Introduction to J

Yes Context

- Developed by Kenneth E. Iverson and Roger Hui in early 1990s based on APL
  - Product of Canada / Produit du Canada
- Array-based: array is the universal data structure
- Famous for its brevity, infamous for its obscurity
  - Cartesian product
- Function-level programming via tacit function definitions
- Strong in data analysis, statistics, math-related fields

Write-Only?

"Programs must be written for people to read, and only incidentally for machines to execute."

— *Structure and Interpretation of Computer Programs*

Which of the following four 60-character lines is J code?

- A

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

- B

  ```
  +?-@_->?$@$[|$\$}>$#'^!+)~.-=;#'>@#'`^&?@$^'@;:_>:*!`([+<]_&
  ```

- C

  ```
  ((?@#{])@]($:@(]#~[>]),(]#~[=]),$:@(]#~[<])))])`(''"_)@.(0=#)
  ```

- D

  ```
  (+#?~)+;/,|;}+}([/!~\*+_++]_)\~|^)`*;;"~]=[[]]*-+`:+|\~`%.|?
  ```

Read J

- Part of Speech
- Right-to-left Evaluation
- Monad vs. Dyad
- Rank
- Composition/Train of Verbs

Read J: Part of Speech

4 categories:

- Noun: data to be manipulated
- Verb
  - input: noun(s)
  - output: noun(s)
- Adverb: one type of *modifiers*
  - input: *one* verb or noun
  - output: a verb
- Conjunction: the other type of *modifiers*
  - input: *two* verbs and/or nouns
  - output: a verb

Other borrowed concepts:

- Gerund: one or more verbs grouped together as a noun
- Inflection: appending `:` or `.` to a verb to form a new verb
  - Thus we can use ASCII character for everything

Read J: Right-to-left Evaluation

```
   7 - 6 + 5
_4
   (7 - 6) + 5
6
```

Note that `_4` is the literal negative four but `-4` is the `-` (*negate*) verb applied on four.

Read J: Monad vs. Dyad

- Monads take one argument: `u y`

```
   !50x
30414093201713378043612608166064768844377641568960512000000000000
   i.10
0 1 2 3 4 5 6 7 8 9
```

- Dyads take two arguments: `x u y`

```
   NB. randomly select 3 numbers from [0, 9] without repetition
   3 ? 10
6 3 9
```

- A verb may behave differently when called as monad and dyad.

```
   NB. increment as monad
   >: 99
100
   NB. greater-or-equal as dyad
   4 >: 99
0
```

Read J: Rank 1/3

Length, shape, rank of arrays:

```
   NB. more on train of verbs later
   inspect=: ('length';'shape';'rank') ,: # ; $ ; #@$
   inspect 1
+------+-----+----+
|length|shape|rank|
+------+-----+----+
|1     |     |0   |
+------+-----+----+
   inspect ''
+------+-----+----+
|length|shape|rank|
+------+-----+----+
|0     |0    |1   |
+------+-----+----+
   inspect i.3 8
+------+-----+----+
|length|shape|rank|
+------+-----+----+
|3     |3 8  |2   |
+------+-----+----+
```

Read J: Rank 2/3

Verbs applied on different ranks:

```
   ]a=: 2 4 $ i.12   NB. 2x4 array initialized with 0-11 inclusively
0 1 2 3
4 5 6 7
   $a                NB. find the shape of a
2 4
   < b. 0            NB. query rank info of verb < (box)
_ 0 0
   <a                NB. box a on maximum rank
+-------+
|0 1 2 3|
|4 5 6 7|
+-------+
   <"1 a             NB. box a on rank 1: rows via rank conjunction "
+-------+-------+
|0 1 2 3|4 5 6 7|
+-------+-------+
   <"0 a             NB. box a on rank 0: scalars
+-+-+-+-+
|0|1|2|3|
+-+-+-+-+
```

Read J: Rank 3/3

A more confusing example:

```
   ]a=: 2 4 $ i.12
0 1 2 3
4 5 6 7
   +/ b. 0
_ _ _
   NB. apply on a as a 2D array: insert + between rows
   NB.   0 1 2 3
   NB.      +
   NB.   4 5 6 7
   +/ a
4 6 8 10
   NB. apply on a's 1D arrays: insert + inside each row
   NB.   0 + 1 + 2 + 3
   NB.   4 + 5 + 6 + 7
   +/"1 a
6 22
```

Read J: Composition of Verbs

```
   NB. monad tracking monad: square then increment
   (>: @: *:) 5
26
   NB. monad tracking dyad: average of two numbers
   3 (-: @: +) 5
4
   NB. dyad tracking monad: sum of squares
   3 (+ &: *:) 4
25
```

Actually there are four: `@`, `@:`, `&`, `&:`.

https://code.jsoftware.com/wiki/File:Funcomp.png

Read J: Train of Verbs 1/2

Hooks

- Monadic: `(f g) y` equals `y f (g y)`
- Dyadic: `x (f g) y` equals `x f (g y)`

```
   NB. ,. y (raval items) creates an array whose rows come
   NB. from the items of the argument so we get a column vector
   ]a=: ,. i.5
0
1
2
3
4
   NB. x ,. y (stitch) joins the corresponding items of x and y
   NB. this equals a ,. (*: a)
   (,. *:) a
0  0
1  1
2  4
3  9
4 16
```

Read J: Train of Verbs 2/2

Forks

- Monadic: `(f g h) y` equals `(f y) g (h y)`

```
   NB. monadic <. is floor, monadic >. is ceiling
   (<. ; >.) 2.646
+-+-+
|2|3|
+-+-+
```

- Dyadic: `x (f g h) y` equals `(x f y) g (x h y)`

```
    NB. dyadic +. is GCD, dyadic *. is LCM
    16 (+. ; *.) 24
    +-+--+
    |8|48|
    +-+--+
```

Now let's see more (bad) examples

Examples 1/4

Project Euler Problem 1: Multiples of 3 and 5

> If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The sum of these multiples is 23.
>
> Find the sum of all the multiples of 3 or 5 below 1000.

Answer: 233168

Examples 2/4

Rewrite the function at the beginning of the slides:

```
((?@#{])@]($:@(]#~[>]),(]#~[=]),$:@(]#~[<]))])`(''"_)@.(0=#)
```

```
   pivot=: ?@# { ]
   left =: ([>])#]
   right=: ([<])#]
   mid  =: ([=])#]
   empty=: 0=#
   qsort=: (pivot@] (qsort@left , mid , qsort@right) ]) ` (''"_) @. empty
   qsort ?10#100
6 9 28 32 33 38 60 62 90 92
```

Take `right` as an example, where `x` is the pivot and `y` is the array:

```
              # (copy)
         /              \
      < (less-than)      ] (right)
     /        \         / \
   [ (left)    ]       x   y
  / \         / \
 x   y       x   y
```

Examples 3/4

Actually the other 3 choices were generated by the following function:

```
   NB. usage: <length> noise <alphabet>
   noise=: ] {~ ?@([ $ #@])
   alphabet=: '!"#$%&''()*+,-./:;<=>?@[\]^_`{|}~'

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

Examples 4/4

Conway's Game of Life

In a 2D grid, considering the 8 neighbours of a cell

- If a cell is live:
  - it dies if it has < 2 or has > 3 live neighbours
  - it lives if it has 2 or 3 live neighbours
- If a cell is dead:
  - it lives if it has 3 live neighbours

Otherwise a cell remains its state.

Questions?

After this F!F you will all be J'ing.