

## Calculator

---

## Announcements

## Hog Strategy Contest

---

Each submitted strategy will play against all other submissions. The player to go first will be determined by a flip of a fair coin. A submission scores a match point each time it has an expected win rate strictly above 50.0001%. We will rank submissions based on the number of matches they won.

Third Place: Alexander Tian and Miles Hua

First Place (tie): (Kevin Yang and Davis Jin) and (Brian Zhou and Lin Jiang)

One more contest coming up: Scheme Recursive Art



Recursive Art Contest  
Fall 2015

# Symbolic Programming

## Symbolic Programming

---

Symbols normally refer to values; how do we refer to symbols?

```
> (define a 1)
> (define b 2)
> (list a b)
(1 2)
```

No sign of “a” and “b” in the resulting value

Quotation is used to refer to symbols directly in Lisp.

```
> (list 'a 'b)
(a b)
> (list 'a b)
(a 2)
```

Short for (quote a), (quote b):  
Special form to indicate that the expression itself is the value.

Quotation can also be applied to combinations to form lists.

```
> '(a b c)
(a b c)
> (car '(a b c))
a
> (cdr '(a b c))
(b c)
```

(Demo)

## List Processing



## Built-in List Processing Procedures

**(append s t):** list the elements of s and t; append can be called on more than 2 lists

**(map f s):** call a procedure f on each element of a list s and list the results

**(filter f s):** call a procedure f on each element of a list s and list the elements for which a true value is the result

**(apply f s):** call a procedure f with the elements of a list s as its arguments

(Demo)

```
(1 2 3 4)           ; count
((and a 1) (and a 2) (and a 3) (and a 4)) ; beats
(and a 1 and a 2 and a 3 and a 4)         ; rhythm
```

```
(define count (list 1 2 3 4))
(define beats (map (lambda (x) (list 'and 'a x)) count))
(define rhythm (apply append beats))
```

[pollev.com/cs61a](http://pollev.com/cs61a)

## Cons Count

Return how many cons cells appear in the diagram for a value `s`.

```
(define (cons-count s)
  (if (list? s)
      (+ (length s)
         (apply + (map cons-count s) ))
      0
      ))
```

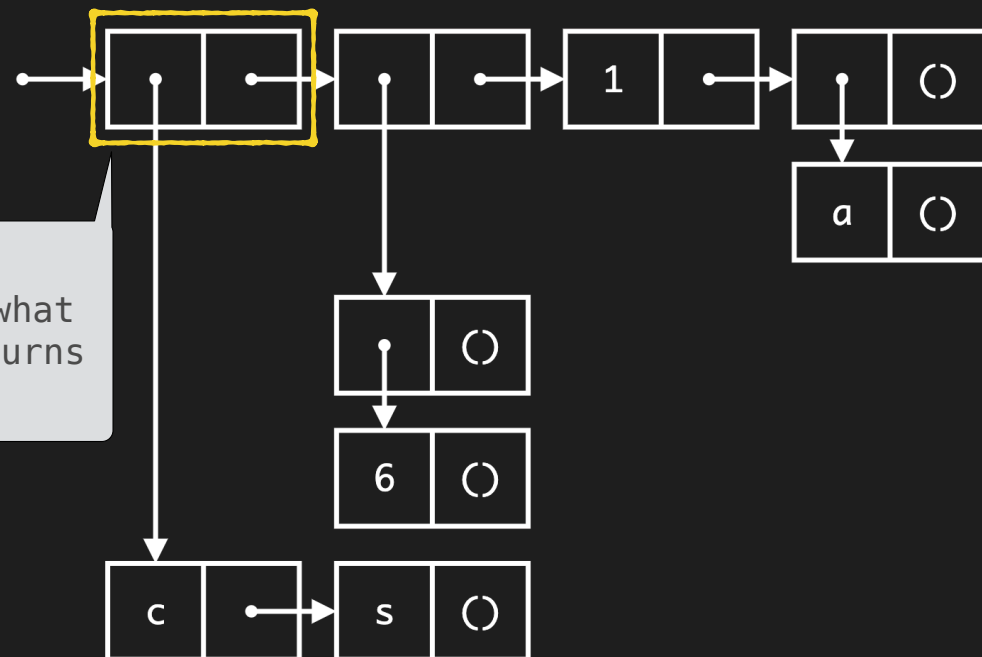
[pollev.com/cs61a](http://pollev.com/cs61a)

A "cons cell" is what  
cons creates & returns

```
scm> '((c s) ((6)) 1 (a))
```

```
((c s) ((6)) 1 (a))
```

```
scm> (draw '((c s) ((6)) 1 (a)))
```



# Scheme-Syntax Calculator

(Demo)

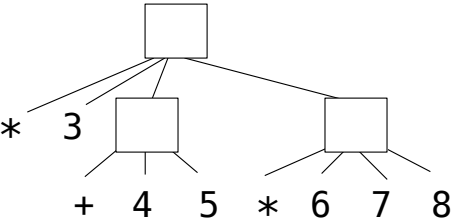
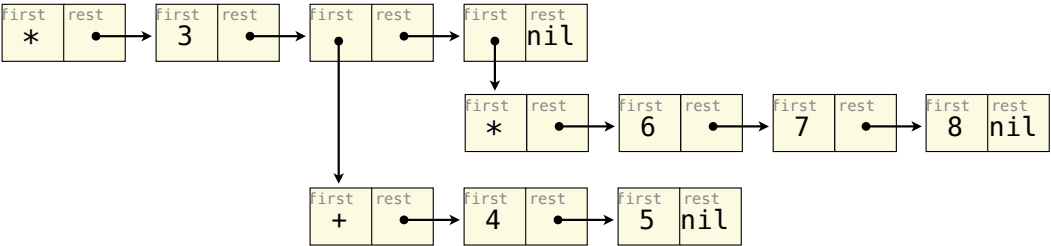
# Calculator Syntax

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 (Link instances) that encode tree structures.

Expression	Expression Tree	Representation as Link objects
<pre>(* 3   (+ 4 5)   (* 6 7 8))</pre>		

## 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**

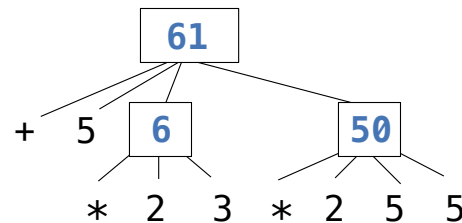
**-: If one argument, negate it. If more than one, subtract the rest from the first.**

**/: 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)

## 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

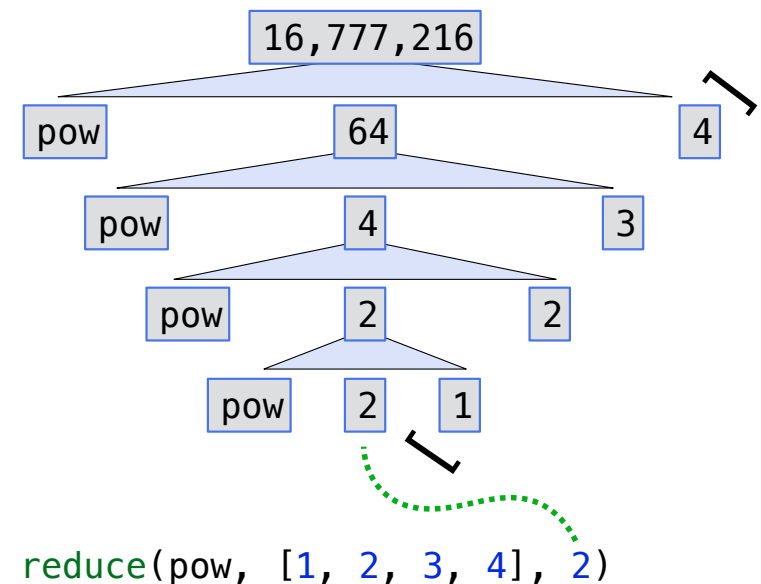
## Reducing a Sequence to a Value

```
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



(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)
```

[pollev.com/cs61a](http://pollev.com/cs61a)

## 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))
```

pollev.com/cs61a

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

nil = Link.empty
```

## 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, 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*

## 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)