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.

\[
\text{(define } \text{(cons-count } s) \text{)} \\
\text{(if } \text{(list? } s) \text{)} \\
\quad (\text{+ } \text{(length } s) \text{)} \\
\quad (\text{apply + } (\text{map cons-count } s))) \\
\text{0 } \text{\hfill )}
\]

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

```python
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)
Scheme-Syntax Calculator

(Demo)
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.

\[
\begin{array}{ccc}
\text{Expression} & \text{Expression Tree} & \text{Representation as Pairs} \\
(* 3 \\
  (+ 4 5) \\
  (* 6 7 8)) & \\
 & * \\
 & + 4 5 * 6 7 8 & \\
\end{array}
\]
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.

\begin{itemize}
  \item \textbf{+}: Sum of the arguments
  \item \textbf{\*}: Product of the arguments
  \item \textbf{-}: If one argument, negate it. If more than one, subtract the rest from the first.
  \item \textbf{/}: If one argument, invert it. If more than one, divide the rest from the first.
\end{itemize}

Expression:  
\[(+ 5  
  \text{(\*) 2 3)  
  (\*) 2 5 5))\]

Expression Tree: 
\[
\text{+} \\
\text{6} \\
\text{x} 2 3  \\
\text{x} 2 5 5
\]
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).

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Language Semantics</th>
</tr>
</thead>
</table>
| 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
| A number evaluates... to itself
| A call expression evaluates... to its argument values combined by an operator

Recursive call returns a number for each operand

A Scheme list of numbers

'+', '-', '*', '/'
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: +, -, *, /

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Language Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>def calc_apply(operator, args):</code></td>
<td><code>+:</code></td>
</tr>
<tr>
<td><code>if operator == '+':</code></td>
<td><code>Sum of the arguments</code></td>
</tr>
<tr>
<td><code>    return reduce(add, args, 0)</code></td>
<td><code>-:</code></td>
</tr>
<tr>
<td><code>elif operator == '-':</code></td>
<td><code>...</code></td>
</tr>
<tr>
<td><code>    ...</code></td>
<td><code>...</code></td>
</tr>
<tr>
<td><code>elif operator == '*':</code></td>
<td><code>...</code></td>
</tr>
<tr>
<td><code>    ...</code></td>
<td><code>...</code></td>
</tr>
<tr>
<td><code>elif operator == '/':</code></td>
<td><code>...</code></td>
</tr>
<tr>
<td><code>    ...</code></td>
<td><code>...</code></td>
</tr>
<tr>
<td><code>else:</code></td>
<td><code>else:</code></td>
</tr>
<tr>
<td><code>    raise TypeError</code></td>
<td><code>raise TypeError</code></td>
</tr>
<tr>
<td><code>(Demo)</code></td>
<td><code>(Demo)</code></td>
</tr>
</tbody>
</table>
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)