APL

Is APL Dead?

Is APL dead? Not anymore. Changes in modern hardware, business needs, development methods, and our understanding of programming makes APL in its modern state perhaps more relevant and viable than ever before.

AARON W. HSU
23 APR 2020 • 8 MIN READ

A recent post by Hillel Wayne about influential dead languages (well, mostly dead, he notes) has reminded me that there is a lot about APL that people do not know, and with the obscurity of APL in the wide world of programming, one must be forgiven for thinking of APL in terms of a dead (mostly) language. But this question comes up so much that I think it is worthwhile to examine APL in this light and clarify the understanding of APL in the modern world. Let us examine this question in three parts:

1. What makes a language dead?
2. Why isn't APL dead, and why might it appear dead to some people?
3. What makes APL not just "not dead," but also uniquely alive?

What makes for a dead language? In short, lack of life. But how does this apply to programming languages? The proxy measures of popularity and userbase can help us, because popular languages are never dead. The reason for this is that the languages with many developers and users are

Nonetheless, popularity is not everything. After all, Haskell spent years in obscurity before becoming at least reasonably well known, even if its popularity has never topped the global charts.

Life is creation and movement and potential. A language dies when it ceases to provide value in these dimensions. In other words, when a language no longer has anything to contribute, when it is no longer suitable for creating new ideas, when it no longer keeps pace with the needs of the user, it dies.

Languages that maintain usability while embedding unique or innovative models of thinking and problem solving are much more resilient to losses in popularity. Indeed, most new languages rise from obscurity precisely because of a unique combination of usability and newness/innovation in some way that matters. But languages like APL are a great example of how even great languages may lapse into obscurity due to market forces and failures to deliver value in the right niche. What is the maxim? Change is the only constant. But let's not forget that history repeats itself.

Often, a new language may die relatively quickly, as newness wears off and its essence is absorbed into other languages. The less fundamental uniqueness and value in a language, the faster a language can die. That's why we see so many languages that might be legacy still kicking in certain niches: their value is insanely high, or perceived to be so in those areas.

The truth is, APL very nearly did die.

APL thrived in the 70's and 80's as an ultra-high productivity language for solving a problem rapidly and accurately. It gave access to the mainframes that others could not. Universities' Computer Science departments actually found it too easy, and many departments embracing the software engineering model or formal methods model a la Dijkstra ran a heavy campaign against APL.

and Object-oriented Programming (tm), swallowed up any alternatives. The APL vendors did not adequately respond and the fashion of the day meant that the user-focused, rapid development of APL became persona non grata in short order. The growing popularity of spreadsheets meant APL was fighting a two-front war and it lost ground (mindshare) quickly. Efforts such as J failed to achieve sufficient marketshare to turn the tide. Despite this, APL continued to provide very high value to its existing customers, and Dyalog continued to invest not only in maintaining the language, but evolving and growing it to remain relevant and in many ways still ahead of the times (by about 20 years it seems). The result is a very solid language with some of the best tooling out there. The late great John Scholes introduced functional programming into the language at least a decade before anyone in mainstream programming (including John himself) realized how relevant it would become.

The willingness of the APL community to continue chasing what a "tool of thought" means in the fast-paced and practical world of business helped to insulate APL from too much fad programming. Ironically, it has also positioned APL in this decade to emerge like a phoenix from obscurity with a mature, fast, and experienced platform of tools and techniques for solving today's problems, because a few things happened while APL was "away" that changed the value proposition immensely.

Not only has APL continued to evolve and improve over the years, the rest of the programming community has been learning lessons the hard way which only serve to make APL more relevant today than ever before.

It turns out, OOP and software engineering are not the Godsends we thought them to be, particularly in the presence of rapid change. The compositional Lego block approach to programming has proven highly effective, both in functional programming as well as in the library-centric Python models. This was discussed years ago in MIT's famous shift away from its 6.001 curriculum of the day (decade?).

practice. Methods like the Agile Method and the like emphasize code and a close, efficient communication with the user of a system. The methods used in many APL shops have been using and mastering "agile" in very tough corporate conditions since long before it was a blip on the radar of "hot" terms. Not only is there a cohesive culture around it, APL is one of the only languages intentionally designed around such principles. In some cases, users and developers pair program together, and the source is their common language.

We have also had some time to learn the pitfalls of super compilation and "Super High-level Declarative Programming" with the sometimes insane performance hits in these systems either in usability or runtime performance or, sometimes, both. A classic example is the now standard practice of in-memory data stores instead of using an RDBMS for everything. APL applications have been maximizing in-memory database systems and high-performance queries for decades now, and the out of the box language support for them and high-performance serialization support is second to none.

In a post-Ruby on Rails, big data, data-driven world, it turns out that performance does matter, long term scalability matters, but productivity also matters a great deal. And for many, the lesson has been, you cannot have it all. But there is another element to this. In a world of Lego blocks, if you can be the block designer and builder, you have tremendous power and advantage compared to people who can only slap together the big blocks/components of others.

When it comes to performance and productivity, no other general purpose language comes close nor has as many lessons in doing just that. This is because big picture performance gains matter most as long as you can manage the low-level performance. As I show in my thesis, APL is uniquely capable of very high-level, direct solutions that are extremely concise, yet maintaining mechanical sympathy and near byte-level available today. In an ironic twist, the very things that turned off University Professors in the past, notably Dijkstra, are the same things that make it so powerful today. Its high-level, compositional, functionally oriented, mechanically sympathetic features mean that a pure, "from scratch" solution is often less code than the library call and boilerplate from another language, and much more flexible, adaptable, stable/bit-rot resistant, compositional, and simple.

But what about those pesky symbols? Well, ask yourself what the trend in PL is these days. How many people embrace COBOL's keyword-heavy approach? We live in an international world full of non-English scripts. You know the closest thing to an international language we have? Math notation, and heavy on the symbols. APL is designed as executable math notation. Now that Unicode and modern fonts and operating systems make it possible to use and type any symbols we want, we need not be constrained by limitations imposed on us by 90's technical restrictions. Here again, APL has proven itself far ahead of the curve. (Note: I get asked about symbols a lot, and I answer that question in just about every Q&A session I do after my talks, so if you want, browse some of my talks and you should be able to find my ramblings about symbols in more depth.)

Of course, it is not all sunshine and roses. The near death of APL in the 90's still weighs heavy on the community. Most users today are young enthusiasts or old greybeards, with precious little in the middle. It is still a small, albeit passionate (sometimes too passionate), community. Many of its users are simply not present in the Internet channels so precious and vital to the view of the world for many of us.

Sometimes this is because they are under NDA's and cannot have a presence or talk about their work. Others simply have better things to do and would not bother. Still others may not love to share, but they do not stay connected with the broader community of programmers, because they are users of APL for their passion somewhere else, and they would

The end result is a severe iceberg effect for APL users that only compounds. Fortunately, things are looking up, with small user groups showing up throughout the world and places like the APL Orchard providing a chance to hang out with a passionate subset of the community who enjoy helping new users.

Still, APL is a small community no matter how you slice it, but do not let that deter you from joining it. APL's community is still largely human, so please reach out to someone at Dyalog or myself or Adam at the APL Orchard with questions you may have and we can help answer your questions and point you to the right documentation.

Is APL dead? Not anymore. Changes in modern hardware, business needs, development methods, and our understanding of programming makes APL in its modern state perhaps more relevant and viable than ever before.

Here are some things that make APL a huge value-add in the modern space:

- It is extremely high-level.
- Solutions in many programming domains are simple to express in direct APL without complex libraries and dependency hell.
- APL is mechanically sympathetic to modern GPU and CPU architectures, leading to fast code that does not need to be hard to read or write or port.
- Dyalog APL comes with extensive tooling, examples, documentation, integrations, visualization, debugging, ingest tools/functions, libraries, serialization, and platform support.
- Idiomatic APL admits automatic or near automatic complexity analysis and is performance predictable across many architectures.

- APL idioms are often portable across domains, meaning that techniques and strategies you learn apply in more areas (versus a library approach that restricts how applicable a given function is).
- More concise APL enables reductions in boilerplate and a more holistic analysis of systems for better whole architecture optimization.
- APL keeps a high-level focus, helping you to avoid premature optimization.
- APL is a language designed from the ground up on principles aligned with "agile" styles of programming.
- APL programs can be fundamentally simpler and more transparent, leading to better understanding the problem and better solutions.
- APL enables the benefits of data flow, functional, Lego style programming without needing to resort to large and complex sets of black-box libraries that limit your freedom and understanding as well as restrict your cross-domain transfer.
- APL is particularly suited for innovative spaces where simply leveraging off the shelf solutions can lead to sub-optimal results, while APL can allow domain experts to iterate on novel and existing solutions together in a compositional way to address complex and even not-so-complex domains with aplomb.
- APL is easy to integrate into existing or new codebases through extensive integration support for REST, JSON, .NET, Python, C/C++, shared objects, TCP, CSV, Office, HTTP, Docker, COM, R, and many other interfaces.
- APL enables you to have a single codebase that is performance portable across both CPU and GPU architectures.
- Many more common tech stacks can be fully and directly implemented in pure APL without any libraries needs, such as

as short or shorter than the library calls to the existing implementations.

When it comes right down to it, APL offers a unique platform in Dyalog that can often deliver the trifecta of increased runtime performance and control while reducing long term maintenance costs and increasing developer productivity on the same codebase when compared to common platforms in vogue today. In my book, such a radically value-driven language is far from dead, it's back from a long hiatus and here to show the youngsters a thing or three.

MORE IN **APL**

Dyalog Tooling Resources List (LambdaConf Talk 2020)
22 Jul 2020 – 1 min read

Configuring your APL keyboard on Linux
28 Nov 2019 – 1 min read

Installing Dyalog APL on Nix
28 Nov 2019 – 1 min read

See all 85 posts →

PRODUCTIVITY

Productivity Systems: 7 Principles of Evaluation

Handwritten VersionProductivity is a hot topic. Who knows how much time is potentially lost trying out different systems. It is fun to experiment and see what you may like, but there is just so much out there, and while some guides try to help

AARON W. HSU
6 MAY 2020 • 4 MIN READ

APL

Configuring your APL keyboard on Linux

My preferred APL keyboard setup on Linux is as follows: setxkbmap -layout us,apl -variant ,dyalog -option grp:lswitchThis will use the left-Alt key as a Mode switch key to allow you to type APL symbols by holding down Alt and hitting the correct

AARON W. HSU
28 NOV 2019 • 1 MIN READ