Examples (and some documentation) for the Words in the Library

```
from notebook_preamble import J, V
```

NOTE: Stack print out order.

When printing out the stack the *top* is on the *left*.

```
J('1 2 3')
```

```
3 2 1
```

In the traces, however, the top of the stack is on the *right*.

```
V('1 2 3')
```

```
      . 1 2 3
    1 . 2 3
  1 2 . 3
1 2 3 .
```

Stack Chatter

This is what I like to call the functions that just rearrange things on the stack. (One thing I want to mention is that during a hypothetical compilation phase these "stack chatter" words effectively disappear, because we can map the logical stack locations to registers that remain static for the duration of the computation. This remains to be done but it's "off the shelf" technology.)

`clear`

```
J('1 2 3 clear')
```

`dup dupd`

```
J('1 2 3 dup')
```

```
3 3 2 1
```

```
J('1 2 3 dupd')
```

```
3 2 2 1
```

enstacken disenstacken stack unstack

(I may have these paired up wrong. I.e. `disenstacken` should be `unstack` and vice versa.)

```
J('1 2 3 enstacken') # Replace the stack with a quote of itself.
```

```
[3 2 1]
```

```
J('4 5 6 [3 2 1] disenstacken')  # Unpack a list onto the stack.
```

```
1 2 3 6 5 4
```

```
J('1 2 3 stack')  # Get the stack on the stack.
```

```
[3 2 1] 3 2 1
```

```
J('1 2 3 [4 5 6] unstack')  # Replace the stack with the list on top.
```

```
4 5 6
```

pop popd popop

```
J('1 2 3 pop')
```

```
2 1
```

```
J('1 2 3 popd')
```

```
3 1
```

```
J('1 2 3 popop')
```

```
1
```

roll< rolldown roll> rollup

The "down" and "up" refer to the movement of two of the top three items (displacing the third.)

```
V('1 2 3 roll<')
```

```
      . 1 2 3 roll<
    1 . 2 3 roll<
  1 2 . 3 roll<
1 2 3 . roll<
2 3 1 .
```

```
V('1 2 3 roll>')
```

```
      . 1 2 3 roll>
    1 . 2 3 roll>
  1 2 . 3 roll>
1 2 3 . roll>
3 1 2 .
```

swap

```
J('1 2 3 swap')
```

```
2 3 1
```

tuck over

```
J('1 2 3 tuck')
```

```
3 2 3 1
```

```
J('1 2 3 over')
```

```
2 3 2 1
```

unit quoted unquoted

```
J('1 2 3 unit')
```

```
[3] 2 1
```

```
J('1 2 3 quoted')
```

```
3 [2] 1
```

```
J('1 [2] 3 unquoted')
```

```
3 2 1
```

```
V('1 [dup] 3 unquoted')  # Unquoting evaluates.  Be aware.
```

```
              . 1 [dup] 3 unquoted
            1 . [dup] 3 unquoted
      1 [dup] . 3 unquoted
    1 [dup] 3 . unquoted
    1 [dup] 3 . [i] dip
1 [dup] 3 [i] . dip
      1 [dup] . i 3
            1 . dup 3
          1 1 . 3
        1 1 3 .
```

List words

concat swoncat shunt

```
J('[1 2 3] [4 5 6] concat')
```

```
[1 2 3 4 5 6]
```

```
J('[1 2 3] [4 5 6] swoncat')
```

```
[4 5 6 1 2 3]
```

```
J('[1 2 3] [4 5 6] shunt')
```

```
[6 5 4 1 2 3]
```

cons swons uncons

```
J('1 [2 3] cons')
```

```
[1 2 3]
```

```
J('[2 3] 1 swons')
```

```
[1 2 3]
```

```
V('[1 2 3] uncons')
```

```
        . [1 2 3] uncons
[1 2 3] . uncons
1 [2 3] .
```

first second third rest

```
J('[1 2 3 4] first')
```

```
1
```

```
J('[1 2 3 4] second')
```

```
2
```

```
J('[1 2 3 4] third')
```

```
3
```

```
J('[1 2 3 4] rest')
```

```
[2 3 4]
```

flatten

```
J('[[1] [2 [3] 4] [5 6]] flatten')
```

```
[1 2 [3] 4 5 6]
```

getitem

```
J('[10 11 12 13 14] 2 getitem')
```

```
12
```

remove

```
J('[1 2 3 1 4] 1 remove')
```

```
[2 3 1 4]
```

reverse

```
J('[1 2 3 4] reverse')
```

```
[4 3 2 1]
```

size

```
J('[1 1 1 1] size')
```

```
4
```

swaack

"Swap stack" swap the list on the top of the stack for the stack, and put the old stack on top of the new one. Think of it as a context switch.

```
J('1 2 3 [4 5 6] swaack')
```

```
[3 2 1] 4 5 6
```

choice select

```
J('23 9 1 choice')
```

```
9
```

```
J('23 9 0 choice')
```

```
23
J('[23 9 7] 1 select')  # select is basically getitem, should retire it?
9
J('[23 9 7] 0 select')
23
```

zip

```
J('[1 2 3] [6 5 4] zip')
[[6 1] [5 2] [4 3]]
J('[1 2 3] [6 5 4] zip [sum] map')
[7 7 7]
```

Math words

+ add

```
J('23 9 +')
32
```

- sub

```
J('23 9 -')
14
```

*** mul**

```
J('23 9 *')
207
```

/ div floordiv truediv

```
J('23 9 /')
2.5555555555555554
J('23 -9 truediv')
```

```
-2.5555555555555554
J('23 9 div')
2
J('23 9 floordiv')
2
J('23 -9 div')
-3
J('23 -9 floordiv')
-3
```

% mod modulus rem remainder

```
J('23 9 %')
5
```

neg

```
J('23 neg -5 neg')
5 -23
```

^ xor

```
J('1 1 ^')
0
J('1 0 ^')
1
```

pow

```
J('2 10 pow')
1024
```

sqr sqrt

```
J('23 sqr')
```

```
529
```

```
J('23 sqrt')
```

```
4.795831523312719
```

++ succ -- pred

```
J('1 ++')
```

```
2
```

```
J('1 --')
```

```
0
```

<< lshift >> rshift

```
J('8 1 <<')
```

```
16
```

```
J('8 1 >>')
```

```
4
```

average

```
J('[1 2 3 5] average')
```

```
2.75
```

range range_to_zero down_to_zero

```
J('5 range')
```

```
[4 3 2 1 0]
```

```
J('5 range_to_zero')
```

```
[0 1 2 3 4 5]
```

```
J('5 down_to_zero')
```

```
0 1 2 3 4 5
```

`product`

```
J('[1 2 3 5] product')
```

```
30
```

`sum`

```
J('[1 2 3 5] sum')
```

```
11
```

`min`

```
J('[1 2 3 5] min')
```

```
1
```

`gcd`

```
J('45 30 gcd')
```

```
15
```

`least_fraction`

If we represent fractions as a quoted pair of integers [q d] this word reduces them to their … least common factors or whatever.

```
J('[45 30] least_fraction')
```

```
[3 2]
```

```
J('[23 12] least_fraction')
```

```
[23 12]
```

Logic and Comparison

`? truthy`

Get the Boolean value of the item on the top of the stack.

```
J('23 truthy')
```

```
True
```

```
J('[] truthy')  # Python semantics.
```

```
False
```

```
J('0 truthy')
```

```
False
```

```
? == dup truthy
```

```
V('23 ?')
```

```
        . 23 ?
     23 . ?
     23 . dup truthy
  23 23 . truthy
23 True .
```

```
J('[] ?')
```

```
False []
```

```
J('0 ?')
```

```
False 0
```

`& and`

```
J('23 9 &')
```

```
1
```

`!= <> ne`

```
J('23 9 !=')
```

```
True
```

The usual suspects: - `<` `lt` - `<=` `le`
- `=` `eq` - `>` `gt` - `>=` `ge` - `not` - `or`

Miscellaneous

`help`

```
J('[help] help')
```

```
Accepts a quoted symbol on the top of the stack and prints its docs.
```

parse

```
J('[parse] help')
```

```
Parse the string on the stack to a Joy expression.
```

```
J('1 "2 [3] dup" parse')
```

```
[2 [3] dup] 1
```

run

Evaluate a quoted Joy sequence.

```
J('[1 2 dup + +] run')
```

```
[5]
```

Combinators

app1 app2 app3

```
J('[app1] help')
```

```
Given a quoted program on TOS and anything as the second stack item run
the program and replace the two args with the first result of the
program.

            ... x [Q] . app1
   -----------------------------------
      ... [x ...] [Q] . infra first
```

```
J('10 4 [sqr *] app1')
```

```
160 10
```

```
J('10 3 4 [sqr *] app2')
```

```
160 90 10
```

```
J('10 2 3 4 [sqr *] app3')
```

```
160 90 40 10
```

anamorphism

Given an initial value, a predicate function `[P]`, and a generator function `[G]`, the `anamorphism` combinator creates a sequence.

```
   n [P] [G] anamorphism
---------------------------
          [...]
```

Example, `range`:

```
range == [0 <=] [1 - dup] anamorphism
```

```
J('3 [0 <=] [1 - dup] anamorphism')
```

```
[2 1 0]
```

`branch`

```
J('3 4 1 [+] [*] branch')
```

```
12
```

```
J('3 4 0 [+] [*] branch')
```

```
7
```

`cleave`

```
... x [P] [Q] cleave
```

From the original Joy docs: "The cleave combinator expects two quotations, and below that an item `x` It first executes `[P]`, with `x` on top, and saves the top result element. Then it executes `[Q]`, again with `x`, and saves the top result. Finally it restores the stack to what it was below `x` and pushes the two results P(X) and Q(X)."

Note that `P` and `Q` can use items from the stack freely, since the stack (below `x`) is restored. `cleave` is a kind of *parallel* primitive, and it would make sense to create a version that uses, e.g. Python threads or something, to actually run `P` and `Q` concurrently. The current implementation of `cleave` is a definition in terms of `app2`:

```
cleave == [i] app2 [popd] dip
```

```
J('10 2 [+] [-] cleave')
```

```
8 12 10
```

`dip dipd dipdd`

```
J('1 2 3 4 5 [+] dip')
```

```
5 7 2 1
```

```
J('1 2 3 4 5 [+] dipd')
```

```
5 4 5 1
```

```
J('1 2 3 4 5 [+] dipdd')
```

```
5 4 3 3
```

dupdip

Expects a quoted program `[Q]` on the stack and some item under it, `dup` the item and `dip` the quoted program under it.

```
n [Q] dupdip == n Q n
```

```
V('23 [++] dupdip *')  # N(N + 1)
```

```
        . 23 [++] dupdip *
     23 . [++] dupdip *
23 [++] . dupdip *
     23 . ++ 23 *
     24 . 23 *
  24 23 . *
    552 .
```

genrec primrec

```
J('[genrec] help')
```

```
General Recursion Combinator.

                        [if] [then] [rec1] [rec2] genrec
  ---------------------------------------------------------------------
     [if] [then] [rec1 [[if] [then] [rec1] [rec2] genrec] rec2] ifte

From "Recursion Theory and Joy" (j05cmp.html) by Manfred von Thun:
"The genrec combinator takes four program parameters in addition to
whatever data parameters it needs. Fourth from the top is an if-part,
followed by a then-part. If the if-part yields true, then the then-part
is executed and the combinator terminates. The other two parameters are
the rec1-part and the rec2-part. If the if-part yields false, the
rec1-part is executed. Following that the four program parameters and
the combinator are again pushed onto the stack bundled up in a quoted
form. Then the rec2-part is executed, where it will find the bundled
form. Typically it will then execute the bundled form, either with i or
with app2, or some other combinator."

The way to design one of these is to fix your base case [then] and the
```

```
test [if], and then treat rec1 and rec2 as an else-part "sandwiching"
a quotation of the whole function.

For example, given a (general recursive) function 'F':

    F == [I] [T] [R1] [R2] genrec

If the [I] if-part fails you must derive R1 and R2 from:

    ... R1 [F] R2

Just set the stack arguments in front, and figure out what R1 and R2
have to do to apply the quoted [F] in the proper way.  In effect, the
genrec combinator turns into an ifte combinator with a quoted copy of
the original definition in the else-part:

    F == [I] [T] [R1]   [R2] genrec
      == [I] [T] [R1 [F] R2] ifte

(Primitive recursive functions are those where R2 == i.

    P == [I] [T] [R] primrec
      == [I] [T] [R [P] i] ifte
      == [I] [T] [R P] ifte
)
```

```
J('3 [1 <=] [] [dup --] [i *] genrec')
```

```
6
```

i

```
V('1 2 3 [+ +] i')
```

```
            . 1 2 3 [+ +] i
          1 . 2 3 [+ +] i
        1 2 . 3 [+ +] i
      1 2 3 . [+ +] i
1 2 3 [+ +] . i
      1 2 3 . + +
        1 5 . +
          6 .
```

ifte

```
[predicate] [then] [else] ifte
```

```
J('1 2 [1] [+] [*] ifte')
```

```
3
```

```
J('1 2 [0] [+] [*] ifte')
```

```
2
```

infra

```
V('1 2 3 [4 5 6] [* +] infra')
```

```
                    . 1 2 3 [4 5 6] [* +] infra
                  1 . 2 3 [4 5 6] [* +] infra
                1 2 . 3 [4 5 6] [* +] infra
              1 2 3 . [4 5 6] [* +] infra
      1 2 3 [4 5 6] . [* +] infra
1 2 3 [4 5 6] [* +] . infra
              6 5 4 . * + [3 2 1] swaack
               6 20 . + [3 2 1] swaack
                 26 . [3 2 1] swaack
         26 [3 2 1] . swaack
         1 2 3 [26] .
```

loop

```
J('[loop] help')
```

```
Basic loop combinator.

   ... True [Q] loop
-----------------------
     ... Q [Q] loop

   ... False [Q] loop
------------------------
          ...
```

```
V('3 dup [1 - dup] loop')
```

```
              . 3 dup [1 - dup] loop
            3 . dup [1 - dup] loop
          3 3 . [1 - dup] loop
3 3 [1 - dup] . loop
            3 . 1 - dup [1 - dup] loop
          3 1 . - dup [1 - dup] loop
            2 . dup [1 - dup] loop
          2 2 . [1 - dup] loop
```

```
2 2 [1 - dup] . loop
            2 . 1 - dup [1 - dup] loop
          2 1 . - dup [1 - dup] loop
            1 . dup [1 - dup] loop
          1 1 . [1 - dup] loop
1 1 [1 - dup] . loop
            1 . 1 - dup [1 - dup] loop
          1 1 . - dup [1 - dup] loop
            0 . dup [1 - dup] loop
          0 0 . [1 - dup] loop
0 0 [1 - dup] . loop
            0 .
```

map pam

```
J('10 [1 2 3] [*] map')
```

```
[10 20 30] 10
```

```
J('10 5 [[*][/][+][-]] pam')
```

```
[50 2.0 15 5] 5 10
```

nullary unary binary ternary

Run a quoted program enforcing arity.

```
J('1 2 3 4 5 [+] nullary')
```

```
9 5 4 3 2 1
```

```
J('1 2 3 4 5 [+] unary')
```

```
9 4 3 2 1
```

```
J('1 2 3 4 5 [+] binary')  # + has arity 2 so this is technically pointless...
```

```
9 3 2 1
```

```
J('1 2 3 4 5 [+] ternary')
```

```
9 2 1
```

step

```
J('[step] help')
```

Run a quoted program on each item in a sequence.

```
        ... [] [Q] . step
     -----------------------
               ... .


       ... [a] [Q] . step
    ------------------------
             ... a . Q


   ... [a b c] [Q] . step
----------------------------------------
             ... a . Q [b c] [Q] step
```

The step combinator executes the quotation on each member of the list on top of the stack.

V('0 [1 2 3] [+] step')

```
              . 0 [1 2 3] [+] step
            0 . [1 2 3] [+] step
    0 [1 2 3] . [+] step
0 [1 2 3] [+] . step
      0 1 [+] . i [2 3] [+] step
          0 1 . + [2 3] [+] step
            1 . [2 3] [+] step
      1 [2 3] . [+] step
  1 [2 3] [+] . step
      1 2 [+] . i [3] [+] step
          1 2 . + [3] [+] step
            3 . [3] [+] step
        3 [3] . [+] step
    3 [3] [+] . step
      3 3 [+] . i
          3 3 . +
            6 .
```

times

V('3 2 1 2 [+] times')

```
            . 3 2 1 2 [+] times
          3 . 2 1 2 [+] times
        3 2 . 1 2 [+] times
```

```
      3 2 1 . 2 [+] times
    3 2 1 2 . [+] times
3 2 1 2 [+] . times
  3 2 1 [+] . i 1 [+] times
      3 2 1 . + 1 [+] times
        3 3 . 1 [+] times
      3 3 1 . [+] times
  3 3 1 [+] . times
    3 3 [+] . i
        3 3 . +
          6 .
```

b

```
J('[b] help')
```

```
b == [i] dip i

... [P] [Q] b == ... [P] i [Q] i
... [P] [Q] b == ... P Q
```

```
V('1 2 [3] [4] b')
```

```
            . 1 2 [3] [4] b
          1 . 2 [3] [4] b
        1 2 . [3] [4] b
    1 2 [3] . [4] b
1 2 [3] [4] . b
        1 2 . 3 4
      1 2 3 . 4
    1 2 3 4 .
```

while

```
[predicate] [body] while
```

```
J('3 [0 >] [dup --] while')
```

```
0 1 2 3
```

x

```
J('[x] help')
```

```
x == dup i
```

```
... [Q] x = ... [Q] dup i
... [Q] x = ... [Q] [Q] i
... [Q] x = ... [Q]  Q
```

```
V('1 [2] [i 3] x')  # Kind of a pointless example.
```

```
            . 1 [2] [i 3] x
          1 . [2] [i 3] x
      1 [2] . [i 3] x
1 [2] [i 3] . x
1 [2] [i 3] . i 3
      1 [2] . i 3 3
          1 . 2 3 3
        1 2 . 3 3
      1 2 3 . 3
    1 2 3 3 .
```

void

Implements **Laws of Form** *arithmetic* over quote-only datastructures (that is, datastructures that consist soley of containers, without strings or numbers or anything else.)

```
J('[] void')
```

```
False
```

```
J('[[]] void')
```

```
True
```

```
J('[[][[]]] void')
```

```
True
```

```
J('[[[]][[][]]] void')
```

```
False
```