Interpreters

Announcements

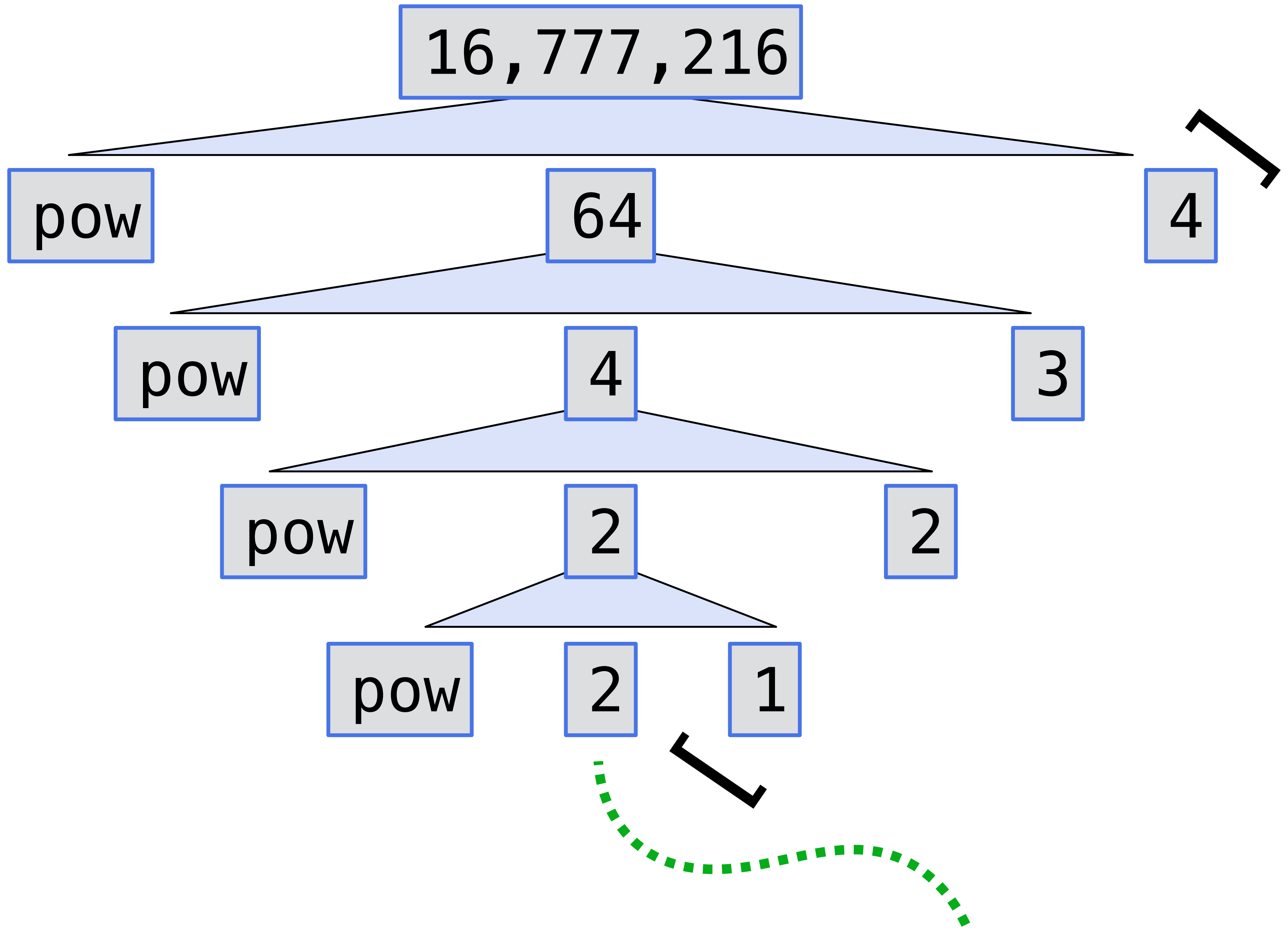
Exceptions

Reducing a Sequence to a Value

```
def reduce(f, s, initial):
    """Combine elements of s pairwise using f, starting with initial.

    E.g., reduce(mul, [2, 4, 8], 1) is equivalent to mul(mul(mul(1, 2), 4), 8).

    >>> reduce(mul, [2, 4, 8], 1)
    64
    """
```



`f` is ...
  a two-argument function that returns a first argument

`s` is ...
  a sequence of values that can be the second argument

`initial` is ...
  a value that can be the first argument

`reduce(pow, [1, 2, 3, 4], 2)`

(Demo)

Reduce Practice

Implement sum_squares, which returns the sum of the square of each number in a list s.

```
def reduce(f, s, initial):
    """Combine elements of s pairwise using f, starting with initial.

    E.g., reduce(mul, [2, 4, 8], 1) is equivalent to mul(mul(mul(1, 2), 4), 8).

    >>> reduce(mul, [2, 4, 8], 1)
    64
    """

def sum_squares(s):
    """Return the sum of squares of the numbers in s.

    >>> sum_squares([3, 4, 5])  # 3*3 + 4*4 + 5*5
    50
    """
    return reduce( lambda x, y: x + y * y , s, 0)
```

Scheme-Syntax Calculator

(Demo)

The Pair Class

(Demo)

Reducing a Pair

A **reduce** that takes a function, a Scheme list represented as a Pair, and an initial value.

```
def reduce(fn, scheme_list, initial):
    """Reduce a Scheme list made of Pairs using fn and an initial value.

    >>> reduce(add, Pair(1, Pair(2, Pair(3, nil))), 0)
    6
    """
    if scheme_list is nil:
        return initial

    return  reduce(fn, scheme_list.rest, fn(initial, scheme_list.first))


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

Calculator Syntax

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

A primitive expression is a number: 2   -4   5.6

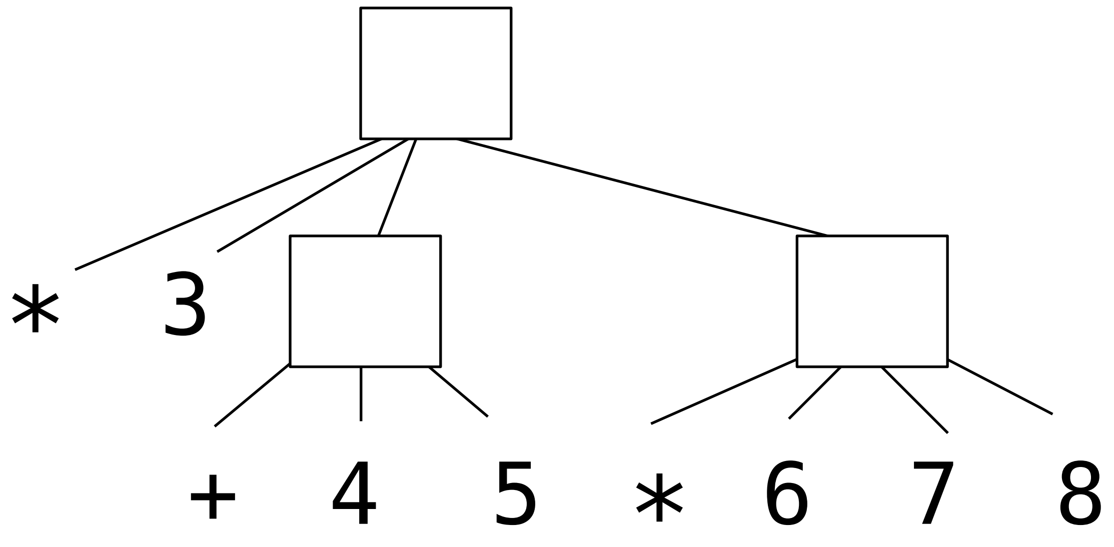
A call expression is a combination that begins with an operator (+, -, *, /) followed by 0 or more expressions: (+ 1 2 3)   (/ 3 (+ 4 5))

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

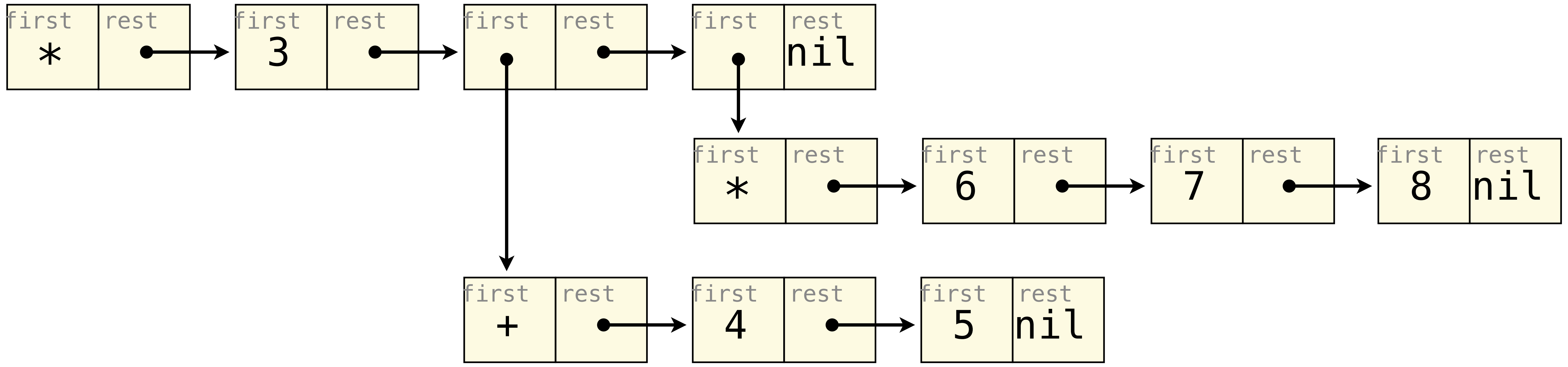
Expression

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

Expression Tree



Representation as Pairs



Calculator Semantics

The value of a calculator expression is defined recursively.

Primitive: A number evaluates to itself.

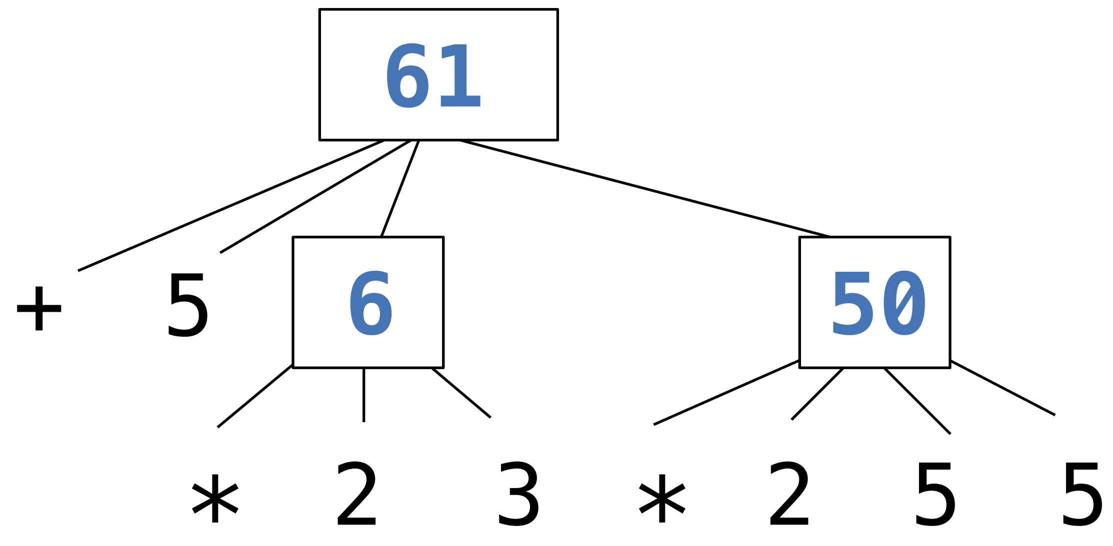
Call: A call expression evaluates to its argument values combined by an operator.

- **+**: 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



Evaluation

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)

Interactive Interpreters

Read-Eval-Print Loop

The user interface for many programming languages is an interactive interpreter

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

(Demo)

Break: 5 minutes

Interpreting Scheme

The Structure of an Interpreter

Eval

Base cases:
- Primitive values (numbers)
- Look up values bound to symbols

Recursive calls:
- Eval(operator) & Eval(operand) of calls
- Apply(procedure, arguments)
- Eval(sub-expression) of special forms

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

Project 4

Pairs in Project 4: Scheme

https://cs61a.org/proj/scheme/ (released on Wed.)

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

(Demo)