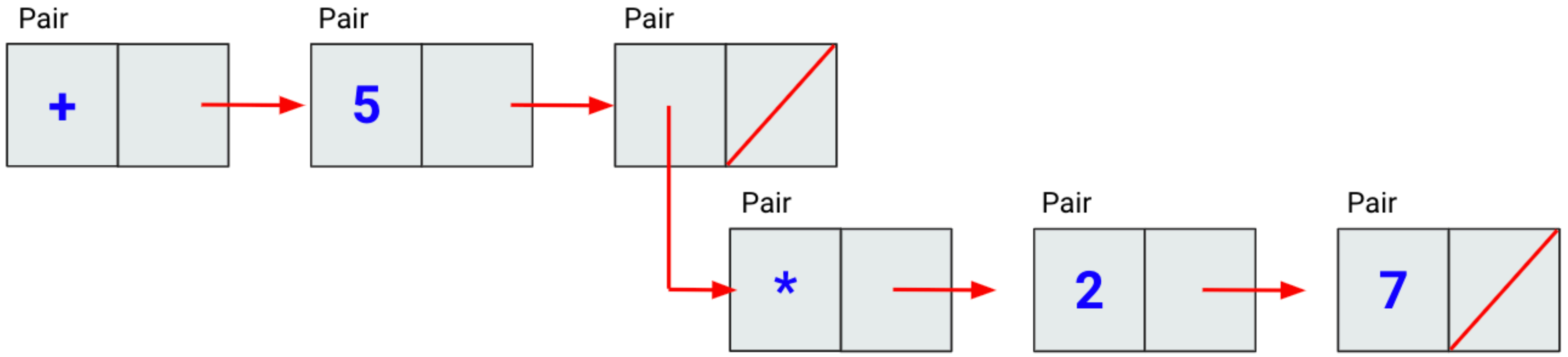


We can also create nested expressions:

Syntactic Analysis

```
'(', '+', 5,                    Pair('+', Pair(5, Pair(
     '(', ‘*’, 2, 7, ')', ')'   →     Pair(‘*’, Pair(2, Pair(7, nil))), nil)))
```

(+ 5 (* 2 7))



Generating Pairs

We define a function called scheme_read that will consume the input tokens for exactly one expression.

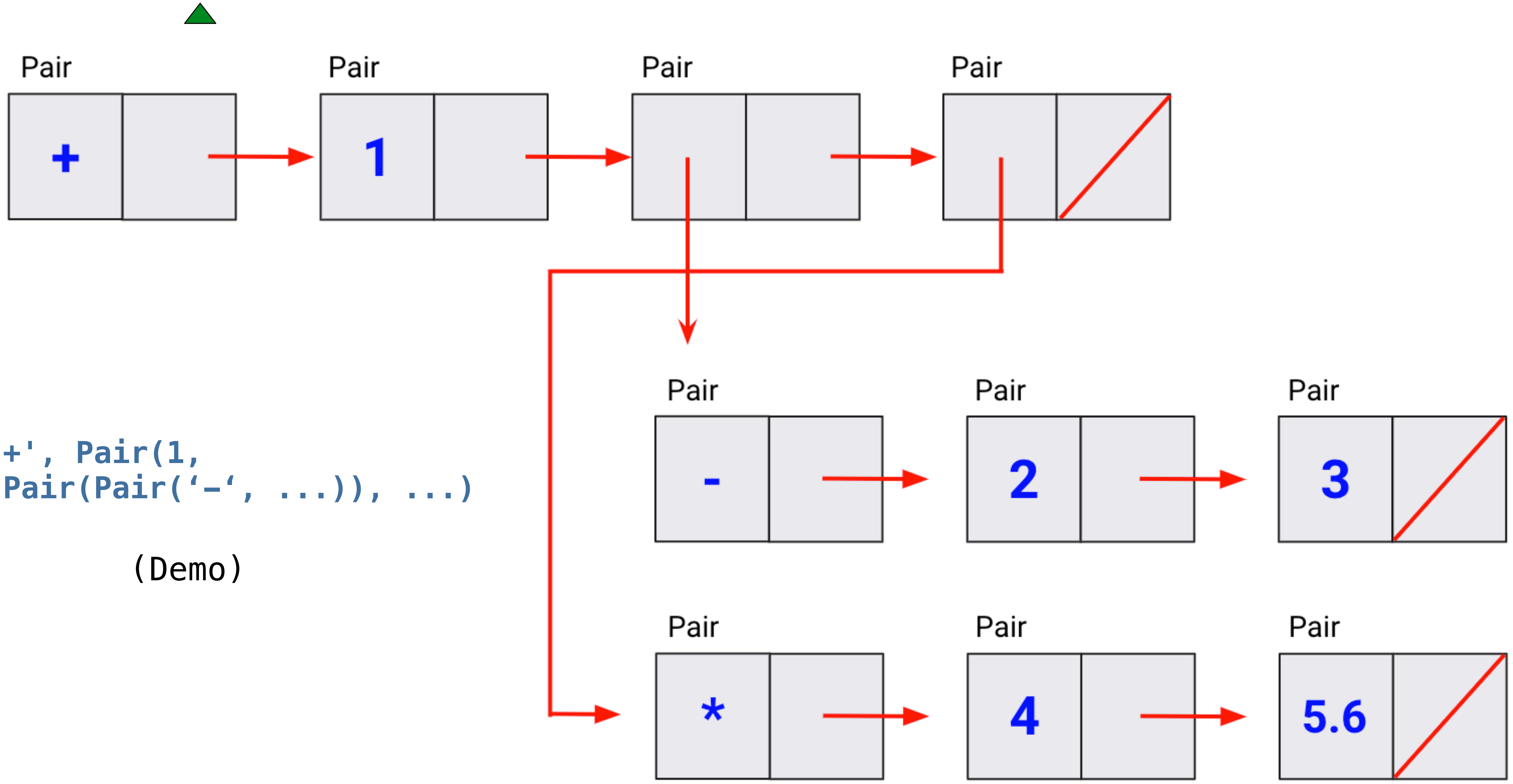
- This expression can have nested expressions
- Recursive problem in nature
- **Builds** the Pair object for us

Base case: symbols and numbers

Recursive call: scheme_read sub-expressions and combine them

Generating Pairs

```
'(', '+', 1, '(', '-', 2, 3, ')', '(', '*', 4, 5.6, ')', ')'
```




```
Pair('+', Pair(1,
      Pair(Pair('-', ...)), ...)
```

(Demo)

Syntactic Analysis

Syntactic analysis identifies the hierarchical structure of an expression

- Formal way of representing the tokens generated from lexical analysis
- Symbols can be “nested”


```
'(', '+', 1
'(', '-', 23, ')'
'(', '*', 4, 5.6, ')', ')'
```

```
Pair('+', Pair(1, ...))
```

printed as

```
(+ 1 (- 23) (* 4 5.6))
```

- Recursive process
- Processes multiple lines
- Balances parentheses
- Returns tree structure

The Calculator Language

`(You’ll implement this in Lab 11!)`

Calculator Syntax

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

A primitive expression is a number: 2 -4 5.6

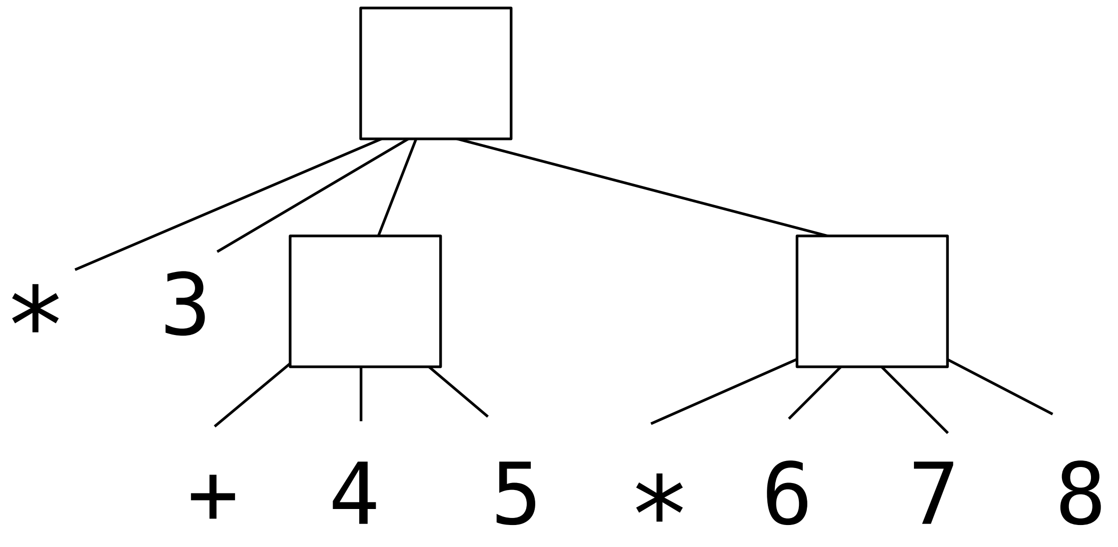
A call expression is a combination that begins with an operator (+, -, *, /) followed by 0 or more expressions: (+ 1 2 3) (/ 3 (+ 4 5))

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

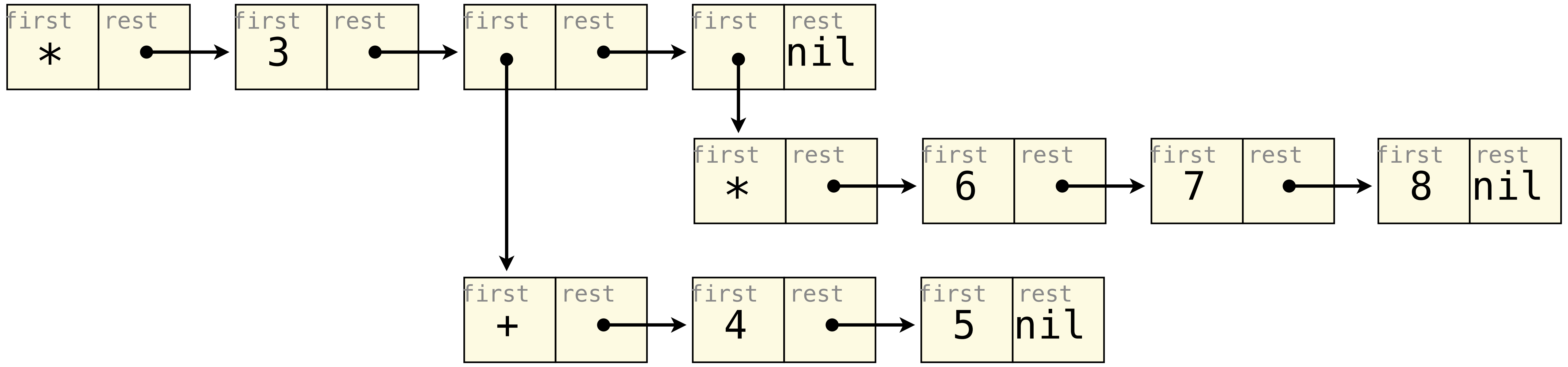
Expression

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

Expression Tree



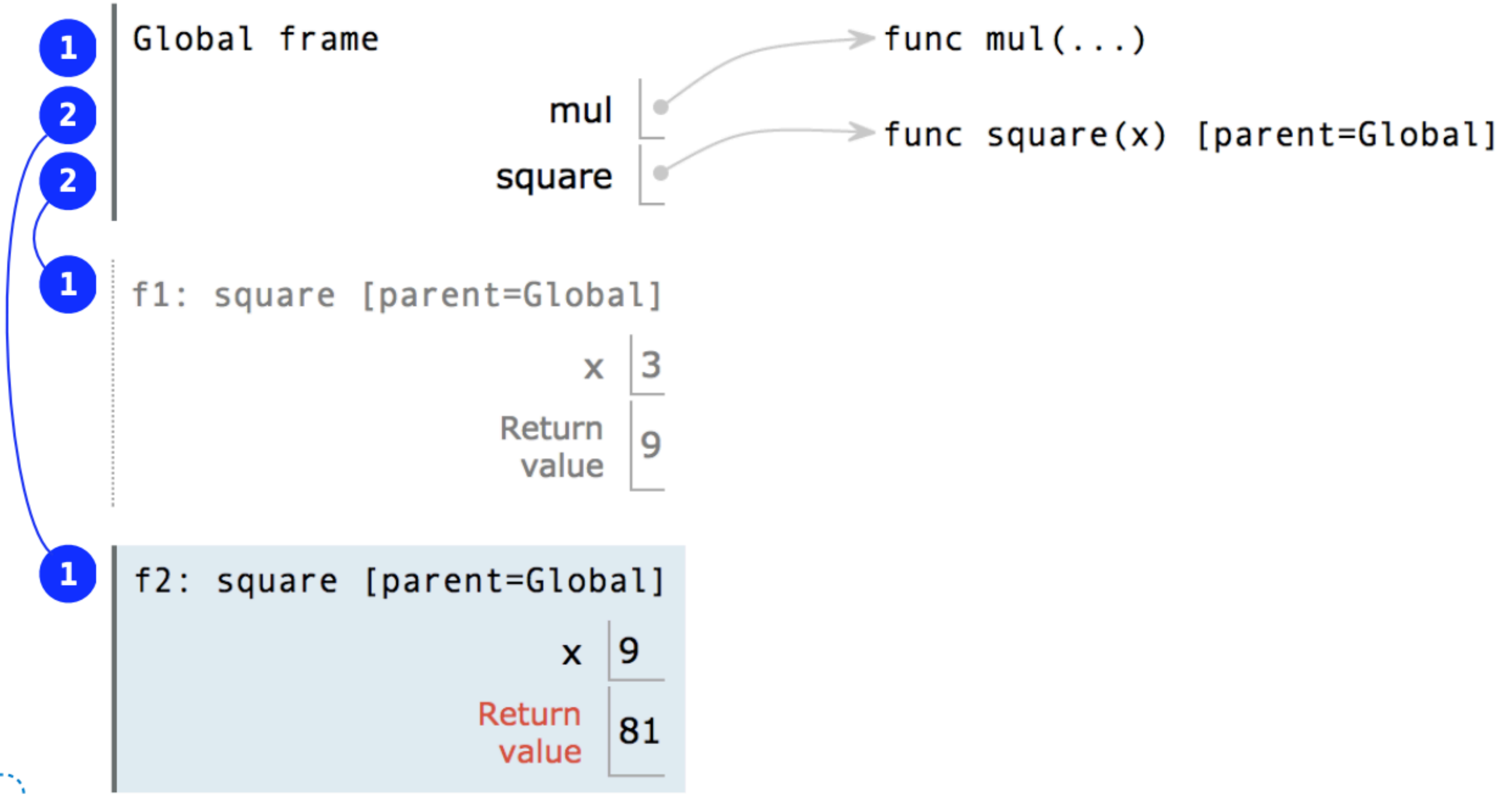
Representation as Pairs

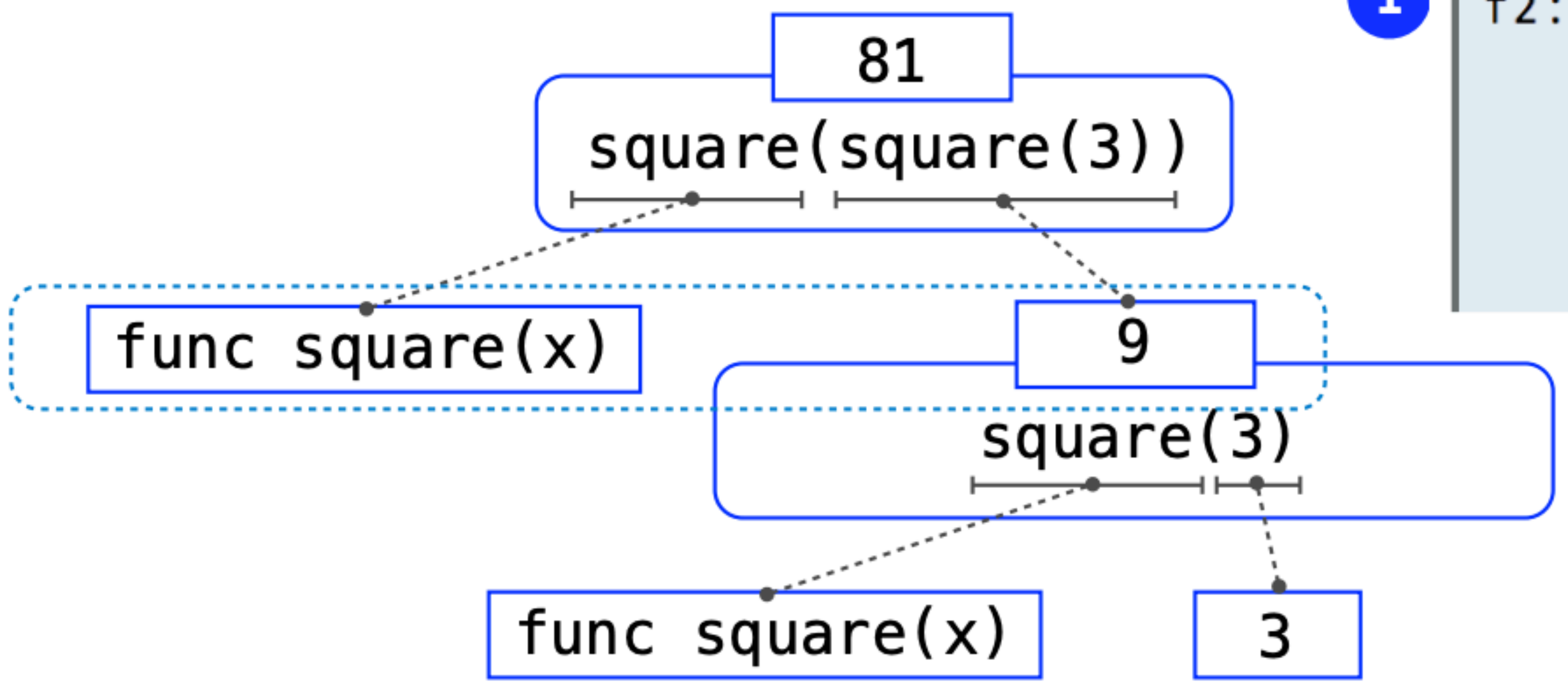


Expression Trees

We've seen expression trees before! Think back to Lecture 3 [Control]:

```
from operator import mul
def square(x):
    return mul(x, x)
square(square(3))
```





An environment is a sequence of frames.

- The global frame alone
- A local, then the global frame

Calculator Semantics

The value of a calculator expression is defined recursively.

Primitive: A number evaluates to itself.

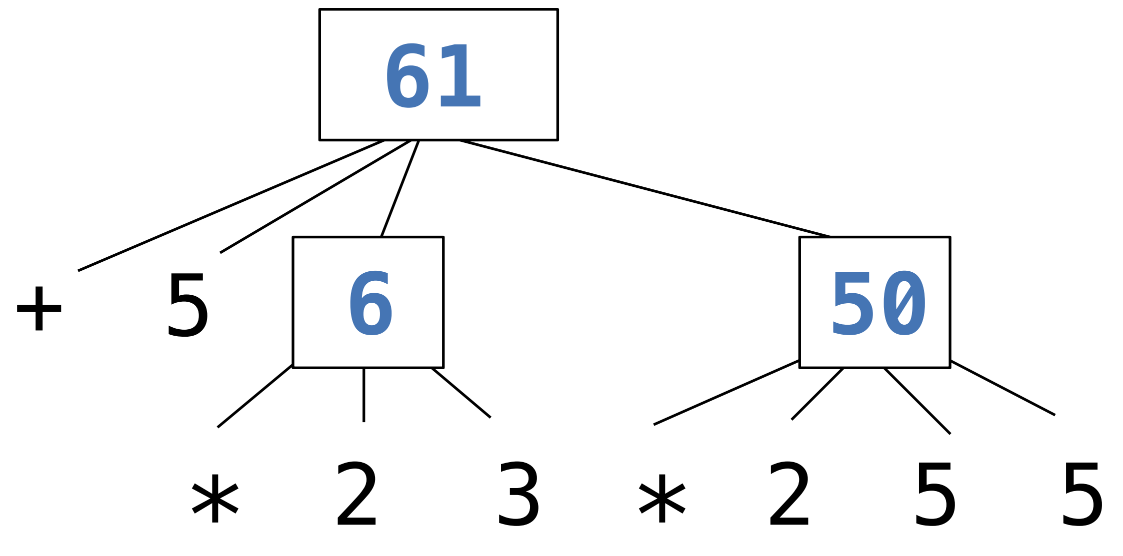
Call: A call expression evaluates to its argument values combined by an operator.

- **+**: Sum of the arguments
- *****: Product of the arguments
- **-**: If one argument, negate it. If more than one, subtract the rest from the first.
- **/**: If one argument, invert it. If more than one, divide the rest from the first.

Expression

```
(+ 5
   (* 2 3)
   (* 2 5 5))
```

Expression Tree



(Demo)

Evaluation

The Eval Function

```
(+ 5
   (* 2 3)
   (* 2 5 5))
```

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

In calculator, an expression is either a **number** or a **Pair**

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

-:

...

...

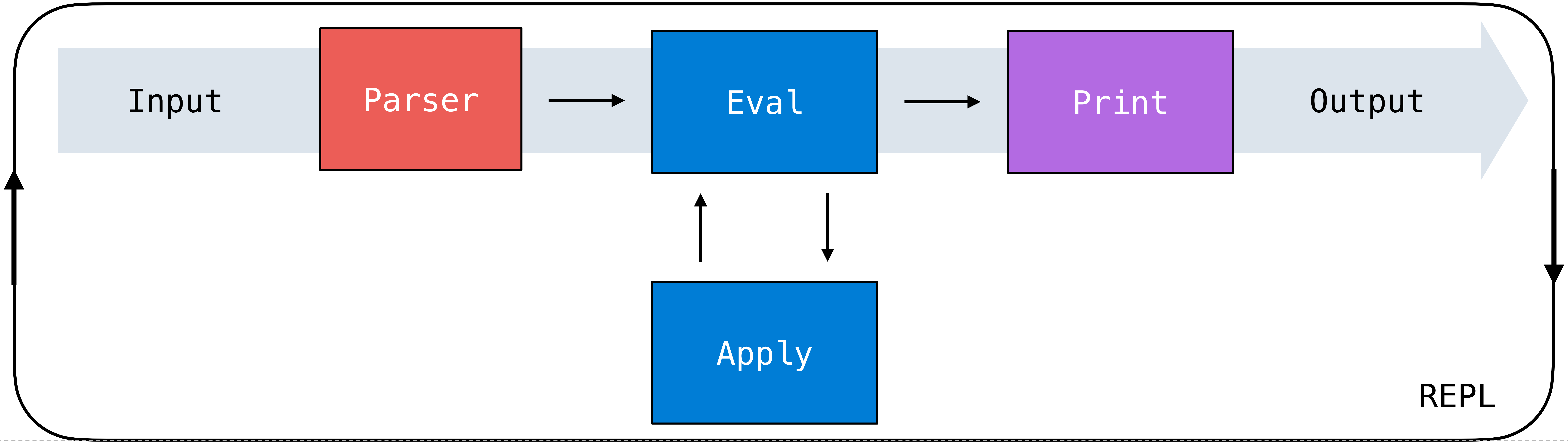
(Demo)

Interactive Interpreters

Read-Eval-Print Loop

The user interface for many programming languages is an interactive interpreter

1. **Read** text input from the user
2. Parse the text input into an expression
3. **Evaluate** the expression
4. If any errors occur, report those errors, otherwise
5. **Print** the value of the expression and repeat



Raising Exceptions

Exceptions are raised within lexical analysis, syntactic analysis, eval, and apply

Example exceptions

- **Lexical analysis:** The token 2.3.4 raises ValueError("invalid numeral")
- **Syntactic analysis:** An extra) raises SyntaxError("unexpected token")
- **Eval:** An empty combination raises TypeError("() is not a number or call expression")
- **Apply:** No arguments to - raises TypeError("- requires at least 1 argument")

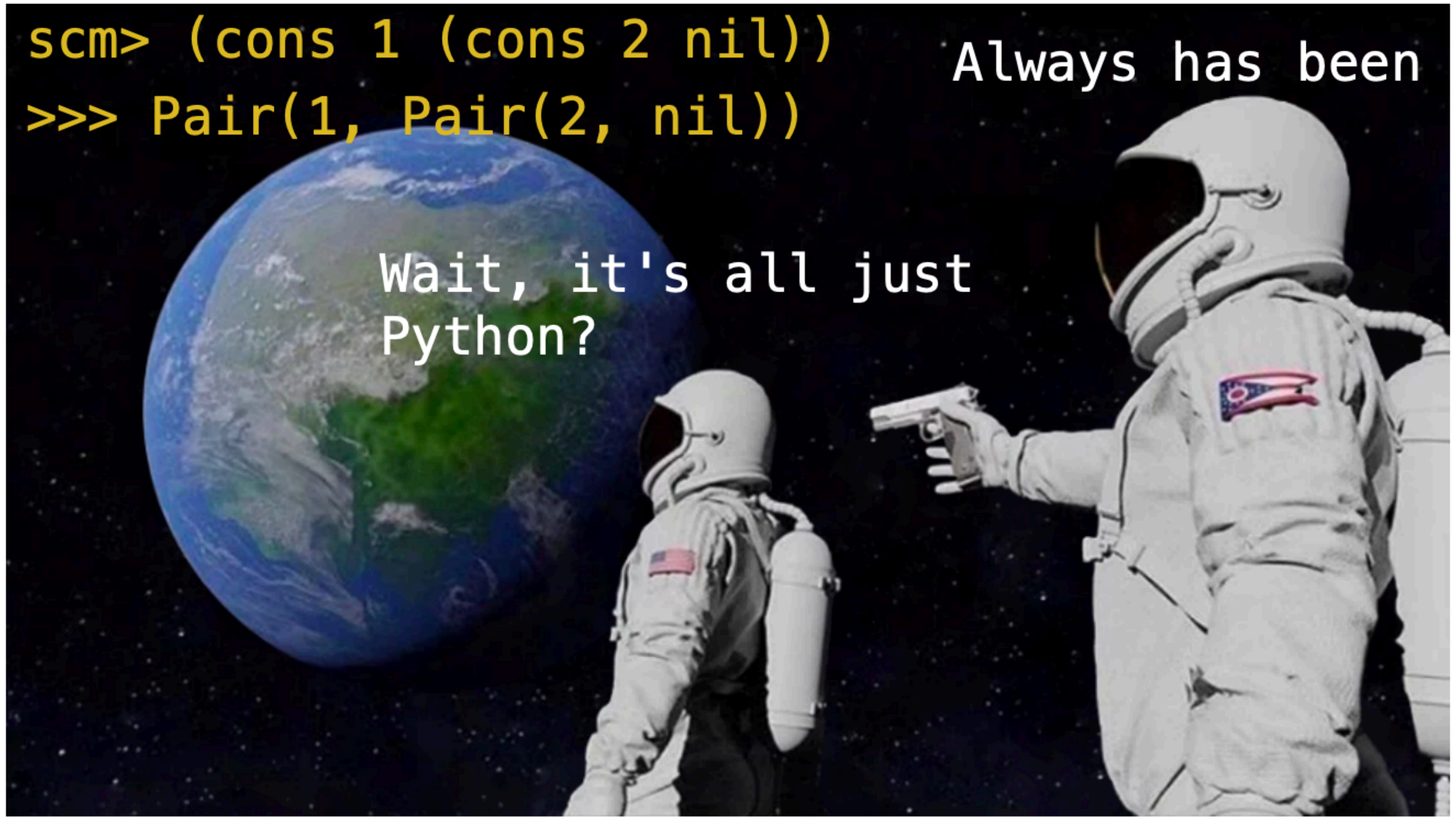
(Demo)

Handling Exceptions

An interactive interpreter prints information about each error

A well-designed interactive interpreter should not halt completely on an error, so that the user has an opportunity to try again in the current environment

Break



scm> (cons 1 (cons 2 nil))
>>> Pair(1, Pair(2, nil))
Always has been
Wait, it's all just Python?

Interpreting Scheme

The Structure of an Interpreter

Eval

Base cases:

- Primitive values (numbers)
- Look up values bound to symbols (i.e. variables)

Recursive calls:

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

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

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

```
(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 forms may only evaluate some sub-expressions

- **If** expression: (**if** <predicate> <consequent> <alternative>)
- **And** and **or**: (**and** <e1> ... <en>), (**or** <e1> ... <en>)
- **Cond** expression: (**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 to get the value of the whole expression

do_if_form

(Demo)

Quotation

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 that starts with quote

(Demo)

Lambda Expressions

Lambda Expressions

Lambda expressions evaluate to user-defined procedures

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

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

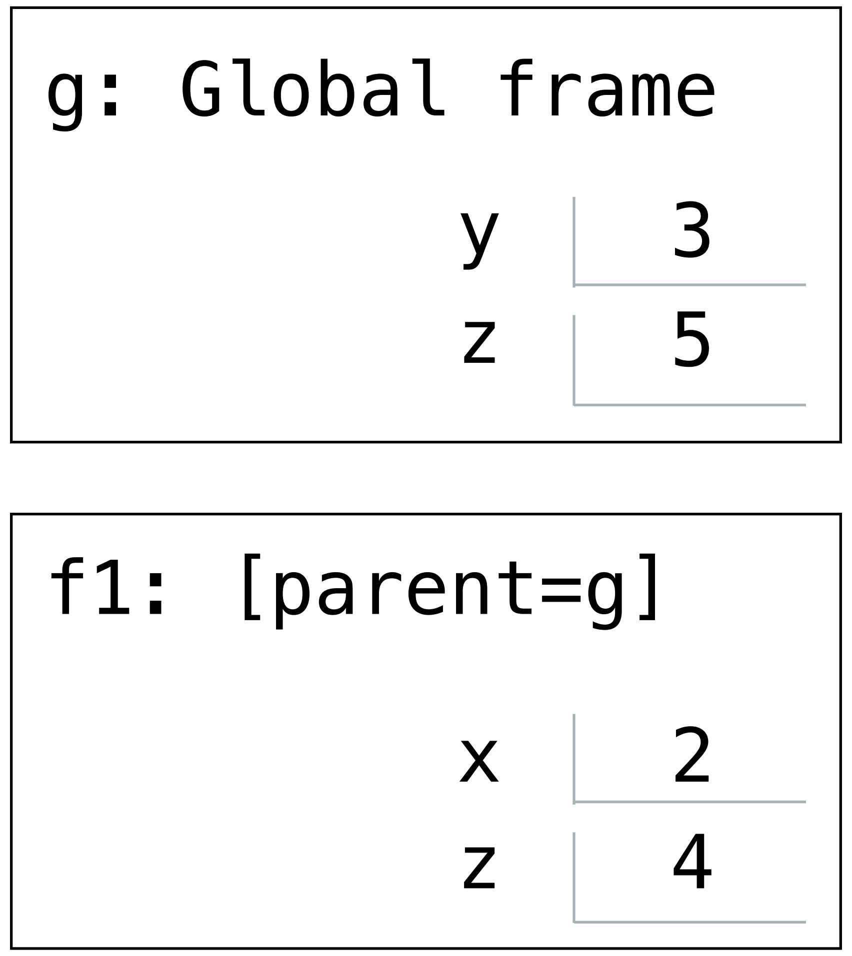
```
class LambdaProcedure:
    def __init__(self, formals, body, env):
        self.formals = formals ........ A scheme list of symbols
        self.body = body .............. A scheme list of expressions
        self.env = env ................ A Frame instance
```

Frames and Environments

A frame represents an environment that has variable bindings and a parent frame (if not the Global frame)

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

In Project 4, Frames do not hold return values



(Demo)

Define Expressions

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

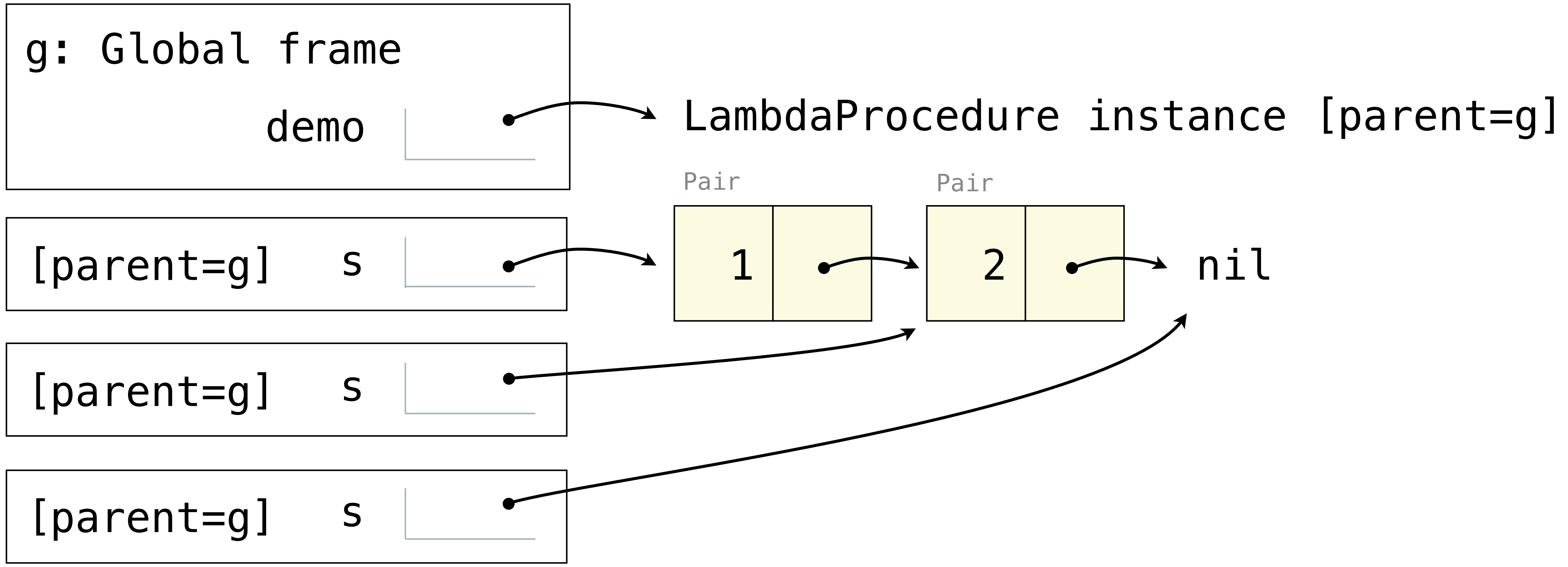
To apply a user-defined procedure, create a new frame where…

- Formal parameters (variables) are bound to argument values
- Whose parent frame is the **env** attribute 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))
```



Why Do We Teach Interpreters?

Why Interpreters?

- From the syllabus: “In CS 61A, we are interested in teaching you about **programming**, not about how to use one particular programming language.”
  - Programming: creating a set of instructions for a computer to execute
- Learning about interpreters provides better insight into how Python operates
  - Most elements of the Scheme interpreter (special forms, creating call/environment frames, etc.) are also present in Python
- Explains why programming languages are so brittle
  - One small syntax error makes a huge difference!
- Small introduction into programming systems
  - If you think interpreters are cool, take CS 164 (Programming Languages & Compilers)