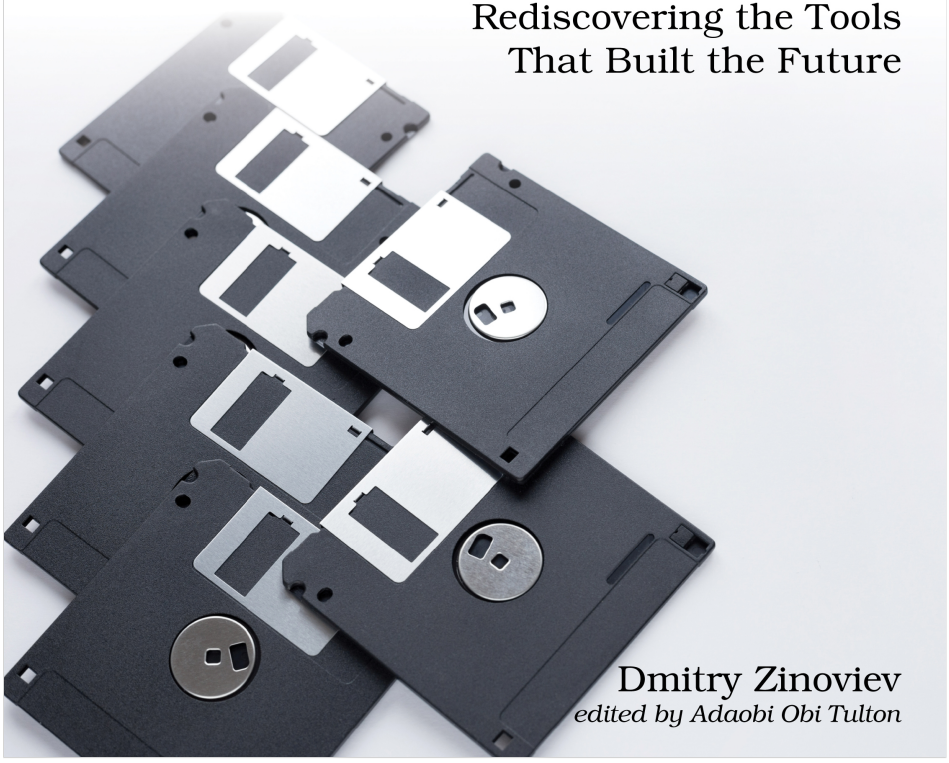
The Pragmatic Programmers

Seven Obscure Languages in Seven Weeks

Rediscovering the Tools That Built the Future



Dmitry Zinoviev
edited by Adaobi Obi Tulton

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

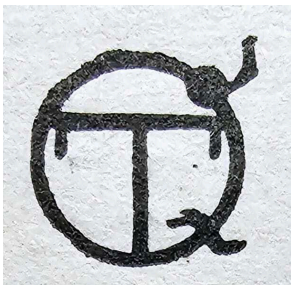
When it comes to massively parallel processing of multidimensional arrays—vectors, matrices, datacubes, and so on—nothing seems to beat NumPy, a numerical Python library (hence the name), and Matlab, the Matrix Laboratory (hence the name). However, this wasn't always the case. The first array processing language, APL, was designed much earlier.

The original version of APL dates back to 1962 when Kenneth Iverson, a Canadian computer scientist, introduced it as a form of algebraic and algorithmic notation in his book *A Programming Language [Ive62]*. APL became a powerful interactive problem-solving system, first optimized for legendary IBM OS/360 mainframes (under the name APL\360) and, later, for minis and personal computers. Unless you have a better alternative, you'll work with well-supported and well-documented GNU APL (apl).[1] But first, we must have a serious conversation about the APL character set and the need for a specialized keyboard.

Deciphering APL Character Set

If someone asks you what the only thing that differentiates APL from 99.9 percent of other programming languages is, answer without hesitation: it's APL's character set.

The APL founding fathers were of a solid mathematical background. They intended to create a programming language resembling familiar mathematical notation as much as possible. Ideally, the user could type a formula in an APL interpreter window and instantly execute it. That's how APL ended up with 65 special characters (in addition to the familiar alphanumerics, spaces, and punctuation), including ⊖, ⍞, ⍈, a "thumbnail" (⍝), and even a grotesque overlay of O, Q/U, and T (the end of input—see Performing Input and Output, on page ?):



1. https://www.gnu.org/software/apl/

The extended alphabet of the language permitted users to write concise expressions. For example, the following expression calculates the value of e≈2.718281828 through Taylor series expansion with 170 members—1++/1÷(!⍳170)—and I don't blame you if you fail to recognize the formula at first glance.

On the bright side, the value of π in APL is at your fingertips: ○1. The function ○ represents multiplication by π and understandably looks like a circle; though it would be even more natural for such a function to look like a half-circle ◠ or represent multiplication by 2π.

Modern physical computer keyboards don't show the extended APL characters. As an APL coder, you must use a virtual keyboard, buy a pricey specialized APL keyboard—from Dyalog,[2] for example—or install a secondary APL keyboard layout, not unlike a layout for a foreign language (see Activating the APL Keyboard Layout, on page 5). Well, APL *is* a foreign language.

To conclude, the complete APL character set consists of the ASCII alphanumeric characters (A through Z, a through z, and 0 through 9), ASCII punctuation, white spaces, and 65 or more special characters, making APL the black sheep of programming languages, as the sidebar on page 6 explains.

Some special characters went out of use as early as 1970, making the APL reader's life somewhat more manageable.

2. https://www.dyalog.com/

I hope I haven't scared you. You can study J or K instead—they're remote relatives of APL that use only standard ASCII characters—or continue with APL anyway.

J and K

The J programming language[a] is another baby of Kenneth Iverson. It appeared first in 1990. J inherits the compactness and expressiveness of APL but does away with any special characters. Sadly, along with losing the APL special characters, it also lost APL's charm.

The K programming language[b] is from 1993 and out of Morgan Stanley. (To be fully honest, K is a descendant of two more APL-style languages, A and A++.) K's purpose was to facilitate the migration of APL code from IBM mainframes to Sun workstations. K uses heavy operator overloading to make up for the absence of silly special characters. It's not clear to me if `10#{1+1.0%x}\1` in K is more readable than `1++/1÷(!170)` in APL.

a. https://www.jsoftware.com/help/learning/contents.htm
b. https://xpqz.github.io/kbook/Introduction.html

Activating the APL Keyboard Layout

As a Linux or macOS user, you can switch to the secondary APL keyboard layout with the program `setxkbmap`. The following command makes the combination `Right`-`Alt` a layout switch. Note the comma just in front of `dyalog`. No space is between them (the invisible "empty" variant before the comma refers to the `us` layout):

```
setxkbmap -layout us,apl -variant ,dyalog -option grp:switch
```

Press the combination to activate the secondary APL layout. Otherwise, the standard U.S. layout is used. The same program with different options removes the `Right`-`Alt` binding:

```
setxkbmap -layout us -option grp:switch
```

If you're a Windows user or none of the above worked, visit the Dyalog website[3] for more options.

This introduction to APL programming was longer than expected—blame the APL character set. You're ready to move on to the rest of the language, for it deserves it.

3. https://www.dyalog.com/apl-font-keyboard.htm

The Black Sheep of Programming Languages

Aside from APL, special (non-ASCII) characters can be found in PL/I (¬), Fortress (→, ⊂, ⊆, ∞), TI-BASIC (≤, ≠, ≥, √‾, →), Scala (←, ⇒), Haskell (::, ∀, ⇒, ↣), Agda (ℕ, ∀), and perhaps some other exotic languages. It's APL, however, that makes wild and unconstrained use of the special symbols.

In APL's defense, the first version of the American Standard Code for Information Interchange (ASCII) wasn't published until 1963, and before that, no standard character set existed. Any character, technically, was "special."

IBM devised its staple encoding, Extended Binary Coded Decimal Interchange Code (EBCDIC), only in 1963–64. At this point, we can only guess the original encoding of the APL symbols. Fortunately, with the advent of Unicode, the Tower of Babel of the character codes is once again uniting users and programmers instead of dividing them.

Looking at Data Types

Numbers can be integer and real, positive and negative, and here's the catch: APL strongly promotes the one-to-one correspondence between a symbol and its function. In most programming languages, the minus is used as a constituent of a negative literal expression (-5 is a negative 5) or a unary negation function (-X is the negation of X, not necessarily a negative number by itself). In APL, -X is the negation of X, but negative 5 is written as ¯5.

The original APL\360 doesn't support complex numbers.

A one-dimensional numeric array—a vector—is a sequence of scalars separated by one or more spaces. Notice that the APL code traditionally starts in the seventh column. The first six positions are reserved for the output and line numbers within function definitions (see Define and Call Functions, on page ?; Fortran has a similar arrangement). Also, when in the interactive mode, APL displays the value of the most recently entered expression:

```
      1 2.0 3E¯4 ¯5e¯6
1 2 0.0003 ¯0.000005
```

Oddly, you cannot directly define a one-element array (it would be indistinguishable from a scalar), but you can specify a two-element array and truncate it.

APL strings are enclosed in single quotation marks and cannot have line breaks. If a quotation mark is an element of a string, it's represented as two consecutive quotation marks (compare Starset on page ?). A number included in a vector of strings remains a number: string vectors don't have

to be homogeneous. By the way, the symbol ⍝, a thumbnail, denotes a comment throughout the line.

```
      'I am a string' ⍝ 13 elements: 'I', ' ', 'a', 'm', ' ', ...
I am a string
      'Me, ''too''' ⍝ 9 elements: 'M', 'e', ' ', '''', ...
Me, 'too'
      'I' 'am' 'a' 'vector' 'of' 'strings' ⍝ 6 elements: 'I', 'am', ...
I am a vector of strings
      'Me,' 2 ⍝ 2 elements: 'Me', 2
Me, 2
```

You can assign variable names to scalars, vectors, and higher-dimensional arrays (the operation known in APL literature as *specification* and *respecification*). A variable name is any combination of letters, underlined letters (obsolete), digits, an underscore, Δ, or ∆ (also obsolete). However, it cannot begin with a digit, SΔ, or TΔ (the latter two are reserved for debugging). Variable names are case-sensitive. The assignment function is the left arrow ←.

```
      dataSize←32
      dataSize
32
      DATASIZE
VALUE ERROR
      DATASIZE
      ^
      Data←1 2 3
      Data
1 2 3
```

All APL variables, unless declared local (see Creating User-Defined Functions, on page ?), are global and available to all functions. Once "specified," a global variable becomes a part of a *workspace*—a container for variables, functions, and other objects that the user creates and interacts with during a session (user-defined functions are also stored in workspaces). Workspaces are persistent: they can be saved and later restored. Variables and functions can be listed and erased. Once erased from a workspace, a variable becomes unavailable (compare operator del in Python):

```
      )VARS
Data    dataSize
      )ERASE ata ⍝ Intentional mistake
NOT ERASED: ata
      )ERASE Data
      )VARS
dataSize
      )CLEAR
CLEAR WS
```

Commands whose names begin with a right parenthesis (such as)CLEAR) are system commands. Unlike variable names, they are case-insensitive, but we'll type them in uppercase to emphasize their significance.

Congratulations on your first APL experience! Have some rest, but remember to log off:

```
      )OFF
Goodbye.
Session duration: 59.4107 seconds
```

In the era of the mighty ancient mainframes and remote terminals, a failure to log off might have resulted in a hefty bill!