A Constructionist Journey: 42 years with APL - "A Programming Language”

Jean Rohmer, *jean.rohmer@fr.thalesgroup.com*
Thales RD 128, 91767 Palaiseau France

Abstract

This paper revisits the author’s long experience in using APL in various domains of research and industry. We explain how APL can be seen as a construction tool, rather than as a programming language. APL features are compared in this respect to current languages and methods of software engineering and object-oriented programming and modelling.

We advocate that a constructionist approach combined with a “liberal” tool like APL which empowers programmers yields better results, and makes difficult things possible. A detailed analysis is made to stress weak points of classical object-oriented languages, compared to languages like APL.

We emphasize the notion of architecture by layers of tools and constructions, and show why APL is a tool of choice in such an approach. We examine the managerial and cultural obstacles to be removed for a broad adoption of constructive programming.

Keywords

APL, Languages Interpretation, Software Engineering, Programmers Empowerment, Object Oriented Programming

INTRODUCTION

I recently attended a presentation by James Clayson, where he explained how his students in Humanities, at the American University of Paris, were able to program a computer after only a few weeks, and managed to do sophisticated things like painting landscapes of trees and forests, or generating pictures mimicking the style of contemporary masters. He explained that these students used the programming language Logo. These paintings were extremely impressive. During James Clayson’s conference, I discovered the concept of “constructionism” and the way he applies it with Logo, and I realized that my way of programming might belong to constructionism.

Personally, I am doing a lot of programming since 1967, in various scientific and industrial contexts. I experienced many different programming languages, but, when I have the choice, I prefer the one named “APL” (A Programming Language).

In 2006, I was visiting DFKI in Kaiserslautern, the main academic institution in Germany for Artificial Intelligence, and, during my presentation, I made a live demo of IDELIANCE, a comprehensive multi-users system to manage Knowledge Bases and Semantic Networks. Suddenly a bug occurred. Within 30 seconds, I could examine the guilty part of the system, diagnose, fix the bug, and resume the demonstration.

This was visibly the most impressive point of my presentation. None of the distinguished scientists in the audience could ever imagine such a “live” recovery from this kind of situation. They asked me the secret behind: I answered the truth: “APL”.

HOW I BECAME A RECURSIVE CONSTRUCTOR

I discovered APL in 1968 in Grenoble, when I was a student at ENSIMAG, a Computer Science School.

APL arrived there with the IBM 360-67 computer, inside its CP/CMS Operating System, the first large time-sharing OS with virtual memory and virtual machines.

It was the blessed times where the policies of standardization had not their present virulence, and where many interesting languages flourished all around the world.

APL was presented to us along with LISP, Algol 60, Algol 68, PL/1, GPSS, AED, Scratchpad, SNOBOL, … I do not mention here FORTRAN and COBOL, the true standards of theses times, since our founder and director Jean Kuntzmann wisely decided ***not to*** teach them, ***since they were standards*** *…("you will learn them yourself later in the industry if you need them")*

N.B. Today the situation is ***exactly the opposite***: academic institutions teach ***only*** standard languages.

In 1974, I joined a team at INRIA, the national French research institute in computer science, which was building an « APL machine », i.e. a processor specialized in processing APL programs.

For that purpose, we had not only to know how to use ***APL as a construction tool***, but to imagine how to ***design a machine which could construct APL***.

"A machine which constructs APL which constructs cuboids objects –as explained in the next paragraph"

At this point, we must remember that APL was initially proposed by Kenneth Iverson in the late 50's as a notation to help his IBM colleagues describing computer hardware, in other words, to construct machines. Indeed, APL was the tool we chose at INRIA, to describe the logics of our APL machine.

We used APL to construct a machine which constructs APL (which constructs cuboids)

In computer science, this kind of recursion is known as "layers of interpretation": a language at level N is used to construct a machine which understands another language at level N+1.

Language 1 -> construction 1 -> Language 2 -> construction 2 – Language 3 -> …

I conjecture that understanding and mastering several layers of construction is key to become a good constructor.

I conjecture also that the best way to teach and to understand a language L, is to

construct it from another language L-1

construct from it another language L+1

I then became a "recursive constructor".

Later, every time I needed to build some software, I used APL.

To write my thesis of « Docteur ès Sciences », I first wrote a text editor in APL, because IBM just launched in 1975 the 5100 computer, the first « portable » personal computer: 64 K memory, 55 (fifty-five) pounds, 20 000 dollars, 1.9 MHZ Clock)

In fact, APL was the main software environment (with Visual Basic, but most of 5100 were bought for APL). There was no other Operating System than APL itself.

The first personal portable computer was an APL machine.

I then joined Groupe Bull Corporate Research Centre, in Louveciennes, in 1980. I spent some months with a team building a parallel supercomputer. They were painfully using FORTRAN to simulate the hardware. I introduced them with APL. They immediately changed their mind and continued the simulation with APL. They needed only some hours of hands-on exercises. They were constructors and immediately felt the “constructivism inside”.

They eventually built an impressive hardware prototype, and APL was key in this achievement.

A few months later, an important event happened in the world of programming languages: Japanese Ministry of Industry disclosed its “Fifth Generation Computing” program, aiming at developing a new kind of computers based on Artificial Intelligence, and totally relying on the PROLOG language. PROLOG was invented in 1972, in Marseille, by Alain Colmerauer, a former student of Ensimag.

Prolog was a true revolution: programming was presented as a process of solving logical equations on symbols. Colmerauer invented Prolog because he has to translate bilingual meteorological bulletins in Quebec. Remember that Colmerauer was not polluted by standards during his education in Grenoble.

Prolog was very difficult to understand for scientists like me, used to other languages. It took me several months to understand Prolog, and the definite way I found was to ***construct in APL my own Prolog Machine.***

APL was my Language L and Prolog my Language L+1

I was impressed by the power of Prolog, and I felt empowered by my construction of a Prolog machine with APL as a tool. I advocated for the creation of an Artificial Intelligence group inside Bull, and I had the opportunity to create and lead this group, known as CEDIAG, and grow it up to 200 people worldwide, from 1981 to 1994. We did research, products and many significant operational business applications in natural language, experts systems and constraint programming.

In 1993, Bull suffered a severe crisis, and most of AI group people, including myself, were invited to leave the company in 1994. In the small world of AI, became what was called the “AI Winter”

AI techniques allowed us to build a new generation of applications, which solved real-life problems, which were out of reach of classical software engineering, like monitoring complex industrial processes, crisis management, application of administrative regulations.

However, achieving such applications required sophisticated tools, expensive machines, skilled engineers, and a deep involvement of customers. In other words, this field of programming - known as **knowledge engineering**- was too demanding, was not ***sustainable***.

These limits gave me the idea of a more ecological AI: a ***personal*** one. Could we design a tool which would enable individuals to become programmers of their own knowledge?

But I was alone at home, without system engineers, C++ programmers and their costly Unix graphical workstations, up to 100.000 dollars each! Again, the solution was APL.

I started on my tiny portable computer to write the first line of APL code of a system I boldly named “Intelligence Amplifier”. My idea was that, when performance problems would arise, I could find enough money to rewrite the code in a more standard and serious language like C++.

But, first, performance problems never arose, and, second, I never found enough money to rewrite anything.

I started a small company, and years after years, we were up to 6 people there, and the initial code became a comprehensive multi-users knowledge management system, on Internet and

Intranet. It was marketed under the name IDELIANCE, to large companies like l'Oréal, Air Liquide, Merck, and French Military Intelligence. Now this tool is with Thales Group, and is one of the few "intelligent" systems which has been used in military operations abroad.

The main reason for this achievement were the constructionist properties of APL:

self-sufficient: everything in Ideliance is programmed in APL; we use no database, no application manager, no HTML generator; even the multitasking scheduler is written in APL

easy to learn by doing: the new Ideliance developers were always operational in APL after two weeks, without any formal training. I just showed them what the Ideliance code was, and how they could extend it.

Power of productivity: the language is so concise and powerful that significant new features can always be prototyped in a few days, letting you observe, react, and adapt, -by throwing away everything and recoding if necessary

Transparency: everything is visible and touchable at any time: data and code. That is why I could fix a bug and continue during my presentation in DFKI

Generally, large software engineering projects are developed with too much money and not enough time. By contrast, Ideliance was constructed year after year, with not enough money but enough time. A sort of elegant and continuous epitaxy.

WHY APL IS A GOOD CONSTRUCTION TOOL

N.B. this paragraph is not an introduction to APL. No APL code is shown; we describe and analyse some properties of APL.

APL is good at something apparently[[abstract and virtual:

"construct rows, square or rectangular arrays, cubes or cuboids of numbers or characters"

(cuboid is a shorter word for rectangular parallelepiped)

This seems abstract, but:

- A word, a sentence is a row of characters
- A computer screen is an array of pixels of colours
- A colour is a row of 3 elementary colours
- A computer hard-disk is a cuboid of characters

With APL, you can build such objects, which are organized sets of elements.

More, you can create hyper cubes, hyper cuboids of any number of dimensions.

In the APL "workspace", each of these objects has the name and the content you gave them. Then APL lets you assemble these objects:

- You can add a row to an array, an array to an array, if the adjacency dimension is the same
- If you stack arrays of same dimension, you get a cuboid.
- You can put side-by-side two cubes if they have the same dimension: you get a cuboid.
- You can take a slice of an objects to get smaller ones.

- For instance, take the first three positions in a row, or the last 5 ones, or the positions numbered 3, 5 and 7 (this last operation yields a row of three elements)

At any time, you know the **list** and **name** of the objects inside your workspace, you know their dimension, you can **see** their content, and you can **modify** this content.

Any elementary operation on numbers and characters is generalized to these multidimensional cuboids, if they have the same dimension: the sum of two cubes is a cube. APL provides operators which construct new objects from old ones.

Classical mathematical operators are provided, like arithmetic operations, generalised matrix products, union, intersection, projection, comparison, permutation, rotation …

Less classical operators let you perform operations which transform the dimension of the objects (like flattening, putting side-to-side, expanding …)

A very special point is that APL represents most of these operators by a ***new*** letter of a ***new*** alphabet, invented by Ken Iverson. APL is a language written with special letters expressing operations on complex objects.

You compute new objects exactly as you compute new numbers with usual arithmetic operators. An APL expression looks like an arithmetic expression, with arrays or cubes instead of numbers, and juxtaposition or slicing instead of additions and multiplications. APL expressions are very concise, you can understand what they do just by looking at them.

Note that, in the same way young children learn arithmetic without any reference to programming, we have not yet used the word "programming" about APL.

APL lets you construct objects –rows, arrays, cubes, cuboids, hyper cuboids- in a multidimensional world, each elementary point of theses objects being a number or a character. In other words, APL objects are **sets** of elementary information, and APL operators operate on ***sets***. Whereas in other programming languages you have to program as many loops as you have dimensions to repeatedly execute each elementary operator on elementary information.

Above this powerful set of objects, operators and expressions, APL provides the classical features of programming languages:

- Assign the result of the calculation of an expression to an object
- Execute a sequence or loop of such assignments
- Define a function as the performance of a sequence
- Reuse this function as an operator in other expressions

Another characteristic of APL is that it works like a pocket calculator: you type some expressions, and it instantly creates and computes objects.

When you want to do something, you can do it immediately, and you can keep track of your work, to replay or refine it later.

The consequence is that you become a mighty constructor of objects:

- you can construct
- you can see what you construct
- you can see how you construct

If you can construct, you construct. Using APL empowers you. And when you ***can*** construct, when you ***do*** construct, you can ***think***, you do ***think*** differently. Doing and thinking work in a close loop.

This idea that doing and thinking could be the same in computer programming is unfortunately in total opposition with the current ***doxa*** of software engineering, which strictly isolate designing and doing.

About the etymology of "construction":

It comes from latin "strues", which means "heap". And in latin "structor" means "mason".

A recent theory about Egyptian pyramids construction, proposed by architect Pierre Crozat, is that they were built as heaps: a new layer of stones was laid on a pyramid "N" , simply by climbing it, producing pyramid "N+1". ***The pyramid was its own ramp.***

WHY APL IS SO UNKNOWN

If qualities of APL are so evident, why is it nearly totally unknown? Reasons are deep, they do not come simply from APL characteristics, they are ***cultural***.

When I started programming in the late 60's, ***programming*** was the most prestigious thing you could do with a computer. ***Using computers*** was reserved to clerks, female clerks who were employed for inputting data with punched cards, and male ones for loading and unloading cards racks and tapes, sorting printer listings, night and day. High-level white collars never saw a computer, except through the glass walls which let visitors gaze at the temple of the computer room, with great priests bustling around. After 40 years, things went exactly the other way round.

High-level management is today surrounded by the computer screens of their office and their smartphones. And programmers have become second-class corporate citizens.

High-level management have replaced clerks as major users –in the sense of "hands-on"- of computers. Programmers are relegated to invisible subcontractors, and programming is considered as a dangerous activity.

In fact, our technical civilization has still to understand what programming is. Industry and Academy made the choice of trying to reduce programming to engineering. But building an information system is quite different from building a bridge. (We should think of how constructionism may differ when applied to civil engineering or to computer programming).

Software engineering is a contradiction in terms.

Between management and programmers, engineering interposes **methods and standards**. The idea is to produce a cascade of deterministic transformations which will start from an "expression of needs" and end with an "executable machine code". This does not work, but the reaction of software engineering is to say "this is the proof that we need still more methods and standards".

With this culture in mind, it is clear that a tool like APL which empowers programmers is heretic and subject to the most severe inquisition. This has reached the point that most people cannot imagine that programming is a rich intellectual activity.

When I explain my vision of systems like Ideliance, people are in general very positive, until the killer question comes: "But who programs all theses nice things for you ?".

The hypothesis that Ideliance was possible **only** because I elaborated the concept **while** constructing it **with** an empowering tool like APL is just unthinkable.

In such circumstances, I incline to answer: **“Ich bin ein Programmer!”**

One of the key foundation of software engineering is that we must prevent programmers from making mistakes. It is an application of the precautionary principle.

Management believe that, since there exist methods and standards, all problems are solved by advance. In their eyes, methods and standards depreciate the work of their employees, which become interchangeable.

The true question should be to compare the mistakes induced by an empowered programmer and the mistakes resulting of the specification cascades of software engineering.

Note. Another reason why APL is not popular today is the **NIN** syndrome **: « Not Invented Now ! »**

COMPARING APL WITH OBJECT ORIENTED MODELLING AND PROGRAMMING

Object-Oriented Programming, through languages like C++ and Java, and Object-Oriented Modelling, with formalisms like UML, are today the official and unique way of programming taught in schools and universities, and the only practice accepted by Industry. This ***pensée unique*** has many drawbacks.

Indeed, methods and standards cannot be harmful by themselves. The problem is that they are currently proposed as ***the*** solution, ***independently of the tools*** –i.e. the languages- to which they are applied. And, precisely, they are applied today to languages like Java and C++, which are, in our opinion, very bad choices, because they are crippling tools rather than empowering ones.

We can even ask whether current methods are not around just to try to fight the crippling due to low level languages, like crutches.

Recently I listened to a conversation between two programmers in a bar. They just were exiting a course on methodology, namely “design patterns”. Verbatim: “We need not instructors or patterns telling us how to do our job, we need tools powerful enough to do our job the way we want”. I was happy for them, their clearness had survived the course.

We can fear that some managers think good methods achieve good results with bad tools.

Why are languages like Java and C++ bad construction tools ? Here are some reasons:

- R1: They manipulate very low level objects, and one element at a time: add two numbers, compare two characters, index elements one by one in a list.
- R2: They do not trust programmers: before writing any operation, programmers must make a declaration of what they want to do, when and how. They call it “typing”. A sort of bureaucracy, where you have to declare by advance everything you intend to do in the future, and conform to it.
- R3: When you have finished to “type” –in both senses- your program, it escapes you: the bureaucracy takes your program, checks if you obeyed your declarations, then “compiles” it, links it with other programs, runs the program
- R4: to understand whether your program is running as you expect, you have to add extra –low level- instructions to let you visualize, then imagine, its behaviour

- R5: since all the above-mentioned points are very painful, you try to spend a lot of time in minimizing the number of lines you write, by organizing your code in a modular way, by abstracting and reusing some parts (with, at each step, the obligation to declare your abstraction to the bureaucracy)
- R6: since all the above-mentioned points are very painful, you in fact do not use directly the programming language, but a "programming environment", which you must learn to use in supplement to the language itself (like IBM Eclipse environment)
- R7: the documentation which explains how to perform the above-mentioned points amounts to several thousands of pages, and much more if you include the variants and community tricks found on the Internet
- R8: since all the above-mentioned points are very painful, you are tempted to program less and less, (remember, it was the objective of the Methodists who do not trust you). You are invited to reuse programs written by others. They call it "libraries". But libraries mean still more thick books to retrieve, read and understand (see R7), and still more bureaucracy to declare how to use features typed and declared elsewhere by others.

What is by contrast the constructivist philosophy of APL ?

- R1: manipulate very high level objects, yet simple to understand, set-oriented, mathematics-oriented, with powerful operators
- R2: trust programmers: let them assemble objects as they want. Adopt a **liberal** rather than a **carceral** attitude towards programming. When programmers feel more responsible, they finally make less mistakes
- R3, R4: programs are interpreted instead of being compiled. High-level objects –cuboids- are visible at any time. Concise extra code can be written and executed directly at any time to experiment the current status of your objects
- R5: since code is usually 10 times shorter, since operators are powerful, you need not so many intermediate abstraction layers which hide what is behind: you just read what your code is and does
- R6: APL comes with its own dedicated environment which optimizes the above-mentioned points
- R7: a few hundreds of pages of documentation are enough, and more important, if you need to understand the behaviour of an operator, you just try it online
- R8: the more you program, the better you master the tool, the faster you program, the better you construct what you want. It remains feasible to go down to the very lines of code. What you have constructed is concise, its semantics is clear to somebody who would like to reuse it directly, or –better- transform it.

CONCLUSION

By writing this paper, I constructed some understanding of what I did with APL during the last 42 years.

Complex systems construction is difficult. A layered approach may help. Each layer must bring its own added value.

A layer is constructed with two things:

- a tool T
- a construction C, performed with T

The result of C is to yield a tool T+1, with which construction C+1 will be performed, yielding tool T+2, etc …

If we are in the following configuration:

T-1 -> C -> T -> C+1 -> T+1

Tool T has an added value, a ***raison d'être*** , if the sum of complexity of constructions C and C+1 is smaller than what would be the complexity of constructing T+1 directly from T-1.

APL is a good constructionist tool T because:

- It provides high-level set-oriented objects and operations (cuboids) to be used by C+1
- Its objects and operations embed a direct indexing logic which easily matches with low-level features used by T-1 tools, closer to the Von Neumann architecture

A good tool T is useful only if C+1 actually follows constructionism principles, among which freedom rather than methods, knowing by doing rather than doing after knowing.

Indeed, constructionism is not an enemy of knowledge and methods.

A recent TV document showed a wooden shipbuilder in the Syrian Island of Arwad. He explained that he builds **alone** 50 feet-long ships. He said: "I build them fast, because I use no blueprint, this way I do not loose time watching at blueprints. We are doing like this here for 4000 years"

We also must have constantly in mind the motto of Leonardo da Vinci, the prince of constructionists: **"Ostinato Rigore"**

Acknowkedgement

To the company Dyalog Ltd, www.dyalog.com , the company who produces and sells Dyalog APL since 1983