APL Style: Patterns vs. Anti-patterns

APL is an infamous language that is both beloved and hated across the spectrum of programmers. Ironically, APL has had more success with non-computer scientists than with seasoned programmers. There seems to be a general feeling of “never in my life” among many programmers, while many of the daily users of APL couldn’t imagine life without APL. Why do computer scientists have such trouble with APL?

APL is known for its one-liners and an almost lyrical expression of computation ideas. Alan Perlis phrased it thusly:

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.

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.

# Let’s 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 Expression 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

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:

+/m⍴n ←→ n×m

×/m⍴n ←→ 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

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

To quote Iverson, again:

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 not economical languages however, 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

At the time Iverson designed APL, large scale mechanical proofs were not even on the radar of practical possibility. Proofs by exhaustion and other sort of mechanical explorations were available, but by a wide margin, the ability to physical manipulate 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 to do so. 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 vs. Anti-patterns Dichotomy

So much for the design principles of APL. They give us a hint into what makes APL different, but they don’t really 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. A major problem with progressing in APL is that forward progress and good APL follow a set of design/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.”

When APL code appears before such a specific palate, it leaves a sour taste, because the flavors, as it were, for which APL is famous, directly contradict the prescribed norms of traditional programming methods. Because value judgments about good and bad code have been applied so strongly and deeply into the psyche of the traditional programmer, it is no wonder that such a programmer struggles to see anything desirable in the language. Despite the excellent track record with serious users, one must consciously evaluate APL code by a different metric to understand it and how to use it effectively.

Attempting to use traditional software construction patterns to write APL code will only result in writing code better suited to another language. The APL language is so different in practical use from other schools of language design and program construction that we must abandon traditional software engineering wisdom if we are to identify how to use APL effectively to produce good, maintainable, “beautiful” code.

The following patterns contrast traditional software engineering practice with the patterns of practice that appear in well refined and tuned APL code. This is not to say that all APL code will look like this, but rather, that APL code generally respected and viewed with appreciation, and to which style APL programmers aspire and work when actively improving their code base, tends to follow these principles in one form or another, in direct contrast to the traditional patterns of design.

## Concision vs. Verbosity

The above principles of design given by Iverson make it clear that brevity plays a key role in the language design principles that spawned APL. This probably doesn’t come as a surprise, as APL is often remarked on for its concision and terseness.

This is not tersity for tersity’s sake. Instead, brevity is a recurring feature that supports the specific language design goals mentioned previously. Brevity is critical to the human comprehension and retention of a given code base.

Common programming practice suggests that those unfamiliar with the code base should still get the general idea of the code, which is the idea of "readability." The result is that usually code is deemed to be more readable when it is more verbose, as the expected reader is assumed to now have an intimate knowledge of the entire code base. Indeed, it is presumed that even programmers that spend their whole lives inside of the code base will not be able to keep the whole of the source in their minds at once because of the complexity and size of most non-trivial programs. This leads to a general favoritism to verbose code over concise code, which is considered to be more self-documented even for the experienced programmer.

APL programmers, on the other hand, often optimize their code so that it is more easily manipulated, and they are more easily able to absorb the code base on the whole. Contrary to traditional wisdom, they often expect that they will be able to see and process significant programs at one time (macro vs. micro) which in turn leads them to favor concision, which allows them to see more of the code, rather than verbosity, which hides more code at a time. Concise code is taken to be more readable both at the small level (because it is easier to play with and explore and understand as a whole) but also at the level of integration, where the more concise code is taken to be easier to integrate into a larger code base and not lose sight of the entire source code.

## Macro vs. Micro

Traditional programming problem solving practice encourage breaking down a problem into very small components, and most programming languages are designed around the idea of providing a clear picture of a small piece of code in isolation of the rest of the code. APL instead emphasizes finding ways of expressing solutions that emphasize the macro-view of the problem, and 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 vs. Abstraction

Abstraction is considered harmful. This isn't to say that we don't work with abstractions, or at an abstraction, but the process of programming by which we build progressively upon previous abstractions with new abstractions, eventually resulting in some generic framework that we can then plug in our desired instance and obtain a result is considered to be a harmful practice.

Rather than subordinating detail, abstraction hides it. "Good" abstraction hides it so well that you cannot access it. Good programming languages often provide means of solid abstractions that prevent introspection beyond the abstraction boundaries, while languages are often criticized for permitting too much leakiness in their abstraction boundaries.

APL on the other hand, embraces transparency of expression, and tends to eschew the ever-increasing introduction of abstraction to solve complexity. Instead, transparency of ideas, and directness of expression serve as the weapons with which the APLer falls upon the enemy called "Complexity" and deals a mortal blow.

## Idioms vs. Libraries

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.

## Data vs. 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, achieving the end result, rather than to have a more complicated source structure that embeds the logic in the control flow of the program.

## Structure vs. Names

Other programming languages usually expect to use long, descriptive names that reference often very distant objects or ideas. The use of names serves as the documentation that allows one to track through the code and have an idea of what is going on.

To the APLer, names, particularly long names, often get in the way of revealing the overall structure of the code, which is considered to be more documentary and elucidatory than the names would have been. Because implementations of ideas are often shorter than descriptive names for the same, the use of structure and relatively short (in terms of spatial distance) data dependencies means that names are often chosen to avoid getting in the way of the structure, rather than providing the primary anchor points for understanding the code.

## Implicit vs. Explicit

Often programmers are taught and follow the best practice of being very explicit with types, data encodings, and API boundaries. Often data mappings and control flow are very explicit. APLers tend to prefer to make many things implicit in the code, to make the bigger picture clearer. This leads to code that is often written for special cases, that just happen to also work in the general case, or code that is written implicitly in such a way that it is general, though the specific details of the generality are not expressed anywhere. APLers are also more likely to be willing to endure apparent "type" issues or other "inconsistencies" that might otherwise be solved by being more explicit.

## Syntax vs. Semantics

Most programmers are taught that syntax matters less than semantics, and they are often only considered with ensuring that the semantics of their ideas are doing what they want, and they will often ignore syntax or only pay attention to it in so far as is required for their own limited sanity.

APLers on the other hand, have a strong tradition of focusing on syntax over semantics, and often spend considerable time thinking about the experience that one feels with code, and how that code is "read" as well as how the syntax flows and works. Semantics are often given their simplest possible assignments, and more effort is put into how best to work with the syntax, rather than working on more sophisticated semantics.

# APL Styles

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.

## Traditional APL2

This is the traditional APL that folks like Alan Perlis would have seen compared to the wave of Pascal and BASIC code that was around at the time. It’s use today is largely in support of legacy code.

The major linguistic features are that it applies a dynamically scoped, flat namespace macro view of the world with labeled/computed goto’s for control flow within a given function definition.

For most computer science oriented problems, there are better ways of writing code. The dynamic scoping and goto-heavy style can be a problem for problems that are a more natural fit for a lexically scoped style with either function or structured control flow.

However, there are some business situations in which the incoming specification of what should be written is directly or closely aligned with a dynamically scoped, goto-style of programming. In such cases, the ability to directly mirror the style of writing and thought process used on the business side should not be under-estimated. Following the business patterns closely can better permit the use of Direct Development, where the end-user/customer is reading code along with the programmer.

## Structured APL

Structured APL is closest to more traditional imperative styles. It is still dynamically scoped, but internal, micro-level control flow makes use of traditional structured programming constructs such as While loops, If statements, and Switch/Case statements.

Again, this is probably best reserved for cases where the business specifications or logic closely mirrors this structure. Because of the English keywords for control flow, one should be careful to keep the use of the structured control statements to points where the same terms appear within the business logic.

A problem with this style and the traditional style is that it does not parallelize as easily. It is possible to use this method as a stop-gap between more refined code and existing business logic practices. Once a customer is used to relying on the code for their functional specification instead of relying on other documents, then a careful move towards more parallelism friendly designs can enable improved performance and better readability over time.

## Pedagogical dfns (a.k.a. – Running Commentary dfns)

Probably one of the most useful general-purpose styles, the dfns language is a lexically scoped, functionally oriented style of array programming. It has the benefits of lexical scoping and convenient recursive syntax.

This style consists of utilizing the standard dfns syntax in a two column format, with the source generally divided into a left column of code, and a right column of comments. A running commentary on the right side mirrors the operations on the left side.

This style greatly increases the visual space required for a single function, which can have negative impacts on macro-readability. However, it makes it easy to follow the reasoning and logic behind code inside of a single function. Therefore, it is generally recommended for situations in which the inter-procedural dependencies are essentially non-existent, and where a great deal of new users are likely to need to read the code, but not edit the code. Cases like this might be for a library of utility functions or shared functions for specific classes of problems. In these cases, understanding the function quickly becomes more important, since there are no macro dependencies to consider. Since they are utility functions, it is not anticipated that many changes will be made to the function over time, and therefore the ability to quickly edit the function is less important.

However, because of the verbosity of this style, it is not recommended as a general implementation technique for complex software that developers will be working on regularly. The cost of maintaining the running commentary and the potential loss of global view makes it less valuable within the context of a team of developers working on more complex systems that have inter-dependencies between functions, and that value the ability to refactor and continue forward progress on code over the ability for any single newbie to understand the micro-level of the code quickly.

## Bag o’ dfns

The bag o’ dfns style is like the running commentary style in its use of dfns, which would be generally advocated for most tasks and most problems that are not highly aligned with one of the trad-fns styles. However, it omits the two-column style and instead tends to favor keeping dfns short and fitting on a single line of source. There are clear advantages to refactoring here, and global view is much easier to see.

Care should be taken here not to introduce extraneous names that clutter the overall space. It may be worth it to have a dfns that is more than a single line to avoid excessive names.

The bag o’ dfns method of development is particularly useful as an exploratory development method on the APL session with the use of the ]dfns user-command.

## Tacit/Trains-centric

Tacit/trains-centric programming is one of the most renowned styles of APL programming, but also one of the most alien to those of mainstream programming styles. It has significant benefits of brevity, regularity of code, consistency, and the ability to make rapid changes to the code confidently and efficiently.

However, this model comes at the cost of not usually being a natural model for common business logic specifications. This can make it difficult to use in those situations unless careful design considerations are made to ensure the appropriate semantic density.

The style itself consists of composing your system as a series of trains/tacit program definitions. The model creates exceptionally agile and short code, enabling you to see more of your code than just about any other method.

Users of this style should be careful to use some idiomatic techniques to reduce “boiler plate” code that might make the code density shallower than it appears – never a good thing. It is important in this style to utilize structure to best effect and try to maintain a high degree of meaningful idiomatic coding practice, such as reducing nesting depth of expressions, to ensure that code is easy to read and track.

If you are working in a domain that is heavily data flow oriented, or the “end-user” is near or familiar with the Computer Science fields, the use of the trains style can have significant benefits in terms of long term maintenance and architectural simplicity.

# The Myth of Readability

As we have hinted at above, the issue of readability deserves specific attention. APL has often been accused of being a “write only” language. This starkly contrasts the general sentiment of those writing APL.

We should consider what we really mean by readability. Most people define readability in terms of comfort and familiarity. The more comfortable and familiar a piece of code looks to them, the more likely they are to consider it readable. There is more to it than that, but the vast majority of initial “readability” sentiments are around what feels comfortable. This is a bad definition for readability.

Instead, let’s redefine readability as something more concrete.

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.

Your confidence in changing or *deleting* code is directly tied to your ability to understand the code itself. This definition of readability is better able to succinctly capture what it means to have readable code than the more common definition that readable code is code that an uninitiated programmer can understand with minimal effort.

The betting definition adequately captures readability at the micro and macro levels, and explains the dissonance between APL and traditional languages along this metric.

Traditional abstraction heavy languages make it possible to isolate specific code for improved local reasonability. This ability to reason locally helps to make code locally more readable. Thus, a highly restricted and constrained class hierarchy can make it possible to easily and confidently make changes within a single class at a low-level for any given method, because it is possible to isolate the interactions along the hard abstraction barrier created by the method and the class.

Functional programmers will often do something similar. The use of static type systems allows for an explicit programming contract between two potentially very distant pieces of code that can be dealt with largely independent of one another because they rely only on the type signature. Coding to the type signature therefore allows one to isolate a given piece of code and reason locally about that code. This makes the code more readable, because it is possible to better understand the behavior of a single piece of code without requiring broad knowledge of the rest of the code base.

This makes mainstream programming paradigms likely to give a high degree of confidence and readability in making localized changes to specific functions/methods in a given ecosystem.

APL programmers also have this sense of micro readability in terms of their equational reasoning of specific lines of code. Often a single line of APL code encapsulates the logic in many lines of code for the same task in another language. Local reasoning for the APL programmer then becomes more a matter of proving equivalence of two algebraically styled lines of code. Whereas in an imperative language one might consider the readability of a single loop (which might cover a couple to 10 or so lines of code as a rule), that same computation in APL might be a single character or a handful of characters in APL. The readability of these given phrases is judged by the structural regularity of the expression, the symmetry and ease of parsing, the degree of nesting, and the consistency of data flow through the code. One may also examine the generality of the expression, where a more general expression will work reliably over a wider range of input shapes and ranks than a less general expression, for which one must consider unusual cases or edge cases.

However, the brevity of APL code and the Macro vs. Micro principle of APL style elevates the discussion of readability from just the micro level to the macro level of architectural and inter-procedural levels. And this is where the APLer argues that mainstream programming languages are not readable.

Posed the question of making a micro-edit to a piece of code, both the APL programmer and the mainstream programmer will likely share a great deal in common with regards to their confidence of making changes. A single line of APL code is well encapsulated and its impact is usually quite readily judged. The abstraction and encapsulation heavy style of mainstream programming generally limits the extent of the impact of a single micro-edit to a relatively small region of code which can be also readily judged and understood by most programmers.

When posed with making a broad, sweeping architectural or macro edit to the code base, however, the picture changes. Here are some example questions of this sort:

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?

These might be classed as refactoring problems, or architectural rewrites, or any number of other things that mess with the code graph. Micro edits are those edits that tend to be thought of as occurring inside of the leaf nodes of the total system graph (which describes the dependencies and relationships between all the various units of the program). Macro edits are those changes that begin to work on what the programmer would generally consider as the edges, or in the removal or addition of new nodes in the graph.

The mainstream programming style completely eschews any attempt to explicate this system graph within the code of the text itself. Instead, documentation, type errors, tooling, diagrams, and other extra-source entities serve to provide the visual model by which programmers reason about these macro connections. Module systems, for instance, are an example. They define local connections, and generally stay out of the way of the specific code within a module, but the relationships between multiple modules is usually not readily apparent (and intentionally so). The emphasis is on local reasoning here. Additional tools, such as JavaDocs, folding editors, IDEs, and so forth, serve as the tools to see the “bigger picture.” If one were to try to examine this by hand in the source text itself, the solution would be to crawl through the various files to identify the edges manually, and attempt to build up the structure in your head.

In good APL programs, however, the graph is kept smaller, because the degree of brevity at the local level permits a great deal of detail to be removed from the graph, essentially lifting up what is considered to be a “leaf node” in the graph. This brevity also means that significant portions of this system graph appear together and directly obvious from within the source code. The APL program, in essence, prioritizes the explication of the macro dependencies of the code, while attempting to suppress or reduce the overall impact of the local elements of the code.

The combination of the use of implicit behaviors, brevity, and structure over names means that generally, the good APL program is more open to broad, sweeping changes and refactoring than the equivalent mainstream program, because the APL programmer is significantly more confident about making macro level changes to the code, because significantly more of the code and the ramifications of making changes to the code, are visible at one time, both on the screen and in the mental system graph.

The result is that the APL programmer feels that APL programs are more readable, not because an APL program can be read at the same “lines per minute” rate as another piece of code, nor because one can quickly “get the general idea,” but instead, because one can more readily and confidently get the “complete picture” to the point where one can make confident changes to the code without worry. One might initially feel like a given page in another language is easier to read, but the APL programmer is more confident in being able to get the whole picture from top to bottom and all around, and importantly, to make changes to that big picture, than is often seen in mainstream programming styles.

# Conclusion

There is not one universally lauded APL programming style. Instead, individual styles arise for specific customer requirements. Styles change depending on who is going to be reading the code, their experience levels, and so forth.

Despite the style of various APL programs varying widely, however, we can find very specific design principles and design patterns that are universally recognized as “good style” across these various types. These 8 patterns generally conflict with prevailing software engineering best practices, and for good reason.

The difficulty in experienced programmers readily acquiring, understanding, and utilizing APL effectively can be partially attributed to this dichotomy of practice between more mainstream languages and APL languages. The emotional, even moral, investment made into traditional design practices can make the acquisition and appreciation of skill in working with the APL paradigm difficult.

Fortunately, this difficulty can be remedied somewhat readily simply by being aware of the causes, and being willing to examine the practice holistically, from a contextually aware position that considered culture, social dynamic, development style, and language design as a unified system that exhibits emergent behaviors which may differ from prevailing wisdom.

While this does not mean shaking old habits is any easier, it at least helps to identify when these habits are hindering an understanding of APL programming paradigms. With this in mind, the productivity and efficiency with which APL, Iverson-style programming languages are being used in real practice can be considered rationally, rather than being dismissed as an aberration and accident of circumstance, even in the face of the difficulty in replicating similar results in other languages.