Calculator

Announcements

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)

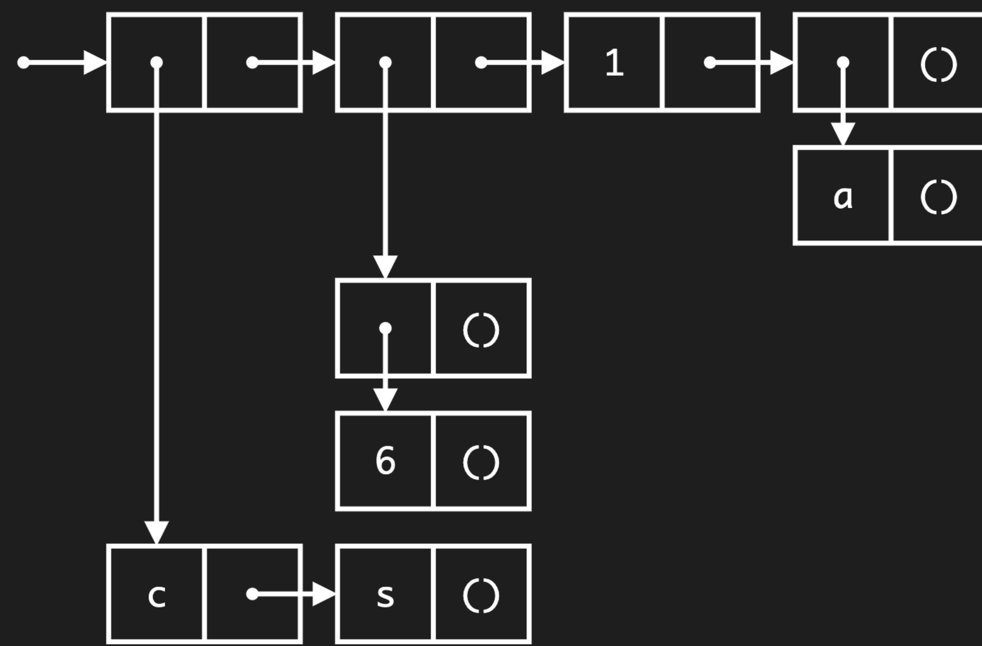
```
(1 2 3 4)                                ; count
((and a 1) (and a 2) (and a 3) (and a 4)) ; beats
(and a 1 and a 2 and a 3 and a 4)         ; rhythm

(define count (list 1 2 3 4))

(define beats (map (lambda (x) (list 'and 'a x)) count))

(define rhythm (apply append beats))
```

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 ))
```

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

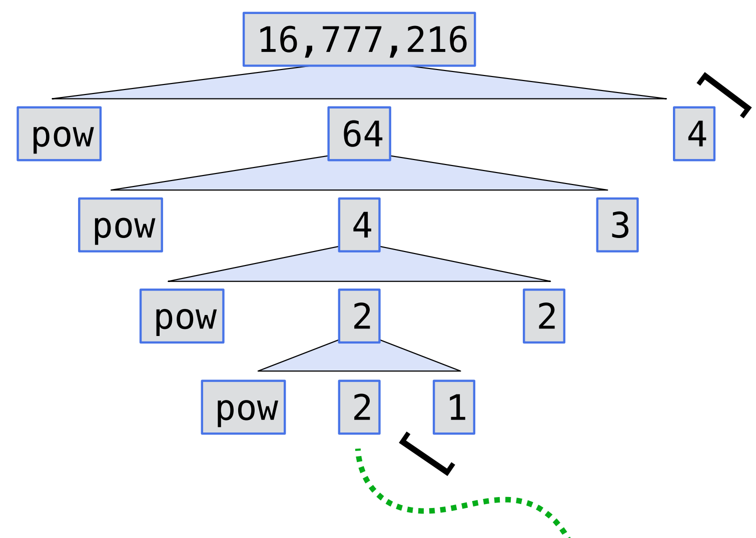


Exceptions

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)

Scheme-Syntax Calculator

(Demo)

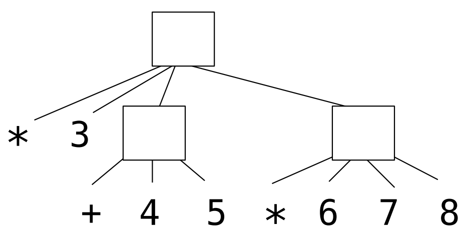
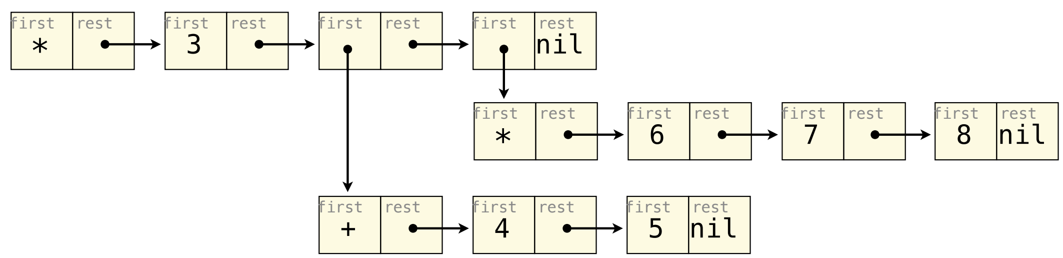
Calculator Syntax

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 (Pair instances) that encode tree structures.

Expression	**Expression Tree**	**Representation as Pairs**
(* 3 (+ 4 5) (* 6 7 8))		

Calculator Semantics

The value of a calculator expression is defined recursively.

Primitive: A number evaluates to itself.

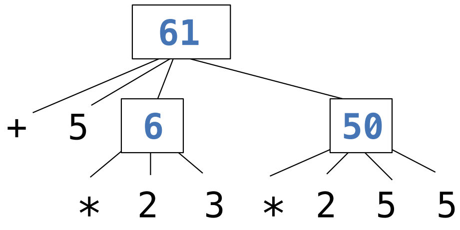
Call: A call expression evaluates to its argument values combined by an operator.

- **+**: 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)