WIKIPEDIA

J (programming language)

The **J** programming language, developed in the early 1990s by Kenneth E. Iverson and Roger Hui,[5][6] is an array programming language based primarily on APL (also by Iverson).

To avoid repeating the APL special-character problem, J uses only the basic ASCII character set, resorting to the use of the dot and colon as *inflections*[7] to form short words similar to *digraphs*. Most such *primary* (or *primitive*) J words serve as mathematical symbols, with the dot or colon extending the meaning of the basic characters available. Also, many characters which in other languages often must be paired (such as `[] {} "" ``` or `<>`) are treated by J as stand-alone words or, when inflected, as single-character roots of multi-character words.

J is a very terse array programming language, and is most suited to mathematical and statistical programming, especially when performing operations on matrices. It has also been used in extreme programming[8] and network performance analysis.[9]

Like John Backus's languages FP and FL, J supports function-level programming via its *tacit programming* features.

Unlike most languages that support object-oriented programming, J's flexible hierarchical namespace scheme (where every name exists in a specific *locale*) can be effectively used as a framework for both class-based and prototype-based object-oriented programming.

Since March 2011, J is free and open-source software under the GNU General Public License version 3 (GPLv3).[10][11][12] One may also purchase source under a negotiated license.[13]

J

Designed by	Kenneth E. Iverson, Roger Hui
Developer	JSoftware
First appeared	1990
Stable release	J901 / 15 December 2019[1]
Typing discipline	dynamic
OS	Cross-platform: Windows, Linux, macOS, Android, iOS, Raspberry Pi[2]
License	GPLv3
Website	www.jsoftware.com (http://www.jsoftware.com)
Major implementations	
J	
Influenced by	
APL	
Influenced	
NumPy,[3] SuperCollider[4]	

Contents

Examples

J permits point-free style and function composition. Thus, its programs can be very terse and are considered difficult to read by some programmers.

The "Hello, World!" program in J is

```
  'Hello, world!'
```

This implementation of hello world reflects the traditional use of J – programs are entered into a J interpreter session, and the results of expressions are displayed. It's also possible to arrange for J scripts to be executed as standalone programs. Here's how this might look on a Unix system:

```
   #!/bin/jc
   echo 'Hello, world!'
   exit ''
```

Historically, APL used `/` to indicate the fold, so `+/1 2 3` was equivalent to `1+2+3`. Meanwhile, division was represented with the mathematical division symbol (`÷`), which was implemented by overstriking a minus sign and a colon (on both EBCDIC and ASCII paper text terminals). Because ASCII in general does not support overstrikes in a device-independent way, and does not include a division symbol *per se*, J uses % to represent division, as a visual approximation or reminder. (This illustrates something of the mnemonic character of J's tokens, and some of the quandaries imposed by the use of ASCII.)

Defining a J function named `avg` to calculate the average of a list of numbers yields:

```
  avg=: +/ % #
```

This is a test execution of the function:

```
  avg 1 2 3 4
2.5
```

`#` counts the number of items in the array. `+/` sums the items of the array. `%` divides the sum by the number of items. Above, *avg* is defined using a train of three verbs (`+/`, `%`, and `#`) termed a *fork*. Specifically `(V0 V1 V2) Ny` is the same as `(V0(Ny)) V1 (V2(Ny))` which shows some of the power of J. (Here V0, V1, and V2 denote verbs and Ny denotes a noun.)

Some examples of using `avg`:

```
  v=: ?. 20 $100     NB. a random vector
  v
46 55 79 52 54 39 60 57 60 94 46 78 13 18 51 92 78 60 90 62
  avg v
59.2
```

```
  4 avg\ v            NB. moving average on periods of size 4
58 60 56 51.25 52.5 54 67.75 64.25 69.5 57.75 38.75 40 43.5 59.75 70.25 80 72.5
```

```
   m=: ?. 4 5 $50     NB. a random matrix
   m
46  5 29  2  4
39 10  7 10 44
46 28 13 18  1
42 28 10 40 12
```

```
   avg"1 m              NB. apply avg to each rank 1 subarray (each row) of m
17.2 22 21.2 26.4
```

Rank is a crucial concept in J. Its significance in J is similar to the significance of `select` in SQL and of `while` in C.

Implementing quicksort, from the J Dictionary yields:

```
    sel=: adverb def 'u # ['

    quicksort=: verb define
     if. 1 >: #y do. y
     else.
      (quicksort y <sel e),(y =sel e),quicksort y >sel e=.y{~?#y
     end.
    )
```

The following is an implementation of quicksort demonstrating tacit programming. The latter involves composing functions together and not referring explicitly to any variables. J's support for *forks* and *hooks* dictates rules on how arguments applied to this function will be applied to its component functions.

```
    quicksort=: (($:@(<#[), (=#[), $:@(>#[)) ({~ ?@#)) ^: (1<#)
```

Sorting in J is usually accomplished using the built-in (primitive) verbs `/:` (sort up) and `\:` (sort down). User-defined sorts such as quicksort, above, typically are for illustration only.

The following example demonstrates the usage of the self-reference verb `$:` to recursively calculate fibonacci numbers:

```
1:`($:@-&2+$:@<:)@.(>&2)
```

This recursion can also be accomplished by referring to the verb by name, although this is of course only possible if the verb is named:

```
fibonacci=:1:`(fibonacci@-&2+fibonacci@<:)@.(>&2)
```

The following expression exhibits pi with n digits and demonstrates the extended precision abilities of J:

```
   n=: 50                      NB. set n as the number of digits required
   <.@o. 10x^n                 NB. extended precision 10 to the nth * pi
314159265358979323846264338327950288419716939937510
```

Verbs and Modifiers

A program or routine - something that takes data as input and produces data as output - is called a *verb*. J has a rich set of predefined verbs, all of which work on multiple data types automatically: for example, the verb `i.` searches within arrays of any size to find matches:

```
   3 1 4 1 5 9 i. 3 1  NB. find the index of the first occurrence of 3, and of 1
0 1
   3 1 4 1 5 9 i: 3 1  NB. find the index of the last occurrence of 3, and of 1
0 3
```

User programs can be named and used wherever primitives are allowed.

The power of J comes largely from its *modifiers*: symbols that take nouns **and verbs** as operands and apply the operands in a specified way. For example, the modifier `/` takes one operand, a verb to its left, and produces a verb that applies that verb between each item of its argument. That is, `+/` is a verb, defined as 'apply `+` between the items of your argument' Thus, the sentence

```
   +/ 1 2 3 4 5
```

produces the effect of

```
   1 + 2 + 3 + 4 + 5

   +/ 1 2 3 4 5
15
```

J has roughly two dozen of these modifiers. All of them can apply to any verb, even a user-written verb, and users may write their own modifiers. While modifiers are powerful individually, allowing

- repeated execution, i. e. *do-while*
- conditional execution, i. e. *if*
- execution of regular or irregular subsets of arguments

some of the modifiers control the order in which components are executed, allowing modifiers to be combined in any order to produce the unlimited variety of operations needed for practical programming.

Data types and structures

J supports three simple types:

- Numeric
- Literal (Character)
- Boxed

Of these, numeric has the most variants.

One of J's numeric types is the *bit*. There are two bit values: *0*, and *1*. Also, bits can be formed into lists. For example, `1 0 1 0 1 1 0 0` is a list of eight bits. Syntactically, the J parser treats that as one word. (The space character is recognized as a word-forming character between what would otherwise be numeric words.) Lists of arbitrary length are supported.

Further, J supports all the usual binary operations on these lists, such as *and*, *or*, *exclusive or*, *rotate*, *shift*, *not*, etc. For example,

```
   1 0 0 1 0 0 1 0 +. 0 1 0 1 1 0 1 0     NB. or
1 1 0 1 1 0 1 0
```

```
   3 |. 1 0 1 1 0 0 1 1 1 1 1             NB. rotate
1 0 0 1 1 1 1 1 1 0 1
```

J also supports higher order arrays of bits. They can be formed into two-dimensional, three-dimensional, etc. arrays. The above operations perform equally well on these arrays.

Other numeric types include integer (e.g., 3, 42), floating point (3.14, 8.8e22), complex (0j1, 2.5j3e88), extended precision integer (12345678901234567890x), and (extended precision) rational fraction (1r2, 3r4). As with bits, these can be formed into lists or arbitrarily dimensioned arrays. As with bits, operations are performed on all numbers in an array.

Lists of bits can be converted to integer using the `#.` verb. Integers can be converted to lists of bits using the `#:` verb. (When parsing J, `.` (period) and `:` (colon) are word-forming characters. They are never tokens alone, unless preceded by whitespace characters.)

J also supports the literal (character) type. Literals are enclosed in quotes, for example, `'a'` or `'b'`. Lists of literals are also supported using the usual convention of putting multiple characters in quotes, such as `'abcdefg'`. Typically, individual literals are 8-bits wide (ASCII), but J also supports other literals (Unicode). Numeric and boolean operations are not supported on literals, but collection-oriented operations (such as rotate) are supported.

Finally, there is a boxed data type. Typically, data is put in a box using the < operation (with no left argument; if there's a left argument, this would be the *less than* operation). This is analogous to C's & operation (with no left argument). However, where the result of C's & has reference semantics, the result of J's < has value semantics. In other words, < is a function and it produces a result. The result has 0 dimensions, regardless of the structure of the contained data. From the viewpoint of a J programmer, < *puts the data into a box* and allows working with an array of boxes (it can be assembled with other boxes, and/or more copies can be made of the box).

```
   <1 0 0 1 0
+---------+
|1 0 0 1 0|
+---------+
```

The only collection type offered by J is the arbitrarily dimensioned array. Most algorithms can be expressed very concisely using operations on these arrays.

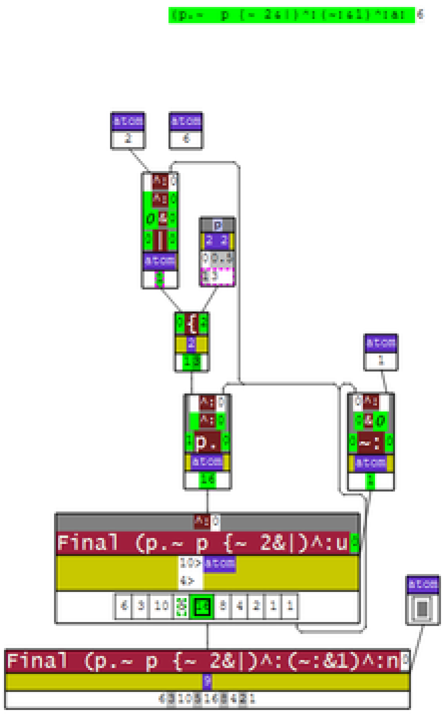
J's arrays are homogeneously typed, for example the list `1 2 3` is a list of integers despite `1` being a bit. For the most part, these sorts of type issues are transparent to programmers. Only certain specialized operations reveal differences in type. For example, the list `1.0 0.0 1.0 0.0` would be treated exactly the same, by most operations, as the list `1 0 1 0` .

J also supports sparse numeric arrays where non-zero values are stored with their indices. This is an efficient mechanism where relatively few values are non-zero.

J also supports objects and classes,[14] but these are an artifact of the way things are named, and are not data types. Instead, boxed literals are used to refer to objects (and classes). J data has value semantics, but objects and classes need reference semantics.

Another pseudo-type—associated with name, rather than value—is the memory mapped file.

Debugging



Dissecting the Collatz sequence starting from 6

J has the usual facilities for stopping on error or at specified places within verbs. It also has a unique visual debugger, called *Dissect* (http://code.jsoftware.com/wiki/Vocabulary/Dissect), that gives a 2-D interactive display of the execution of a single J sentence. Because a single sentence of J performs as much computation as an entire subroutine in lower-level languages, the visual display is quite helpful.

Documentation

J's documentation includes a dictionary (http://code.jsoftware.com/wiki/NuVoc), with words in J identified as nouns (http://code.jsoftware.com/wiki/Vocabulary/Nouns), verbs (http://code.jsoftware.com/wiki/Vocabulary/Verbs), modifiers (http://code.jsoftware.com/wiki/Vocabulary/Modifiers), and so on. Primary words are listed in the vocabulary (http://code.jsoftware.com/wiki/Vocabulary/Words), in which their respective parts of speech (http://code.jsoftware.com/wiki/Vocabulary/PartsOfSpeech) are indicated using markup. Note that verbs have two forms: monadic (arguments only on the right) and dyadic (arguments on the left and on the right). For example, in '`-1`' the hyphen is a monadic verb, and in '`3-2`' the hyphen is a dyadic verb. The monadic definition is mostly independent of the dyadic definition, regardless of whether the verb is a primitive verb or a derived verb.

Control structures

J provides control structures (details here) (http://jsoftware.com/help/dictionary/ctrl.htm) similar to other procedural languages. Prominent control words in each category include:

- `assert.`
- `break.`
- `continue.`
- `for.`
- `goto_label.`
- `if. else. elseif.`
- `return.`
- `select. case.`
- `throw.`
- `try. catch.`
- `while. whilst.`

See also

- K (programming language) – another APL-influenced language
- Q – The language of KDB+ and a new merged version of K and KSQL.

References

1. "J901 release 15 December 2019" (https://code.jsoftware.com/wiki/System/ReleaseNotes/J901).
2. https://www.jsoftware.com/#/README
3. Wes McKinney at 2012 meeting Python for Data Analysis (https://traims.tumblr.com/post/33883718232/python-for-data-analysis-18-oct-2012-london)
4. SuperCollider documentation, Adverbs for Binary Operators (http://doc.sccode.org/Reference/Adverbs.html)
5. A Personal View of APL (https://web.archive.org/web/20040812193452/http://home1.gte.net/res057qw/APL_J/IversonAPL.htm), 1991 essay by K.E. Iverson (archived link)
6. Overview of J history (http://jsoftware.com/pipermail/general/2002-March/010962.html) by Roger Hui (19 March 2002)
7. J NuVoc Words (http://code.jsoftware.com/wiki/Vocabulary/Words)
8. Bussell, Brian; Taylor, Stephen (2006), "Software Development as a Collaborative Writing Project", Extreme programming and agile processes in software engineering, Oulu, Finland: Springer, pp. 21–31, ISBN 978-3-540-35094-1 Missing or empty `|title=` (help)
9. Holt, Alan (2007), *Network Performance Analysis: Using the J Programming Language*, Springer, ISBN 978-1-84628-822-7
10. Jsoftware's source download page (https://www.jsoftware.com/#/source)
11. Eric Iverson (1 March 2011). "J Source GPL" (http://thread.gmane.org/gmane.comp.lang.j.programming/20882). *J programming mailing list*.
12. openj (https://github.com/openj/core) on GitHub
13. Jsoftware's sourcing policy (https://www.jsoftware.com/#/source)
14. Chapter 25: Object-Oriented Programming (http://www.jsoftware.com/help/learning/25.htm)

External links

- Official website (http://www.jsoftware.com) – JSoftware, creators of J
- jsource (https://github.com/jsoftware/jsource) on GitHub – Repository
- Learning J (http://www.jsoftware.com/help/learning/contents.htm) – An Introduction to the J Programming Language by Roger Stokes

Retrieved from "https://en.wikipedia.org/w/index.php?title=J_(programming_language)&oldid=1009106835"