Interpreters

Announcements

Exceptions (in Python)

Raise Statements

Python exceptions are raised with a raise statement

raise <expression>

<expression> must evaluate to a subclass of BaseException or an instance of one

Exceptions are constructed like any other object. E.g., TypeError('Bad argument!')

TypeError -- A function was passed the wrong number/type of argument

NameError -- A name wasn't found

KeyError -- A key wasn't found in a dictionary

RecursionError -- Too many recursive calls

(Demo)

Try Statements

Try statements handle exceptions

```
try:
    <try suite>
except <exception class> as <name>:
    <except suite>
...
```

Execution rule:

The <try suite> is executed first

If, during the course of executing the <try suite>,
an exception is raised that is not handled otherwise, and

If the class of the exception inherits from <exception class>, then

The <except suite> is executed, with <name> bound to the exception

Exceptions Example: Reduce

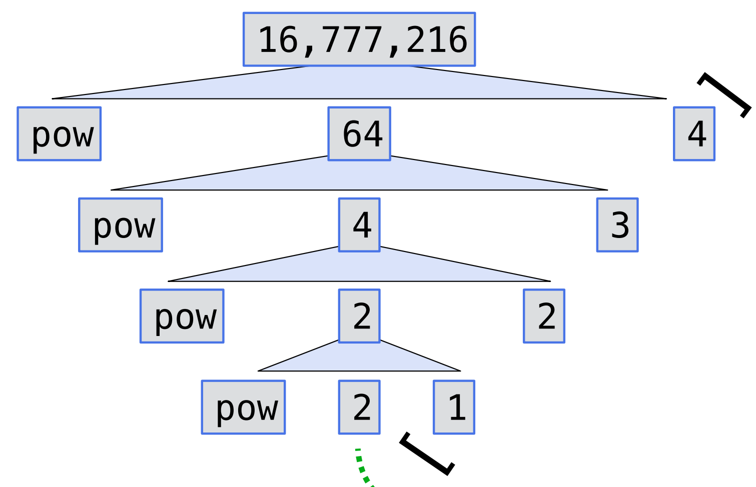
Reducing a Sequence to a Value

```
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)
```

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)
```

(Demo)

Reducing a Linked List

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

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

    >>> reduce(add, Link(1, Link(2, Link(3, nil))), 0) ; (+ (+ (+ 0 1) 2) 3)
    6
    """
    if s is nil:
        return initial

    return ____________reduce(fn, s.rest, fn(initial, s.first))____________
```

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

nil = Link.empty
```

Calculator Evaluation

The Calculator Language (a Small Subset of Scheme)

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

A primitive expression is a number: 2 -4 5.6

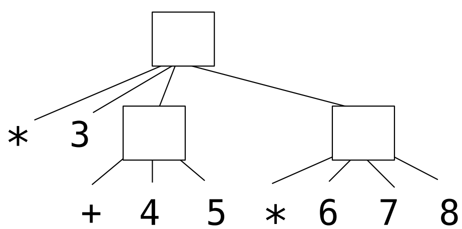
A call expression is a combination that begins with an operator (+, -, *, /) followed by 0 or more expressions: (+ 1 2 3) (/ 3 (+ 4 5))

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

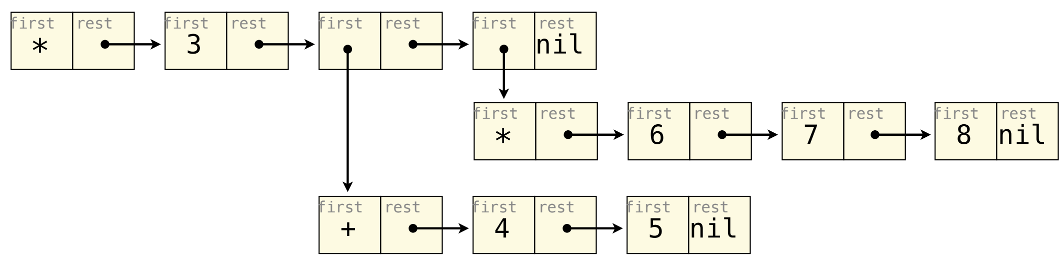
Expression

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

Expression Tree



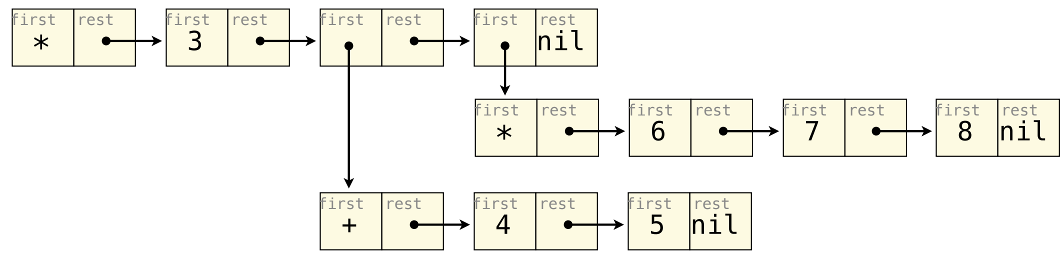
Representation as Link objects



Calculator: The Eval Function

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

exp:



Implementation

```
def calc_eval(exp):
    if isinstance(exp, (int, float)):
        return exp
    elif isinstance(exp, Link):
        arguments = map_link(calc_eval, exp.rest)
        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

(Demo)

Interactive Interpreters

Read-Eval-Print Loop (REPL)

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)

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

Linked Lists 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 Link 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)

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(expr):
    """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 = expr.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

```
(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))
```

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 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

(Demo)