

APL in Research: Prototyping Software

Santiago Núñez-Corrales, Ph.D.

Quantum Lead, National Center for Supercomputing Applications


Faculty Affiliate, Illinois Quantum Science and Technology Center

Senior Personnel, Hybrid Quantum Architectures and Networks

Faculty Affiliate, Center for Global Studies

Faculty Affiliate, Illinois Informatics

nunezco2@illinois.edu

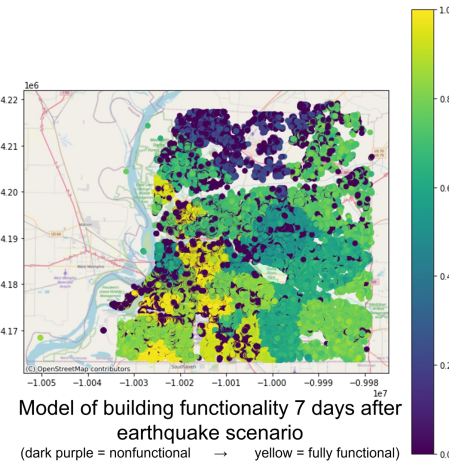
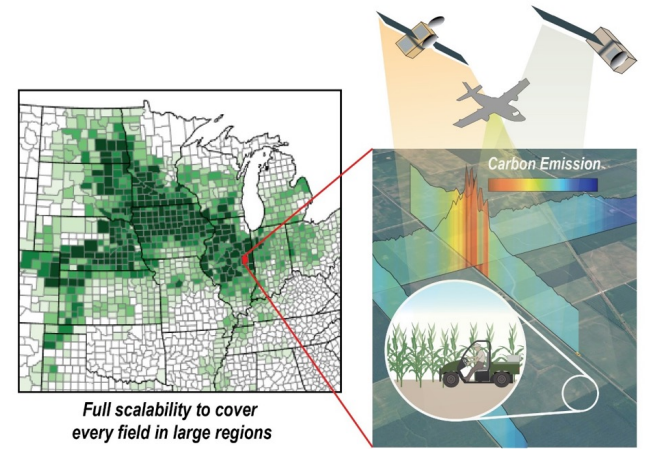
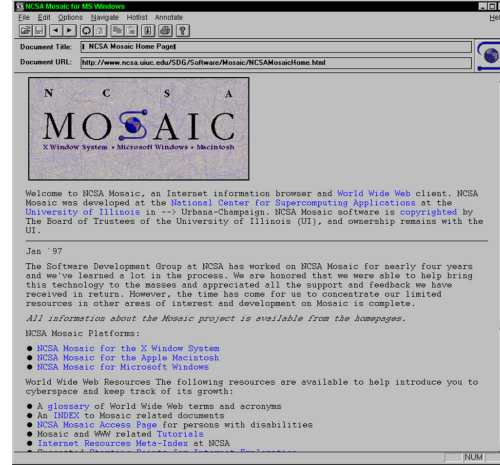
 @snunezcr



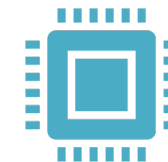
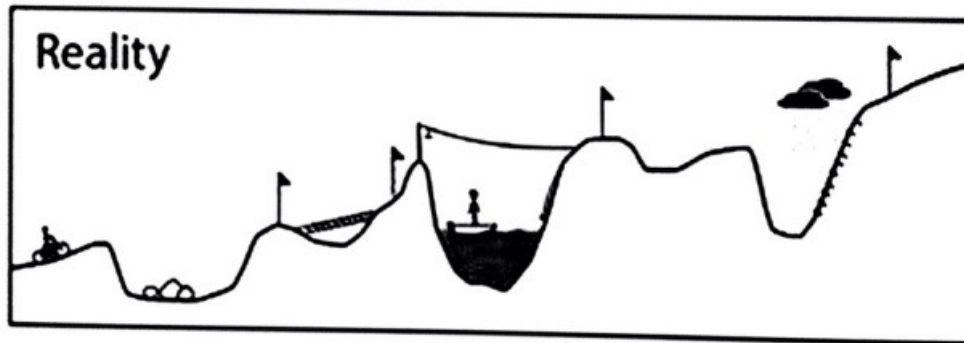
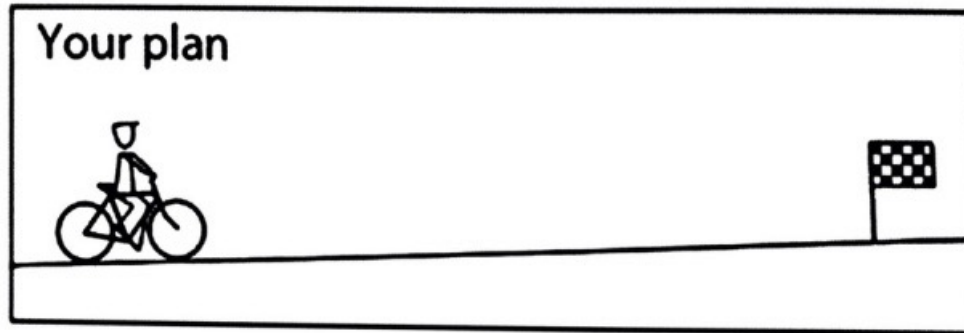
**National Center for
Supercomputing Applications**

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Who we are



Research software is hard



Source: <https://wonderingaround.me/2013/11/13/your-plan-vs-reality/>



...SCIENTISTS AND THEIR SOFTWARE

A survey of nearly 2,000 researchers showed how coding has become an important part of the research toolkit, but it also revealed some potential problems.

> **45%** said scientists spend more time today developing software than five years ago."

> **38%** of scientists spend at least one fifth of their time developing software.

> Only **47%** of scientists have a good understanding of software testing.

> Only **34%** of scientists think that formal training in developing software is important.

Merali, Z. (2010). Error: why scientific programming does not compute. *Nature*, 467(7317), 775-777.

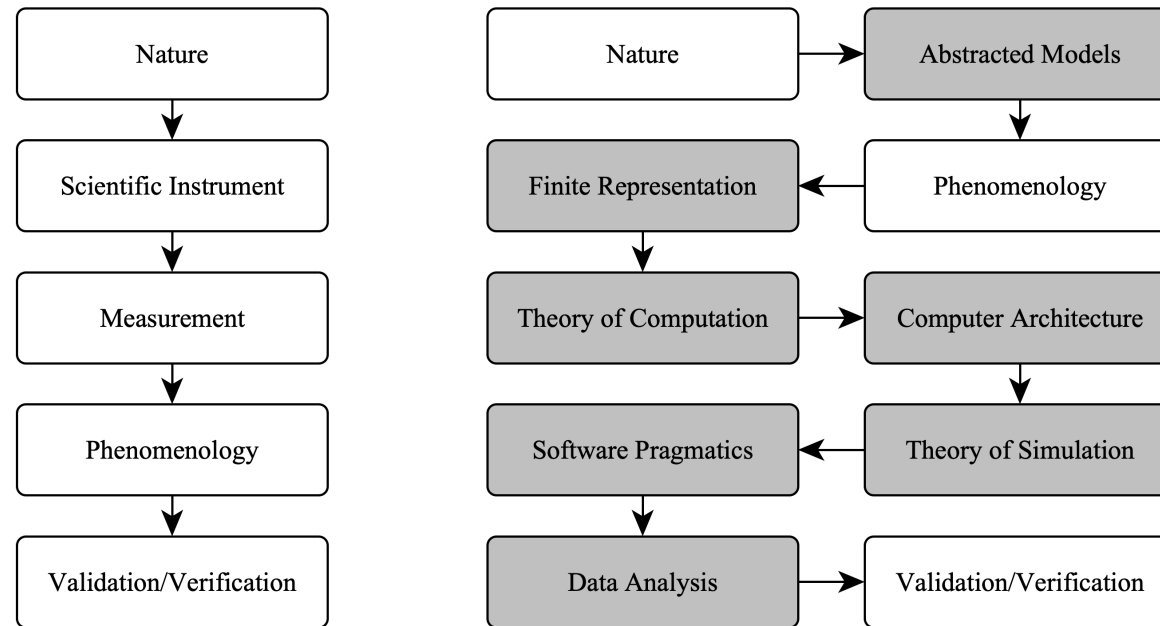
Revolutionizing Science and Engineering Through Cyberinfrastructure:

Report of the National Science Foundation
Blue-Ribbon Advisory Panel on
Cyberinfrastructure

January 2003

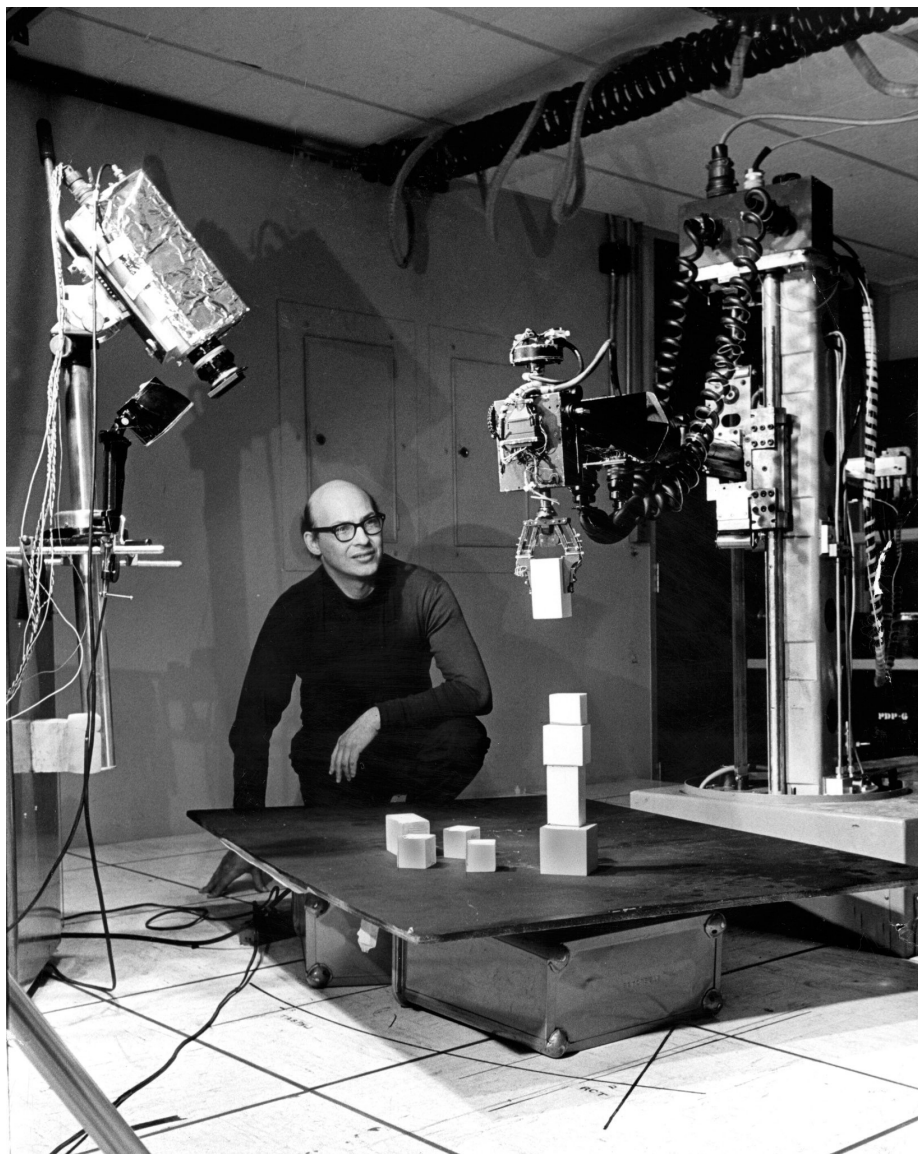


Research software is hard



Núñez-Corrales, S. Dissecting Computational Reproducibility: Fundamental Challenges. *Under review (PNAS)*.





Programming is a good medium for expressing poorly understood and sloppily formulated ideas

Minsky, M., 1967. Why programming is a good medium for expressing poorly understood and sloppily formulated ideas. *Design and Planning II - Computers in Design and Communication*, pp.120-125.



Research Software Engineering

- Scientific statements → software packages
- Our value proposition:
 - no need to learn a whole new discipline (software engineering)
 - accurate translation between science and programs
 - lower project risk with minimal technical debt
- *“Some problems are so complex that you have to be highly intelligent and well informed just to be undecided about them.”*
 - Laurence J. Peter



Research Software Prototyping



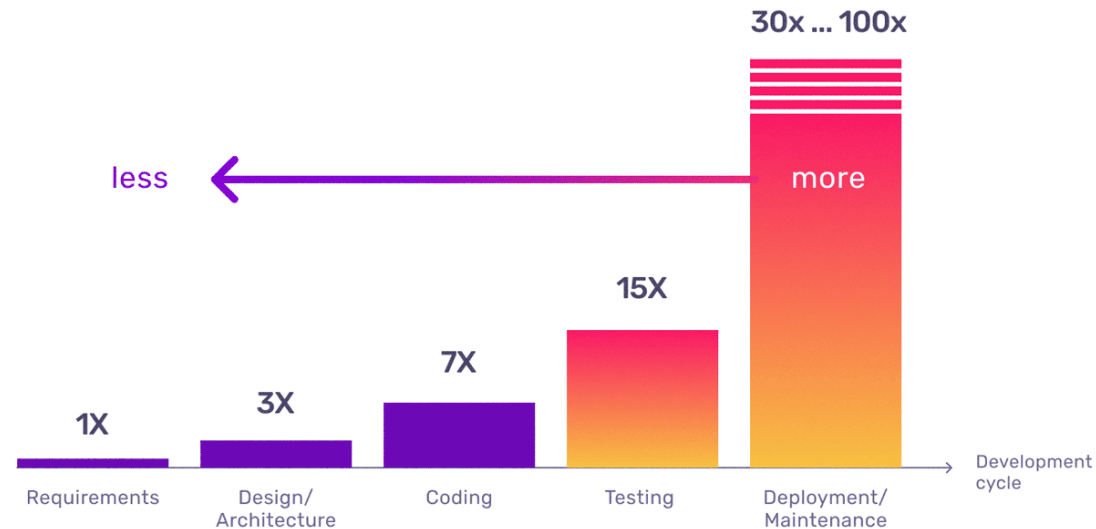
Developing software systems that implement only the essentials to understand a research problem.

A pliable medium for software experimentation to reduce future risk and cost



Bugs = \$\$\$ × time lost

Cost of Defects



The more time we save your team, the more time they have to find bugs sooner.

That Saves Money

Source: <https://www.functionize.com/blog/the-cost-of-finding-bugs-later-in-the-sdlc>

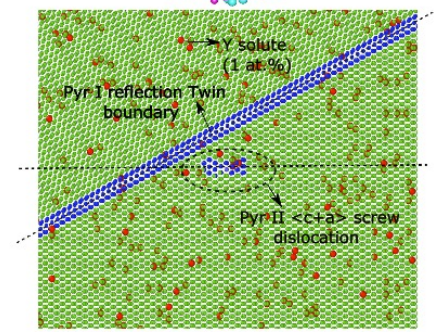
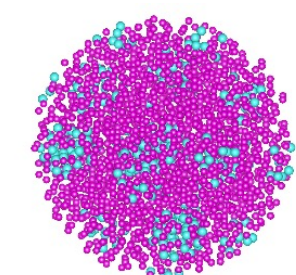


Errors can be **really** expensive

| | Work | Basis | Versatility AE/PSP | Benchmark System | Machine (CPU cores/GPUs) | Wall time (mins) | PFLOPS (% of peak) |
|------------------|---------------------------------|----------|-----------------------|--|--|------------------------|--------------------------------|
| Level 1 | RSDFT [31] (2011) | FD | PSP | Si nanowire 107K atoms, 430K e ⁻ | K (450K cores) | 73.6 / SCF | 7.1 (43.6%) |
| | QBox [23] (2008) | PW | PSP | Mo (1K atoms, 12K e ⁻) \times 8 <i>k</i> -pts [†] | BlueGene/L (125K cores) | 8.8 / SCF | 0.2 (56.5%) |
| Level 2 | DFT-FE [6] (2019) | FE | AE/PSP | Mg dislocation 10K atoms, 100K e ⁻ | Summit (22,800 GPUs) | 2.4 / SCF | 46 (27.8%) |
| | PARSEC [30] (2023) | FD | PSP | Si nanocluster 100K atoms, 400K e ⁻ | Frontera (115K cores) | 2,808 / GS | - |
| Level 3 | Hybrid DFT, ACE [38] (2017) | PW | PSP | Si bulk 4,096 atoms, 16K e ⁻ | NERSC Cori-KNL (8K cores) | 30 / SCF | - |
| Level 4 & beyond | QMCPACK [20] (2018) | PW | PSP | NiO supercell 128 atoms, 1,536 e ⁻ | Titan (18000 GPUs) | 294.7 / GS | - |
| | QMCPACK [19] (2020) | PW | PSP | NiO supercell 512 atoms, 6,144 e ⁻ | - | - | - |
| | LNO-CCSD(T) [17] (2019) | Gaussian | AE | Lipid transfer protein 1,023 atoms, 3,980 e ⁻ | Intel Xeon PC (6 cores) | 26,064 / GS | - |
| | iFCI, QChem [15] (2021) | Gaussian | AE | Transition metal complex 47 atoms, 192 e ⁻ | - | - | - |
| | MCSCF, NWChem [14] (2017) | Gaussian | AE | Cr trimer 3 atoms, 72 e ⁻ | Cori Haswell (2048 cores) | 57.8 / SCF | - |
| | This Work DFT-FE-MLXC | FE | AE/PSP | Extended defects in Mg-Y alloy I. (36K atoms, 76K e ⁻) \times 4 <i>k</i> -pts [†] II. (74K atoms, 155K e ⁻) \times 4 <i>k</i> -pts [†] | Frontier (19,200 GCDs) (64,000 GCDs) | 3.7 / SCF 8.6 / SCF | 226.3 (49.3%) 659.7 (43.1%) |

1.95
days

18.1
days

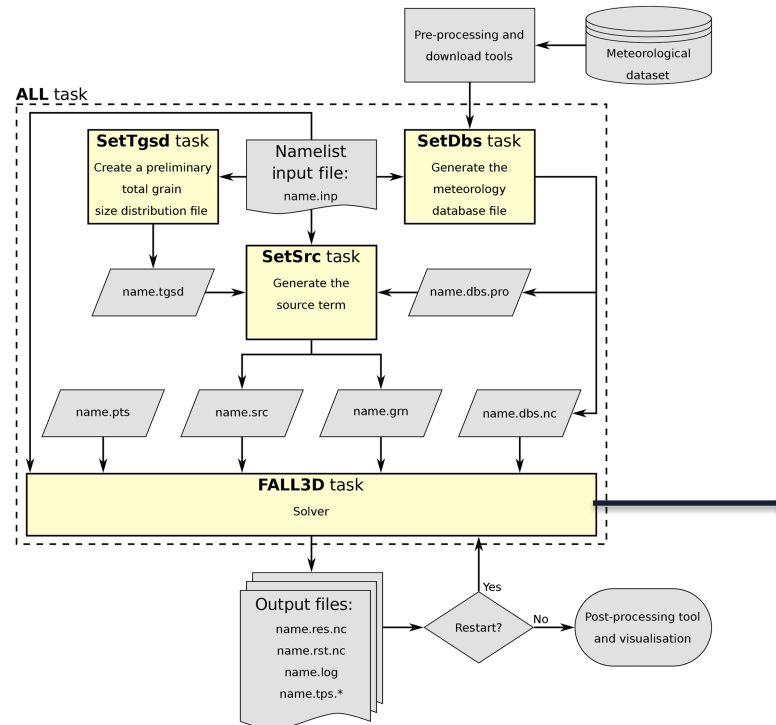
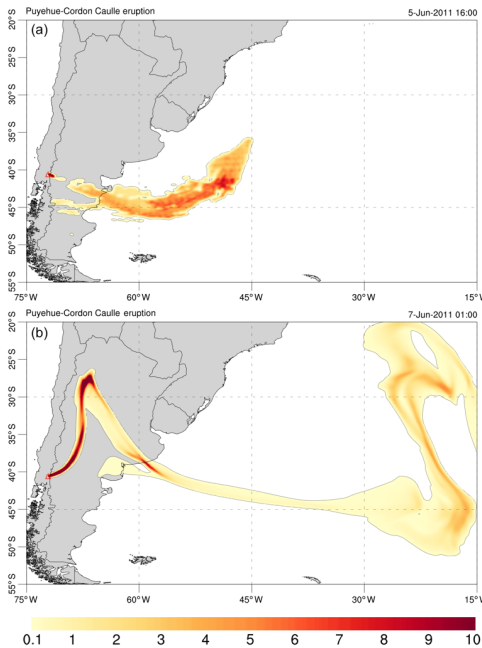


Gordon Bell Prize 2023 (UMich + IIS):
<https://dl.acm.org/doi/pdf/10.1145/3581784.3627037>

[†] In simulations using *k*-point sampling (QBox, DFT-FE-MLXC), the total number of electrons in the supercell are obtained by multiplying with the number of *k*-pts: number of electrons in the supercell for QBox are 96K e⁻, DFT-FE-MLXC system I are 302K e⁻ and system II are 619K e⁻.

The effect of locality

Regular code: 10% of the code ~ 90% of CPU time
 Research code: 1% of the code ~ > 95% of CPU time



$$\frac{\partial C}{\partial t} = S^* - I^*$$

$$\frac{\partial C}{\partial t} + \frac{\partial(CU)}{\partial X_1} - \frac{\partial}{\partial X_1} \left(K_1 \frac{\partial C}{\partial X_1} \right) = 0$$

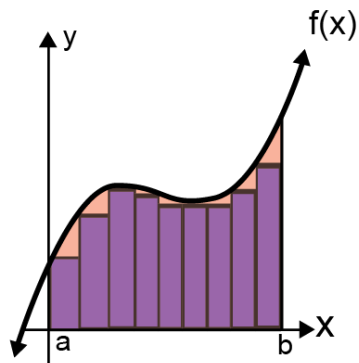
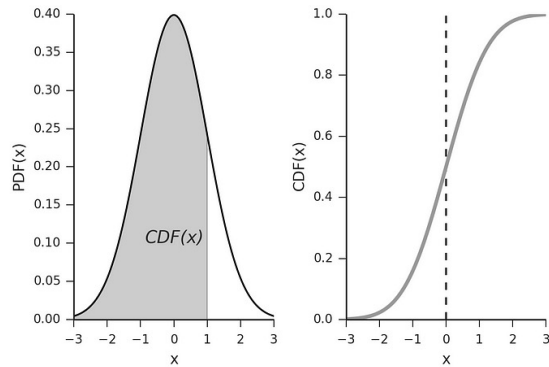
$$\frac{\partial C}{\partial t} + \frac{\partial(CV)}{\partial X_2} - \frac{\partial}{\partial X_2} \left(K_2 \frac{\partial C}{\partial X_2} \right) = 0$$

$$\frac{\partial C}{\partial t} + \frac{\partial [C(W - W_s)]}{\partial X_3} - \frac{\partial}{\partial X_3} \left(K_3 \frac{\partial C}{\partial X_3} \right) = 0$$

See: <https://gmd.copernicus.org/articles/13/1431/2020/#&gid=1&pid=1>



Calculus!



```

_Integrate_Simpson_ ← {
  α ← θ
  a b N ← ω
  1=2|N: □SIGNAL 11
  ds ← (b - a)÷N
  s ← a + ds×(0,τN-1)
  simpson_rule ← {α×((αα ω[1])) + (4×(αα ω[2])) + (αα ω[3]))÷3}
  0.5 × +/(ds α simpson_rule“({cω}⊗3)s)
}

normal_pdf ← {
  (*^-0.5×(ω-α[1])*2÷α[2]*2)÷((2×(o 1)×α[2]*2)*0.5)
}

norm_cdf ← {
  LLIM ← ^-40
  SF5 ← 1000000
  α (normal_pdf)_Integrate_naive_ (LLIM ω SF5)
}

```

Questions we want to answer

- What is the general shape of the core computation?
- How does the math translate into actual code?
- Which simplifications can cause trouble?
- How amenable to parallelization/concurrency is it?
- What are different ways to implement a specific algorithm?
- What are the most performant ones?
- Can performance impact readability?



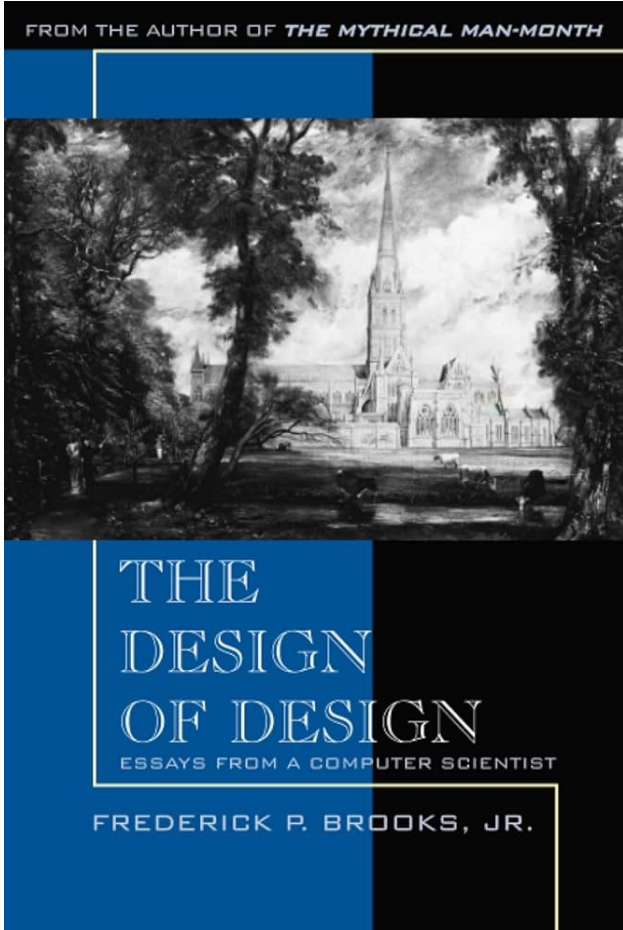


“Programs must be written for people to read,
and only incidentally for machines to execute.”

Harold Abelson, Structure and Interpretation
of Computer Programs



Prototypes are (likely) disposable

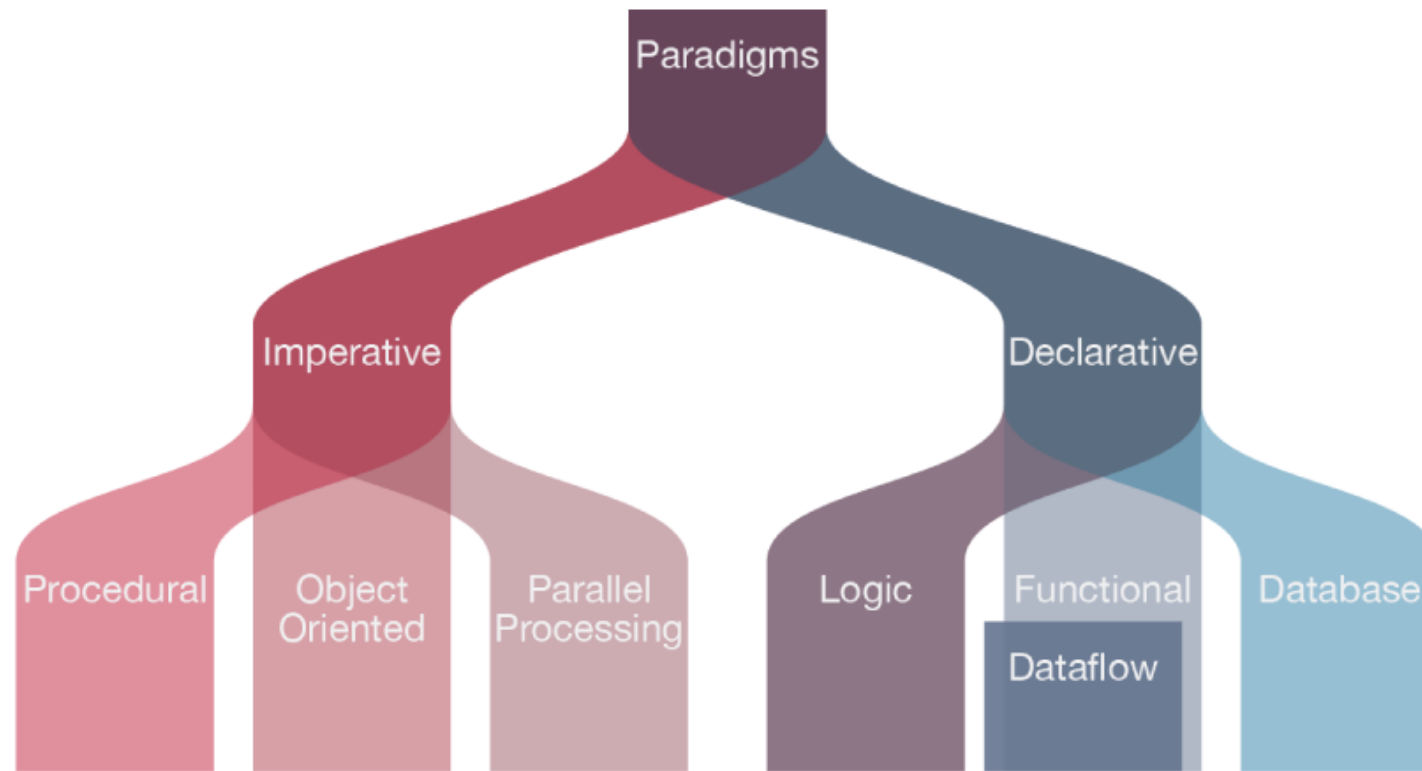


Plan to throw one (implementation)
away; you will, anyhow.

— *Fred Brooks* —

AZ QUOTES





Source: <https://www.watelectronics.com/types-of-programming-languages-with-differences/>



IN PRAISE OF APL

A LANGUAGE FOR LYRICAL PROGRAMMING

by Alan J. Perlis

(Reprinted from SIAM NEWS, June 1977, with permission of the Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania.)

Many reasons can be given for teaching one or more aspects of computer science (defined as the study of the set of phenomena arising around and because of the computer) to all university students. Probably every reader of this note supports some of these reasons. Let me list the few I find most important: (1) to understand and to be able to compose algorithms; (2) to understand how computers are organized and constructed; (3) to develop fluency in (at least) one programming language; (4) to appreciate the inevitability of controlling complexity through the design of systems; (5) to appreciate the devotion of computer scientists to their subject and the exterior consequences (to the student as citizen) of the science's development.



APL is a mistake, carried through to perfection. It is the language of the future for the programming techniques of the past: it creates a new generation of coding bums.

Edsger W. Dijkstra



1. Important Characteristics of Notation

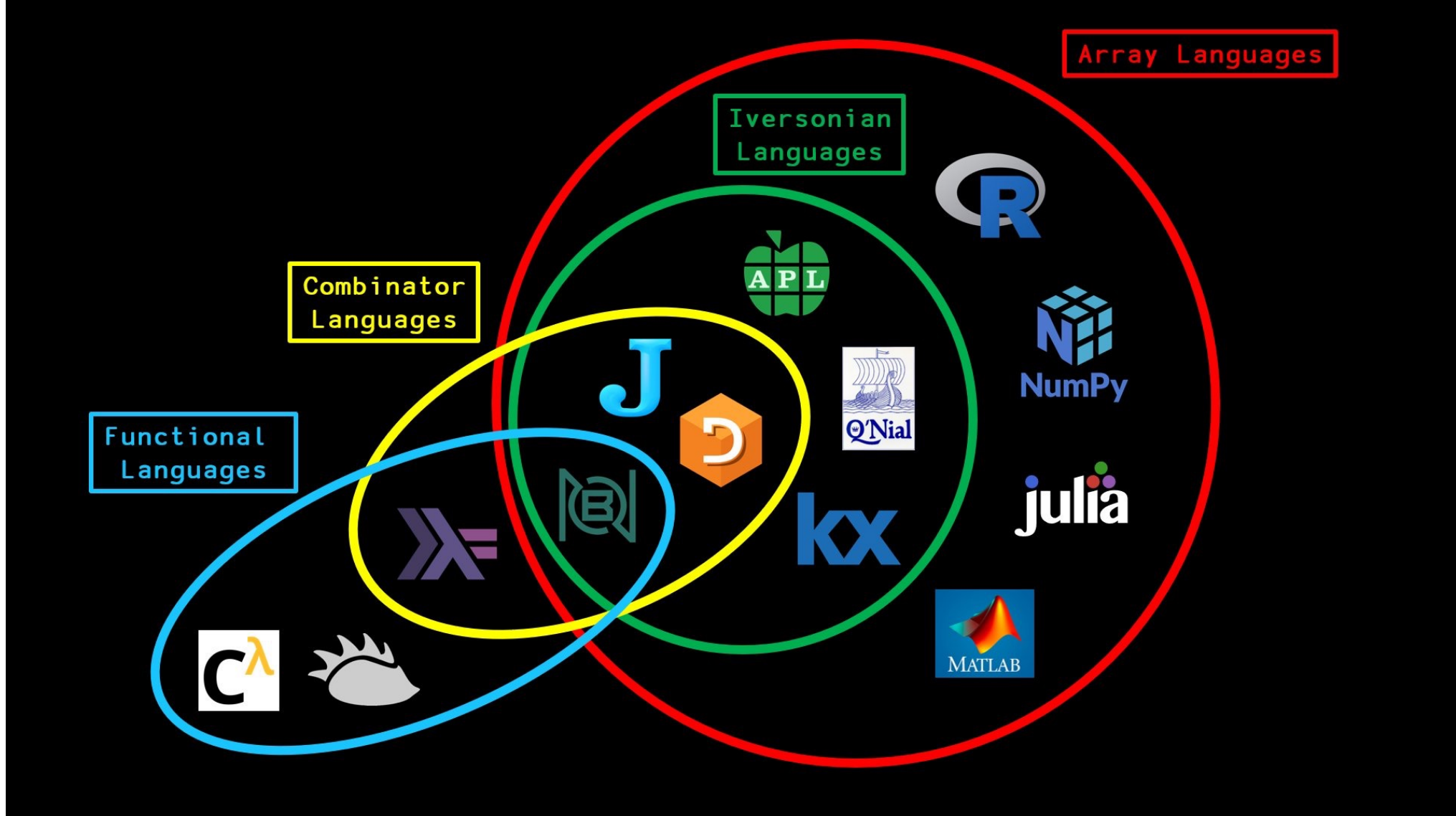
In addition to the executability and universality emphasized in the introduction, a good notation should embody characteristics familiar to any user of mathematical notation:

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

The foregoing is not intended as an exhaustive list, but will be used to shape the subsequent discussion.

“Programming languages, because they were designed for the purpose of directing computers, offer important advantages as tools of thought. Not only are they universal (general-purpose), but they are also executable and unambiguous. Executability makes it possible to use computers to perform extensive experiments on ideas expressed in a programming language, and the lack of ambiguity makes possible precise thought experiments.”





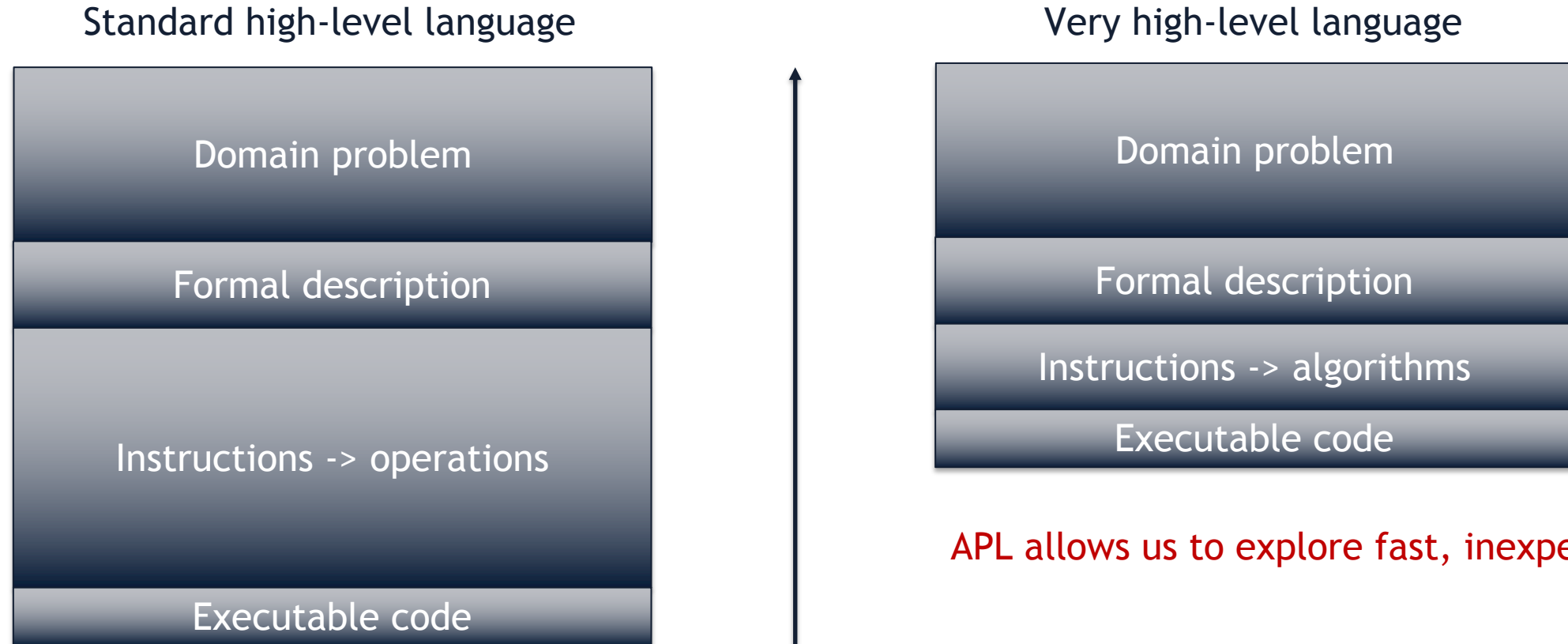
Source: https://twitter.com/code_report/status/1569808096654163969/photo/1

| Combinator | Lambda Expression |
|----------------|-------------------------------|
| I | $\lambda a.a$ |
| K | $\lambda ab.a$ |
| W | $\lambda ab.abb$ |
| C | $\lambda abc.acb$ |
| B | $\lambda abc.a(bc)$ |
| S | $\lambda abc.ac(bc)$ |
| D | $\lambda abcd.ab(cd)$ |
| B ₁ | $\lambda abcd.a(bcd)$ |
| Ψ | $\lambda abcd.a(bc)(bd)$ |
| Φ | $\lambda abcd.a(bd)(cd)$ |
| D ₂ | $\lambda abcde.a(bd)(ce)$ |
| E | $\lambda abcde.ab(cde)$ |
| Φ ₁ | $\lambda abcde.a(bde)(cde)$ |
| Ê | $\lambda abcdefg.a(bde)(cfg)$ |

Hoekstra, C., 2022, June. Combinatory logic and combinators in array languages. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming* (pp. 46-57).



Why all these complications?



APL allows us to explore fast, inexpensively!



APL AS A PROTOTYPING LANGUAGE:
CASE STUDY OF COMPILER DEVELOPMENT PROJECT

Matsuki Yoshino

Software Works, Hitachi, Ltd.
5030 Totsuka-machi Totsuka-ku Yokohama-shi
Kanagawa-ken 244 Japan

ABSTRACT

We are applying prototyping method using APL to a commercial compiler's development Project.

This paper will discuss the following matters based on our one year experience of the project:

- 1) Merits of APL as a prototyping language.
- 2) An environment of the prototype.
- 3) A representation of tables and the intermediate language of compilers in an APL environment.
- 4) A strategy of transforming the APL prototype into final product written in Pascal.
- 5) Evaluation of this method.

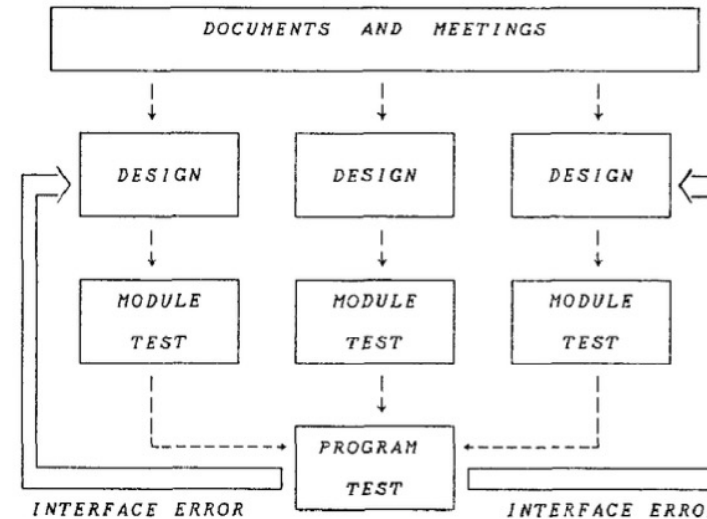
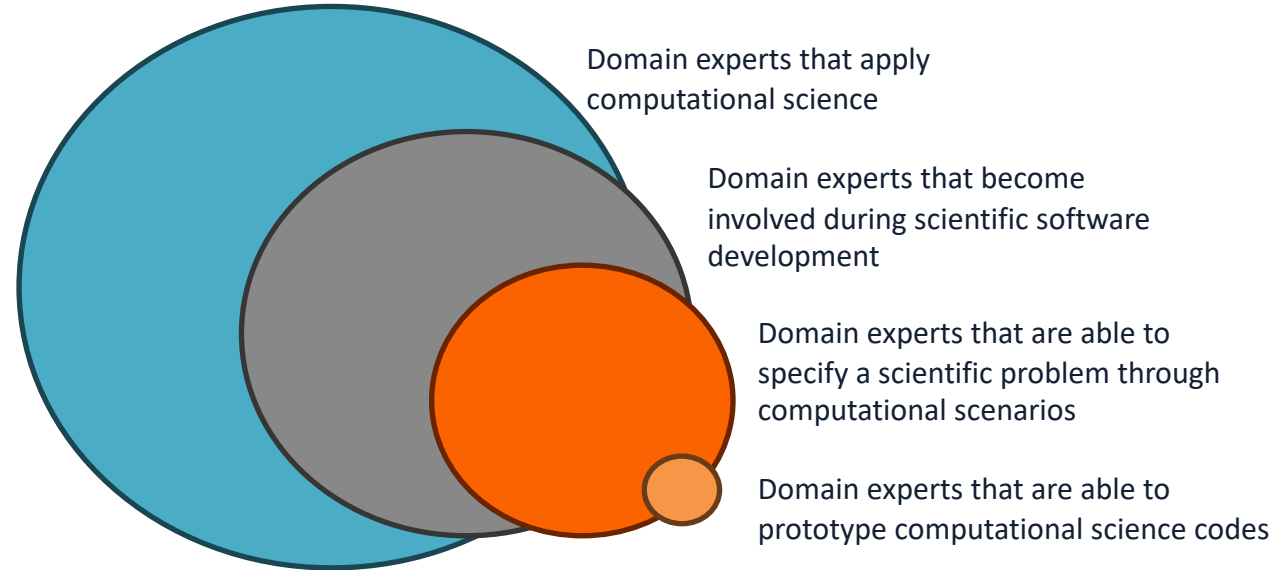


FIG.1 CONVENTIONAL DEVELOPMENT METHOD



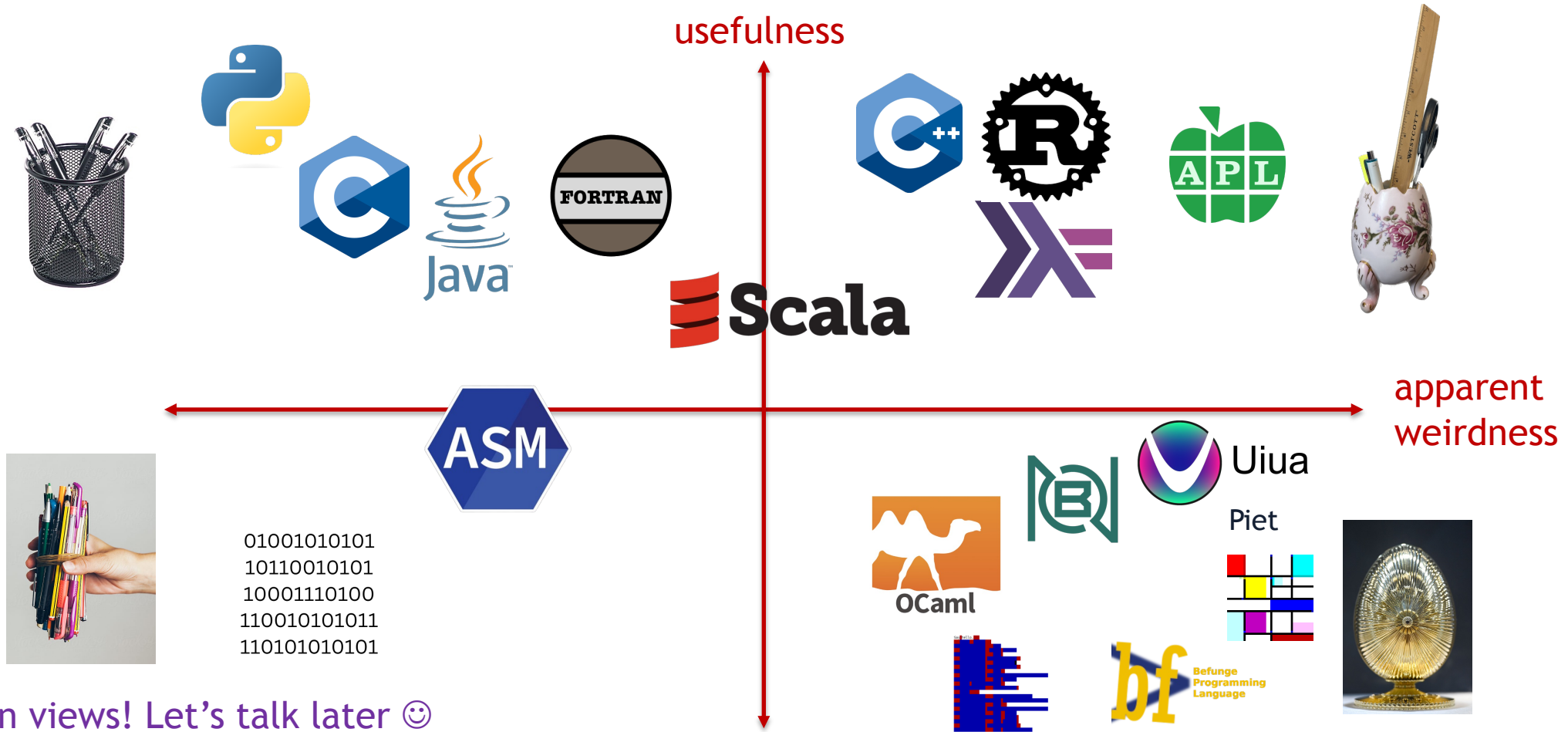
All domain experts in science



Benner, K. M., Feather, M. S., Johnson, W. L., & Zorman, L. A. (2014). Utilizing scenarios in the software development process. *Information system development process*, 30, 117-134.



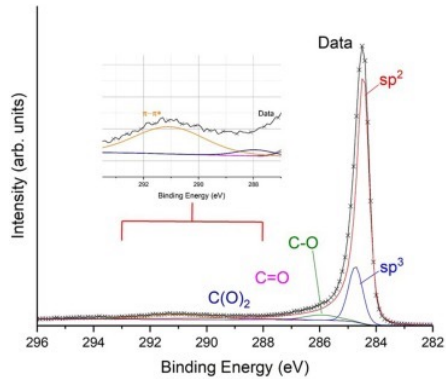
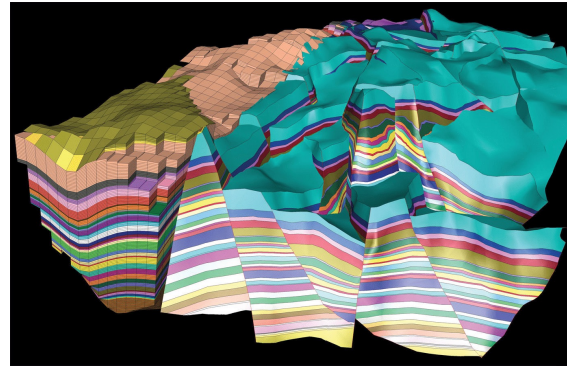
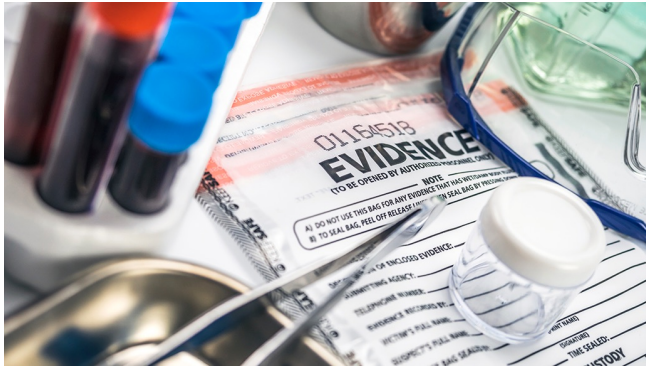
Innovation potential = apparent weirdness × usefulness



My own views! Let's talk later 😊



Where is APL used today?



U-net CNN in APL

Exploring zero-framework, zero-library machine learning

Aaron W. Hsu
aaron@dyalog.com
Researcher
Dyalog, Ltd.
Bloomington, IN, United States

Rodrigo Girão Serrão
rodrigo@dyalog.com
Consultant
Dyalog, Ltd.
Bramley, United Kingdom

ICFP'22, Sep 11 - Sep 16, 2022, Ljubljana, Slovenia

Aaron W. Hsu and Rodrigo Girão Serrão

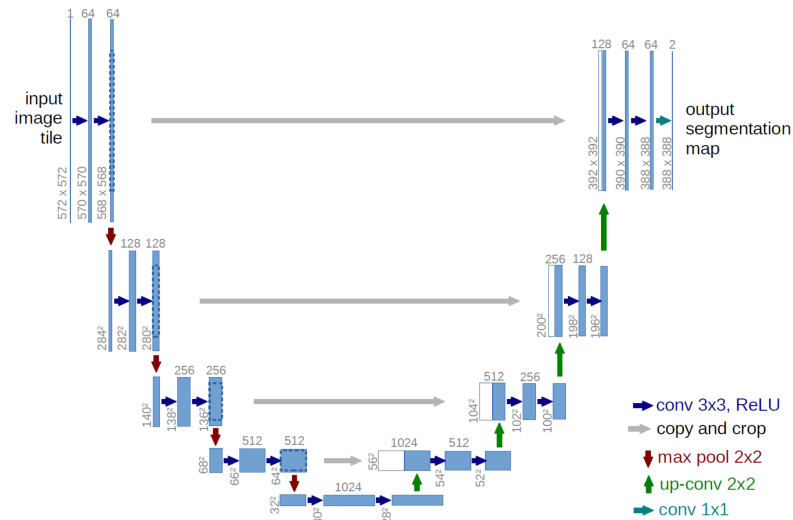


Figure 1. Original u-net architecture, as seen in the original paper [Ronneberger et al. 2015]. Arrows represent operations between the multi-channel feature maps represented by the rectangles. The number on top of each rectangle is its number of channels and the numbers in the lower-left corner are the x and y dimensions of the feature maps.

Appendix A: Complete APL U-net implementation

:Namespace UNET

W+θ ◊ V+θ ◊ Z+θ ◊ LR+1e⁻⁹ ◊ MO+0.99

```
FWD+{Z+-(#W)ρ<θ
CV+{0[Z-Z[α]+cZ[α],cZ+([2+i3]3 3⊞Z[α]+cω)+.x,[i3]α>W}
CC+{ω,⊞([p]+(-[p]+(α>Z)-p+2⊞(ρα>Z)-ρω)
MX+{[f[2],[2 3](2 2ρ2)⊞Z[α]+cω}
UP+{((2×-1+ρω),-1+ρα>W)ρ0 2 1 3 4qω+.xα>W-Z[α]+cω}
C1+{1E-8+z÷[i2]+/z+*z-[i2][z+ω+.xα>W-Z[α]+cω}
LA+{α≥#Z:ω
  down+(α+6)∇(α+2)MX(α+1)CV(α+0)CV ω
  (α+2)CC(α+5)UP(α+4)CV(α+3)CV down}
2 C1 1 CV 0 CV 3 LA ωρ23+1,⊞ρω}
```

```
BCK+{Y+α ◊ YΔ+ω
Δ+{0-W[α]+c(α>W)-LR×V[α]+cω+MO×(ρω)ρα>V}
ΔCV+{w+,[i3]ϕ[1]0 1 3 2qα>W ◊ x+α>Z ◊ Δz+ω×0<1>α>Z
ΔZ+-22ϕ[1](4+2↑ρΔz)†Δz
_+α Δ 3 0 1 2q(ϕ,[i2]Δz)+.x,[i2]3 3⊞x
w+.x⊞,[2+i3]3 3⊞ΔZ}
ΔCC+{x+α>Z ◊ Δz+ω ◊ d+[-2+⊞2†(ρx)-ρΔz ◊ (d)ϕ(1=d)ϕ[1](ρx)†Δz}
ΔMX+{x+α>Z ◊ Δz+ω ◊ y×x=y+(ρx)†2/2/[1]Δz}
ΔUP+{w+α>W ◊ x+α>Z ◊ Δz+ω ◊ cz+(2 2ρ2)⊞Δz
_+α Δ(ϕ,[i2]x)+.x,[i2]cz
([2+i3]cz)+.xq;W}
ΔC1+{w+α>W ◊ x+α>Z ◊ Δz+ω ◊ _+α Δ(ϕ,[i2]x)+.x,[i2]Δz ◊ Δz+.xqω}
ΔLA+{α≥#Z:ω
  down+(α+6)∇(α+3)ΔCV(α+4)ΔCV(α+5)ΔUP ω†[2]-2⊞ϕρω
  (α+0)ΔCV(α+1)ΔCV(ω ΔCC-2α+2)+(α+2)ΔMX down}
3 ΔLA 0 ΔCV 1 ΔCV 2 ΔC1 YΔ-(-Y),[1.5]Y}
```

E+{-+f,ϕ(α×ω[;:1])+(-α)×ω[;:0]}

RUN+{Y YΔ(Y E YΔ)-(Y+[0.5+nm†ω†²⊞(ρω)-nm+2†ρYΔ)BCK+YΔ+FWD α}

:EndNamespace



quAPL

A Motif examples

A -----

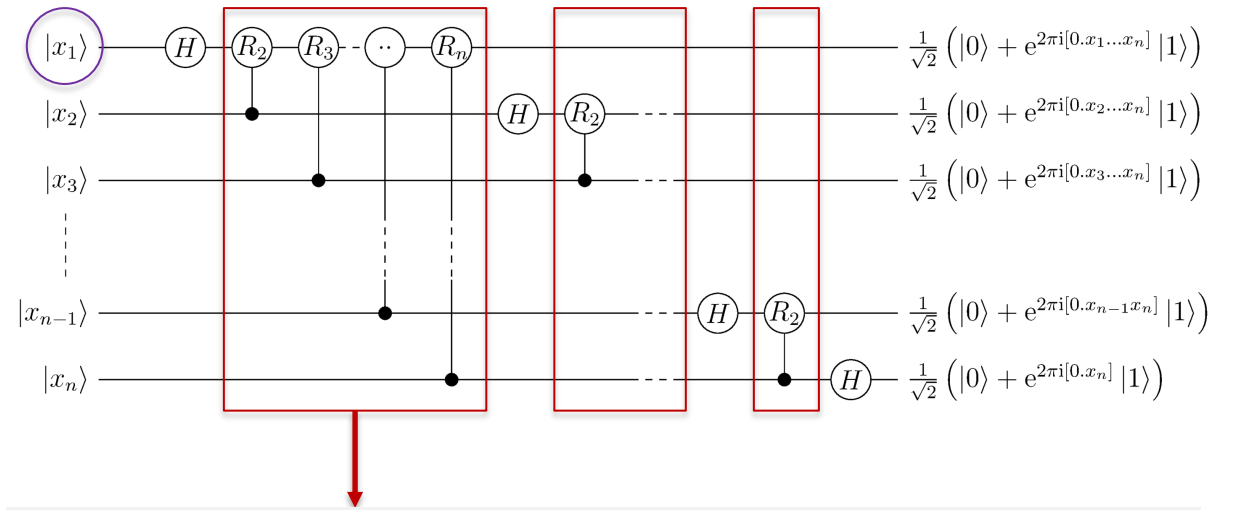
```
template ← {> kpr/ (1⊔2⊗(ρ⊗α)) ρ<ω}
```

```
gCTR ← {
  (n _) ← ρ ω
  ID ← {ω ω ρ 1, ωρ0}
  gate ← ID 2*(α + 2⊗n)
  ((-ρω)†gate) ← ω
  gate
}
```

```
superpose ← {
  stage ← ω template H
  stage ω
}
```

```
nvs ← (idx gtx) stage vs
```

```
(cbits, vs) ← measure vs
```



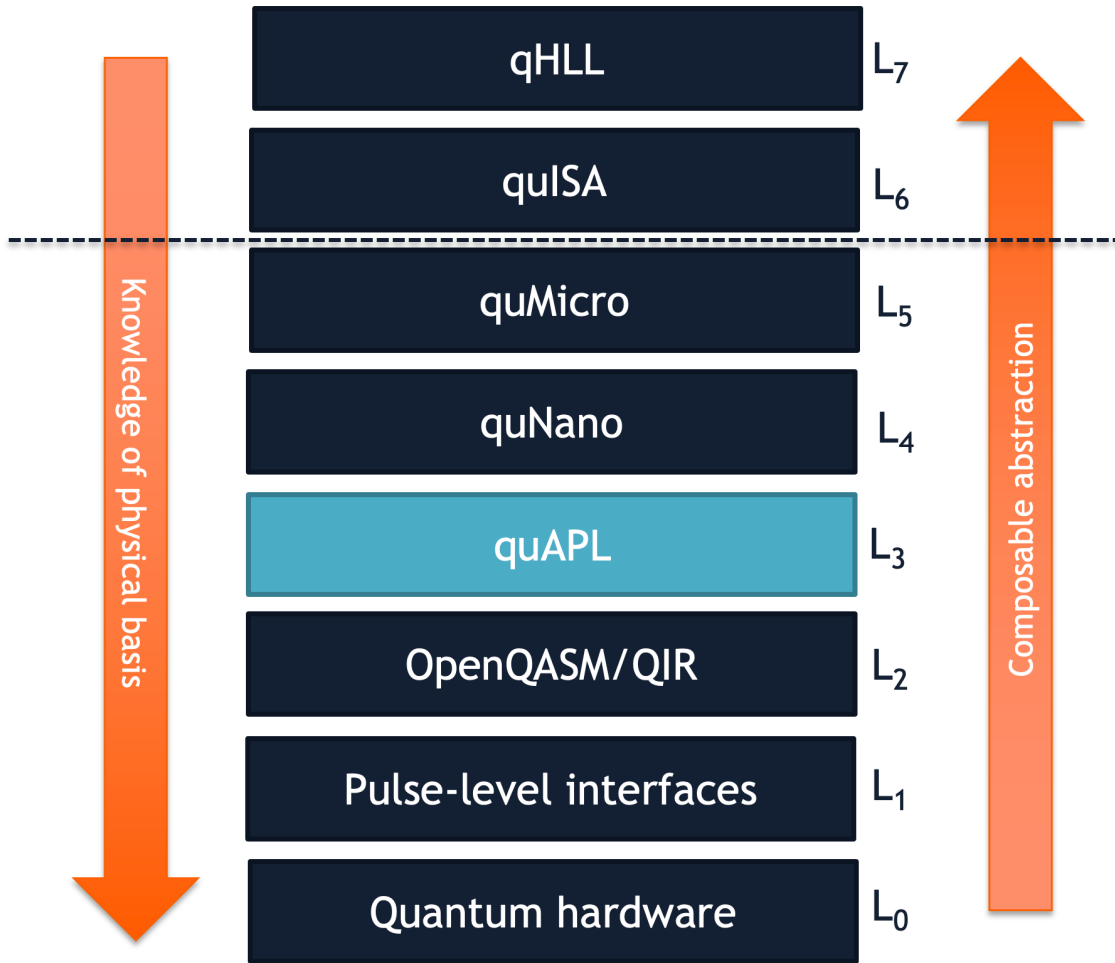
A R_n controlled gate for one R_n sequence

```
 $R_n \leftarrow \{1 \circ gCTR P (2 \times \omega \div 2 * \omega)\}$ 
```

```
qft_Rn_set ←  $R_n \dots (1 \downarrow \uparrow 2 \otimes 1 \uparrow \rho vs)$ 
```



Toward a quantum instruction set architecture

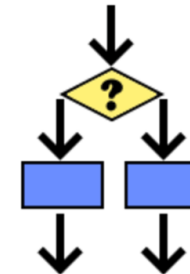


?

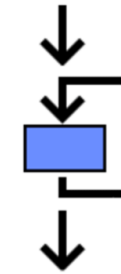
SEQUENCE



SELECTION



ITERATION



?



What have we learned?

- Building research software is hard
- Scientific packages have a core
- APL helps explore, understand, and prototype that core
- Sometimes, APL code becomes your code 😊

