***

***

```
(+ 1 2 3) ;; 6
```

```
(let [x 1 y 2 z 3] (+ x y z)) ;; 6
```

```
(range 10) ;; (0 1 2 3 4 5 6 7 8 9)
```

```
(zero? 0) ;; true
```

```
(ns ns1) (defn average [& nums] (/ (reduce
  + nums) (count nums))) [(average 4 11) (
average 3.0 72 9.6 33)] ;; [15/2 29.4]
```

```
(println (((fn [f] ((fn [x] (f (fn [v
] ((x x) v)))) (fn [x] (f (fn [v]
((x x) v)))))) (fn [g] (fn [name] (
str "Hello, " name "!")))) "John Doe")) ;;
 Hello, John Doe!
```

```clojure
( require '[ clojure.set :as s ] ) ( def a-vowels
  #{ \a \e \i \o \u \x \y \z } ) ( def b-
vowels #{ \a \e \i \o \u } ) [ ( s / difference
a-vowels b-vowels ) ( s / union a-vowels b-vowels )
  ( s / intersection a-vowels b-vowels ) ] ;; [#{\x
\y \z} #{\a \e \i \o \u \x \y \z} #{\a \e \i \o \u}]


( def bhaskara ( fn [ a b c ] ( if ( or ( nil ?
 a ) ( nil ? b ) ( nil ? c ) ) nil ( let [ delta
 ( - ( * b b ) ( * 4 a c ) ) ] ( if ( < delta
 0 ) nil ( list ( / ( + ( - b ) ( Math / sqrt
delta ) ) ( * 2 a ) ) ( / ( - ( - b ) ( Math /
sqrt delta ) ) ( * 2 a ) ) ) ) ) ) ) ( bhaskara 1
 -5 6 ) ;; (3.0 2.0)


( defmacro defexpenses [ name & expenses ] `( def
~ name ( atom ' ~ expenses ) ) ) ( defn add-expense
[ atom-expense amount ] ( swap ! atom-expense conj
amount ) ) ( defn sum-expenses [ & atoms ] ( reduce
 + ( map #( apply + @ % ) atoms ) ) ) ( defexpenses
 person-1 1200 800 450 ) ( defexpenses person-2
1000 600 300 ) ( defexpenses person-3 1500 900
550 ) ( add-expense person-1 200 ) ( add-expense
person-2 100 ) ( add-expense person-3 150 ) ( sum-
expenses person-1 person-2 person-3 ) ;; 7750


( defn dot-product [ v1 v2 ] ( reduce + ( map *
v1 v2 ) ) ) ( defn add-elements [ v1 v2 ] ( mapv +
v1 v2 ) ) ( defn apply-weights [ input layer-weights
 layer-biases ] ( mapv ( fn [ w b ] ( + ( dot-
product input w ) b ) ) layer-weights layer-biases
) ) ( defn activation-function [ input ] ( mapv #(
 Math / tanh % ) input ) ) ( defn neural-network [
input weights biases activation-fn ] ( let [ layer-
outputs ( map ( fn [ w b ] ( activation-fn ( apply-
weights input w b ) ) ) weights biases ) ] ( last
 layer-outputs ) ) ) ( def input-1 [ 0.1 0.2 0.3 ]
) ( def input-2 [ 0.4 0.5 0.6 ] ) ( def weights-
1 [ [ 0.1 0.2 0.3 ] [ 0.4 0.5 0.6 ] [ 0.7 0.8
```

```
0.9 ] ] ) ( def  biases-1  [ 0.1  0.2  0.3 ] ) ( def
weights-2  [ [ 0.1  0.2  0.3 ]  [ 0.4  0.5  0.6 ] ] ) (
def  biases-2  [ 0.1  0.2 ] ) ( let  [ inputs  [ input-
1  input-2 ]  weights  [ weights-1  weights-2 ]  biases
 [ biases-1  biases-2 ] ] ( mapv  # ( neural-network
%  weights  biases  activation-function )  inputs )
)  ;; [[0.23549574953849794 0.47770001216849795] [0.
39693043200507755 0.7487042869693086]]


( def  x  ( ->  ( promise )  ( deliver  " text " ) ) ) @x
;; #'user/x
```

***

***