



*Department of Computer Science*

# Introduction to the J Programming Language

Peter G. Anderson, Computer Science Department  
Rochester Institute of Technology, Rochester, New York  
anderson@cs.rit.edu <http://www.cs.rit.edu/>

November 16, 2009

## Abstract

Here is my latest favorite programming language, introduced with several of my favorite applications.



**“Complexity**  
makes programs hard to build  
and potentially hard to use;  
**beauty**  
is the ultimate defense  
against complexity.”

David Gelernter  
*Elegance and the Heart of Technology*  
Basic Books, 1998



# What is J (My Favorite Language)?

- an interactive language
- an interpreted language
- in the APL tradition
- developed by Ken Iverson
- expressed in ordinary ASCII
- orthogonal
- free from [www.jsoftware.com](http://www.jsoftware.com)



## From the “Dictionary”

J is a dialect of APL, a formal imperative language.

Because it is imperative, a sentence in J may also be called an instruction, and may be executed to produce a result.

Because it is formal and unambiguous it can be executed mechanically by a computer, and is therefore called a programming language.

Because it shares the analytic properties of mathematical notation, it is also called an analytic language.

APL originated in an attempt to provide consistent notation for the teaching and analysis of topics related to the application of computers, and developed through its use in a variety of topics, and its implementation in computer systems.



# Sample J Session

A =. 13

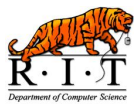
B =. 19

C =. 28

L =. i. 10

C | (A\*L) +/ (B\*L)

```
0 19 10 1 20 11 2 21 12 3
13 4 23 14 5 24 15 6 25 16
26 17 8 27 18 9 0 19 10 1
11 2 21 12 3 22 13 4 23 14
24 15 6 25 16 7 26 17 8 27
9 0 19 10 1 20 11 2 21 12
22 13 4 23 14 5 24 15 6 25
7 26 17 8 27 18 9 0 19 10
20 11 2 21 12 3 22 13 4 23
5 24 15 6 25 16 7 26 17 8
```



# Sample J Session

```
Linux Prompt: j
J 2.06 Copyright (c) 1990-1995, Iverson Software ...
```

```
    A =. 13 [ B =. 19 [ C =. 28
    L =. i. 15
    C | (A*L) +/ (B*L)
  0 19 10  1 20 11  2 21 12  3 22 13  4 23 14
13  4 23 14  5 24 15  6 25 16  7 26 17  8 27
26 17  8 27 18  9  0 19 10  1 20 11  2 21 12
11  2 21 12  3 22 13  4 23 14  5 24 15  6 25
24 15  6 25 16  7 26 17  8 27 18  9  0 19 10
 9  0 19 10  1 20 11  2 21 12  3 22 13  4 23
22 13  4 23 14  5 24 15  6 25 16  7 26 17  8
 7 26 17  8 27 18  9  0 19 10  1 20 11  2 21
20 11  2 21 12  3 22 13  4 23 14  5 24 15  6
 5 24 15  6 25 16  7 26 17  8 27 18  9  0 19
18  9  0 19 10  1 20 11  2 21 12  3 22 13  4
 3 22 13  4 23 14  5 24 15  6 25 16  7 26 17
16  7 26 17  8 27 18  9  0 19 10  1 20 11  2
 1 20 11  2 21 12  3 22 13  4 23 14  5 24 15
14  5 24 15  6 25 16  7 26 17  8 27 18  9  0
```



# J's Operators

= < > \_ + \* - % ^ \$ ~ | . : , ; # ! / \ [ ] { } " ' @ & ?

=	=.	=:	<	<.	<:	>	>.	>:	-	-.	-:
+	+.	+:	*	*.	*:	-	-.	-:	%	%.	%:
^	^.	^:	\$		\$:	~	~.	~:		.	:
.	..	::	:	::	::	,	,.	,:	;	;	;
#	#.	#:	!	!.	!:	/	/.	/:	\	\.	\:
[	[.	[:	]	].		{	{.	{:	}	}.	}:
"	".	":	'		':	@	@.	@:	&	&.	&:
?	?.	?:									

## Some Old Friends

	monadic	dyadic
=		equals
<		less than
>		greater than
+		plus
*		times
-	negate	minus
%		divide
^		power
	abs	
,		join

# Old Friends, New Names

	monadic	dyadic
=.		assignment
=:		assignment
<.	floor	min
>.	ceiling	max
<:	decrement	less than or equal
>:	increment	greater than or equal
+	conjugate	
+. *.		OR, GCD AND, LCM
+: *:	double	
-. %:	square halve square root	
* %	sign	
^ ^.	reciprocal exp log <sub>e</sub>	
~:		log not equal
 ,	abs	mod join
!	factorial	binomial coef

# New Friends

	monadic	dyadic
?	roll	deal
o. NB.	$\pi$ times comment	trig functions
– –: [ ]	negative sign infinity same same	left right

# Precedence

J has an unlimited number of operators.

So, the designers chose right to left evaluation instead of MDAS.

They retained the old friends “(” and “)”.

This takes some getting used to....

But: the rule is rigid and can be useful.



# J's Power: It's an APL!

Arrays

Tables

Matrices

Lists

Strings

# Constants & Operations

```
Linux Prompt: j
J 2.06 Copyright (c) 1990-1995, Iverson Software ...

      a =. 1 2 3 4 5 4 3 2 1
      b =. 7 7 8 8 9 9 0 0 _3
      a+b
8 9 11 12 14 13 3 2 _2
      a*b
7 14 24 32 45 36 0 0 _3
      a<b
1 1 1 1 1 1 0 0 0
      b^a
7 49 512 4096 59049 6561 0 0 _3
      #a
9
```



## Array Operations, Cont.

```

c =. a +/ b
c
8 8 9 9 10 10 1 1 _2
9 9 10 10 11 11 2 2 _1
10 10 11 11 12 12 3 3 0
11 11 12 12 13 13 4 4 1
12 12 13 13 14 14 5 5 2
11 11 12 12 13 13 4 4 1
10 10 11 11 12 12 3 3 0
9 9 10 10 11 11 2 2 _1
8 8 9 9 10 10 1 1 _2

```

```

#c
9
$c
9 9
a
1 2 3 4 5 4 3 2 1
+/a
25
+/c
88 88 97 97 106 106 25 25 _2
+/"1 c
54 63 72 81 90 81 72 63 54

```



# Subscripting

a  
1 2 3 4 5 4 3 2 1

c  
8 8 9 9 10 10 1 1 \_2  
9 9 10 10 11 11 2 2 \_1  
10 10 11 11 12 12 3 3 0  
11 11 12 12 13 13 4 4 1  
12 12 13 13 14 14 5 5 2  
11 11 12 12 13 13 4 4 1  
10 10 11 11 12 12 3 3 0  
9 9 10 10 11 11 2 2 \_1  
8 8 9 9 10 10 1 1 \_2

6 { a  
3  
2 { c  
10 10 11 11 12 12 3 3 0  
4 { 2 { c  
12  
2 4 7 { c  
10 10 11 11 12 12 3 3 0  
12 12 13 13 14 14 5 5 2  
9 9 10 10 11 11 2 2 \_1  
1 { 2 4 7 { c  
12 12 13 13 14 14 5 5 2



## Array Creation Ops: “i.”

```
i. 10
0 1 2 3 4 5 6 7 8 9
i. 4 6
0 1 2 3 4 5
6 7 8 9 10 11
12 13 14 15 16 17
18 19 20 21 22 23
i. 3 2 5
0 1 2 3 4
5 6 7 8 9

10 11 12 13 14
15 16 17 18 19

20 21 22 23 24
25 26 27 28 29
```

## Array Creation Ops: “#”

```
10 # 6
6 6 6 6 6 6 6 6 6 6
4 5 6 # 1 2 3
1 1 1 1 2 2 2 2 2 3 3 3 3 3 3
3 4 $ 4 5 6 # 1 2 3
1 1 1 1
2 2 2 2
2 3 3 3
a
1 2 3 4 5 4 3 2 1
b
7 7 8 8 9 9 0 0 _3
a < b
1 1 1 1 1 1 0 0 0
(a<b) # a
1 2 3 4 5 4
```

# Operators

	monadic	dyadic
#	count	copy
\$	shape of	shape
{		subscript
{.	head	take
	tail	
}.	behead	drop
}::		curtail
.	reverse	rotate
:	transpose	

## The Adverb “/”

	monadic	dyadic
+/	sum of	addition table
-/	alternating sum of	subtraction table
*/	product of	multiplication table
>./	max of	comparison table
<./	min of	comparison table
=/		comparison table
/		remainder table

## The Adverb “ $\sim$ ”

$x \text{ f} \sim y$  means  $y \text{ f} x$

$\text{f} \sim y$  means  $y \text{ f} y$

## Example: Find Primes

```
] T =. (P =. 2 3 5 7) |/ L =. 8 + i. 30
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1
3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2
1 2 3 4 5 6 0 1 2 3 4 5 6 0 1 2 3 4 5 6 0 1 2 3 4 5 6 0 1 2 3 4 5 6 0 1 2
] M =. *. / * T
0 0 0 1 0 1 0 0 0 1 0 1 0 0 0 1 0 0 0 0 0 1 0 1 0 0 0 0 0 1 0 1 0 0 0 0 1
] Q =. M # L
11 13 17 19 23 29 31 37
] P =. P , Q
2 3 5 7 11 13 17 19 23 29 31 37
```

## Find Primes “Functionally”

Now that we know how this works, we can express it J-functionally. Note:

Right to left evaluation

Use of `>./` for “maximum”

Use of `2^~` for “square of”

Use of `#~`

```

P =. 2 3 5 7
>./P
7
2 ^~ >./P
49
] P =. P,L #~ *./ * P | / L =. 2+i. 2^~ >./P
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47

```



## “Functions are First Class Objects”

```
sqr =. ^&2 NB. "bind" 2 as right argument of ^
max =. >./ NB. give the verb a name
P =. 2 3 5 7
] P =. P,L #~ *./ * P |/ L=. 2+i. sqr max P
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
```

We “bound” 2 on the right of ^

Alternatively:

```
TwoToThe =. 2&^
P =. 2 3 5 7
TwoToThe P
4 8 32 128
```



## Function Composition: @

```
P
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
sum =. +/
sqr =. ^&2
SumOfSquares =. sum @ sqr
SquareOfSum =. sqr @ sum
SumOfSquares P
10466
SquareOfSum P
107584
```

# Function Composition: Forks

```
fun =. f g h
```

`fun y` means `(f y) g (h y)` We could write: `(f g h) y`

```
P
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
sum =. +/
average =. sum % #      NB. The "fork"
average P
21.8667
```



# Function Composition: Hooks

```
fun =. g h
```

`fun y` means `y g (h y)` We could write: `(g h) y`

```
normalize =. % average
normalize P
0.0914634 0.137195 0.228659 0.320122 0.503049 0.594512
#P
15
3 5 $ normalize P
0.0914634 0.137195 0.228659 0.320122 0.503049
0.594512 0.777439 0.868902 1.05183 1.32622
1.41768 1.69207 1.875 1.96646 2.14939
```



# Statistics Examples

```
average  
sum % #
```

```
stddev =. %: @ average @ sqr @ ( - average )  
stddev 1 1 1  
0  
stddev 1 1 1 2 3 2  
0.745356
```

## Statistics Examples

ave            =. average

rms            =. %: @: ave @: \*:

geoMean      =. ^ @: ave @: ^.

harmMean     =. % @: ave @: %

```
(ave , rms , geoMean , harmMean) 100+i.100
149.5 152.261 146.642 143.75
```

The material inside parentheses is a “fork”



## Statistics Examples

```
sum      =. +/  
average =. sum % #  
rms      =. %: @: ave @: *:  
ave      =. average
```

```
ave 1 2 3  
2  
rms 1 2 3  
2.16025
```

```
stddev =. rms @: ( - ave )  
stddev 1 1 1 2 3 2  
0.745356
```

# Cosine

```
cos =. 2 & o.  
4 4 $ cos 0.1 * i.16  
1 0.995004 0.980067 0.955336  
0.921061 0.877583 0.825336 0.764842  
0.696707 0.62161 0.540302 0.453596  
0.362358 0.267499 0.169967 0.0707372
```

# Power of a Function

```
cos ^: 5 ( 1.1 )
0.687737
cos ^: 15 ( 1.1 )
0.738116
cos ^: 115 ( 1.1 )
0.739085
5 2 $ cos ^: (*: i. 10) ( 1.1 )
1.1 0.453596
0.81243 0.728675
0.739738 0.739066
0.739085 0.739085
0.739085 0.739085
```

# Drunk Hat-Checker: How Many Get Their Own Hat?

```

20 ? 20
17 4 9 7 12 14 18 13 0 6 15 1 16 10 2 8 3 19 5 11
?~ 20
6 7 1 2 18 13 4 15 3 0 16 19 12 17 14 8 10 11 5 9
(?~ 20) = i. 20
1 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0
+/(?~ 20) = i. 20
0
+/(?~ 20) = i. 20
0
+/(?~ 20) = i. 20
0
+/(?~ 20) = i. 20
2

```



# Drunk Hat-Checker: How Many Get Their Own Hat?

0  
+/(?~ N) = i. N =. 1000

3  
+/(?~ N) = i. N =. 1000

0  
+/(?~ N) = i. N =. 1000

1  
+/(?~ N) = i. N =. 1000

0  
+/(?~ N) = i. N =. 1000

0  
+/(?~ N) = i. N =. 1000



# The Drunk Hat-Checker: A Tacit Function

```
dhc =. +/ @ (?~ = i.)
dhc 100000
2
dhc 100000
0
dhc 100000
2
dhc 100000
0
dhc 100000
0
dhc "0 [ 30 # 10000
2 1 1 2 0 0 1 1 0 1 1 0 1 1 2 0 1 0 1 1 3 1 1 1 0 0 3 2 1
```

## Fibonacci Computing: Extend a List

```
last2 =. _1 _2 & {
last2 4 5 6 5 4 3
3 4
(sum @ last2) 4 5 6 5 4 3
7
NextFib =. , sum @ last2
NextFib 0 1
0 1 1
NextFib 0 1 1 2 3 5
0 1 1 2 3 5 8
NextFib NextFib NextFib NextFib 0 1
0 1 1 2 3 5
NextFib ^: 15 (0 1) NB. Need the parens
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```



# LPS Computing

Extend a list of Fibonacci-like numbers.

```
last31 =. _1 _3 & {  
MySeqNext =. , sum @ last31
```

```
MySeqNext 0 1 1  
0 1 1 1
```

```
MySeqNext MySeqNext MySeqNext MySeqNext 0 1 1  
0 1 1 1 2 3 4
```

```
MySeqNext ^: 15 (0 1 1)  
0 1 1 1 2 3 4 6 9 13 19 28 41 60 88 129 189 277
```



# Pascal's Triangle: From One Row to the Next

A row is the sum of shifted copies of the previous row.

```
row =. 1 4 6 4 1
      (0 , row) + (row , 0)
1 5 10 10 5 1
      NextRow =. (0 & ,) + (, & 0)
      NextRow NextRow NextRow row
1 7 21 35 35 21 7 1
      NextRow ^: 10 (1)
1 10 45 120 210 252 210 120 45 10 1
      +/ NextRow ^: 10 (1)
1024
```



# Poker

i. 52 = 0 1 2 ... 51  
represents a deck of  
cards.

Divide a card's number  
by 4 to get the value.

Remainder it mod 4 to  
get the suit.

```
NB.  define the functions
value =. <. @ % & 4
suit =. 4 & |
(value 5 ? 52) =/ i. 13
```

```
0 0 0 0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0 0 0 0
```

```
NB.  how many of each value?
```

```
+/(value 5 ? 52) =/ i. 13
0 0 1 0 0 1 1 0 0 0 0 1 1
+/(value 5 ? 52) =/ i. 13
1 1 0 0 1 2 0 0 0 0 0 0 0 0
```



## Poker: Deal Some Hands & See Their Types

```
(*M) # M =. +/(value 5 ? 52) =/ i. 13
2 1 1 1
(*M) # M =. +/(value 5 ? 52) =/ i. 13
1 2 1 1
(*M) # M =. +/(value 5 ? 52) =/ i. 13
1 1 1 2
(*M) # M =. +/(value 5 ? 52) =/ i. 13
1 1 1 1 1
(*M) # M =. +/(value 5 ? 52) =/ i. 13
1 2 2
(*M) # M =. +/(value 5 ? 52) =/ i. 13
1 1 1 1 1
(*M) # M =. +/(value 5 ? 52) =/ i. 13
1 1 2 1
```



## Poker Deal & Evaluate Routine

Create a routine,  
"DealValue."

Put it in a file "deal.j."

Call it with one (necessary)  
arg.

```
val =. <. @ % & 4
```

```
DealValue =. 3 : 0
```

```
  (*M)#M =. +/(val 5?52) =/ i. 13
```

```
  NB. Last computation is returned.
```

```
)
```

```
DealValue 1
```

```
DealValue 1
```

```
DealValue 1
```

```
DealValue 1
```

```
DealValue 1
```

```
DealValue 1
```



## Try Several Poker Hands

Run the program and see the results:

```
Linux Prompt: j < deal.j
```

```
1 1 1 1 1
```

```
1 1 1 2
```

```
1 1 1 1 1
```

```
1 1 1 1 1
```

```
1 1 1 2
```

```
1 1 1 1 1
```

```
Linux Prompt:
```



## Get Clever!

We have extracted a short list of numbers whose sum is five to summarize poker hands.

description	code	product	4-count	value
bust	1 1 1 1 1	1	0	1
one pair	2 1 1 1	2	0	2
two pair	2 2 1	4	0	4
three of a kind	3 1 1	3	0	3
full house	3 2	6	0	6
four of a kind	4 1	4	1	5

Add the product and 4-count to get a unique code.

Straights and flushes are left as exercises.



## Add a Driver Loop

```
print =. 1!:2 & 2
value =. <. @ % & 4
DealValue =. 3 : 0
    c =. (*M) # M =. +/(value 5 ? 52) =/ i. 13
    ((* / c) + (+ / 4 = c)) { 0 0 1 3 2 5 4
)
RunDV =. 3 : 0
    while. (y =. <: y) >: 0 do.
        print DealValue 1
    end.
)
trash =. RunDV 100
```

## Run the Program and See the Results

```
Linux Prompt: j < deal.j | fmt -40
0 1 0 0 1 0 0 0 1 0 1 1 1 1 0 2 0 1 0 1
0 0 1 3 0 0 1 0 1 0 0 1 0 2 1 0 0 1 1 0
1 0 1 1 1 2 0 0 1 0 0 0 2 0 0 0 0 0 1 1
3 1 1 0 1 1 2 1 0 0 1 1 0 1 0 1 1 0 1 1
1 1 1 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
```

Run the program with 1000 calls, use Unix tools, and see the results:

```
Linux Prompt: j < deal.j | sort | uniq -c
496 0
429 1
48 2
24 3
3 4
```



## Avoid Explicit Loops: Use a List of Arguments

```
value =. <. @ % & 4
```

```
DealValue =. 3 : 0
```

```
  c =. (*M) # M =. +/(value 5 ? 52) =/ i. 13
```

```
  ((*/c) + (+/4=c)) { 0 0 1 3 2 5 4
```

```
)
```

```
DealValue "0 (30 # 0)    NB.  run the program 30 times
```

Run it:

```
Linux Prompt: j < deal.j
```

```
0 1 0 0 1 0 0 0 1 0 1 1 1 1 0 2 0 1 0 1 0 0 1 3 0 0 1 0 1 0
```



# Frequency Calculation in J

Use tricks we learned about: build a table and sum it. (File is `deal.j`.)



```
value =. <. @ % & 4
```

```
DealValue =. 3 : 0
```

```
  c =. (*M) # M =. +/(value 5 ? 52) =/ i. 13
```

```
  ((*/c) + (+/4=c)) { 0 0 1 3 2 5 4  NB. Re-code values  
)
```

```
+/ (DealValue "0 (10000 # 0)) =/ i. 6
```

```
Linux Prompt: j < deal.j
```

```
5038 4235 499 209 16 3
```



## Simulate & Count All

Prompt: `j < deal2.j | sort | uniq -c`

50322 0

42107 1

4808 2

2064 3

339 4

184 5

149 6

25 7

2 8

Hand codes:

0	bust
1	1 pair
2	2 pair
3	3 of a kind
4	straight
5	flush
6	full house
7	4 of a kind
8	straight flush

## The Full Program

```
print =. 1!:2 & 2
value =. <. @ % & 4
suit =. 4 & |
flush =. 3 : '1 = # ~. suit y'
straight =. 3 : 0
    t =. +/ (value y) =/ i. 13
    2 = +/ (0,t) ~: (t,0)
)

DealValue =. 3 : 0
    c =. (*M) # M =. +/(value y) =/ i. 13
    ((*c) + (+/4=c)) { 0 0 1 3 2 7 6
)

NB. . . . continued on next slide . . .
```

## The Full Program, Cont.

```
RunDV =. 3 : 0
  while. y > 0 do.
    v =. DealValue hand =. 5 ? 52
    if. v = 0 do.
      s =. straight hand
      f =. flush hand
      if. s *. f do. v =. 8
      else. if. s do. v =. 4
      else. if. f do. v =. 5
      end. end. end. end.
    print v
    y =. <: y
  end.
)
```

# Cows and Bulls: Guess My Secret!

Secrets & guesses have 5 unique digits.

**Bull's eye:** a right digit in right place. **Cow's eye:** a right digit.

```
      ] secret =. 5 ? 10 NB. Create a secret.
8 2 4 3 7
      ] guess =. 5 ? 10 NB. Make a guess.
9 2 8 0 3
      bull =. sum @: =
      secret bull guess
1
      cow =. sum @ sum @ ( [ =/ ] )
      secret cow guess
3
```



# Permutations & Queens: Place 8 Non-attacking Queens

8?8  
6 1 3 2 5 4 0 7

row	column	0	1	2	3	4	5	6	7
0	6							Q	
1	1		Q						
2	3				Q				
3	2			Q					
4	5						Q		
5	4					Q			
6	0	Q							
7	7								Q

# Attacking Queens

Queen at  $(r, c)$  attacks  $(r', c')$  if  $r + c = r' + c'$  or  $r - c = r' - c'$

x = . 6 1 3 2 5 4 0 7

x - i. # x                      NB.      look for duplicates here

6 0 1 \_1 1 \_1 \_6 0

x + i. #x                      NB.      or here

6 2 5 5 9 9 6 14

row	column	0	1	2	3	4	5	6	7
0	6							Q	
1	1		Q						
2	3				Q				
3	2			Q					
4	5						Q		
5	4					Q			
6	0	Q							
7	7								Q

# Nub

```
x =. 6 1 3 2 5 4 0 7
] R =. x - i. # x
6 0 1 _1 1 _1 _6 0
] L =. x + i. #x
6 2 5 5 9 9 6 14
~. R NB. "nub" of R
6 0 1 _1 _6
~. L
6 2 5 9 14
# ~. R
5
# ~. L
5
```

Permutation x solves the queens problem if:

1. if length of nub of R = length of x
2. & length of nub of L = length of x

## Permutation Evaluator by the Function “solvesQ”

```
solvesQ =. 3 : 0
  R =. ~. y - i. # y
  L =. ~. y + i. # y
  ((#y) = (#L)) *. ((#y) = (#R))
)
solvesQ 8?8 NB. Try 4 random permutations.
solvesQ 8?8
solvesQ 8?8
solvesQ 8?8
solvesQ 8?8
```

Running this program yields the expected:

```
Linux Prompt: j < queens.j | fmt
0 0 0 0
```



# Permutation Maker: "A.": (i.!4) A. i. 4

0 1 2 3  
0 1 3 2  
0 2 1 3  
0 2 3 1  
0 3 1 2  
0 3 2 1  
1 0 2 3  
1 0 3 2  
1 2 0 3  
1 2 3 0  
1 3 0 2  
1 3 2 0  
2 0 1 3  
2 0 3 1  
2 1 0 3  
2 1 3 0  
2 3 0 1  
2 3 1 0  
3 0 1 2  
3 0 2 1  
3 1 0 2  
3 1 2 0  
3 2 0 1  
3 2 1 0



# Try All Permutations

```
solvesQ =. 3 : 0
  R =. ~. y - i. # y
  L =. ~. y + i. # y
  ((#y) = (#L)) *. ((#y) = (#R))
)

solvesQ "1 (i.!4) A. i. 4
, ,

solvesQ "1 (i.!5) A. i. 5
, ,

+ /solvesQ "1 (i.!N) A. i. N =. 4
+ /solvesQ "1 (i.!N) A. i. N =. 5
+ /solvesQ "1 (i.!N) A. i. N =. 6
+ /solvesQ "1 (i.!N) A. i. N =. 7
+ /solvesQ "1 (i.!N) A. i. N =. 8
```

```
Linux Prompt: j < queens.j | fmt -24
0 0 0 0 0 0 0 0 0 0 1 0
0 1 0 0 0 0 0 0 0 0 0 0

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

2 10 4 40 92
```



# Neural Nets: Perceptrons

The input is a vector,  $\vec{x} = (x_1, x_2, \dots, x_n)$  (prepend a “bias” value  $x_0 = 1$ ).

The output is a truth value  $z = \pm 1$ .

The transfer function is given by:  $z = \text{sign} \sum_{i=0}^n w_i x_i$

In J:

`mp = . +/ . * NB. Matrix product.`

`z = . * W mp x`



# Perceptron Training

Given a “training set” of in-out pairs  $\{(\vec{x}^{(p)}, t^{(p)}) ; p = 1, \dots, P\}$  Goal: determine a weights vector  $\vec{w}$  so that  $\forall p : t^{(p)} = \text{sign}(\vec{w} \cdot \vec{x}^{(p)})$

The training algorithm:

initialize  $\vec{w}$  randomly, and repeat the following

pick a training pair  $(\vec{x}^{(p)}, t^{(p)})$ ;

if  $t^{(p)} \neq \text{sign}(\vec{w} \cdot \vec{x}^{(p)})$ ,

then add  $t^{(p)} \vec{x}^{(p)}$  to  $\vec{w}$

until the system gets them all right.



## Perceptron Training

```
mp =. +/ . *      NB. Matrix mult.
perceptron =. 3 : 0
:
  X =. 1 , x [ Y =. y
  W =: (0 { $X ) $ 0 NB. W is global.
  steps =. 0
  whilst. ((steps =. >: steps) <: 1000) *. 0 < +/ errVec do.
    errVec =. Y neq * W mp X
    W =: W + +/ errVec # Y * |: X
  end.
  steps      NB. Return iteration count.
)
input =. M %~ ? 2 20 $ M =. 1000
] wTarget =. 20 15 _30 NB. For comparison.
output =. * wTarget mp 1 , input
input perceptron output
W NB. Print the global variable W.
```



# Perceptron Training

The output of the program is:

```
20 15 _30
42
10 11.187 _16.527
```



## My Procedure for Writing a PBM File

```
pbm_out =. 4 : 0          NB.  data pbm_out file
  ('P1', LF =. 10{a.) 1!:2 file =. < y , '.pbm'
  ((" |. $ x), LF) 1!:3 file NB.  append
  ( , (": x), "1 LF) 1!:3 file NB.  append
)
```

Invoke this procedure with:

```
BigBinaryMatrix pbm_out 'filename'
```

It will create **filename.pbm**

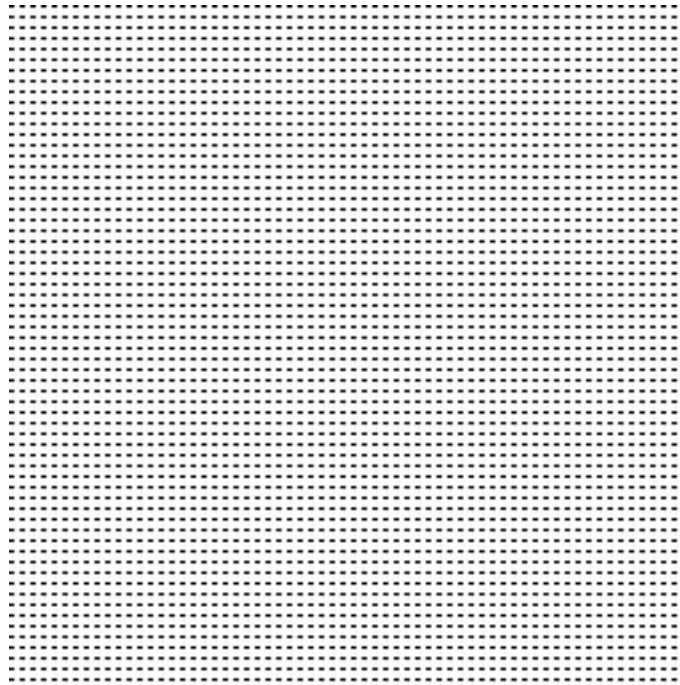


# Creating Patterns for Laser Printer Tests

```
] tile =. 2 > i. 4 4
1 1 0 0
0 0 0 0
0 0 0 0
0 0 0 0
    ,~ tile
1 1 0 0
0 0 0 0
0 0 0 0
0 0 0 0
1 1 0 0
0 0 0 0
0 0 0 0
0 0 0 0
    ,. ~ tile
1 1 0 0 1 1 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
    (,~^:6 ,~^:6 tile) pbm_out 'tile'
```



# A Texture Image for Testing a Printer: `tile.pbm`



# Portable Gray Maps: PGM Files

A  $130 \times 120$ -pixel file with maximum gray value 255 begins:

```
P2
# comment....
# comment....
120 130
255
141 152 161 160 160 164 152 128
120 104 72 56 45 33 23 24
22 21 23 25 28 30 30 35
37 41 41 43 45 51 51 49
38 36 34 32 30 26 25 25
23 22 23 26 28 30 35 36
```

The image looks like this:



Read, squeeze, append,  
repeat.....



## Useful Code for Fractals

`lf =. 10 { a.` NB. 10th ASCII is line feed

`write =. 1!:2` NB. foreign construction

`fmt =. ":` NB. format converts to ASCII

`Write =. 4 : '( , ( fmt x ) , "1 lf) write < y'`



## Useful Code for Fractals

Run the program; convert j's signs; start gnuplot.

```
j < dragon.j ; _to- dragon.data ; gnuplot
```

The code `_to-`

```
for i in $*
do
    sed 's/_/-/g' < $i > ,_$$
    mv ,_$$ $i
done
```



# Dragon Curve Maker

```
TL    =. (_1 1 & *) @ |.
```

```
TR    =. (1 _1 & *) @ |.
```

```
L =. 4 : '2 2 $ x, 2 %~ y + x + (s =: -s) * TL y - x'
```

```
R =. 4 : '2 2 $ x, 2 %~ y + x +                TR y - x'
```

```
build =. 3 : 0
```

```
    s =: 1
```

```
    ,/ y L " 1 1 (1 |. y)
```

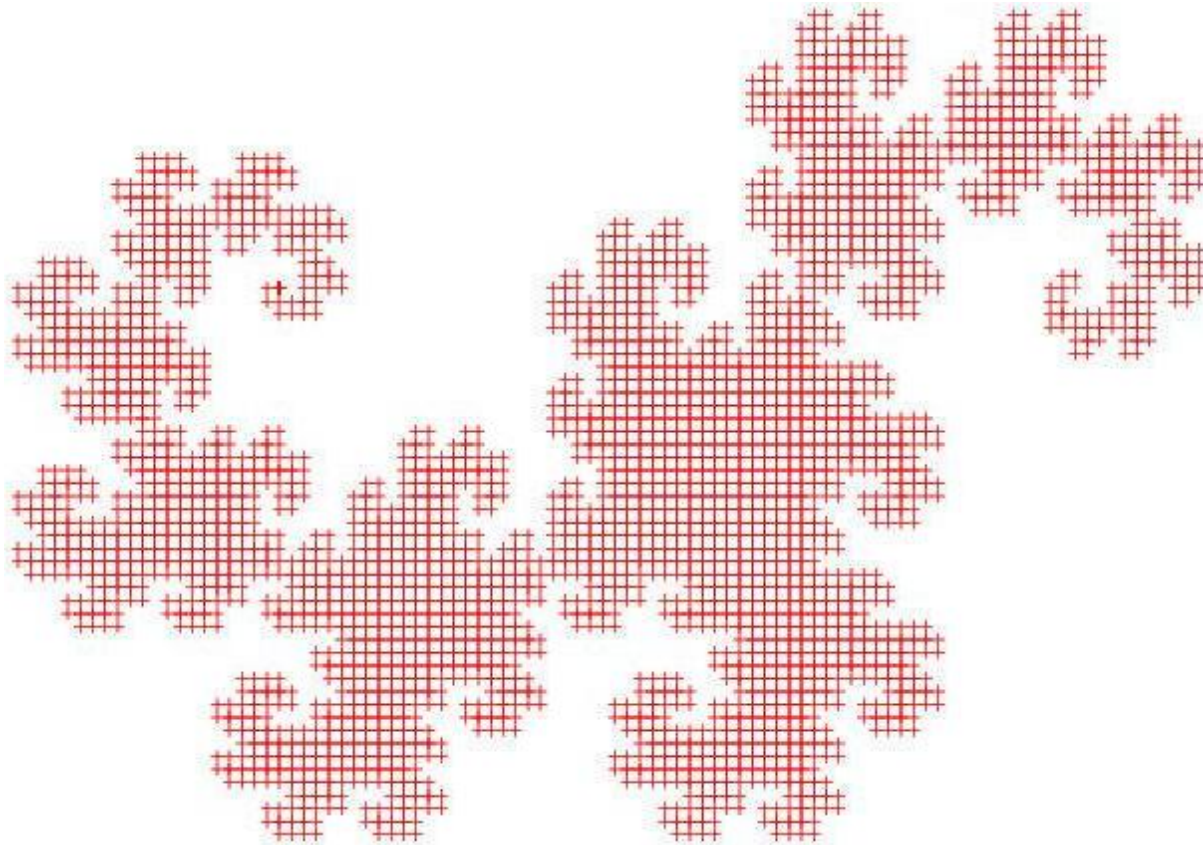
```
)
```

```
a =. 3 2 $ 0 0 1 0 0 0
```

```
(build ^:12 a) Write 'dragon.data'
```



# The Dragon



# Randomized Dragon Curve Maker

```
TL    =. (_1 1 & *) @ |.
```

```
TR    =. (1 _1 & *) @ |.
```

```
L =. 4 : '2 2 $ x, 2 %~ y + x + (s =: 1-2*?2) * TL y - x'
```

```
R =. 4 : '2 2 $ x, 2 %~ y + x + TR y - x'
```

```
build =. 3 : 0
```

```
    s =: 1
```

```
    ,/ y L " 1 1 (1 |. y)
```

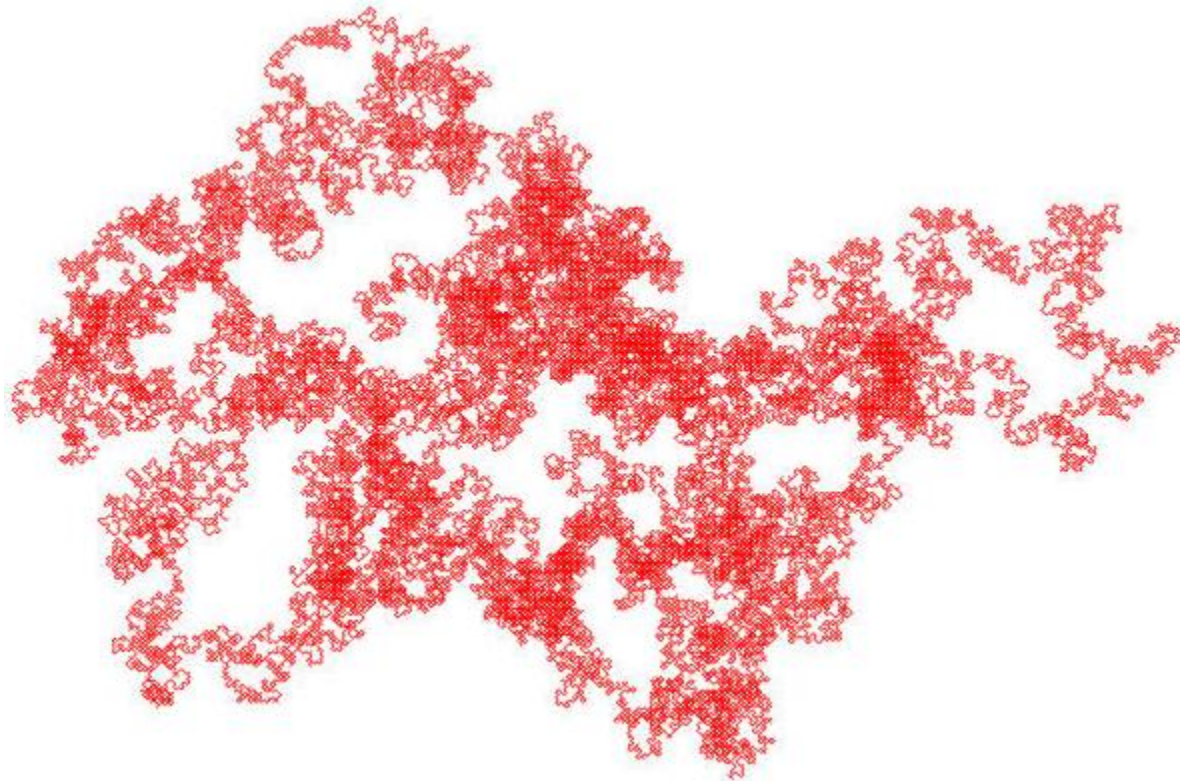
```
)
```

```
a =. 3 2 $ 0 0 1 0 0 0
```

```
(build ^:15 a) Write 'random_dragon.data'
```



# The Random Dragon

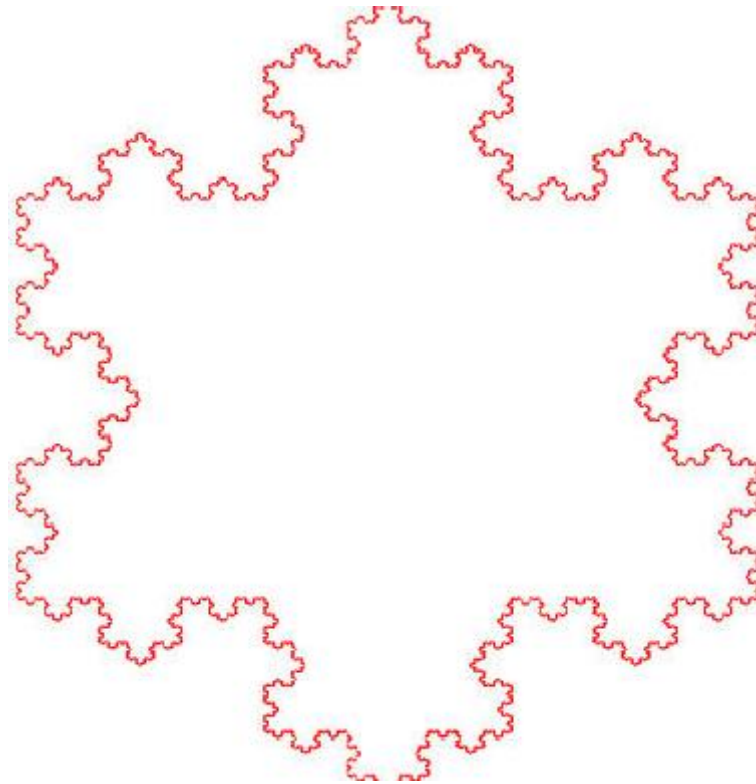


# Koch's Snowflake

```
TL =. (_1 1 & *) @ |.  
alph =. 6 %~ 3^0.5  
L =. 3 : 0  
:  
  p =. 2 %~ x + y  
  NB. q =. (1 -~ 2 * ? 2 ) * alph * TL y - x  
  q =. alph * TL y - x  
  4 2 $ x, (3%~y+2*x), (p+q) , (3%~x+2*y)  
)  
build =. 3 : ',/y L " 1 1 (1 |. y)'  
a =. 3 2 $ 0 0 0.5, (2%~3^0.5), 1 0  
(build ^:7 a) Write 'koch.data'
```



# Koch's Snowflake

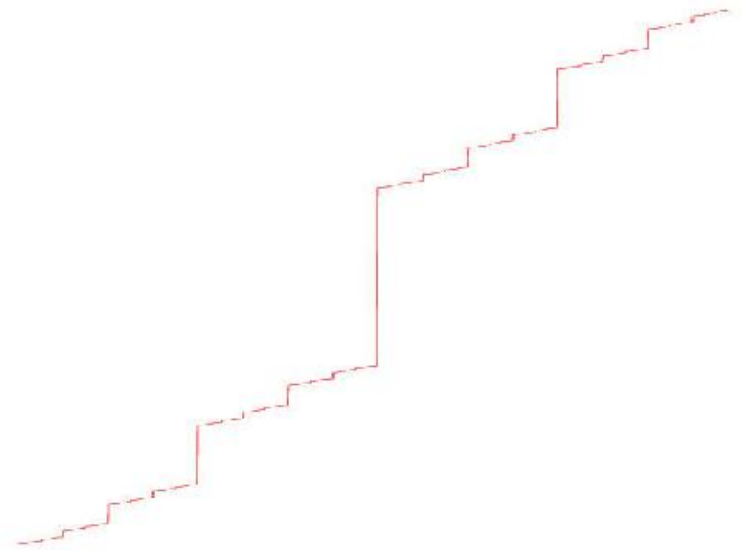


# Cantor's Middle-Third Set & The Devil's Staircase

```
Write =. 4 : '(,(" : x) ,"1 (10{a.)) 1!:2 <y'  
sort =: /:~  
N =. 1000  
L =. N %~ i. N  
mask =: (2*N) $ 1 0  
A =: 1 + a =: 2  
ifs =. 3 : 'mask # sort A %~ y , y + a'  
(L =. ifs ^: 20 L) Write 'dist.data'
```



# The Devil's Staircase



## The Modulo-2 Pascal Triangle

```
pt =. , ((1&|. ) ~: ( _1&|. )) @ {:
```

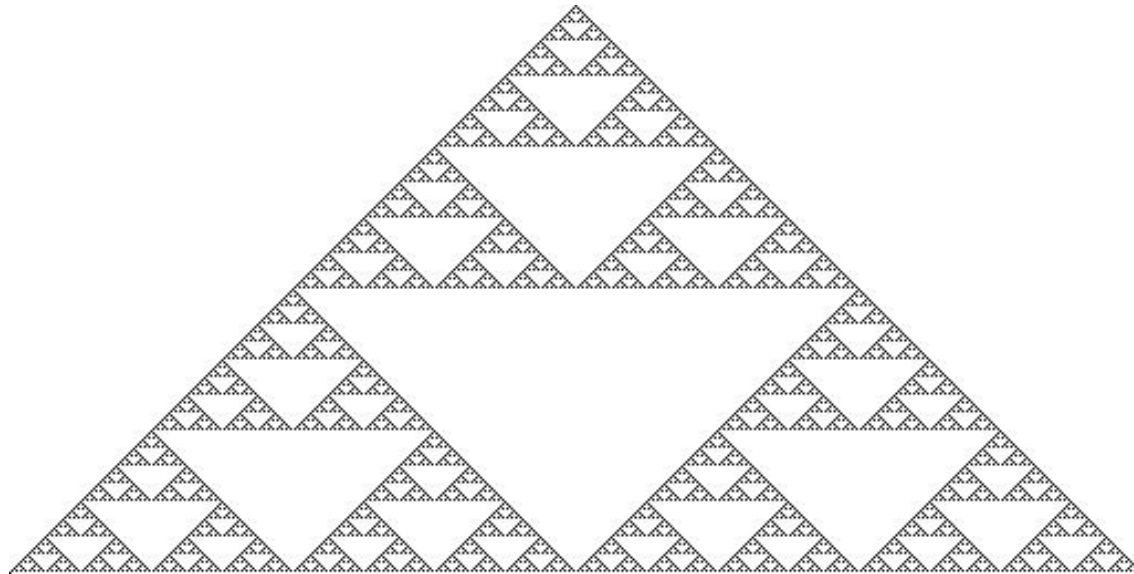
```
pt ^:N (1, >: +: N) $ (N,1,N =. 10) # 0 1 0
```

```
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 1 0 1 0 1 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 1 0 1 0 0 0 0 0 1 0 1 0 0 0 0
0 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0
0 0 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 0
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0
1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1
```

```
(pt ^:N (1, >: +: N) $ (N,1,N =. 256) # 0 1 0) pbm_out 'pascal'
```



# The Modulo-2 Pascal Triangle



## Boxed Items

```

    <123
+----+
|123|
+----+
    (<1), (<2), (<3)
+--+--+
|1|2|3|
+--+--+
    1;2;3
+--+--+
|1|2|3|
+--+--+

```

```

    ] a =. (<i.5), (<i.2), (<'box')
+-----+--+--+
|0 1 2 3 4|0 1|box|
+-----+--+--+
    ] b =. ,. (<i.5), (<i.2), (<'box')
+-----+
|0 1 2 3 4|
+-----+
|0 1      |
+-----+
|box      |
+-----+

```

# Visualizing Ranks

```

      <i. 2 3 4
+-----+
| 0  1  2  3|
| 4  5  6  7|
| 8  9 10 11|
|          |
|12 13 14 15|
|16 17 18 19|
|20 21 22 23|
+-----+

```

```

      < "2 i. 2 3 4
+-----+-----+
|0 1  2  3|12 13 14 15|
|4 5  6  7|16 17 18 19|
|8 9 10 11|20 21 22 23|
+-----+-----+

```



# Visualizing Ranks

```

    < "1 i. 2 3 4
+-----+-----+-----+
|0 1 2 3   |4 5 6 7   |8 9 10 11 |
+-----+-----+-----+
|12 13 14 15|16 17 18 19|20 21 22 23|
+-----+-----+-----+
  
```

```

    < "0 i. 2 3 4
+--+--+--+--+
|0 |1 |2 |3 |
+--+--+--+--+
|4 |5 |6 |7 |
+--+--+--+--+
|8 |9 |10|11|
+--+--+--+--+
  
```

```

+--+--+--+--+
|12|13|14|15|
+--+--+--+--+
|16|17|18|19|
+--+--+--+--+
|20|21|22|23|
+--+--+--+--+
  
```



# Visualizing Ranks

```

    < i. 3 4
+-----+
|0 1 2 3|
|4 5 6 7|
|8 9 10 11|
+-----+
    < "1 i. 3 4
+-----+-----+-----+
|0 1 2 3|4 5 6 7|8 9 10 11|
+-----+-----+-----+

```

```

    ,.< "1 i. 3 4
+-----+
|0 1 2 3 |
+-----+
|4 5 6 7 |
+-----+
|8 9 10 11|
+-----+
    < "0 i. 3 4
+--+--+--+--+
|0|1|2 |3 |
+--+--+--+--+
|4|5|6 |7 |
+--+--+--+--+
|8|9|10|11|
+--+--+--+--+

```



# From the “Dictionary”—Grammar & Six Parts of Speech

<pre> Fahrenheit=: 50 (Fahrenheit-32)*5%9 10 prices=: 3 1 4 2 orders=: 2 0 2 1 orders * prices 6 0 8 2 +/orders*prices 16 +/\1 2 3 4 5 1 3 6 10 15 bump=: 1&amp;+ bump prices 4 2 5 3 </pre>	<pre> 50 Fahrenheit + - * % bump / \ &amp; ( ) =: </pre>	<pre> Nouns/Pronouns Verbs/Proverbs Adverbs Conjunction Punctuation Copula </pre>
--	--	---



## From the “Dictionary”—Parsing

Parsing proceeds by moving successive elements (or their values except in the case of proverbs and names immediately to the left of a copula) from the tail end of a queue to the top of a stack, and eventually executing some eligible portion of the stack and replacing it by the result of the execution. For example, if  $a =: 1\ 2\ 3$ , then  $b =: +/2*a$  would be parsed and executed as follows:

input queue	stack
M b =: + / 2 * a	
M b =: + / 2 *	1 2 3
M b =: + / 2	* 1 2 3
M b =: + /	2 * 1 2 3
M b =: +	/ 2 * 1 2 3
M b =: +	/ 2 4 6
M b =:	+ / 2 4 6
M b	=: + / 2 4 6
M b	=: 12
M	b =: 12
M	12
	M 12



## From the “Dictionary” —Parse Rules

The executions in the stack are confined to the first four elements only, and eligibility for execution is determined only by the class of each element (noun, verb, etc., an unassigned name being treated as a verb), as prescribed in the following parse table. The classes of the first four elements of the stack are compared with the first four columns of the table, and the first row that agrees in all four columns is selected. The underlined elements in the row are then subjected to the action shown in the final column, and are replaced by its result. If no row is satisfied, the next element is transferred from the queue.

## From the “Dictionary” —Parse Rules

EDGE	<u>VERB</u>	<u>NOUN</u>	ANY	0 Monad
EDGE+AVN	VERB	<u>VERB</u>	<u>NOUN</u>	1 Monad
EDGE+AVN	<u>NOUN</u>	<u>VERB</u>	<u>NOUN</u>	2 Dyad
EDGE+AVN	<u>VERB+NOUN</u>	<u>ADV</u>	ANY	3 Adverb
EDGE+AVN	<u>VERB+NOUN</u>	<u>CONJ</u>	<u>VERB+NOUN</u>	4 Conj
EDGE+AVN	<u>VERB</u>	<u>VERB</u>	<u>VERB</u>	5 Trident
EDGE	<u>CAVN</u>	<u>CAVN</u>	<u>CAVN</u>	6 Trident
EDGE	<u>CAVN</u>	<u>CAVN</u>	ANY	7 Bident
NAME+NOUN	<u>ASGN</u>	<u>CAVN</u>	ANY	8 Is
<u>LPAR</u>	<u>CAVN</u>	<u>RPAR</u>	ANY	9 Paren

Legend: **AVN** denotes ADV+VERB+NOUN; **CAVN** denotes  
 CONJ+ADV+VERB+NOUN;  
**EDGE** denotes MARK+ASGN+LPAR

