APL Style: Patterns/Anti-patterns

Disclaimer: *This talk includes hyperbole, moralistic rambling, and optimistic speculation. While it is nice to live in a world of black and white --- I can't really help myself in this presentation --- programming is more artistic than it is dogmatic. There's something to always remember...*

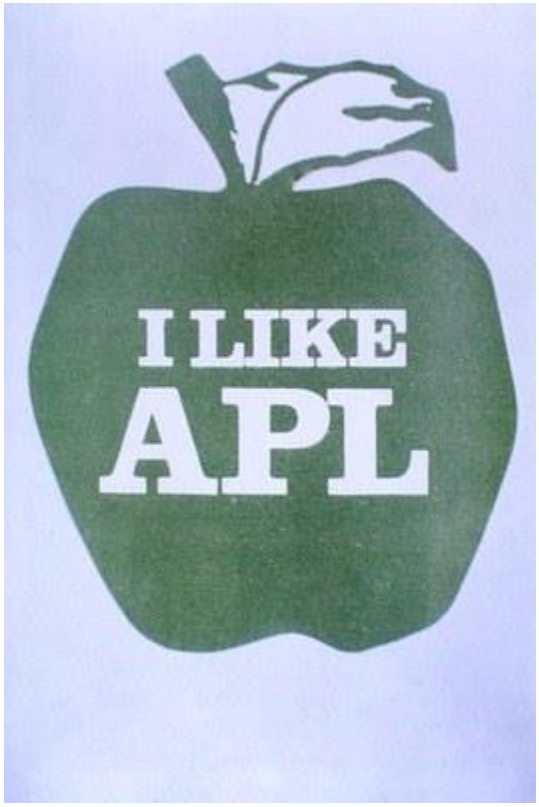
Don't be fanatical, enjoy yourself, and have fun!

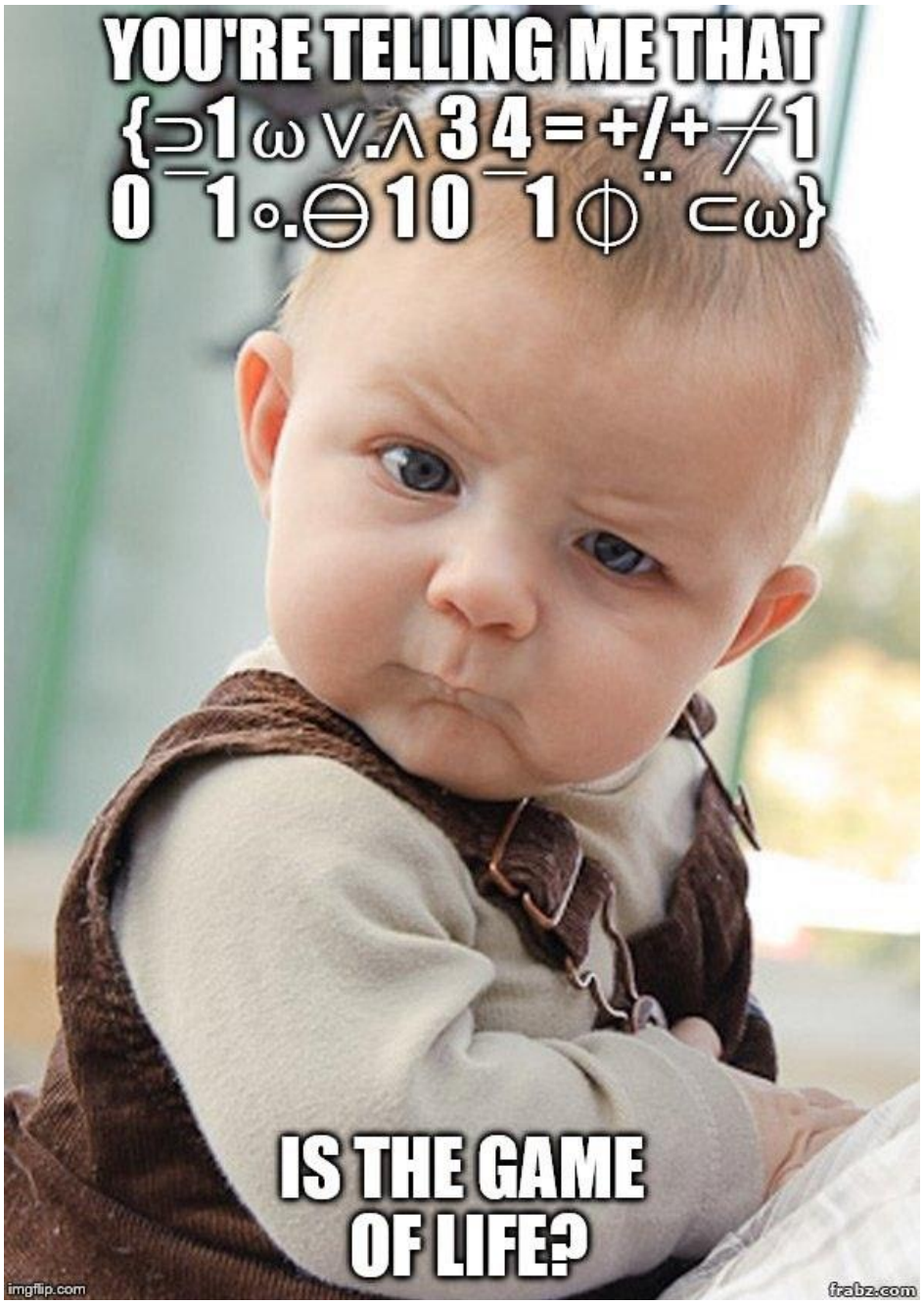
APL is an infamous language, both loved and hated.

It has had more success with non-computer scientists than with seasoned programmers.

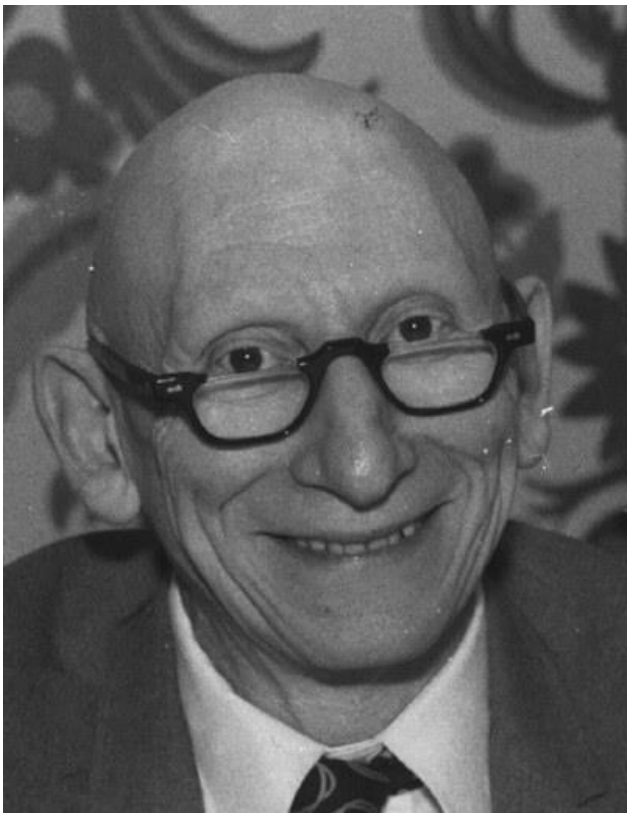
Programmers: "Never in my life."

Users: "Can't imagine life without it!"





Why do Computer Scientists have such trouble with APL?



---

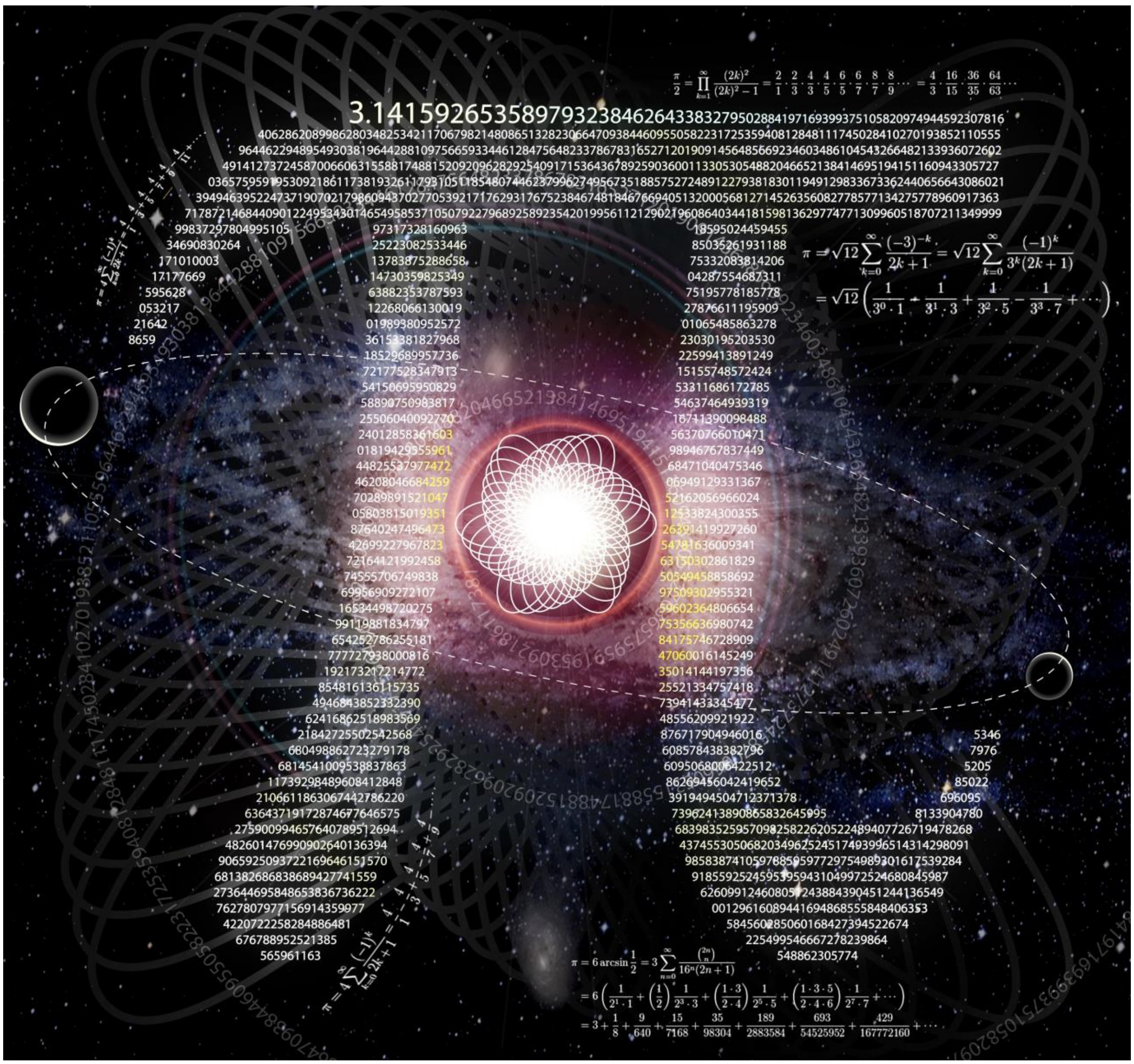
*The virtues of APL that strike the programmer most sharply are its **terseness** — complicated acts can be described briefly, its **flexibility** — there are a large number of ways to state even moderately complicated tasks (the language provides choices that match divergent views of algorithm construction), and its **composability** — there is the possibility to construct sentences — one-liners as they are commonly called — that approach in the flow of phrase organization, sequencing and imbedding, the **artistic possibilities** achievable in **natural language prose**.*

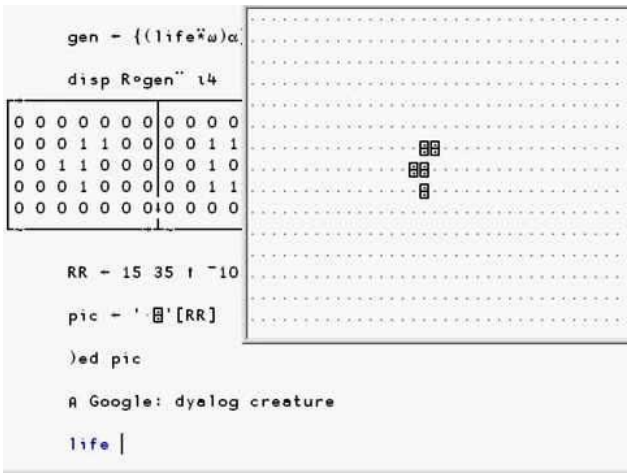
-- Alan Perlis on APL One-liners and Lyrical Programming

But he acknowledges that APL is harder to learn than some more traditional languages. Whatever the case may be, APL has a reputation for a difficult, inscrutable language from the outside.

Perhaps though, after understanding why APL is what it is, and how it truly distinguishes itself from most every other language, we can instead see APL as a **powerful tool of thought** that is **worth the initial learning experience**.

So you want to learn APL?





Most people get to this Game of Life YouTube video, watch it with a degree of awe and appreciation, and then think one of two things:

1. That's super cool, but I'm a fool, and never could write code like that.
2. What painful horror! What disarray! No types, no ADTs, no way!

There is an undeniable "learning wall" upon which eager students with prior programming experience dash their skulls and struggle to climb. Why is this? And why does this appear to be a much greater wall for programmers with traditional, extant computing experience, when many students with little to no programming experience actually find the experience somewhat more pleasant than traditional languages?

The Birth of APL

Kenneth Iverson is the Father of APL. His notation and subsequent book, *A Programming Language*, spawned a family of array languages all centered around a common theme. You might think this is the array datatype, since this is what all these languages have in common. However, you would be wrong. The secret to understanding the difficulty of APL to the outsider begins with an understanding of the design principles that birthed APL.

Iverson describes these design principles in his Turing Award Lecture, *Notation as a Tool of Thought*. He quotes the following great minds to set the stage:

By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and in effect increases the mental power of the race. -- A. N. Whitehead

The quantity of meaning compressed into small space by algebraic signs, is another circumstance that facilitates the reasonings we are accustomed to carry on by their aid. -- Charles Babbage

You can see then, that the focus of Iverson's work is distinctly human and experiential, as opposed to mechanical and semantic.

In his lecture, Iverson lays out the following five principles of good language design:

- *Ease of expressing constructs arising in problems*
- *Suggestivity*
- *Ability to subordinate detail*
- *Economy*
- *Amenability to formal proofs*

Ease of Expressing Constructs

Ease of Expressing Constructs. APL was designed as a method for directly expressing solutions to problems. These *direct expressions* are solutions to problems that are written in the native *primitive ideas* of APL.

Contrast this against more common languages, in which the first thing you do is either look for the appropriate library for solving the problem, or you build up the appropriate abstractions to then solve the problem using this mini-infrastructure that you have built up.

Iverson expressed this idea by saying: "*If it is to be effective as a tool of thought, a notation must allow convenient expression not only of notions arising directly from a problem, but also of those arising in subsequent analysis, generalization, and specialization.*"

Suggestivity

Suggestivity. Iverson was big on this idea, and it's one that does not readily appear in other language design motifs. He says, "*A notation will be said to be suggestive if the forms of the expressions arising in one set of problems suggest related expressions which find application in other problems.*" An example of such suggestivity is the similarity of form:

+/mρn ←→ n×m

×/mρn ←→ n*m

Through further examples, he makes the point that *"[p]art of the suggestive power of a language resides in the ability to represent identities in brief, general, and easily remembered forms.*"

Ability to Subordinate Detail

Ability to Subordinate Detail. Iverson says, "As Babbage remarked in the passage cited by Cajori, brevity facilitates reasoning. Brevity is achieved by subordinating detail…."

This should draw a sharp contrast between **Subordination of Detail** and *Abstraction* as the term is commonly used in Computer Science. Specifically, Iverson's notion of subordination is the **elimination of *notational obligations*** through the use of generalization, systematic extension, and implicit guarantees, in contrast to the usual notion of abstraction as the means by which "API" barriers may be introduced to

implement conceptual frameworks that suppress underlying implementation considerations in order to allow a “black box” reasoning at a different, non-native abstraction level.

Economy

---

*Iverson has much to say about **Economy**:*

The utility of a language as a tool of thought increases with the range of topics it can treat, but decreases with the amount of vocabulary and the complexity of grammatical rules which the user must keep in mind. Economy of notation is therefore important.

Economy requires that a large number of ideas be expressible in terms of a relatively small vocabulary. A fundamental scheme for achieving this is the introduction of grammatical rules by which meaningful phrases and sentences can be constructed by combining elements of the vocabulary.

To this I would add an implication that follows from the above. Specifically, the **composability** of ideas is important, that is, the grammatical rules, but also the **scalability** of the general vocabulary available to the programmer. Today we know that there are many languages with full generality, and exceptionally small vocabularies, and simple rules. These are nonetheless not economical languages, because the language does not scale as a method of **directly** addressing or treating a wide enough range of topics. The economy of a language in addressing a given topic must be evaluated only *after the necessary intermediate abstractions, vocabulary, grammar, and syntax have been layered on top of the core language* in order to solve the given problem.

Thus, economy is not just the number of core forms and syntax you have in your language, but the **number of additional forms, both syntactic and semantic, that arise while solving a wide range of topics**. Every new name/binding that you introduce in order to address a given topic must be considered as part of the vocabulary, and not just the core vocabulary from which it was constructed. Thus, a language like Scheme or C may be very small in core forms, but to evaluate their economy, we must examine how real world problems and topics are addressed using these languages. In doing so, the number of libraries, new forms, syntactic extensions, and APIs developed to make the language generally able to address a host of topics all must be considered in judging the economy of that language.

Amenability to Formal Proofs

Amenability to Formal Proofs. At the time Iverson designed APL, large scale mechanical proofs were not even on the radar of practical possibility. *Physical malleability* and work with the expressions by hand was considered highly valuable. Today, this is an undervalued trait, but one that you find consistently in the APL community.

More than any other community with which I am aware, the formal derivation and manipulation of programs by hand as a method for reasoning and discussing code forms an integral part of the human to human communication in the APL sphere. This is not surprising given that the language designed from

the outset as a **blackboard language** for use before automatic program manipulation would have been considered the norm.

The side-effects of having such a language that is so easily “symbol pushed” can be stated in terms of the facility with which one may play with, explore, and generate code and derivations in rapid order in any given media **without needing sophisticated tool support**. This makes it uniquely well suited to human to human programming, where ideas like *Direct Development* have found unique expressions inside of the developer space because of this feature.

Design Patterns/Anti-patterns Dichotomy

The above principles give a hint, but fail to tell the whole story. What really causes the difficulties is the natural consequences of a language designed around the above principles and a community that leverages those principles to good effect.

Good APL follows a set of best practices that **directly contradict and conflict with traditional programming wisdom**. Indeed, APL design patterns appear as **Anti-patterns** in most other programming languages.

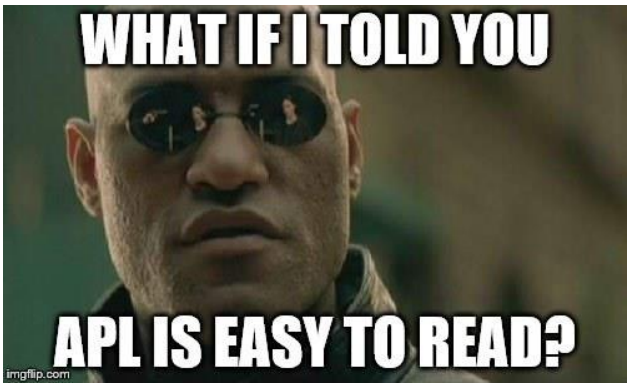
It is this dichotomy of best practices that can give us an insight into why Computer Scientists or those trained in traditional programming methods often find APL jarring and difficult, while those with no prior background can fall in love with the language as a method for getting real work done.

The Computer Scientist has been trained over an intense and rigorous career to think in a specific way, and to apply a specific suite of practices into the expression of code as a craft. They have a highly refined sense of aesthetics developed over a long period of time in order to more efficiently and intuitively identify "good code" from "bad code."

Before such a palate, APL leaves a sour taste: the flavors for which APL is famous directly contradict the prescribed norms of traditional programming methods.

Because **value judgments** about **good** and **bad** code reach deeply into the psyche of the traditional programmer, no wonder the programmer struggles with seeing such code, despite the positive record with serious users.

Writing APL code the same way you write traditional software only defeats both sanity and efficiency.



Free your mind and write beautiful APL code: that black hole over the edge of the cliff is only in your mind.

The following patterns contrast traditional software engineering practice with the patterns of practice that appear in well refined and tuned APL code.

---

*Not all APL code looks like this, but when APLers work on "streamlining" and refactoring their code, they tend to follow these principles, in **direct contrast to the traditional patterns of design**.*

Brevity over Verbosity

---

*Brevity plays a key role in Iverson's language design principles. While not surprising, this is **not terseness for code golf's sake.***

Less code is easier than more code, in more ways than one.

+ / ÷ ≠

Brevity supports the design goals and contributes to the ease of retention and comprehension of code.

Common practice assumes readability is a function of comfort to the uninitiated, and that code at scale cannot be fully retained, so verbosity is presumed to be more self-documenting and easier to understand for all experience levels, since **no one is more lost than the original author after a year of not looking at the code.**

APL programmers often optimize code for **malleability** and **expect to understand their code as a whole**.

Contrary to prevailing wisdom, they expect significant programs to have to relatively small solutions, easily memorized if so desired, which favors concision to see more of the code.

Brief code is readable both at the small level (easier to manipulate and understand as a whole) and during integration, where the more concise code is taken to be **easier to integrate into a larger code base and not lose sight of the big picture**.

Macro over Micro

Common practice encourages decomposing a problem into very small components, and most programming languages emphasize a clear picture of a small piece of code in isolation — ideally enforced isolation.

APL emphasizes finding ways of **expressing solutions around the macro-view of the problem**, and it attempts to eliminate as much as possible the small details or the decomposition of a problem into parts that are viewed separately or independently of the whole.

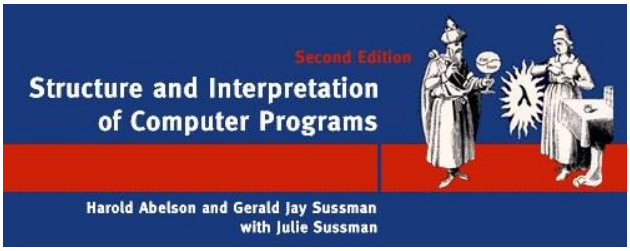
```
(⍺⍺ ⍵) ⍵⍵⌸ ⍵
```

Transparency over Abstraction

The psychological profiling [of a programmer] is mostly the ability to shift levels of abstraction, from low level to high level. To see something in the small and to see something in the large.

-- Donald Knuth, Dr. Dobb's Journal Interview with Jack Woehr

---



Abstraction Considered Harmful

Current design practices obscure both the small and large pictures in favor of hard abstraction barriers.

Pascal is for building pyramids.... Lisp is for building organisms.... It is better to have 100 functions operate on one data structure than to have 10 functions operate on 10 data structures. As a result the

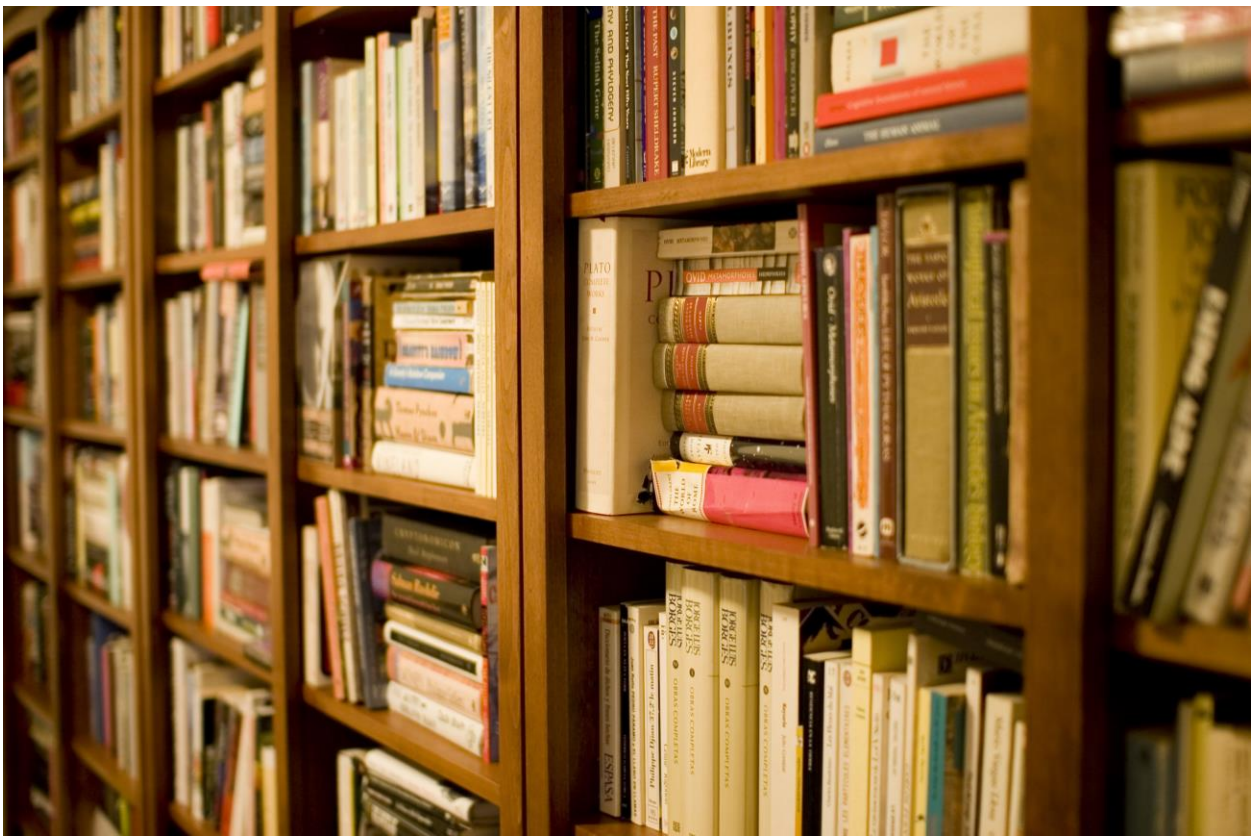
pyramid must stand unchanged for a millennium; the organism must evolve or perish. -- Structure and Interpretation of Computer Programs

Abstraction, in the current sense, hides, rather than subordinates, detail. In modern practice, the best abstractions are the most impenetrable. This intentionally obfuscates the main ideas and the root ideas.

Good APL programs embrace transparency of expression, **eschewing a multitude of idiosyncratic abstractions in favor of directness of solution** as a weapon against complexity.

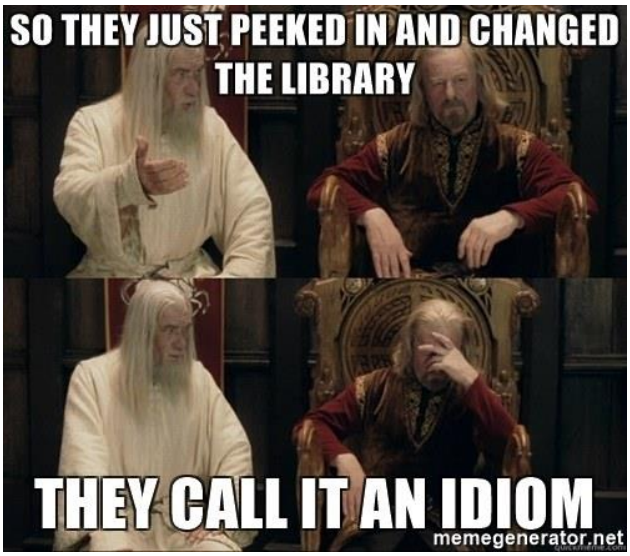
Two abstractions are considered vitally important to the APL programmer: the vocabulary of the **native problem domain** and the **core linguistic primitives** (the APL language) that efficiently spans multiple domains.

Idioms over Libraries



I also must confess to a strong bias against the fashion for reusable code. To me, "re-editable code" is much, much better than an untouchable black box or toolkit. I could go on and on about this. If you're totally convinced that reusable code is wonderful, I probably won't be able to sway you anyway, but you'll never convince me that reusable code isn't mostly a menace.

-- Donald Knuth, Interview with Andrew Binstock



Libraries create hard barriers in the abstraction landscape. Like Knuth's view on black box libraries, APLer's generally tend to favor the use of idiomatic implementations of common actions, rather than using library calls, whenever possible.

Benefit #1: Idioms enable editable code where libraries inhibit it.

Benefit #2: Idioms simplify the name space, because idioms do not require names. Thus, the implementation of an idea, expressed as code, often serves as the symbol/name for the functionality itself. Names can be reserved for more important tasks.

What about idioms in other languages?

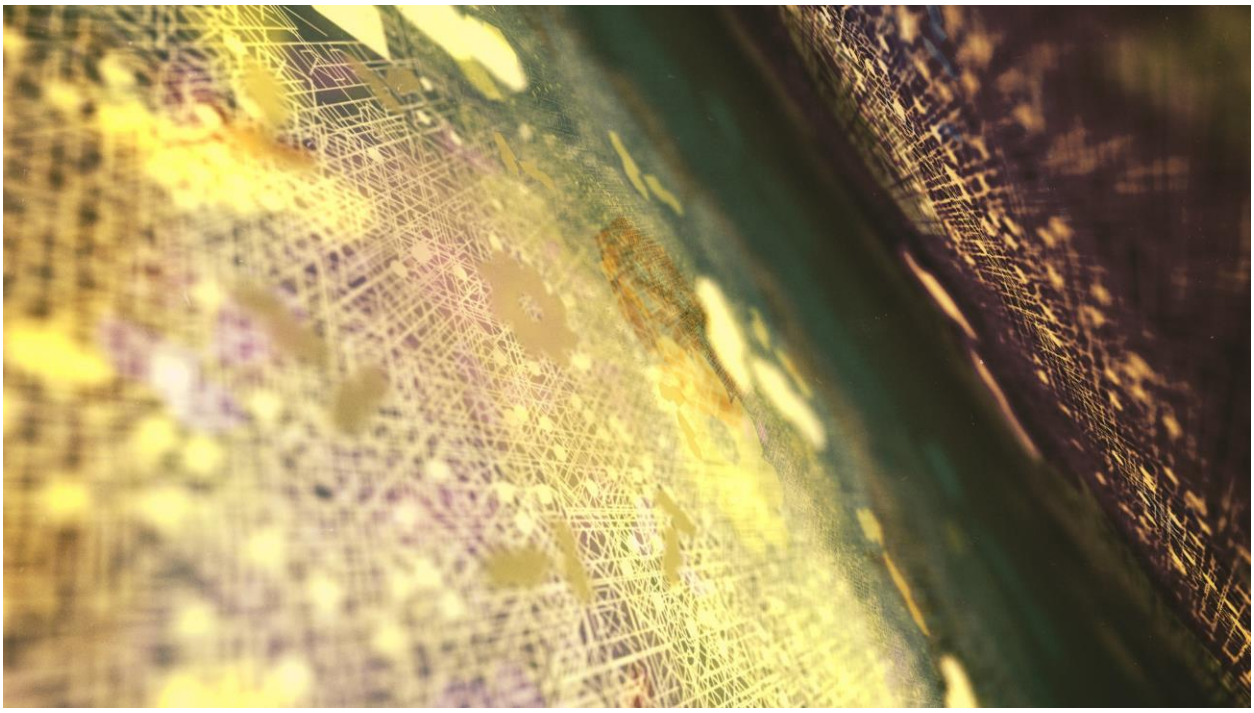
They exist, but rarely as "idioms." Trivial programs may be idioms, but complex operations usually require elevation to "design pattern." However, because of the verbosity of design patterns, they rarely help with the naming situation, and they fail to scale as readily when used pervasively as do idioms.

However, design patterns are used as "re-editable" code templates to good effect in other languages. They often exist at the top level, rather than idioms, which are lower-level elements. They do not replace libraries in other languages.

A single line or two of APL could have three or four idiomatic phrases in it.

```
∧.=
∧.(=∨0=⊣)
```

Data over Control Flow

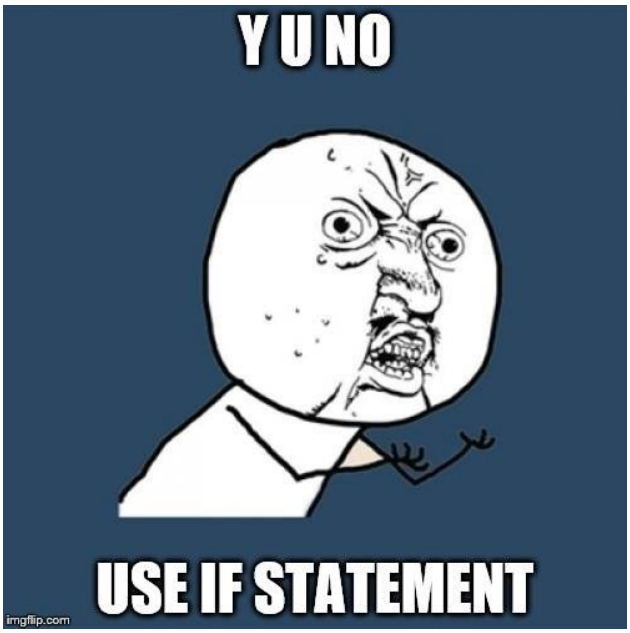


Most programming languages emphasize the use of control flow for managing programming logic.

Logic in APL is often most efficiently expressed in a data representation that is then processed using array operations.

This contributes to concision of code and simplicity of expression. Simplifying the control flow simplifies the syntax.

This is similar to a pervasive use of data flow models of programming at the lowest levels of code composition.

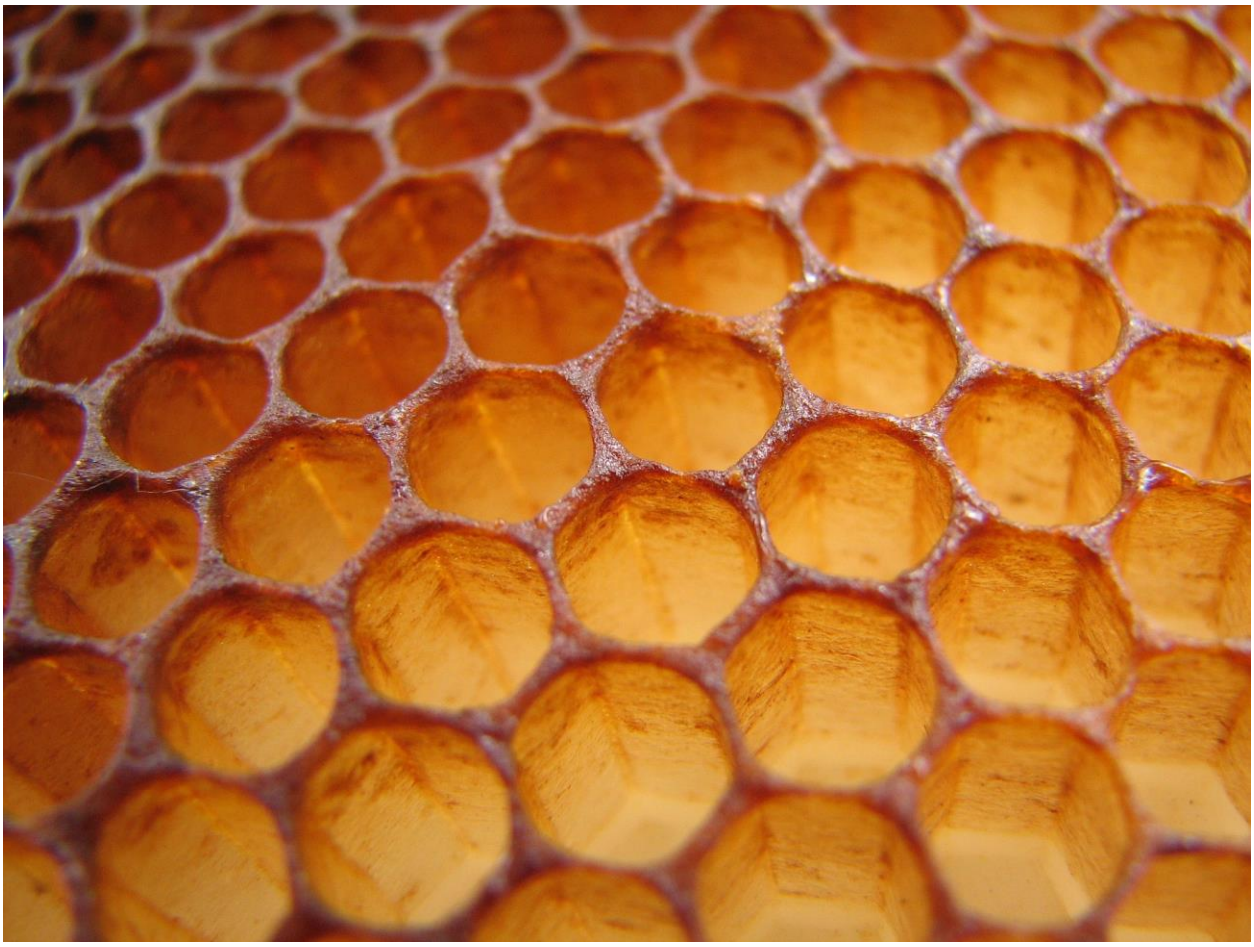


Features that contribute to data logic:

- Trains/Tacit Programming
- Array-at-a-time primitives
- Efficient native support for Boolean arrays
- Computability of shapes, values, and meta-information about arrays

```
⊢(≠⍤⍤)(+\1=d)(~⊣∈⊣(/⍤⍤)(1=d)∧(~'b'∈⍤⍤k)∧Om∨Fm)⊢
```

Structure over Names



As we increase the number of names, we must increase the length to improve comprehension. As we increase their length and decrease the length of implementation, the implementation itself becomes the shortest usable name, and simultaneously, the unequivocally most descriptive.

```java
/**
 * Returns all prime numbers within a given range, in order.
 *
 * @param lowerBound The lower bound (exclusive) of the range.
 * @param upperBound The upper bound (exclusive) of the range.
 * @return A List of the prime numbers that are strictly between the
 * given parameters.  This list is in ascending order.  The returned
 * value is never null; if no prime numbers exist within the given
 * range, then an empty list is returned.
 */
public List<Long> primeNumbersBetween(long lowerBound, long upperBound);
```

```
(2=+⌿0=X∘.|X)/X←⍳N
```

By leveraging the concision of the language and the lyrical nature of APL **idiomatic phrases**, the regularity of structure and the use of pattern recognition (Iverson's Suggestivity at work) can **replace a large number of traditional name-based abstractions**.

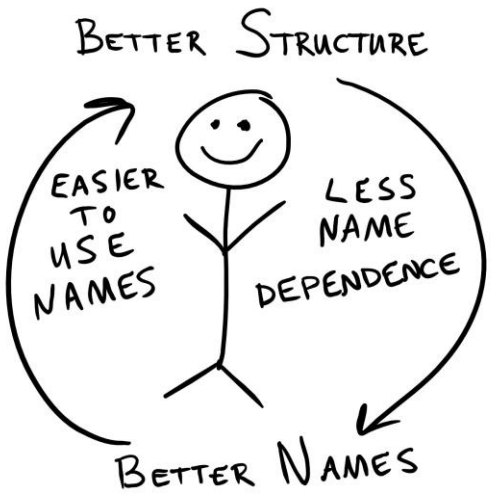
This in turn reduces the total vocabulary, simplifying the namespace.

This allows us to reduce the length of names, which increases brevity.

This further enables the use of structure to communicate over names, and the cycle continues.

This permits us to **utilize names more carefully**, and with more consideration to their effect on the overall picture.

We can begin to reserve names for only the most important elements of our structure, and we have a great chance of keeping the names within the linguistic domain of the end-user, instead of a number of intermediate domains that may appear between the core APL abstractions and the domain being addressed.

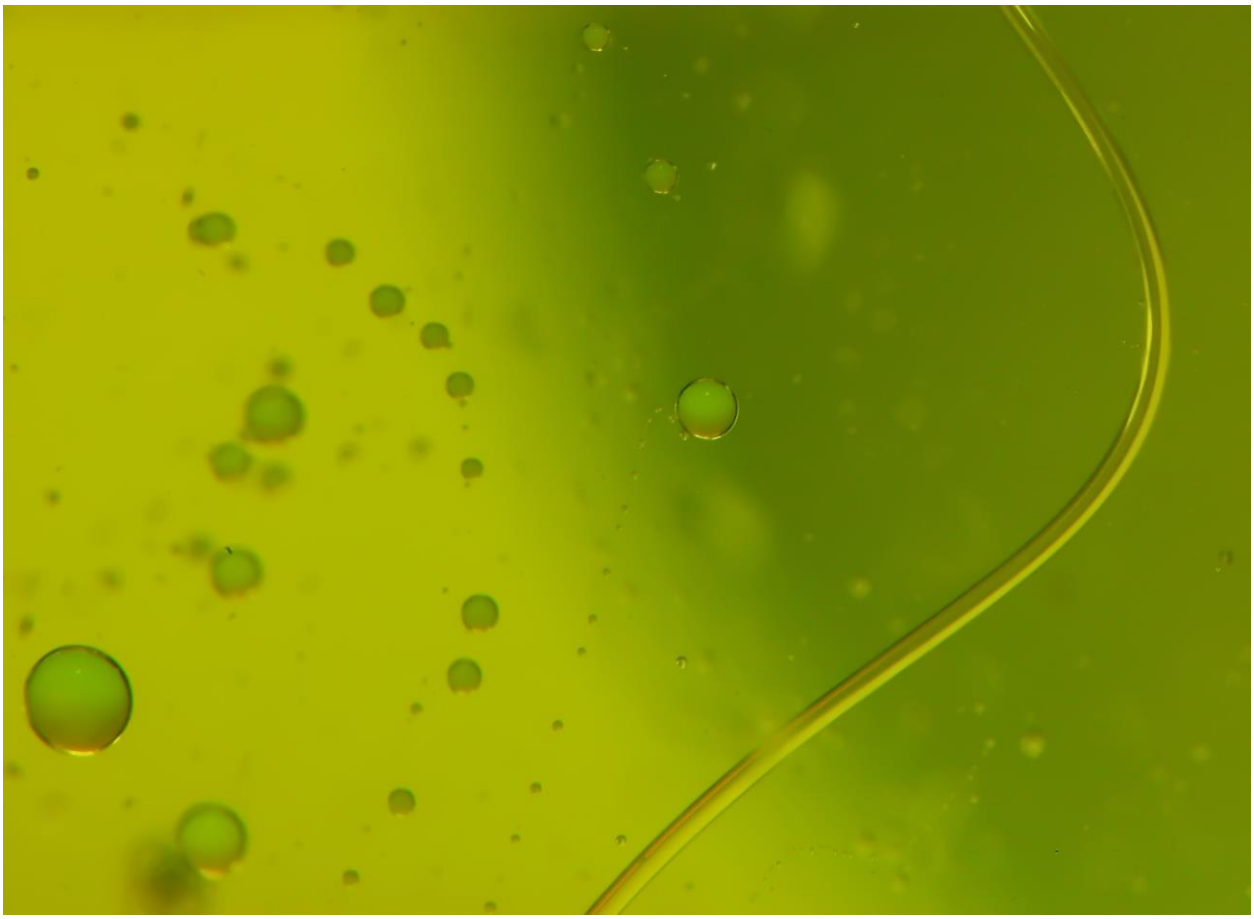


When structure is able to replace names, there is no question of clarity: the iconographic symbol for a given functionality fully articulates all semantic questions, including edge cases, domain, and performance; it admits minor adjustments without loss of clarity.

What's in a name? Potentially everything.

The interplay between name and structure is not binary, and one must consider questions of flow, transition, domain, audience, and so forth.

Implicit over Explicit

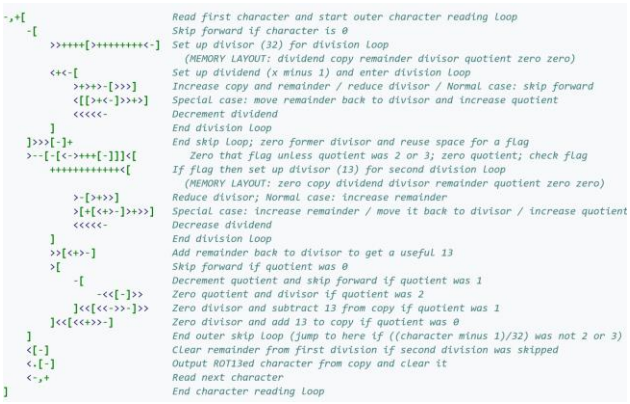


Subordination of Detail!

APL's notation is itself very "implicit." However, APL's lack of explicit "encoding" boundaries such as datatype abstractions, static types, or robust module designs leads to a huge strength: **suggestive generality.**

That is, good APL code often applies **beyond the originally intended domain**, without the explicit intent of the author.

By **avoiding the over-specification** of intention and detail, leaving much of it implied, code **more easily trends towards general usefulness**.



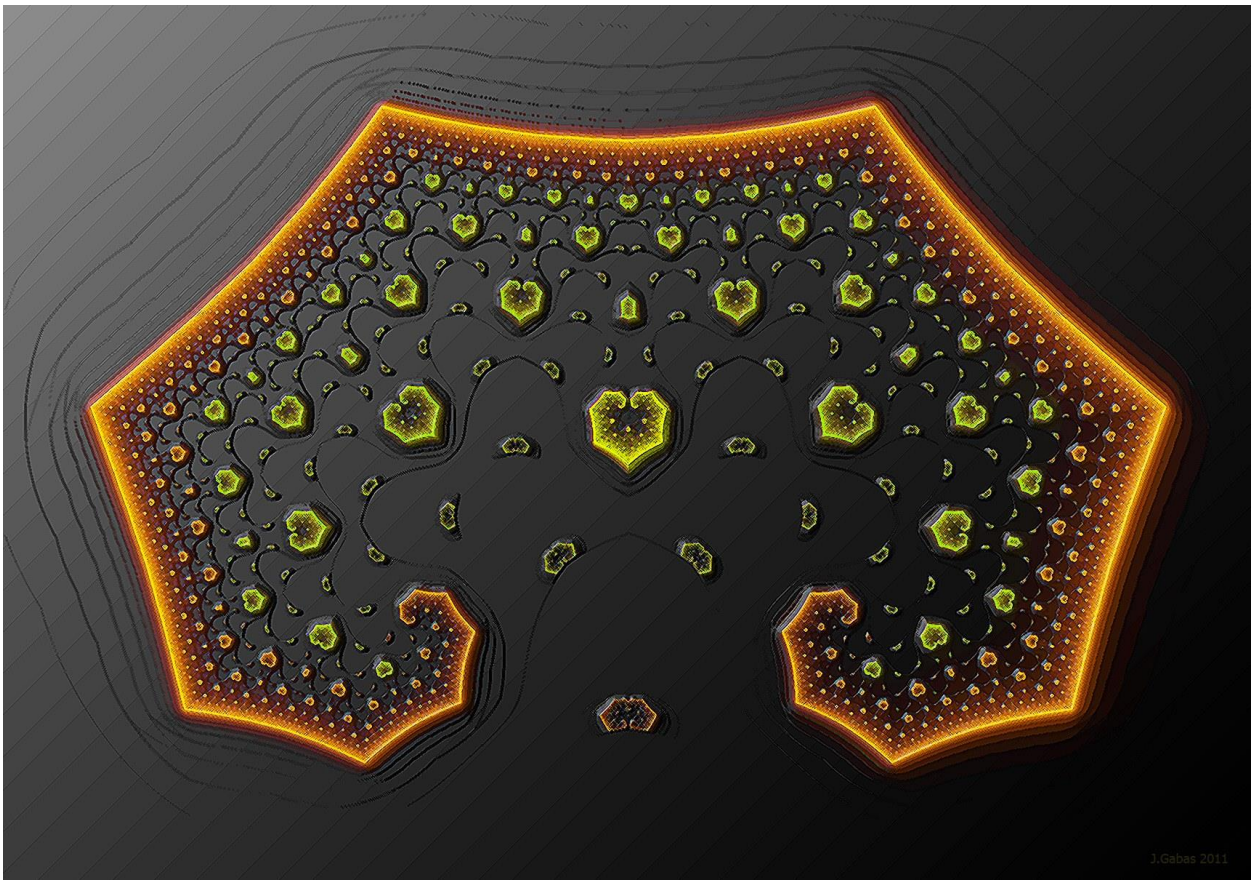
```
-,+[                         Read first character and start outer character reading loop
   -[                        Skip forward if character is 0
      >>++++[>++++++++<-]    Set up divisor (32) for division loop
                               (MEMORY LAYOUT: dividend copy remainder divisor quotient zero zero)
      <+<-[                  Set up dividend (x minus 1) and enter division loop
          >+>+>-[>>>]        Increase copy and remainder / reduce divisor / Normal case: skip forward
          <[[>+<-]>>+>]      Special case: move remainder back to divisor and increase quotient
          <<<<<-             Decrement dividend
      ]                      End division loop
   ]>>>[-]+                  End skip loop; zero former divisor and reuse space for a flag
   >--[-[<->+++[-]]]<[         Zero that flag unless quotient was 2 or 3; zero quotient; check flag
      ++++++++++++<[         If flag then set up divisor (13) for second division loop
                               (MEMORY LAYOUT: zero copy dividend divisor remainder quotient zero zero)
          >-[>+>>]           Reduce divisor; Normal case: increase remainder
          >[+[<+>-]>+>>]     Special case: increase remainder / move it back to divisor / increase quotient
          <<<<<-             Decrease dividend
      ]                      End division loop
      >>[<+>-]               Add remainder back to divisor to get a useful 13
      >[                     Skip forward if quotient was 0
          -[                 Decrement quotient and skip forward if quotient was 1
              -<<[-]>>       Zero quotient and divisor if quotient was 2
          ]<<[<<->>-]>>      Zero divisor and subtract 13 from copy if quotient was 1
      ]<<[<<+>>-]            Zero divisor and add 13 to copy if quotient was 0
   ]                         End outer skip loop (jump to here if ((character minus 1)/32) was not 2 or 3)
   <[-]                      Clear remainder from first division if second division was skipped
   <.[-]                     Output ROT13ed character from copy and clear it
   <-,+                      Read next character
]                            End character reading loop
```

By utilizing generic computations that happen to apply to a given problem when fed appropriately **encoded data**, you can shrink the size of code, extend its usefulness, and avoid issues around the edge cases.

Care should be exercised to ensure that the source code still elevates the core computation directly, and makes implicit the details. *A turing machine written in APL is implicit, but not helpfully so.*

Avoid Turing Tar Pits and DSL explosion by utilizing sub-turing complete expressions that minimally restrict the type of inputs and maximize the salient computation.

Syntax over Semantics



Good APL emphasize the human interactive experience with the code. Bad APL emphasizes mechanical control over human accessibility and freedom of expression.

Robust semantic guarantees are the bread and butter of formal academic Computer Science.

This attitude has spread to the industrial complex by the success of functional programming such as Lisp, Scheme, and Haskell; it has also spread through the Software Engineering community and the tantalizing promises of Object-Oriented Programming.

The **conventional wisdom** (born out by the rapid ability of seasoned programmers to learn, read, and understand new languages) says that the **syntax of a language matters little, but its semantic power makes or breaks it**.

In fact, today's most popular languages are better thought of as a smorgasbord of dialects centered around a few semantically cohesive continents of thought.

Paraphrasing a colleague: "It seems like there's this secret society of Computer Science that broke off from the mainstream early on and has existed on its own secretly for decades."

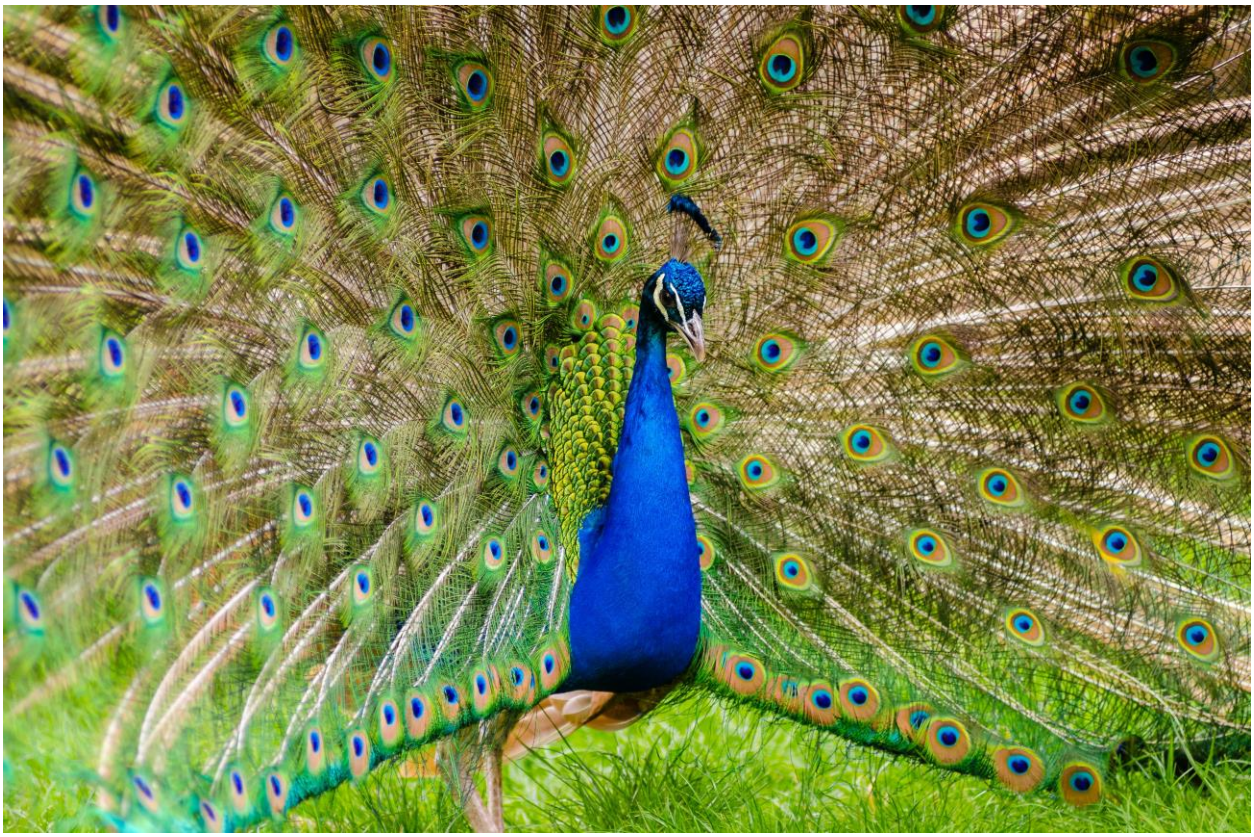
APL inventions like *Direct Development* and *Notational Thinking* contrast against analogous concepts of *Formal Specification* and *Type Directed Programming* in traditional Computer Science primarily by the **APL focus on the syntactic feeling and human interactivity with the symbolic representation itself** compared to the **traditional focus on semantic clarity and mechanically constrained derivation of code**.

There is a strong visual and spatial element to program development in APL; normal coding relegates these issues to matters of presentation and readability after development or as a side effect of automatic tooling.

An APL program that solves the problem with simpler semantics and neater expressions is generally preferred over code that is more semantically robust (defensive coding, error messages, abstractive boundaries, data hiding, ⍰.)

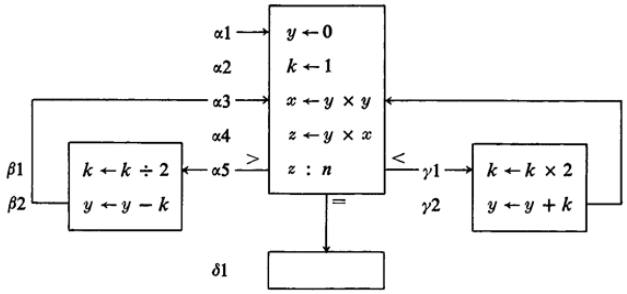
```
_s←{0<⊃c a e r←z←α αα ω:z ⋄ 0<⊃c2 a2 e r←z←e ωω r:z ⋄ (c⌈c2)(a,a2) e r}
_o←{0≥⊃c a e r←z←α αα ω:z ⋄ 0≥⊃c a e r2←z←α ωω ω:z ⋄ c a e(r↑⍨-⌊/≠⍤r r2)}
_any←{α(αα _s ∇ _o _yes)ω}
_some←{α(αα _s (αα _any))ω}
_opt←{α(αα _o _yes)ω}
_yes←{0 θ α ω}
_t←{0<⊃c a e r←α αα ω:c a e r ⋄ e ωω a:c a e r ⋄ 2 θ α ω}
_set←{(0≠≠ω)∧(⊃ω)∈αα:0(,⊃ω)α(1↓ω) ⋄ 2 θ α ω}
_tk←{((≠,αα)↑ω)≡,αα:0(⊂,αα)α((≠,αα)↓ω) ⋄ 2 θ α ω}
_as←{0<⊃c a e r←α αα ω:c a e r ⋄ c (,⊂ωω a) e r}
_ign←{c a e r←α αα ω ⋄ c θ e r}
_env←{0<⊃c a e r←p←α αα ω:p ⋄ c a (e ωω a) r}
_peek←{0<p←⊃α αα ω:p ⋄ 0 θ α ω}
_noenv←{0<⊃c a e r←p←α αα ω:p ⋄ c a α r}
_then←{0<⊃c a e r←p←α αα ω:p ⋄ 0<⊃c a e _←p←e (ωω _s eot) a:p ⋄ c a e r}
_eat←{0=≠ω:2 θ α ω ⋄ 0 (αα↑ω) α (αα↓ω)}
_not←{0<⊃c a e r←α αα ω:0 a α ω ⋄ 2 a α ω}
```

APL Styles



```
box←{                              ⍝ Box the simple text array ⍵.

    (⎕IO ⎕ML)←1 3 ⋄ α←⍬ ⍬ 0 ⋄ ar←{⍵,(⍴⍵)↓⍬ ⍬ 0}{2>≡⍵:,⊂,⍵ ⋄ ⍵}α  ⍝ controls

    ch←{⍵:'+++++++++-|+' ⋄ '┌┐└┘┬┤├┴┼|+'}1=3⊃ar             ⍝ char set
    z←,[⍳⍴⍴⍵],[0.1]⍵ ⋄ rh←⊃z                               ⍝ matricise
                                                          ⍝ simple boxing? ↓
    0∊⍴⊂2↑ar:{q←ch[9]⍪(ch[10],⍵,10⊃ch)⍪9⊃ch ⋄ q[1,⍳⍴q;1,2⊃⍴q]←2 2⍴ch ⋄ q}z

    (r c)←rh{⍺a{(⍵∊0,⍳⍺)/⍵}⍵,(~¯1∊⍵)/0,⍺}⍨2↑ar             ⍝ rows and columns
    (rw cl)←rh{{⍵[⍋⍵]}⍵⍳0,⍺}⍨r c

    (~(0,2⊃rh)∊c){                                        ⍝ draw left/right?
        (⍳⍺)↓[2](-2⊃⍺)↓[2]⍵[;⍋(⍳2⊃rh),cl]                  ⍝ rearrange columns
    }(~(0,1⊃rh)∊r){                                       ⍝ draw top/bottom?
        (⍳⍺)↓[1](-2⊃⍺)↓[1]⍵[⍋(⍳1⊃rh),rw;]                  ⍝ rearrange rows
    }{
        (h w)←(⍴rw),⍴cl ⋄ q←h⍴⍵⍴11⊃ch                      ⍝ size; special,
        hz←(h,2⊃rh)⍴9⊃ch                                  ⍝  horizontal and
        vr←(rh[1],w)⍴10⊃ch                                ⍝  vertical lines
        v/0∊⍨⍤1⊢rw cl:(ω⍪hz),vr⍪q                         ⍝ one direction only?
        q[1;]←5⊃ch ⋄ q[;w]←6⊃ch ⋄ q[;1]←7⊃ch ⋄ q[h;]←8⊃ch  ⍝ end marks
        q[1,h;1,w]←2 2⍴ch ⋄ (ω⍪hz),vr⍪q                   ⍝ corners, add parts
    }z
}
```

1 - Didactic dfns



2 - Box Diagram APL

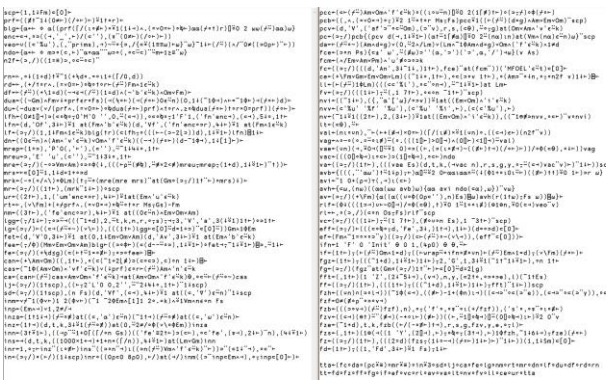
```
at←{⍵+(⍴⍵)↑(-⍺⍺)↑⍺}
avl←{(⍳⊃⍴⍵)~⍵×⊃⍺⌷⍺⍺}
box←{⍵≠⍵/⍵ ⍴⍳⍵*2}
cmap←{⊂[⍳⍴⍴⍵]1⍷¨⍵∘.=⍵}
emt←{(,⍵=0)/,⍳⍴⍵}
pvec←{(⍺(⍺⍺ avl)⍵)(⍺ at)¨⊂⍵}
pvex←{⊃,/⍺∘(⍺⍺ pvec)¨⍵}
rcb←{(⍳⍵),¨box⊃⍵*÷2}
svec←{⊃(cmap rcb⍴⍵)pvex/(emt ⍵),⊂⊂⍵}
```

3 - Bag o' dfns

```
fix
0⍴⎕fx f9 ⍞

d←f9 e;f;i;k
f←(,(e='⍵')∘.≠5↑1)/,e,(⍬⍴,⍴e)⍴' y9 '
f←(,(f='⍵')∘.≠5↑1)/,f,(⍬⍴,⍴f)⍴' x9 '
f←1↓⍴d←(0,+/¯6,i)↓(-(3×i)++\i←':'=f)⍬f,(⍬6,⍴f)⍴' '
d←3⍬c9[1+(1+'⍺'⍷e),i,0;],⍉d[;1,(i←2↓⍳f),2]
k←k+2×k<1⍬k←i∧k⍷(>≠1 0⍬'←⎕'∘.=e)/k←+\~i←e⍷a9
f←(0,1+⍴e)⌈⍴d←d,(f,⍴e)↑⍉0 ¯2↓k⍬' ',e,[1.5]';'
d←(f↑d),[1]f[2] 'ⒶⒷ',e
```

```
      c9                  a9
   z9←                 012345678
 y9z9←                 9abcdefgh
 y9z9←x9               ijklmnopq
)/3→(0=1↑,             rstuvwxyz
   →0,0⍴z9←            ABCDEFGHI
                       JKLMNOPQR
                       STUVWXYZ⎕
```

4 - Traditional/Canonical APL



5 - Tacit/Direct/Trains Style

```
:If not InUWP
    answer←doesNotApply
:Else
    :If termUnder5 and RetirementAge≥75
        answer←refer

    :ElseIf RetirementAge≥75
        answer←doesNotApply

    :ElseIf ExitDate after oneMonthBefore 75 yrsAfter Dob
        answer←doesNotApply

    :ElseIf ExitDate before 183 daysBefore OrigNRD
        answer←any MVRs>0

    :ElseIf IsVista
        answer←ExitDate before ThreePolAnniversariesBefore OrigNRD
```

6 - Structured APL

There are a few major styles of APL code. Some of them are rarer than others, and some of them are best suited for certain situations over others.

- Box Diagram APL
- Traditional APL (Canonical Representation)
- Structured APL
- Didactic dfns
- Bag o' dfns
- Tacit/Direct/Trains Style

Traditional APL2

```
fix
0ρ□fx f9 □

d←f9 e;f;i;k
f←(,(e='⍵')∘.≠5↑1)/,e,(⌽4,⍴e)⍴' y9 '
f←(,(f='⍵')∘.≠5↑1)/,f,(⌽4,⍴f)⍴' x9 '
f←1↓⍴d←(0,+/¯6,i)↓(-(3×i)++\i←':'=f)⌽f,(⌽6,⍴f)⍴' '
d←3⌽c9[1+(1+'⍺'∈e),i,0;],⍉d[;1,(i←2↓⍳f),2]
k←k+2×k<1⌽k←i∧k∈(>≠1 0⌽'⍞'←□'∘.=e)/k←+\~i←e∈a9
f←(0,1+⍴e)⌈⍴d←d,(f,⍴e)↑⍉0 ¯2↓k⌽' ',e,[1.5]';'
d←(f↑d),[1]f[2] '⍝',e
      c9                    a9
   z9←                 012345678
 y9z9←                 9abcdefgh
 y9z9←x9               ijklmnopq
)/3→(0=1↑,             rstuvwxyz
   →0,0⍴z9←            ABCDEFGHI
                       JKLMNOPQR
                       STUVWXYZ□
```

Traditional style, also called *Canonical Representation*, this is what most people in the first generation of APL would have seen (Alan Perlis, Donald Knuth, Edsger Dijkstra, ⍰.).

Largest Modern Use, in my opinion, is to **maintain legacy systems.**

Major Linguistic Features:

- Dynamic Scope
- Flat Namespaces

- Global Definitions
- Labeled and Computed GOTO's for control flow
- Higher use of the Execute function

I do not think this should be considered *canonical* anymore.

Most major Computer Science oriented problems are better solved with **lexically scoped, more functional solutions**.

Dynamic scope and the increased namespace pollution of this style can make code harder to comprehend.

The use of GOTO's is **not** the primary difficulty in this style if they are used properly.

Nonetheless, this style might be appropriate when dynamic scope is an implicit element in the problem domain, or where the global name population matches closely to the problem domain.

In most cases that are not legacy or problem domain specific, Traditional style has the primary benefit of leveraging dynamic scope to inject definitions into arbitrary lexical scopes.

Lacking a macro system, Traditional style functions may be used to more accurately model a problem domain through code generation by coupling dynamic scope with the Execute function to eliminate or reduce boilerplate.

Structured APL

```
:If not InUWP
    answer←doesNotApply
:Else
    :If termUnder5 and RetirementAge≥75
        answer←refer

    :ElseIf RetirementAge≥75
        answer←doesNotApply

    :ElseIf ExitDate after oneMonthBefore 75 yrsAfter Dob
        answer←doesNotApply

    :ElseIf ExitDate before 183 daysBefore OrigNRD
        answer←any MVRs>0

    :ElseIf IsVista
        answer←ExitDate before ThreePolAnniversariesBefore OrigNRD
```

Structured style most closely matches modern, **imperative** languages.

- Dynamically Scoped
- Structured control flow instead of GOTOs
- Otherwise similar to **Traditional**

Exhibits some of the same limitations as the Traditional style, but some business logic domains closely mirror this structure, making it particularly appropriate where there is a high concentration of domain vocabulary with structured control words mapping naturally to the problem domain.

It can be very helpful pedagogically when describing traditional Algorithms, as the APL code can often match the pseudocode nearly character for character.

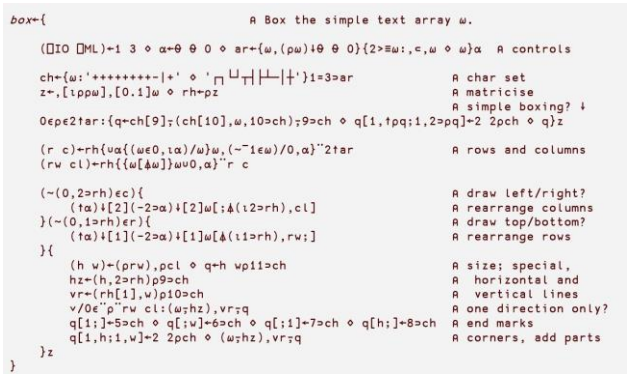
Traditional and Structured styles both suffer heavily from sequential bias, making them very difficult to parallelize if control flow is heavily emphasized over data-centric APL sentences.

Structured style in particular can be an excellent migratory step in refactoring specifications and code to be more parallelism friendly.

Didactic dfns (a.k.a. – Running Commentary dfns)

---

di' dac tic, ADJECTIVE --- intended to teach, particularly in having moral instruction as an ulterior motive: "A didactic dfns that set out to expose wasteful programming practices."



```
box←{                              ⍝ Box the simple text array ⍵.

    (⎕IO ⎕ML)←1 3 ⋄ α←⊖⍬ 0 ⋄ ar←{⍺,(⍴⍵)↓⍬ 0}{2>≡⍵:,⊂,⍵ ⋄ ⍵}⍺  ⍝ controls

    ch←{⍵:'+++++++++-|+' ⋄ '┌┐└┘┬┤├┴─│┼'}1=3⊃ar            ⍝ char set
    z←,[ι⍴⍴⍵],[0.1]⍵ ⋄ rh←⍴z                            ⍝ matricise
                                                        ⍝ simple boxing? ↓
    0∊ρ∊2↑ar:{q←ch[9]⍪(ch[10],⍵,10⊃ch)⍪9⊃ch ⋄ q[1,↑⍴q;1,2⊃⍴q]←2 2⍴ch ⋄ q}z

    (r c)←rh{⍺α{(⍵∊0,ια)/⍵}⍵,(~¯1∊⍵)/0,α}⍨2↑ar           ⍝ rows and columns
    (rw cl)←rh{{⍵[⍋⍵]}⍵∨0,α}⍨r c

    (~(0,2⊃rh)∊c){                                      ⍝ draw left/right?
        (↑α)↓[2](-2⊃α)↓[2]⍵[;⍋(ι2⊃rh),cl]                ⍝ rearrange columns
    }(~(0,1⊃rh)∊r){                                     ⍝ draw top/bottom?
        (↑α)↓[1](-2⊃α)↓[1]⍵[⍋(ι1⊃rh),rw;]                ⍝ rearrange rows
    }{
        (h w)←(⍴rw),⍴cl ⋄ q←h⍴11⊃ch                      ⍝ size; special,
        hz←(h,2⊃rh)⍴9⊃ch                                ⍝  horizontal and
        vr←(rh[1],w)⍴10⊃ch                              ⍝  vertical lines
        ∨/0∊¨⍴¨rw cl:(⍵⍪hz),vr⍪q                        ⍝ one direction only?
        q[1;]←5⊃ch ⋄ q[;w]←6⊃ch ⋄ q[;1]←7⊃ch ⋄ q[h;]←8⊃ch  ⍝ end marks
        q[1,h;1,w]←2 2⍴ch ⋄ (⍵⍪hz),vr⍪q                 ⍝ corners, add parts
    }z
}
```

Didactic dfns style is one of the most useful general-purpose styles.

- Lexically scoped
- Functionally oriented
- Convenient recursion-centric syntax
- dfns syntax with a two-column format, source on the left, and a column of comments on the right
- Think "Cliff Notes for Code"

This style **greatly increases the visual space** required for a single function.

Optimizes single function clarity over inter-function relationships.

Advisable for code that is likely to be **read very often** by new users, but **edited little**, such as library functions or utility functions.

Inadvisable for code that requires frequent editing or contains significant macro-level interdependencies.

Usefulness declines as the overall system complexity increases.

Instead of using structured control flow, focus on finding good APL lines that are controlled overall by recursion (if necessary) with simple base cases and utilize the nested dfns block *technique to create visually suggestive nested scopes without excessive helper names, but keep to a functional programming motif.*

Leverage the two column format to create a guide of how much to put on a single line based on how well you can talk about that line.

Bag o' dfns

```
at←{⍵+(⍴⍵)↑(-⍺⍺)↑⍺}
avl←{(⍳⊃⍴⍵)~⍵×⊃⍺[]⍺⍺}
box←{⍵≠⍵/⍵ ⍵⍴⍳⍵*2}
cmap←{⊂[⍳⍴⍴⍵]1∊¨⍵∘.=⍵}
emt←{(,⍵=0)/,⍳⍴⍵}
pvec←{(⍺(⍺⍺ avl)⍵)(⍺ at)¨⊂⍵}
pvex←{⊃,/⍺∘(⍺⍺ pvec)¨⍵}
rcb←{(⍳⍵),¨box⊃⍵*÷2}
svec←{⊃(cmap rcb⍴⍵)pvex/(emt ⍵),⊂⊂⍵}
```

Bag o' dfns style is a dfns style like the Didactic style is. It is also a general-purpose style that works well for a number of problems.

- Single line dfns, or very short dfns
- Increased dfns names, decreased local variable bindings
- Minimal comments, usually at the top of a group of dfns

The advantage here is that **Global dependencies are easier to see**. There is also a clear edge in terms of **editability and refactoring**.

A balance must be achieved here between too many functions, and overly large functions.

This style works particularly well as an exploratory or evolutionary development style. There is strong support for it in the Dyalog session through the]defs class of user commands. It scales reasonably well as the domain complexity increases, and gives a good deal of flexibility in terms of naming.

This style is most analogous to traditional Functional Programming styles such as Scheme or ML, optimized to have less verbosity.

It makes it particularly easy to delete and replace code instead of requiring complex debugging sessions.
(We call this Friedman debugging at IU.)

*The **Bag o' dfns** style probably has the most **flexibility** and **scalability**, since you can transition to Didactic Style or Tacit style from Bag o' dfns with minimal effort.*

Tacit/Trains/Direct Style

Conflict of Interest Disclosure: this is my favorite style.

```
⎕fh←(0≠1⌷⍵)⊃(⊂⍬⍴⍵)⍪0'M'0 '',0,⍤(⊂⍤⊣),⍬⊂⍬⍴⍵⍪1'F'1,('fn'enc⍤⊣),(⊂⍤⊣),5↓⍬,1↑⊢
⎕fn←(d,'Of',3↓⊢)⍎1 at(Fm∧'b'⍷⍤k)(d,'Vf',('fn'enc⍬⊃r),4↓⊢)⍎1 at(Fm∧1⍷⍤k)
⎕f←(⊃⌿/)(1,1↓Fm∧1⍷⍤k)blg(⍉r)(⊂⎕fh⍪⍬(((⊢-(⊃-2⌊⊃))d),1↓⍎1⊢)⎕fn)⌸1↓⊢
```

A favored style in the J and Rationalized APL traditions, this style is analogous to the "points-free" styles that are now seeing some use (sparingly) in places like the Haskell community.

- Points-free
- Minimal scope (generally all top level)
- Data-flow/Pure functional
- Function explicit rather than data explicit
- Minimal control flow
- Favors short, terse naming conventions
- Usually restricted to a single line per definition

This style has the advantage of generally **rendering moot the issues of lexical vs. dynamic scope, control flow constructs, and variable naming**. It is also the most **parallel friendly** option bar none.

The disadvantage is that it is the most **alien to non-APL programmers**, and requires swallowing the absurdity of APL's terseness. Points-free programming in other communities, where it exists, is often considered **controversial**.

---

*The Direct style most closely matches the ideals of **Lyrical programming** and natural language compositionality.*

*It exhibits the **maximal malleability** of any of the styles, and is also the most **regular** in terms of appearance. You can **see more of your code** with this method than any other.*

Problem domains that exhibit data flow tendencies are a natural fit for this style.

Depending on the domain, this style may be the most **difficult to map to some procedural business logics**. In these cases, it may be worth it if the benefits of parallelism or concision outweigh supporting older legacy process documents, but this will not always be the case.

Users of this style should be careful to leverage **idiomatic expressions** and careful use of **structure and naming conventions** to reduce or limit boiler plate. Attention should be paid to maintain an **appropriate level of density**, so that code doesn't become very shallow and then very dense, which can be confusing.

Be very careful about needlessly introducing helper functions that increase the **edit distance** of your code.

Mix and Match

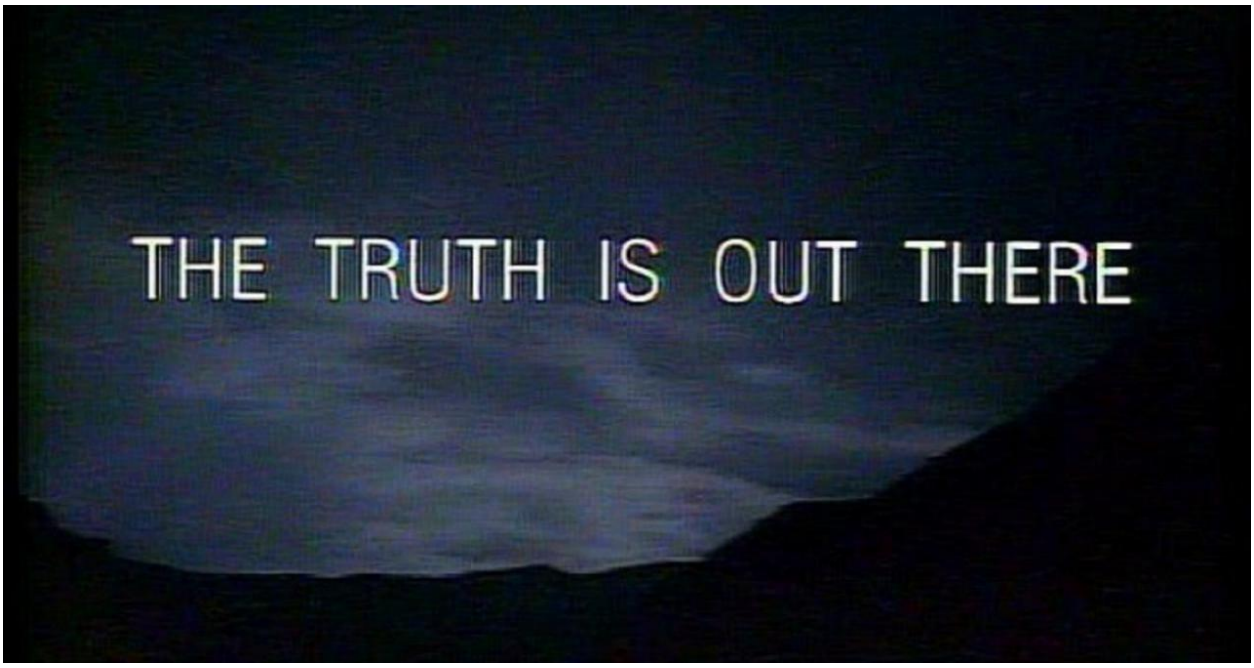
```
Sfn←aws _s (('□sp' _tk)_o('□XOR' _tk)) _s aws _as {P ∈ω}
Prim←prim _as P
Vt←(((O□⍁⍺)⍳ω)1□⍺⍀'' ¯1}
Var←{α(aw _o (name _as ϕ) _t (αα←Vt) _as (ωω V∘,∘⊃))ω}
Num←float _o int _as (N∘ϕ)
Strand←0 Var 'a' _s (0 Var 'a' _some) _as ('s'A∘ϕ)
Pex←{α(rpar _s Ex _s lpar)ω}
Atom←Strand _o (0 Var'a' _as ('v'A)) _o (Num _some _as ('n'A∘ϕ)) _o Pex
Idx←{α(rbrk _s Ex _s lbrk _s Atom _as ('i'E∘ϕ))ω}
Blrp←{α⍺ _s (ωω Slrp ∇))ω}
Slrp←{α(⍺⍺ _o (ωω _s ∇) _o ((1 _eat) _s ∇))ω}
Fn←{ns←n z≠⍤⍤m←{(Fm ω)∧1∈⍤⍤kω}z←⊃⍀/ω ⋄ 0=≠ns:0(,⊂z)α ''
 p←(α⍀0,⍤⍤⍀'⍺ω')∘(Gex _o Ex _o Fex Stmts _then Fn)¨ns ⋄ 0<c←⌈/⊃¨p:c '' α ω
 0(⍀/(<0 4ρθ),⍤⍤p{((3⍳ω),<'')⍀ω((d+∘⊃→),1↓⍤⍤1⊢)wrap⊃⍀/1⊃α}¨at{m}↓z) α ''}
Pfe←{α(rpar _s Fex _s lpar)ω}
Fnp←Prim _o (1 Var 'f') _o (rbrc Blrp lbrc _as ('F'new 1,∘←∘ϕ1↓¯1⊢)) _o Pfe
Mop←{α((mop _as P) _s Afx _as (1 O))ω}
Dop1←{α((dop1 _as P) _s Afx _as (2 O∘ϕ))ω}
Dop2←{α(Atom _s (dop2 _as P) _s Afx _as (2 O∘ϕ))ω}
Dop3←(dop3 _as P) _s Atom _as (2 O∘ϕ) _o (dot _s jot _as (P∘ϕ) _as (1 O))
Bop←{α(rbrk _s Ex _s lbrk _s Afx _as ('i'O∘ϕ))ω}
Afx←Mop _o (Fnp _s (Dop1 _o Dop3 _opt) _as (⊃wrap/∘ϕ)) _o Dop2 _o Bop
Bind←{α(gets _s (name _as ϕ) _env (⍺⍺{(⊃ϕω)α⍺⍀α}) _as (ωω new 'b',1↓⊢))ω}
Fex←Afx _s (1 Bind 'F' _any) _as (⊃wrap/∘ϕ)
App←Afx _s (Idx _o Atom _s (dop2 _not) _opt) _as {(≠ω)Eϕω}
Ex←Idx _o Atom _s {α(0 Bind 'E' _o App _s ∇ _opt)ω} _as (⊃wrap/∘ϕ)
Gex←Ex _s grd _s Ex _as (G∘ϕ)
Nlrp←sep _o eot Slrp (lbrc Blrp rbrc)
Stmts←{α(sep _any _s (Nlrp _then (⍺⍺ _s eot∘ϕ)) _any _s eot)ω}
Ns←nss Blrp nse _then (Ex _o Fex Stmts _then Fn) _s eot _as M

ps←{0≠⊃c a e r←(0 2ρθ)Ns ∈{ω/⍨∧\'⍝'≠ω}¨ω,¨□UCS 10:□SIGNAL c ⋄ (⊃a)e}
```

```
scp←(1,1↓Fm)⊂[0]⊢
prf←((≠⍤¯1↓(0≠⊢)(/∘⊢)⊢)⍠1⍤⍤r)⊢
blg←{α←⊢ ⋄ α((prf(⌈/(⍳∘≠⊢)×⍠1(1↓→)∧.(=∨0=⊢)∘⍁⊢)⍺⍺(≠∘⍤)r)□⍠0 2 ⍵⍵(≠⍨)⍺⍺)ω}
enc←⊂→,∘⊃((→,'_',⊢)/(⊂''),(⍕¨(0≠⊢)(/∘⊢)⊢))
veo←∪((⊂'⍝u'),(,¨prims),→)~⍨∘{⊃,/{⊂⍠(1≡≡ω)⊢ω}¨ω}¯1↓⊢(/⍨)(∧/¨0≠((⊃0ρ⊢)¨⊢))
ndo←{α←⊢ ⋄ m⊃∘(⊂,⊢)¨⍺∘⍺⍺⍨¨ω⊃∘(,∘⊂⍨⊂)⍨¨⍨m←1≥≡¨ω}
n2f←(⊃,/)((1=≡)⊃,∘⊂⍨∘⊂)¨

rn←⊢,∘↓(1+d)⍤⍠¯1(+⍁d∘.=∘⍳1+(⌈/0,d))
rd←⊢,(+/⍤⍤∘r∧.(=∨0=⊢)∘⍁∘⍤∘r⊢(≠⍨)Fm∧1∈⍨k)
df←⊢(≠⍨)(+\1=d)(~→∈→(/⍨)(1=d)∧(~'b'∈⍨k)∧Om∨Fm)⊢
dua←((~Gm)∧Fm∨↓∘prf∈r∘Fs)(→(⍁∘⊢)(→(≠∘⊢)0∈⍨n)(0,1↓(¯1ϕ→)∧⊢=¯1ϕ⊢)→(≠∘⊢)d)⊢
du←(~duav(∨/(prf∧.(=∨0=⊢)∘⍁dua(≠∘⊢)prf)∧⍤∘r∧.≥∘⍁dua(≠∘⊢)⍤⍤r×0=prf))(≠∘⊢)⊢
lfh←(0≠1□→)⊃(⊂∘⍁∘⍀⍤0'M'0 '',0,¨(⊂→)),∘⊂∘⍁∘⍀⍤1'F'1,('fn'enc→),(⊂→),5↓∘,1⍤⊢
lfn←(d,'Of',3⊢)⍠1 at(Fm∧'b'∈⍨k)(d,'Vf',('fn'enc∘⊃r),4↓⊢)⍠1 at(Fm∧1∈⍨k)
lf←(⊃⍀/)(1,1↓Fm∧1∈⍨k)blg(⍤r)(⊂lfh⍀∘(((⊢←(⊃-2⌊⊃))d),1↓⍠1⊢)lfn)⌸1↓⊢
dn←((0∈⍨n)∧(Am∧'v'∈⍨k)∨Om∧'f'∈⍨k)((~→)(≠∘⊢)(d-¯1ϕ→),1↓[1]⊢)⊢
mrep←(1+⊃),'P'0(,'⊢'),(⊂''),¨¯1↓4↓∘,1⍤⊢
mreu←⊃,'E' 'u',(⊂''),¨¯1↓3↓∘,1⍤⊢
mre←(⊃⍀/)(-∘∘Vm∨Am)∘⊃∘ϕ(↓,(((⊢ρ¨(≠⍁),¨≠×2<≠)mreu⍪mrep⍀(1+d),1↓⍠1⊢)¨⍤))⊢
mrs←⊢⊂[0]¨1,1↓d=1+∘⊃d
mrk←(-∘(+/∧\)∘ϕLm)(⍤⍀¨∘(mre(mre mrs)¨at(Gm∘(⊃⍀/)1⍤¨⊢)∘mrs)↓)⊢
mr←(⊃⍀/)((1⍤⊢),(mrk¨1↓⊢))∘scp
ur←((2⍤⊢),1,('um'enc∘⊃r),4↓⊢)⍠1at(Em∧'u'∈⍨k)
rt←⊢,(∨\Fm)+(+≠prf∧.(=∨0=→)∘⍁¨∘⍤or Ms⍀Gs)-Fm
nm←((3⍤⊢),('fe'enc∘⊃r),4↓⊢)⍠1 at((0∈⍨n)∧Em∨Om∨Am)
lgg←(⍀/1↓⊢)⍀∘⊃¨→(((¯1+d),2,¨⍳t,k,n,r,∘⍀s)⍀→⍀3,'V','a',3(↓⍠1)1⍤⊢)∘⊃1⍤⊢
lg←(⊃⍀/)⊢((⊂→(⍷¨∘~)(∨\⊢)),(((1⍤⊢)lgg⊢⊂[0]¨d=1+⊃)¨⊂[0]¨))Gm∧1ϕEm
fet←(d,'V'0,3⊢)⍠1 at(0,1↓Em∨Om∨Am)(d,'Av',3⊢)⍠1 at(Em∧'b'∈⍨k)
fee←(⍀/ϕ)(Mm∨Em∨Om∨Am)blg⊢((⊃∘ϕ⊢)(⊂(d--¨∘⊃),1↓⍠1⊢)∘fet→⍀¯1↓⍠1⊢)⌸⊃,¨1↓⊢
fe←(⊃⍀/)(+⍁d⍵g)(⊂(⊢⍤¨1=∘≠⊢)⍀∘⊃∘fee⊢)⌸⊢
can←(+\Am∨Om)((,1⍤⊢),∘(⊂¯1+2⌊≠)⊃(⊂∘⊂⊃),⊂)∘n 1↓⊢)⌸⊢
```

You can, of course, mix and match styles to suit your particular case.

Be careful to not impair readability by increasing the abstractive overhead too much when doing this.

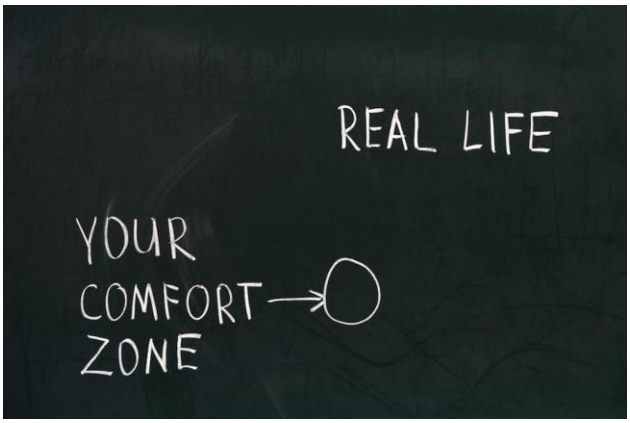
The Myth of Readability

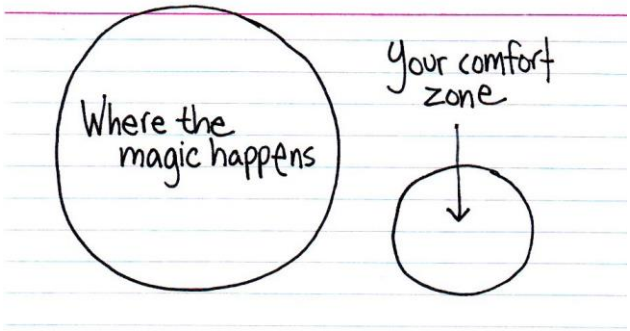


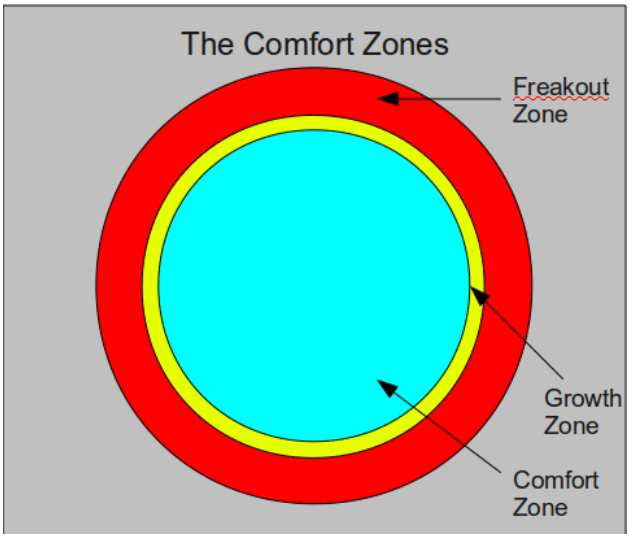
APL is practically the poster child for "write only" languages.

This statement simultaneously falls short and overstates the case.

When you come from a relatively insulated linguistic community that is not APL, not only is APL hard to read, but it also turns out to be almost impossible to write.


REAL LIFE
YOUR
COMFORT
ZONE


Where the magic happens
your comfort zone


The Comfort Zones
Freakout Zone
Growth Zone
Comfort Zone

Readability *in Programming is more about* ***Comfort*** *than it is about* ***Comprehension****.*

It is not a universal constant; it is in fact the ***wrong metric*** *to use when evaluating code.*

A New Definition of Readability

Readability is the ***amount of money*** *you're willing to bet that a change you make to the code will* ***work the first time****, without breaking anything, without testing it mechanically first.*

Confidence vs. Comfort

Your *confidence* in changing or deleting code is directly tied to your ability to understand the code itself.

Confidence is a better measure of usability than minimum effort by uninitiated programmers.

This definition captures micro and macro levels, and unifies the apparently different philosophies of APL and traditional languages. It explains the dissonance between the two camps adequately.

Local vs. Global Readability

Traditional languages encourage **isolating specific code** elements to improve **local reasonability**.

A restricted, carefully architected class hierarchy can make it possible to protect other objects/classes from changes in a single class.

Functional programmers with static type system create similar boundaries by utilizing strong typing guarantees that contractually obligate distant pieces of code in a mechanically verified way. As long as the types stay constant, certain guarantees allow local changes with confidence that they will not break long distant contracts, so long as the types can express the contract.

Locally, these techniques improve readability, since they narrow the scope of observable effects, making it easier to see.

APL micro readability is a matter of equational equivalence, regularity of expression, symmetry and ease of parsing, nesting level, consistency of data flow, and the generality of a single line of APL code.

That is, micro-readability of APL is about **single lines, rather than single functions**.

This is only a part of the picture...

The heart of the readability debate is in the Macro vs. Micro principle.

*APL's **brevity makes discussion about inter-procedural and architectural readability a matter of coding style**, while external documentation often supplements traditional languages to document these higher-level issues, without which code is often considered hard to read at a macro level.*

*Many APLers may argue that these macro level issues **make other languages hard to read**.*

At the micro level, APL programmers and mainstream programmers share common ground about changing things. Impact of micro level changes are usually readily apparent in well designed code bases of either type.

What about broad, sweeping architectural and **macro-level edits** to the code base?

1. Do we really need this function? Why not just delete it entirely?
2. Can we remove this data type from the code base and merge it with another?
3. Can we split out functionality for this class into two classes and have things work?
4. How about we change this function to take a higher-order operation?
5. Can we curry or uncurry this function?
6. What if we merge these three functions into a single function that uses a new, merged data type?

Micro edits are changes inside of the conceptually leaf nodes of the system graph. **Macro edits are changes that alter the system graph itself, such as its edges or the actual nodes in the graph.**

Mainstream programming style **generally eschews explicating the system graph in the text of the code**.

Documentation, type errors, tooling, diagrams, and **extra-source artifacts serve the purpose of describing the system graph**.

Reasoning about the system graph generally happens with these extra-source artifacts.

Code level system graph elements are kept **intentionally implicit or minimal**, in order to emphasize the local reasoning of the code.

Seeing the bigger picture is usually not the point of the source code itself.

The system graph of an APL program is generally kept **simpler**, since the amount of computation happening inside of a leaf node is increased.

The Macro emphasis combined with Brevity also means that **system relationships are more visually prominent**.

This **prioritizes the "big picture" while suppressing the overall impact of local elements**.

The 8 Patterns generally means that good APL programs are more **amenable to broad, sweeping changes** and refactoring when compared against traditional source code.

Since more code is visible at the same time, ramifications of **large edits are usually easier to model** mentally.

The APL programmer feels that APL programs are more readable:

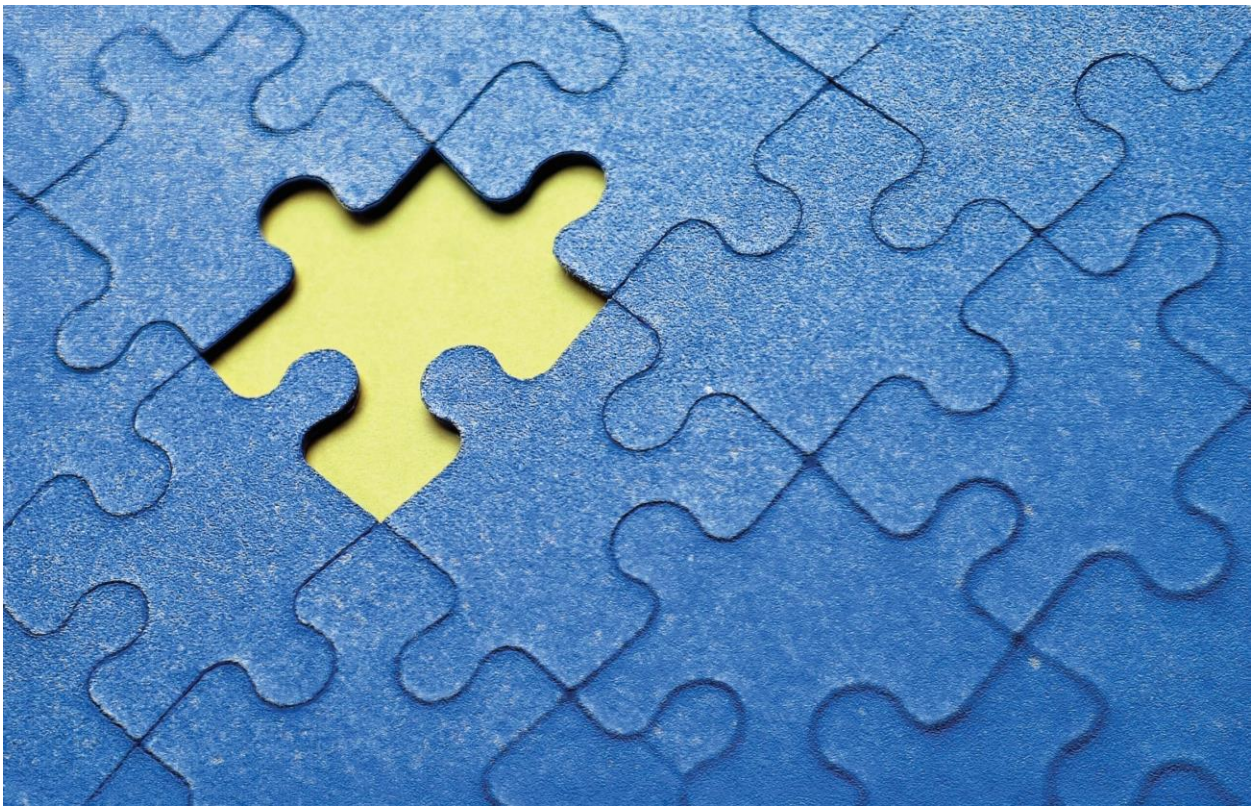
- *Not because one reads more lines per minute*
- *Not because it is easier to get the general idea*

---

*Instead, one is more confident of the complete picture. The idea of a **code rewrite should be less daunting** in an APL codebase. They might not even be thought of as rewrites.*

*Optimizing at the "whole system" level can have important ramifications for code debt, performance, and codebase longevity. **Optimizing for the "system rewrite" may be more important than other types of "readability".***

Conclusion



Unlike many other languages, there isn't one generally recognized "APL style."

Different APL styles result in very different looking code, suitable for different environments.

Nonetheless, there are **specific design principles and patterns** that arise across the various APL styles. These principles help to anchor a common theme throughout the APL community.

Unfortunately, these patterns tend to **push against prevailing best practices** in the Computer Science and Software Engineering industries.

We can attribute some of the difficulty with effectively using APL to the systematic internment of these best practices in experienced programmers, even to an emotional and moral level. Overcoming the visceral response to APL's design principles and methods of use and learning to approach the use of APL differently, as a tool of thought, rather than as a tool of systems design, will greatly aid in applying APL to real problems.

This requires an appreciation of the whole picture, the social, cultural, systemic, and linguistic elements of APL development and the associated emergent properties, rather than attempting to insert APL as a replacement for an existing language without concern for the context of that use.

This may not make APL easier to integrate or learn, but it does at least give a path of adoption that can be better planned, with a better understanding of how to use APL properly.

I hope that it also gives pause, so that we can consider how other languages affect this larger system, and put a better eye towards the human response and effects rather than taking the human factor for granted or as a constant.

Thank You.

Image Credits

Fireworks for a Bee -- Vincent Brassine (CC BY-NC-ND 2.0)

Ken Iverson -- Computer History Museum

Alan Perlis -- The Vantage Point (CC BY-NC-ND 3.0)

Pi: The Transcendental Number -- Tom Blackwell (CC BY-NC 2.0)

Micro flowers -- Orest Ukrainsky (CC BY-SA 2.0)

Taking in a Morning with Mount Rainier (Mount Rainier National Park) -- Mark Stevens (CC BY-NC-SA 2.0)

Cork Oak -- melystu (CC BY-NC-ND 2.0)

Pattern -- Sam Cox (CC BY-NC 2.0)

Brevity -- Pulpolux !!! (CC BY-NC 2.0)

Transparency -- Manuel (CC BY-NC-SA 2.0)

Warm Data -- beachmobjellies (CC BY-SA 2.0)

Library -- Stewart Butterfield (CC BY 2.0)

Honeycomb -- Justus Thane (CC BY-NC-SA 2.0)

Subtle -- B, K ?G (CC BY-NC-ND 2.0)

Gemstones fractal -- J. Gabas Esteban (CC BY-NC-SA 2.0)

Truth is out there -- therichest.com. Some rights reserved.

Thank You -- bunnicula (CC BY-ND 2.0)

Your Comfort Zone and Real Life -- achurchforstarvingartists (CC BY-NC-SA 3.0)

Comfort Zones -- Flustered Grade (CC BY-NC-ND 3.0)

Comfort Zone, Magic Happens -- AmirHossein lifestyle + Indexed

Current Practice Comfort Zone -- Maths No Fear (CC BY-SA 3.0)

Brainwaves -- The Controversial Files (CC BY 4.0)

Bigger on the inside -- Anna (CC BY-NC 2.0)

Secret Society -- The World's Best Ever (CC BY-SA 3.0 US)

Rarity It is On -- TheFlutterKnight (CC BY-NC-SA 3.0)