

# DYALOG

Elsinore 2019



## A Decade of APL Extensions – Trains and High Rank Operations

*Adám Brudzewsky · Marshall Lochbaum  
Richard Park · Richard Smith · Nicolas Delcros*

# Function Trains



# Function Trains: *a bit of history*

expression

TMN

APL

$$f(g(x))$$

$$f\ g\ x$$

$$f(x) \times g(x)$$

$$(f\ x) \times g\ x$$



# Function Trains: *a bit of history*

expression

composition

TMN

APL

TMN

$$f(g(x))$$

$$f \ g \ x$$

$$(f \circ g)(x)$$

$$f(x) \times g(x)$$

$$(f \ x) \times g \ x$$



# Function Trains: *a bit of history*

expression

composition

TMN

APL

TMN

APL

$$f(g(x))$$

$$f \ g \ x$$

$$(f \circ g)(x)$$

$$(f \circ g) \ x$$

$$f(x) \times g(x)$$

$$(f \ x) \times g \ x$$



# Function Trains: *a bit of history*

expression

composition

TMN

APL

TMN

APL

$$f(g(x))$$

$$f \ g \ x$$

$$(f \circ g)(x)$$

$$(f \circ g) \ x$$

$$f(x) \times g(x)$$

$$(f \ x) \times g \ x$$

$$(f \times g)(x)$$



# Function Trains: *a bit of history*

expression

composition

TMN

APL

TMN

APL

$$f(g(x))$$

$$f \ g \ x$$

$$(f \circ g)(x)$$

$$(f \circ g) \ x$$

$$f(x) \times g(x)$$

$$(f \ x) \times g \ x$$

$$(f \times g)(x)$$

$$(f \bar{\times} g) \ x$$



# Function Trains: *a bit of history*

expression

composition

TMN

APL

TMN

APL

$$f(g(x))$$

$$f \ g \ x$$

$$(f \circ g)(x)$$

$$(f \circ g) \ x$$

$$f(x) \div g(x)$$

$$(f \ x) \div g \ x$$

$$(f \div g)(x)$$

$$(f \bar{\div} g) \ x$$





# Function Trains: *a bit of history*

expression

composition

TMN

APL

TMN

APL

$$f(g(x))$$

$$f\ g\ x$$

$$(f \circ g)(x)$$

$$(f \circ g)\ x$$

$$f(x) - g(x)$$

$$(f\ x) - g\ x$$

$$(f - g)(x)$$

$$(f \bar{-} g)\ x$$



# Function Trains: *a bit of history*

expression

composition

TMN

APL

TMN

APL

$$f(g(x))$$

$$f \ g \ x$$

$$(f \circ g)(x)$$

$$(f \circ g) \ x$$

$$f(x) \times g(x)$$

$$(f \ x) \times g \ x$$

$$(f \times g)(x)$$

$$(f \times g) \ x$$



# Function Trains: *valence*

$(+, -)2$       a monadic train

$2 \ ^{-}2$

$3(+, -)2$       a dyadic train

$5 \ 1$

TMN

$3 \pm 2$



APL

$(f \ g \ h)$



TMN

$(f \times g)(x)$



# Function Trains: *in isolation*

1      3 +, - 2      A not a train

5   1      3(+, -)2      A yes a train



# Function Trains: *in isolation*

1	3 +, - 2	A not a train
5	3 (+, -) 2	A yes a train
1	f ← +, -	A train assignment
5	3 f 2	A train application



# Function Trains: *step-by-step evaluation*

⇔  $(+ \neq \div \neq) \ 1 \ 2 \ 3 \ 4$



# Function Trains: *step-by-step evaluation*

⇔  $(+ \neq \div \neq) 1 2 3 4$

⇔  $(+ \neq 1 2 3 4) \div (\neq 1 2 3 4)$



# Function Trains: *step-by-step evaluation*

⇔  $(+ \neq \div \neq) 1 2 3 4$

⇔  $(+ \neq 1 2 3 4) \div (\neq 1 2 3 4)$

⇔  $10 \div 4$





# Function Trains: *step-by-step evaluation*

⇔  $(+ \neq \div \neq) 1 2 3 4$

⇔  $(+ \neq 1 2 3 4) \div (\neq 1 2 3 4)$

⇔  $10 \div 4$

⇔  $2.5$



## *Example problem: Range*

Write an ( f g h ) train that finds a numeric vector's range by subtracting the smallest element from the largest element.

Examples:

Range 3 1 4 1 5 9

8



## Example problem: Range

Write an ( f g h ) train that finds a numeric vector's range by subtracting the smallest element from the largest element.

Range  $\leftarrow \lceil / - \lfloor /$

Range 3 1 4 1 5 9

8

( $\lceil / - \lfloor /$ ) 27 18 28

10



# Function Trains: *definition*

A sequence of 3 functions *in isolation* forms a train.

The functions can be primitive, derived or defined:

$\phi$     $+$  .  $\times$    `foo`   a 3 functions

They can even be trains:

$(+ \neq \div \neq)$     $+$     $|$    a 3 functions



# SkewSymmetric

A *skew symmetric* matrix  $M$  satisfies  $M \equiv -\phi M$

Write an (f g h) train that tests for skew symmetry!

Examples:

```
SkewSymmetric 2 2 p0 2 ^2
```

1

```
SkewSymmetric 3 3 p4 ↑1
```

0



# Function Trains: *definition*

A sequence of **3 or more** functions in isolation forms a train:

$+$  ,  $-$  ,  $\times$  ,  $\div$       a 7 functions

6 (  $+$  ,  $-$  ,  $\times$  ,  $\div$  ) 2

8 4 12 3



# Function Trains: *fork*

Functions in a train are grouped in threes from right:

$$+, -, \times, \div \Leftrightarrow \left( +, \left( -, \left( \times, \div \right) \right) \right)$$


]Box on -trains=parens    ⑈ for diagnostics

$+, -, \times, \div$

$+, (-, (\times, \div))$



# Function Trains

Odd-numbered functions starting from the right are applied to the train's argument(s):

$$\begin{array}{ccccccc}
 6 & (+ & , & - & , & \times & , & \div) & 2 \\
 & (6+2) & , & (6-2) & , & (6\times 2) & , & (6\div 2) \\
 & 8 & , & 4 & , & 12 & , & 3
 \end{array}$$

*Intervening*, even-numbered, functions are applied between results of the odd-numbered functions





# Function Trains: *definition*

A sequence of **2** functions in isolation also forms a train:

– ×                      a 2 functions

6 (– ×) 2

–12



# Function Trains: *step-by-step evaluation*

⇔ 6 ( - × ) 2



# Function Trains: *step-by-step evaluation*

⇔ 6 ( - × ) 2

⇔ - ( 6 × 2 )



# Function Trains: *step-by-step evaluation*

⇔ 6 ( - × ) 2

⇔ - ( 6 × 2 )

⇔ - 12



# Function Trains: *step-by-step evaluation*

⇔  $6 \ (- \times) \ 2$

⇔  $- \ (6 \times 2)$

⇔  $- \ 12$

⇔  $\bar{1}2$



# Function Trains: *definition*

A sequence of **2 or more** functions in isolation forms a train:

$- + / \div \neq$       a 4 functions

$(- + / \div \neq) 2 7 1 8$

$\neg 4.5$



## Function Trains: *atop*

After making zero or more groups of three, there may be a function left over:

$$\underbrace{- \ +/ \ \div \ \neq}_{\text{group of three}} \Leftrightarrow \left( - ( +/ \ \div \ \neq ) \right)$$

]Box on -trains=parens      a for diagnostics

- +/ ÷ ≠

- (( +/ ) ÷ ≠ )



## Function Trains: *atop*

After making zero or more groups of three, there may be a function left over:

(*-* *+/* *÷* *≠*) n←2 7 1 8

*-* (*+/*n) *÷* (*≠*n)

*-* 18 *÷* 4

An *even-numbered leftmost* function is applied to the result.





# Function Trains: *summary*

A 2-train is an *atop*:

$$\begin{array}{lcl} (g \ h) \ \omega & \Leftrightarrow & g \ ( \ h \ \omega) \\ \alpha \ (g \ h) \ \omega & \Leftrightarrow & g \ (\alpha \ h \ \omega) \end{array}$$

A 3-train is a *fork*:

$$\begin{array}{lcl} (f \ g \ h) \ \omega & \Leftrightarrow & ( \ f \ \omega) \ g \ ( \ h \ \omega) \\ \alpha \ (f \ g \ h) \ \omega & \Leftrightarrow & (\alpha \ f \ \omega) \ g \ (\alpha \ h \ \omega) \end{array}$$

The *left tine* of a *fork* (but not an atop!) can be an array:

$$\begin{array}{lcl} (A \ g \ h) \ \omega & \Leftrightarrow & A \ g \ ( \ h \ \omega) \\ \alpha \ (A \ g \ h) \ \omega & \Leftrightarrow & A \ g \ (\alpha \ h \ \omega) \end{array}$$



# IdentityMatrix

The *identity matrix* for a square matrix  $M$  is like an all-zero  $M$  but with ones along the diagonal. Write a train that takes a square matrix as argument and returns the corresponding identity matrix:

```
IdentityMatrix 3 3p5
1 0 0
0 1 0
0 0 1
```



# Function Trains: *addressing arguments*



## Function Trains: *addressing arguments*

$(\phi \equiv \text{⍳}) \text{ 'racecar' } \text{⍳} \text{ palindrome?}$

$(\phi \text{ 'racecar'}) \equiv (\text{⍳} \text{ 'racecar'})$



# Function Trains: *addressing arguments*

$(\phi \equiv \vdash) \text{ 'racecar' } \quad \text{a palindrome?}$

$(\phi \text{ 'racecar' }) \equiv (\vdash \text{ 'racecar' })$

$'_' (\neq \subseteq \vdash) \text{ 'you\_can\_too' } \quad \text{a cut at } \alpha s$

$( '_' \neq \text{ 'you\_can\_too' } ) \subseteq ( '_' \vdash \text{ 'you\_can\_too' } )$



# Function Trains: *runs of monadic functions*



# Function Trains: *runs of monadic functions*

( ? ≠ ⊃ ⌈ ) □ A

( ? ∘ ≠ ⊃ ⌈ ) □ A

⌈ pick random



# Function Trains: *runs of monadic functions*

$$(\textcolor{red}{?} \neq \supset \vdash) \Box A$$

$$(\textcolor{green}{?} \circ \neq \supset \vdash) \Box A$$

$$(\textcolor{green}{?} \circ \neq \Box A) \supset (\vdash \Box A)$$

a pick random





# Function Trains: *runs of monadic functions*

$$(\textcolor{red}{?} \textcolor{blue}{\neq} \supset \vdash) \Box A$$

$$(\textcolor{green}{?} \textcolor{orange}{\circ} \textcolor{blue}{\neq} \supset \vdash) \Box A$$

$$(\textcolor{green}{?} \textcolor{orange}{\circ} \textcolor{blue}{\neq} \Box A) \supset (\vdash \Box A)$$

a pick random

$$7 \ (\vdash \supset \ddot{\sim} \ 2 \ + \ \textcolor{red}{\times} \neg) \ ' - 0 + '$$

$$7 \ (\vdash \supset \ddot{\sim} \ 2 \ + \ \textcolor{red}{\times} \textcolor{red}{\circ} \neg) \ ' - 0 + '$$

$$7 \ (\vdash \supset \ddot{\sim} \ 2 \ + \ \textcolor{orange}{\circ} \textcolor{green}{\times} \neg) \ ' - 0 + '$$

a pick by sign



# Function Trains: *runs of monadic functions*

$$(\textcolor{red}{?} \textcolor{blue}{\neq} \supset \vdash) \Box A$$

$$(\textcolor{green}{?} \textcolor{orange}{\circ} \textcolor{red}{\neq} \supset \vdash) \Box A$$

a pick random

$$(\textcolor{green}{?} \textcolor{orange}{\circ} \textcolor{red}{\neq} \Box A) \supset (\vdash \Box A)$$

$$7 (\vdash \supset \ddot{\sim} 2 + \textcolor{red}{\times} \neg) \text{'-0+'}$$

$$7 (\vdash \supset \ddot{\sim} 2 + \textcolor{red}{\times} \textcolor{red}{\circ} \neg) \text{'-0+'}$$

a pick by sign

$$7 (\vdash \supset \ddot{\sim} 2 + \textcolor{orange}{\circ} \textcolor{green}{\times} \neg) \text{'-0+'}$$

$$(7 \vdash \text{'-0+'}) \supset \ddot{\sim} 7 (2 + \textcolor{orange}{\circ} \textcolor{green}{\times} \neg) \text{'-0+'}$$


# Function Trains: *constants on the right*

$(\circ \div 180) \ 45$

$(180 \div \smiley \circ) \ 45$

$\pi$  degrees to radians



# Function Trains: *constants on the right*

$(\circ \div 180) \ 45$

$(180 \div \smile \circ) \ 45$

$\pi$  degrees to radians

$(! \ \vdash - 1) \ 5$

$(! \ - \circ 1) \ 5$

$\Gamma$  gamma function



# PathLength

The length of a path can be determined from the position of its points when given as complex numbers with the formula  $\{+ / | 2 - / \omega\} \text{ points}$

Translate this dfn into a train!

Example:

```
PathLength 1J1 4J1 4J6
8
```



# Function Trains: *summary*

A 2-train is an *atop*:

$$\begin{array}{lcl} (g \ h) \ \omega & \Leftrightarrow & g \ ( \ h \ \omega) \\ \alpha \ (g \ h) \ \omega & \Leftrightarrow & g \ (\alpha \ h \ \omega) \end{array}$$

A 3-train is a *fork*:

$$\begin{array}{lcl} (f \ g \ h) \ \omega & \Leftrightarrow & ( \ f \ \omega) \ g \ ( \ h \ \omega) \\ \alpha \ (f \ g \ h) \ \omega & \Leftrightarrow & (\alpha \ f \ \omega) \ g \ (\alpha \ h \ \omega) \end{array}$$

The left tine of a fork may also be an array:

$$\begin{array}{lcl} (A \ g \ h) \ \omega & \Leftrightarrow & A \ g \ ( \ h \ \omega) \\ \alpha \ (A \ g \ h) \ \omega & \Leftrightarrow & A \ g \ (\alpha \ h \ \omega) \end{array}$$



# Break



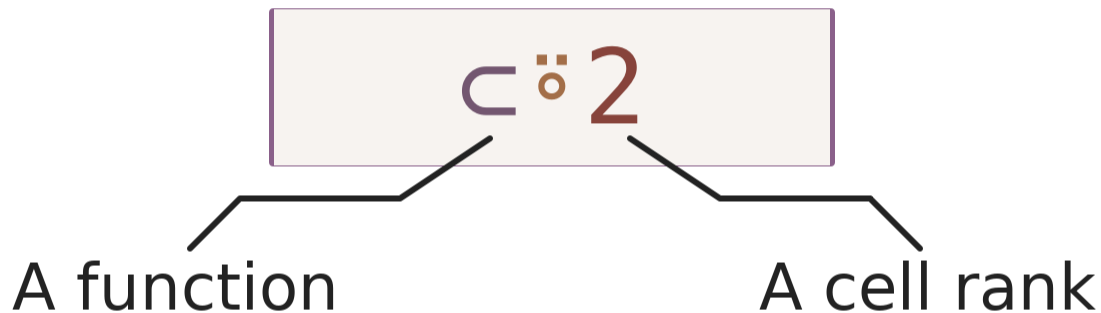
# High Rank Operations





# The Rank (⌞) operator

Like Power (⌘), Rank takes a function left operand and an array right operand:



Rank with a right operand  $k$  calls its function operand on  $k$ -cells of every argument, using the entire argument if its rank is less than  $k$ .

Unlike Each (⌞), Rank works with flat arrays: it never exposes the items of the argument, and always treats them as part of an array.

# Example: Grading cells

Grade with Rank turns each rank-2 cell into a rank-1 grade.

$\vdash a \leftarrow ?4 \ 2 \ 6p100$

34 33 96 68 49 93

98 27 56 20 30 60

58 84 23 23 17 60

12 73 99 74 77 92

86 41 14 96 7 5

88 80 78 35 23 21

98 98 4 76 85 46

72 94 86 22 92 60

$(\uparrow \circ 2) \ a$

1 2

2 1

1 2

2 1

The shape is reduced from 4 2 6 to 4 2, retaining the **frame** 4.

# Result mixing

The result cells generated by `f` in `for` are mixed together, like with `↑`. So if they don't all have the same shape, fills are inserted:

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

# Result mixing

The result cells generated by  $f$  in  $f \circ r$  are mixed together, like with  $\uparrow$ . So if they don't all have the same shape, fills are inserted:

			$\uparrow 0$	3	4	5
1	2	3	0	0		
1	2	3	4	0		
1	2	3	4	5		

Here's how Rank relates to Each, when called monadically:

$(f \circ k)x$	$\leftrightarrow$	$\uparrow f'' \subset [(-k) \uparrow \uparrow \neq \rho x]x$
$(c \circ f \circ k)x$	$\leftrightarrow$	$f'' \subset [(-k) \uparrow \uparrow \neq \rho x]x$
$(c \circ f \circ \supset \circ 0)x$	$\leftrightarrow$	$f'' x$

$$(f \circ k)x \leftrightarrow \uparrow f'' \subset [(-k) \uparrow \neq \rho x]x$$

The function  $\subset [(-k) \uparrow \neq \rho x]$  simply encloses the  $k$ -cells of its argument. Using Rank itself, we might write  $(f \circ k)x \leftrightarrow \uparrow f'' \subset \circ k \vdash x$ .

```
⊢a←3 5p1 0 0 1
```

```
1 0 0 1 1
```

```
0 0 1 1 0
```

```
0 1 1 0 0
```

```
⊂∘1⊢a
```

1 0 0 1 1	0 0 1 1 0	0 1 1 0 0
-----------	-----------	-----------

```
⊂∘1⊢a
```

1 4 5	3 4	2 3
-------	-----	-----

```
⊂∘1⊢a  @ Same as ⊂∘1
```

```
1 4 5
```

```
3 4 0
```

```
2 3 0
```

$\subset \circ k$  is a simple and helpful tool for thinking about Rank.

$$(c \circ f \circ k) x \leftrightarrow f \circ c [(-k) \uparrow 1 \neq \rho x] x$$

To avoid mixing results, you can compose  $\circ$  with the left operand.

$c \circ \text{id} \circ 1 \vdash a$

1	4	5	3	4	2	3
---	---	---	---	---	---	---

$\supset, / \quad c \circ \text{id} \circ 1 \vdash a$

1 4 5 3 4 2 3

$\{\omega[\Delta \omega]\} \supset U / \quad c \circ \text{id} \circ 1 \vdash a$     # Columns with any 1s

1 2 3 4 5

$\text{id} \vee a$

1 2 3 4 5

a

1	0	0	1	1
0	0	1	1	0
0	1	1	0	0

Rank works best when the operand function's result shape is predictable. This trick turns any function into one whose result always has shape  $\emptyset$ .

$$(\subset \circ f \circ \supset \circ \circ \emptyset) x \leftrightarrow f \circ x$$

Each can be implemented in terms of Rank!

In fact, the J language has no Each primitive at all, as it uses Rank instead.

```
x ← 'Each' 'in' 'terms' 'of' 'Rank'
ϕ ∘ x
```

hcaE	ni	smret	fo	knaR
------	----	-------	----	------

```
⊂ ∘ ϕ ∘ ⊃ ∘ ∘ ∅ ⊢ x
```

hcaE	ni	smret	fo	knaR
------	----	-------	----	------

$\supset$  isn't needed if  $x$  is simple, and  $\subset$  isn't needed if each result is a simple scalar.

## Exercise: no-blank split

Rewrite the idiom `~o' ' '↓` as a single function with Rank.

```
t ← 'The' 'idiom' 'is' 'still' 'faster' 'in' '17.1'
```

The  
idiom  
is  
still  
faster  
in  
17.1

~ o | | " ↓ t

The	idiom	is	still	faster	in	17.1
-----	-------	----	-------	--------	----	------

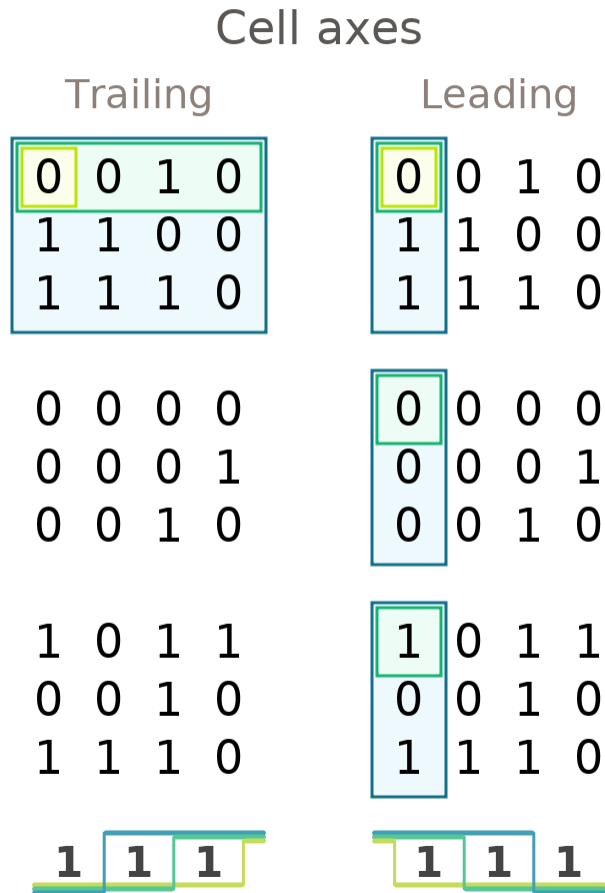


# Leading and trailing axes

We choose to divide axes so that cells are contiguous in the array's ravel, and in memory. For Rank's argument that means:

- Leading axes determine the *frame*, which organises the argument and result cells.
- Trailing axes determine the shape of each *argument cell* (but not the final result).

A cell is a selection of one index from each leading axis, and the whole of each trailing axis. An example might be `A[3;2;;;]`, which selects a 3-cell of `A`.



# Leading and trailing axes

We choose to divide axes so that cells are contiguous in the array's ravel, and in memory. For Rank's argument that means:

- Leading axes determine the *frame*, which organises the argument and result cells.
- Trailing axes determine the shape of each *argument cell* (but not the final result).

A cell is a selection of one index from each leading axis, and the whole of each trailing axis. An example might be `A[3;2;;;]`, which selects a 3-cell of `A`.

Cell axes	
Trailing	Leading
0 0 1 0	0 0 1 0
1 1 0 0	1 1 0 0
1 1 1 0	1 1 1 0
0 0 0 0	0 0 0 0
0 0 0 1	0 0 0 1
0 0 1 0	0 0 1 0
1 0 1 1	1 0 1 1
0 0 1 0	0 0 1 0
1 1 1 0	1 1 1 0
3 2 4	3 2 4

# Leading and trailing axes

We choose to divide axes so that cells are contiguous in the array's ravel, and in memory. For Rank's argument that means:

- Leading axes determine the *frame*, which organises the argument and result cells.
- Trailing axes determine the shape of each *argument cell* (but not the final result).

A cell is a selection of one index from each leading axis, and the whole of each trailing axis. An example might be `A[3;2;;;]`, which selects a 3-cell of `A`.

Cell axes

Trailing

0	0	1	0
1	1	0	0
1	1	1	0

0	0	0	0
0	0	0	1
0	0	1	0

1	0	1	1
0	0	1	0
1	1	1	0

1	3	2
---	---	---

Leading

0	0	1	0
1	1	0	0
1	1	1	0

0	0	0	0
0	0	0	1
0	0	1	0

1	0	1	1
0	0	1	0
1	1	1	0

1	3	2
---	---	---

# Reductions with Rank

How do last- and first-axis reductions interact with Rank?

$a \leftarrow 2 \ 3 \ 4 \rho 124$

$+/\circ 1 \vdash a \quad \text{Ⓢ Shape } 2 \ 3$   
10 26 42  
58 74 90

$+/\circ 2 \vdash a \quad \text{Ⓢ Shape } 2 \ 3$   
10 26 42  
58 74 90

$+/\circ 3 \vdash a \quad \text{Ⓢ Shape } 2 \ 3$   
10 26 42  
58 74 90

$+/\circ 1 \vdash a \quad \text{Ⓢ Shape } 2 \ 3$   
10 26 42  
58 74 90

$+/\circ 2 \vdash a \quad \text{Ⓢ Shape } 2 \quad 4$   
15 18 21 24  
51 54 57 60

$+/\circ 3 \vdash a \quad \text{Ⓢ Shape } \quad 3 \ 4$   
14 16 18 20  
22 24 26 28  
30 32 34 36

# Rank and the leading axis

Rank forces the operand function to work on a suffix of the argument's axes. If it works only on a single axis, then the leading one is the most flexible choice:



Unlike Axis ( $\phi[a]$ ), Rank extends naturally to multi-axis functions.

# Rank versus Axis

Axis treats axes as unordered and chooses one of them. Rank uses the natural ordering to choose smaller or larger cells.

$2 \downarrow \circ 2 \vdash A$

8	6	3	0	0	0	0
4	7	5	3	2	7	1
0	5	9	0	2	0	2
1	1	6	5	0	4	5
4	4	9	4	0	0	7

$2 \downarrow [1] A$

8	6	3	0	0	0	0
4	7	5	3	2	7	1
0	5	9	0	2	0	2
1	1	6	5	0	4	5
4	4	9	4	0	0	7

$2 \downarrow \circ 1 \vdash A$

8	6	3	0	0	0	0
4	7	5	3	2	7	1
0	5	9	0	2	0	2
1	1	6	5	0	4	5
4	4	9	4	0	0	7

$2 \downarrow [2] A$

8	6	3	0	0	0	0
4	7	5	3	2	7	1
0	5	9	0	2	0	2
1	1	6	5	0	4	5
4	4	9	4	0	0	7

# Negative rank

If the right operand of Rank is negative, it is subtracted from the rank of the argument. So `f-1` calls `f` on major cells.

`,-1 ⊢ 'abcd'`

a  
b  
c  
d

`,-1 ⊢ 4 3p□A`

ABC  
DEF  
GHI  
JKL

`,-1 ⊢ 4 3 2p124`

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24

# Negative rank

If the right operand of Rank is negative, it is subtracted from the rank of the argument. So `f¨-1` calls `f` on major cells.

`,¨-1` `⊢ 'abcd'`

a  
b  
c  
d

`,¨-1` `⊢ 4 3p⊠A`

ABC  
DEF  
GHI  
JKL

`,¨-1` `⊢ 4 3 2p⊠24`

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24

For many functions with axis, `f[k] ↔ f¨(-k)` (when `⊠I0` is 1). But good luck using Axis on `{ω[⊠ω]}`!



# Negative rank

If the right operand of Rank is negative, it is subtracted from the rank of the argument. So `f ⍤ -1` calls `f` on major cells.

`, ⍤ -1 ⊢ 'abcd'`

a  
b  
c  
d

`, ⍤ -1 ⊢ 4 3 ρ A`

ABC  
DEF  
GHI  
JKL

`, ⍤ -1 ⊢ 4 3 2 ρ 24`

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24

For many functions with axis, `f[k] ↔ f ⍤ (-k)` (when `⍴I0` is 1). But good luck using Axis on `{ω[⍤ω]}`!

If arrays are thought of as nested lists, the operator `⍤ -1` is analogous to Each.

# Rank with two arguments

With two arguments, corresponding cells are paired. This matches the two frames with each other, but the argument cell sizes are independent.

$\vdash b \leftarrow 2 \downarrow *^{-1} 1 \uparrow 12$

0	0	0	0	0	0	0	1	1	1	1	1
0	0	0	1	1	1	1	0	0	0	0	1
0	1	1	0	0	1	1	0	0	1	1	0
1	0	1	0	1	0	1	0	1	0	1	0

$\vdash e \leftarrow 2 * \phi^{-1} 1 + \uparrow 4$

8 4 2 1

# Rank with two arguments

With two arguments, corresponding cells are paired. This matches the two frames with each other, but the argument cell sizes are independent.

⊢b ← 2 1* <sup>-</sup> 1 112											
0	0	0	0	0	0	0	1	1	1	1	1
0	0	0	1	1	1	1	0	0	0	0	1
0	1	1	0	0	1	1	0	0	1	1	0
1	0	1	0	1	0	1	0	1	0	1	0
⊢e ← 2*ϕ <sup>-</sup> 1+14											
8	4	2	1								

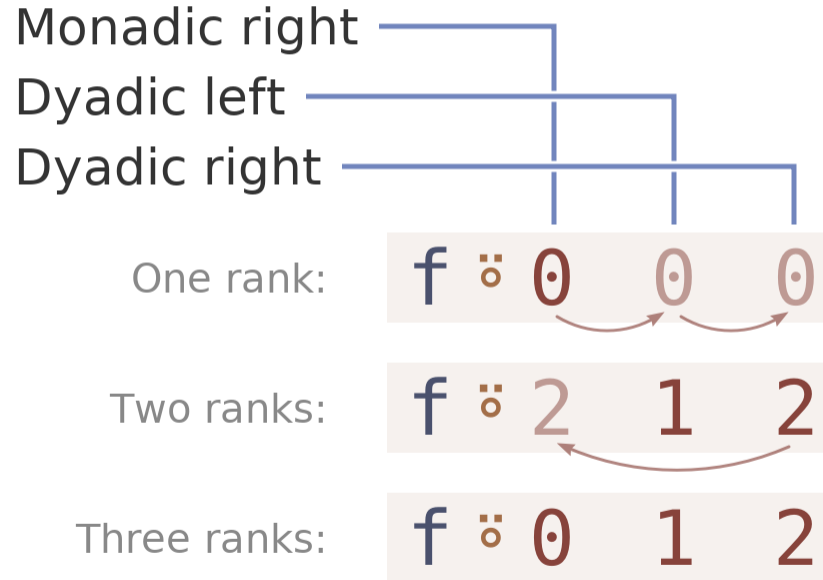
e x <sup>∘</sup> -1 ⊢b											
0	0	0	0	0	0	0	8	8	8	8	8
0	0	0	4	4	4	4	0	0	0	0	4
0	2	2	0	0	2	2	0	0	2	2	0
1	0	1	0	1	0	1	0	1	0	1	0
+ <sup>/</sup> e x <sup>∘</sup> -1 ⊢b											
1	2	3	4	5	6	7	8	9	10	11	12

The function `x` is called four times on a number from `e` and a row from `b`.

Rank allows its right operand to include one rank for each argument, so we could also have written `x∘0 1` instead of `x∘-1`.

# Right operand extension

Rank's right operand controls three possible arguments. If less than three ranks are given, they are extended as follows to cover all cases:



This extension is modelled by  $\phi 3 p \phi \text{rank}$ .

## Exercise (or guessing game)

We can find which rows of a character matrix match a string with monadic Rank:

```
⊢a ← ↑'rank' 'rack' 'tank' 'rank' 'rink'
```

```
rank  
rack  
tank  
rank  
rink
```

```
'rank' ⍳ 1 ⊢a
```

```
1 0 0 1 0
```

But this can be awkward when the string is not known in advance. Can a single function with dyadic Rank perform the same task?

## Exercise (or guessing game)

We can find which rows of a character matrix match a string with monadic Rank:

```
⊢a ← ↑'rank' 'rack' 'tank' 'rank' 'rink'
```

```
rank  
rack  
tank  
rank  
rink
```

```
'rank' ≡ 1 ⊢a
```

```
1 0 0 1 0
```

But this can be awkward when the string is not known in advance. Can a single function with dyadic Rank perform the same task?

```
'rank' ≡ 1 ⊢a
```

```
1 0 0 1 0
```

# Scatter-point indexing

The code below selects elements of the matrix `a` using rows of `i` for indices. It's equivalent to `a[↓i]` but works with flat indices directly.

```
↳ a ← 5 5p A
```

ABCDE  
FGHIJ  
KLMNO  
PQRST  
UVWXY

```
i ← 2 3 2 p 1 4 5 5 1 1 3 2 3 5 2 2
```

```
i (⌈ 1 99) a
```

DYA  
LOG

Version 18.0 introduces special code for `⌈ 1 99`, making it the fastest way to select elements or subcells.

# Scalar extension

For a scalar function such as Power (\*) there are three obvious ways it can apply to an array argument:

```
      2 * 16      ⌘ Some powers of 2
2  4  8 16 32 64

      (16) * 2    ⌘ Some squares
1  4  9 16 25 36

      (16) * 16   ⌘ Some self-exponents
1  4 27 256 3125 46656
```

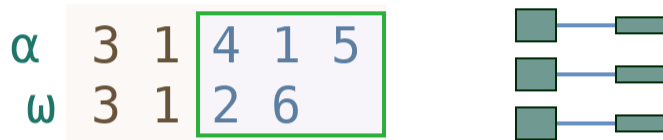
If there's a scalar argument, it *has* to be used in every function invocation, since there's nowhere else to take arguments from. Rank applies the same principle to arguments with only one cell.



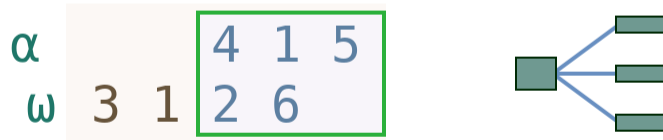
# Three possibilities

When a function with Rank is called dyadically, it may do one of three things:

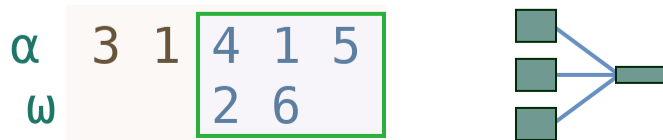
- Pair cells of the arguments one-to-one



- Copy the entire left argument to pair with cells of the right



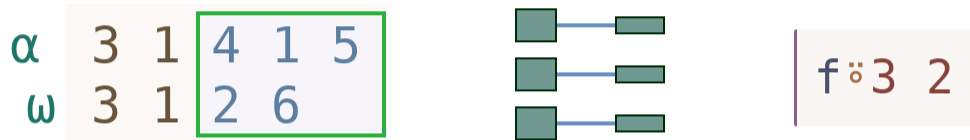
- Copy the entire right argument to pair with cells of the left



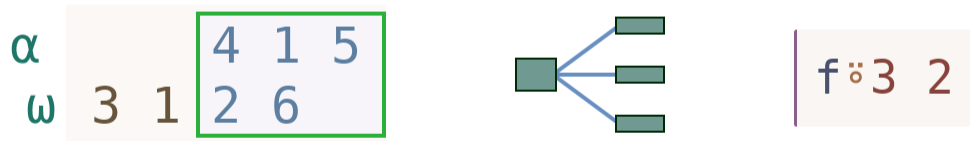
# Three possibilities

When a function with Rank is called dyadically, it may do one of three things:

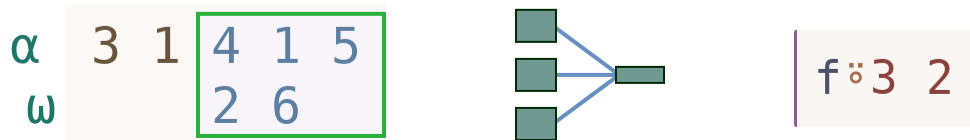
- Pair cells of the arguments one-to-one



- Copy the entire left argument to pair with cells of the right



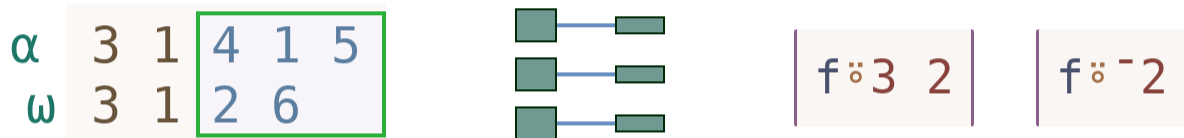
- Copy the entire right argument to pair with cells of the left



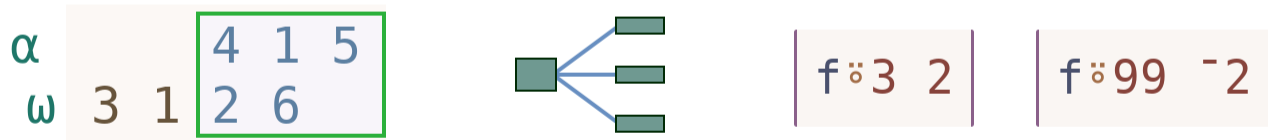
# Three possibilities

When a function with Rank is called dyadically, it may do one of three things:

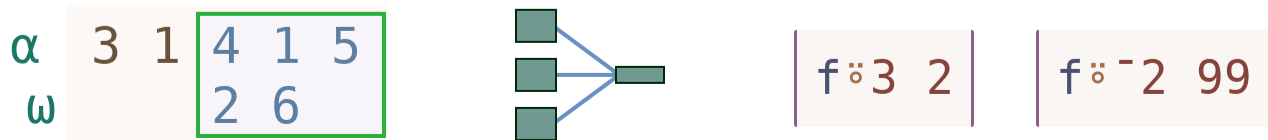
- Pair cells of the arguments one-to-one



- Copy the entire left argument to pair with cells of the right



- Copy the entire right argument to pair with cells of the left



# Vigorous exercise

Given a matrix and a single vector,  
we'd like to join each row of the matrix  
to that vector.

```
⊢ A ← 2 4 ρ 8
```

```
1 2 3 4  
5 6 7 8
```

```
⊢ B ← 0.1 × 4
```

```
0.1 0.2 0.3 0.4
```

```
A{α, [1.5+⊠I0] (ρ α) ρ ω} B
```

```
1 0.1  
2 0.2  
3 0.3  
4 0.4
```

```
5 0.1  
6 0.2  
7 0.3  
8 0.4
```

This solution is pretty awful: it requires an obscure use of Axis and doesn't check its argument shapes. Rewrite it using Rank.