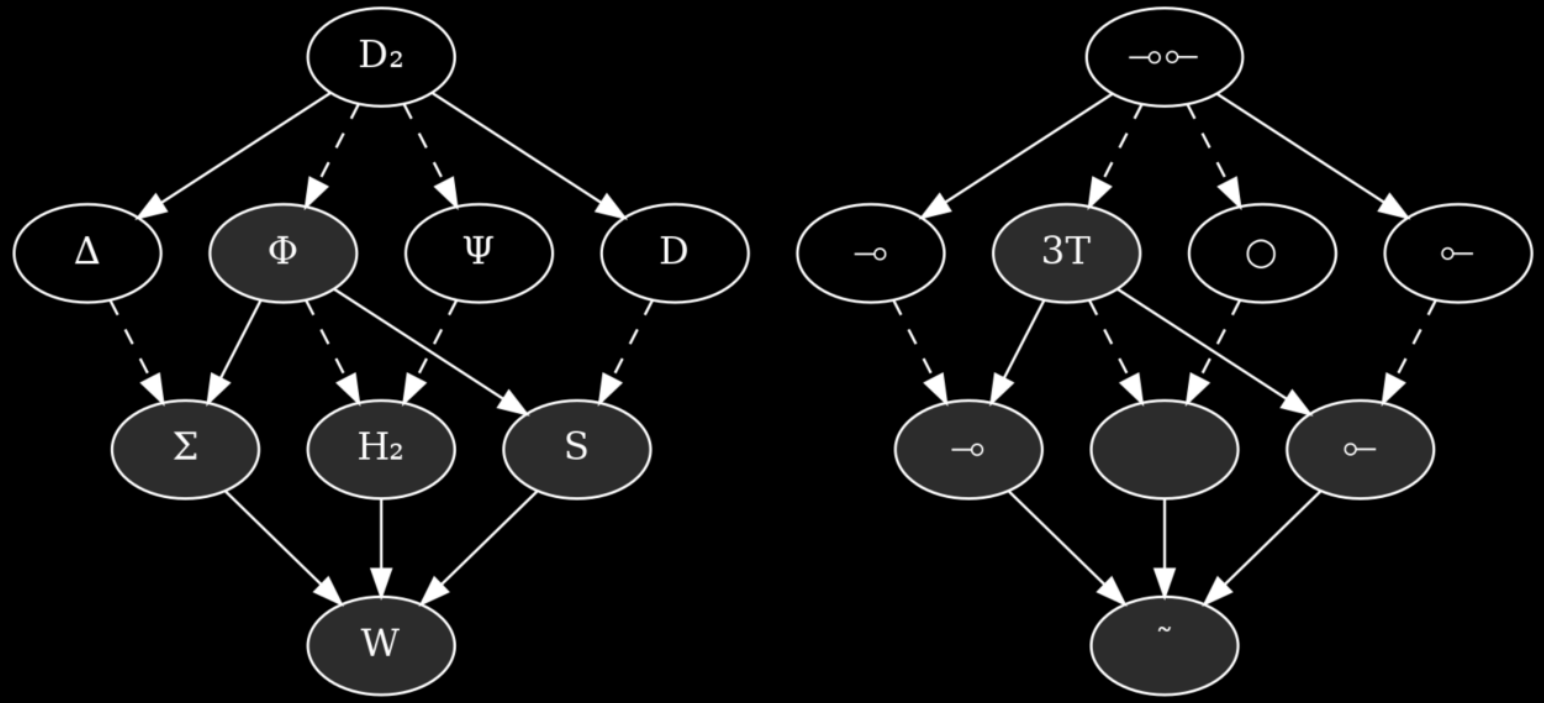
Why Combinators?

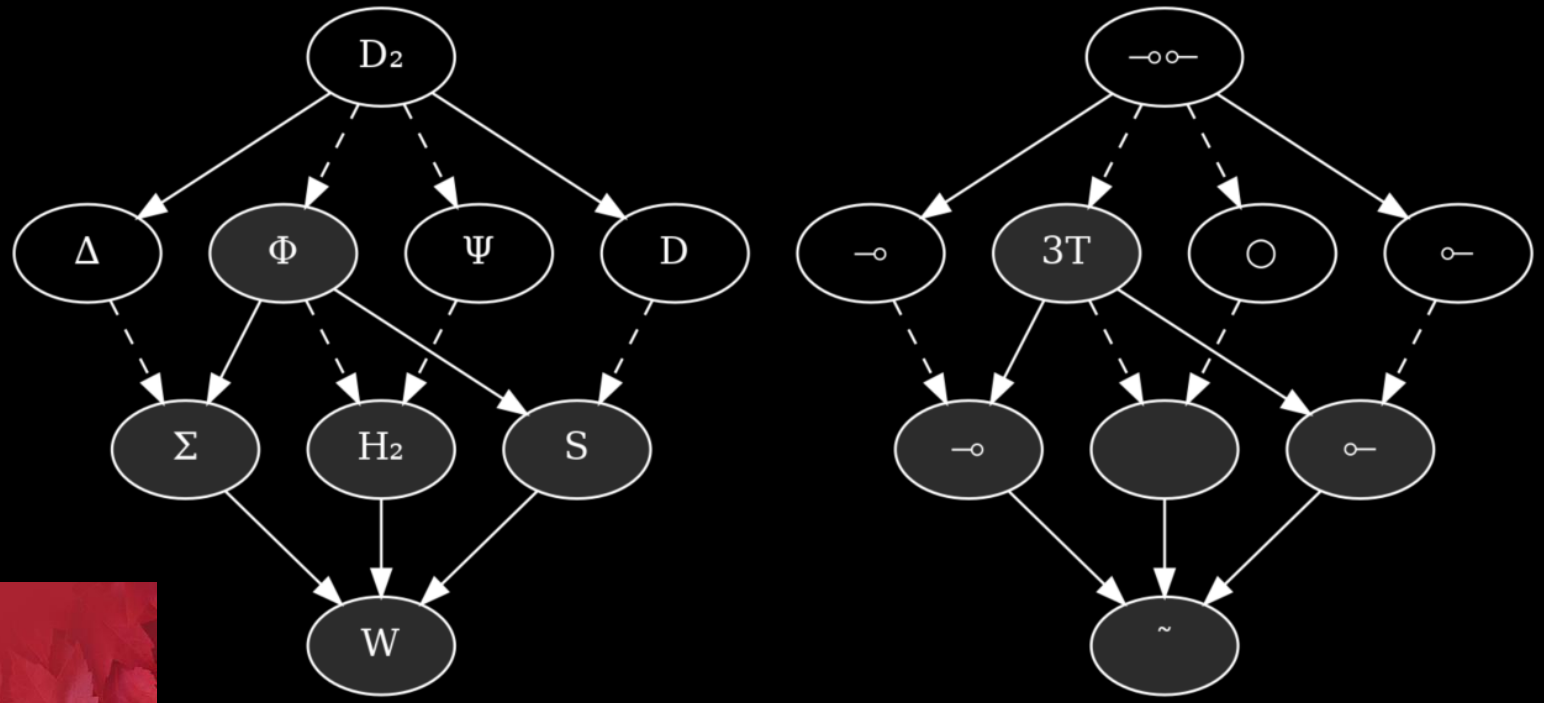


Conor Hoekstra

code_report | codereport

Composition Intuition

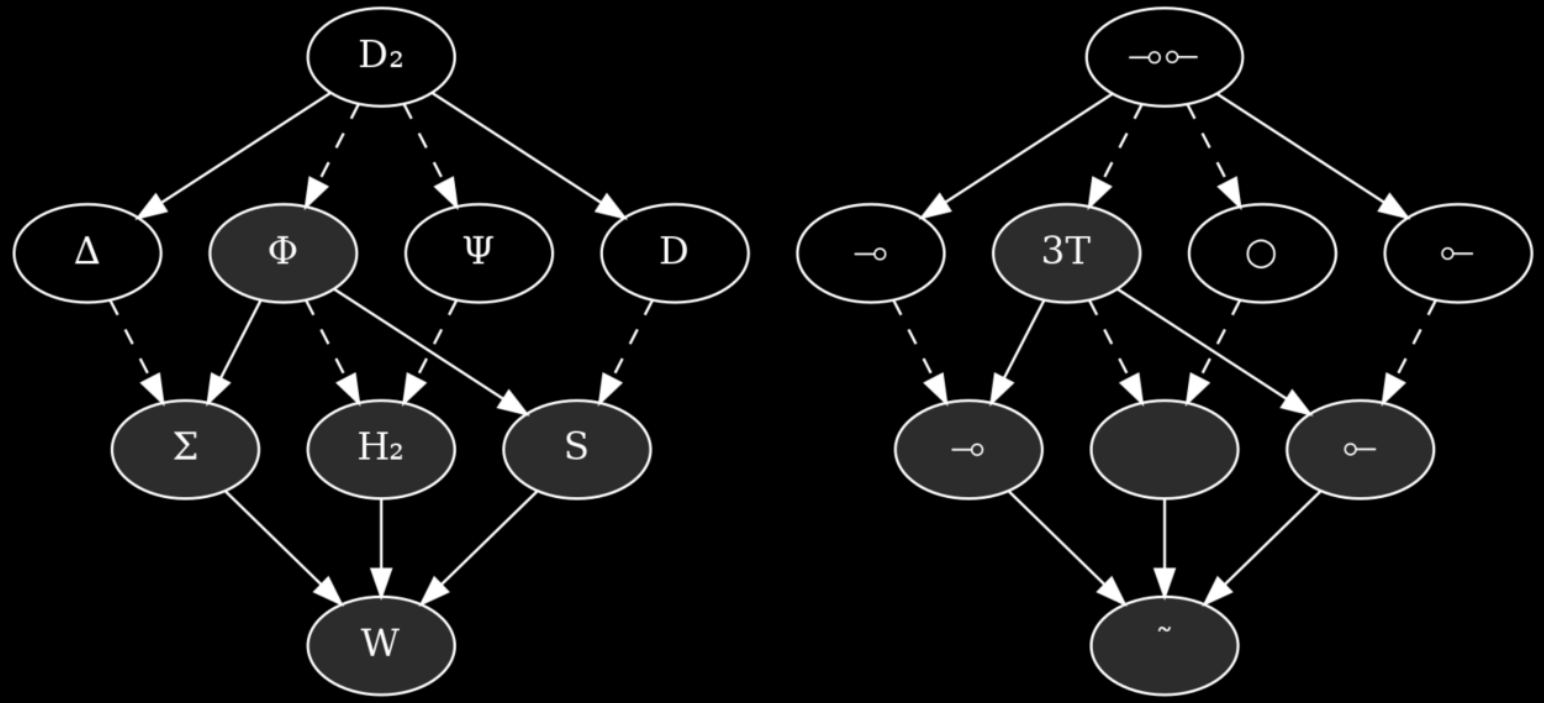
Combinators & Tacit Programming



Conor Hoekstra

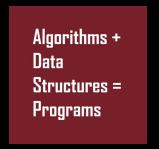
code_report | codereport

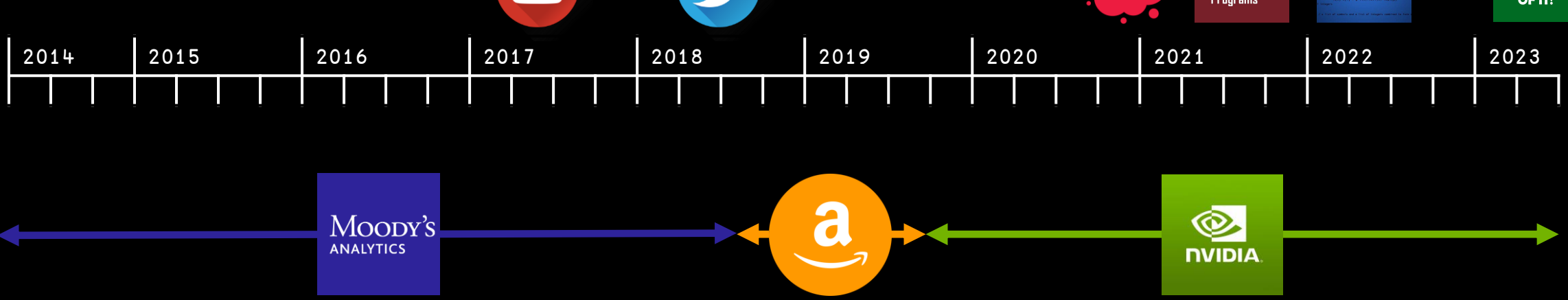
Why Combinators?



Conor Hoekstra

code_report | codereport





About Me

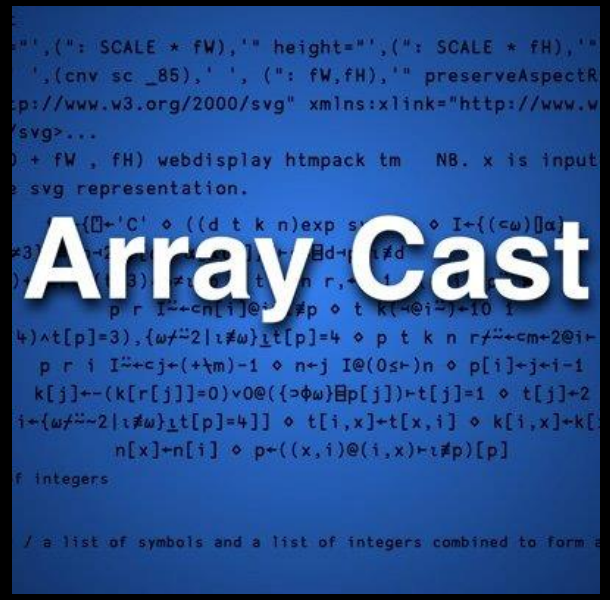
Conor Hoekstra / @code_report

324 Videos

38 (26) Talks

Algorithms + Data Structures = Programs

151 Episodes
@adspthepodcast



63 Episodes
@arraycast

RUN FOR THE FUN OF IT!

13 Episodes
@conorhoekstra

BQN vs Uiua #2

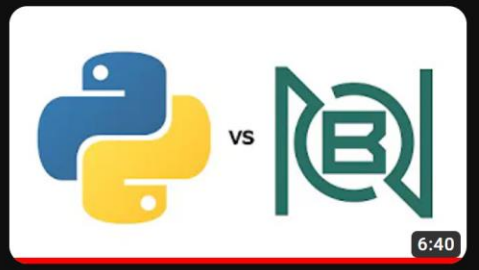
5.6K views • 5 days ago

BQN vs Uiua

24K views • 6 days ago

Uiua! A New Array Language!

10K views • 8 days ago



Python vs BQN (1 Problem)

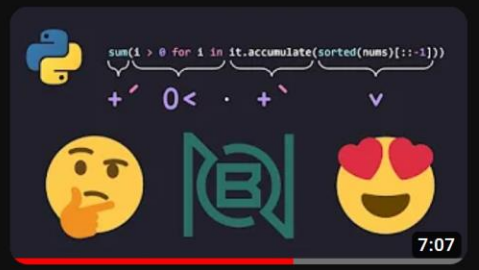
3.1K views • 2 weeks ago

Python + AI = Rust

4.9K views • 1 month ago

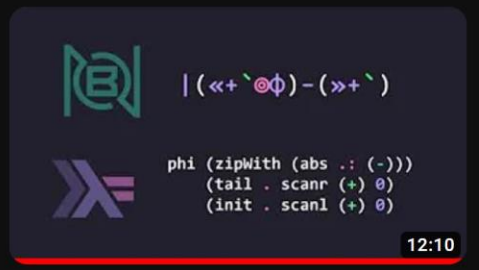
ADSP Episode 124: Vectorizing std::views::filter

2.4K views • 5 months ago



Why I Love BQN So Much! (vs Python)

9.6K views • 6 months ago



I ♥ BQN and Haskell

11K views • 7 months ago

C++ vs Rust: Tuples

13K views • 8 months ago

Top Programming Languages of 2022!

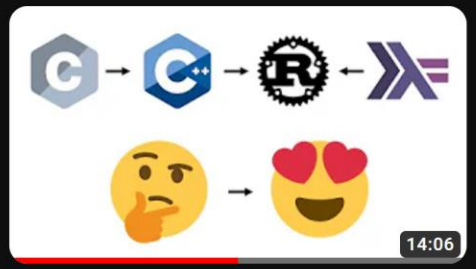
9.8K views • 8 months ago

1 Problem, 24 Programming Languages

226K views • 8 months ago

How to Make Beautiful Code Presentations

108K views • 9 months ago



From C ➡ C++ ➡ Rust

151K views • 9 months ago

Advent of Code 2022 in APL & BQN Day 6!

2.7K views • 9 months ago

Advent of Code 2022 in APL Day 4!

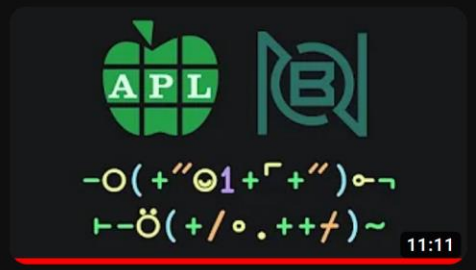
2.3K views • 9 months ago

Advent of Code 2022 in APL & BQN Day 3!

2.8K views • 10 months ago

Advent of Code 2022 in APL & BQN Day 1!

4.1K views • 10 months ago

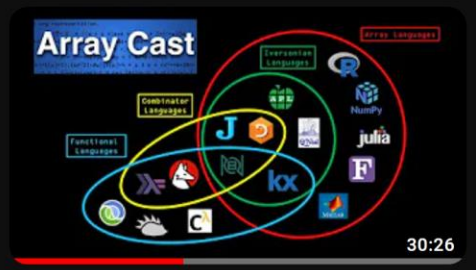


I ❤ APL & BQN

18K views • 10 months ago

APL vs BQN vs J vs Q vs NumPy vs Julia vs R

21K views • 1 year ago



ArrayCast Episode 36 BTS: Array Programming Language Venn Diagram

5.6K views • 1 year ago

How To Install APL

5.2K views • 1 year ago

Seven Languages in Seven Weeks: Chapter 4 - Prolog

5K views • 1 year ago

Seven Languages in Seven Weeks: Chapter 3 - Io

3.2K views • 1 year ago

Seven Languages in Seven Weeks: Chapter 2 - Ruby

4.2K views • 1 year ago

www.plrank.com

Programming Language Rankings (2023 Oct)

by code_report

StackOverflow Octoverse RedMonk Languish
PYPL IEEE Spectrum TIOBE (rankings overview)

Exclude "Edge Languages" | Number of Languages: 20
Months for Delta (Δ): 3 | All Languages

	Language	Avg	StDev	n[1]	3mΔ
1	JavaScript	1.25	0.5	4	-
2	Python	2	0.81	4	-
3	Java	4.25	1.89	4	-
4	TypeScript	4.75	1.7	4	-
5	C#	5.75	1.5	4	-
6	C++	7	1.41	4	-
7	PHP	8.5	3.69	4	-
8	C	9.5	0.57	4	-
9	Shell	10	3.91	4	-
10	Go	11	2.64	3	-
11	Rust	13.66	2.51	3	-
12	Ruby	14.25	6.75	4	-
13	Kotlin	15	1	3	-
14	Swift	16	5.29	3	(1)
15	R	16.33	5.03	3	(1)
16	PowerShell	17.33	6.8	3	(1)
17	Dart	17.66	1.52	3	(1)
18	Lua	22.5	7.77	2	(2)
19	Scala	23	9.64	3	(1)
20	Objective-C	25.66	11.15	3	(1)

1 - The number of (selected) ranking websites this language shows up in.

If you have suggestions or find a bug, you can open an issue here.

Array Programming Language Rankings (2023 Oct)

by code_report

☑ StackOverflow ☑ Octoverse ☑ RedMonk ☑ Languish
☐ PYPL ☐ IEEE Spectrum ☐ TIOBE (rankings overview)

☑ Exclude "Edge Languages" | Number of Languages: 20
Months for Delta (Δ): 3 | Array Languages

	Language	Avg	StDev	n[1]	3mΔ
1	R	16.33	5.03	3	-
2	MATLAB	34.5	16.26	2	-
3	Julia	40	5.65	2	-
4	Fortran	54	21.21	2	-
5	APL	126.5	109.6	2	-
6	Mathematica	132	0	1	-
7	q	152	0	1	-
8	Chapel	239	0	1	-
9	J	330	0	1	-
10	Futhark	356	0	1	-

1 - The number of (selected) ranking websites this language shows up in.

If you have suggestions or find a bug, you can open an issue here.

https://github.com/codereport/Content

https://github.com/codereport/Content

https://github.com/codereport/Content
https://www.arraycast.com/
https://www.youtube.com/c/codereport

[[ talks]]

Thank you! Minnowbrook is awesome!

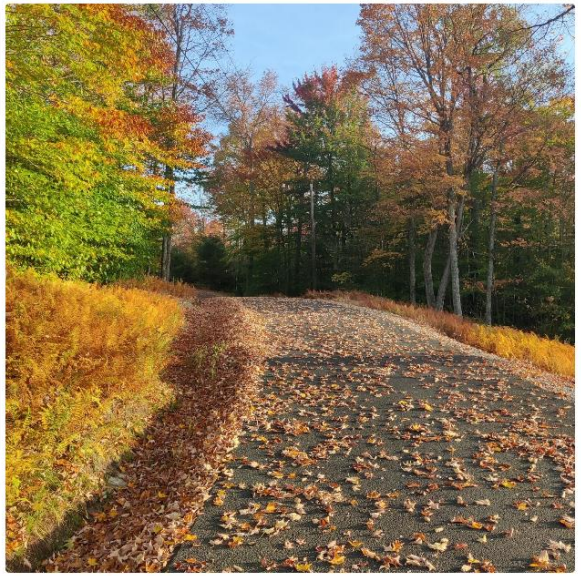
Conor Hoekstra (he/him)

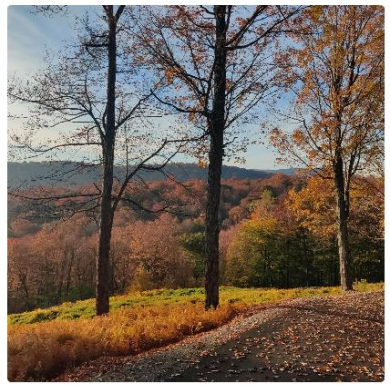
Today at 7:32 AM · Town of Indian Lake, New York

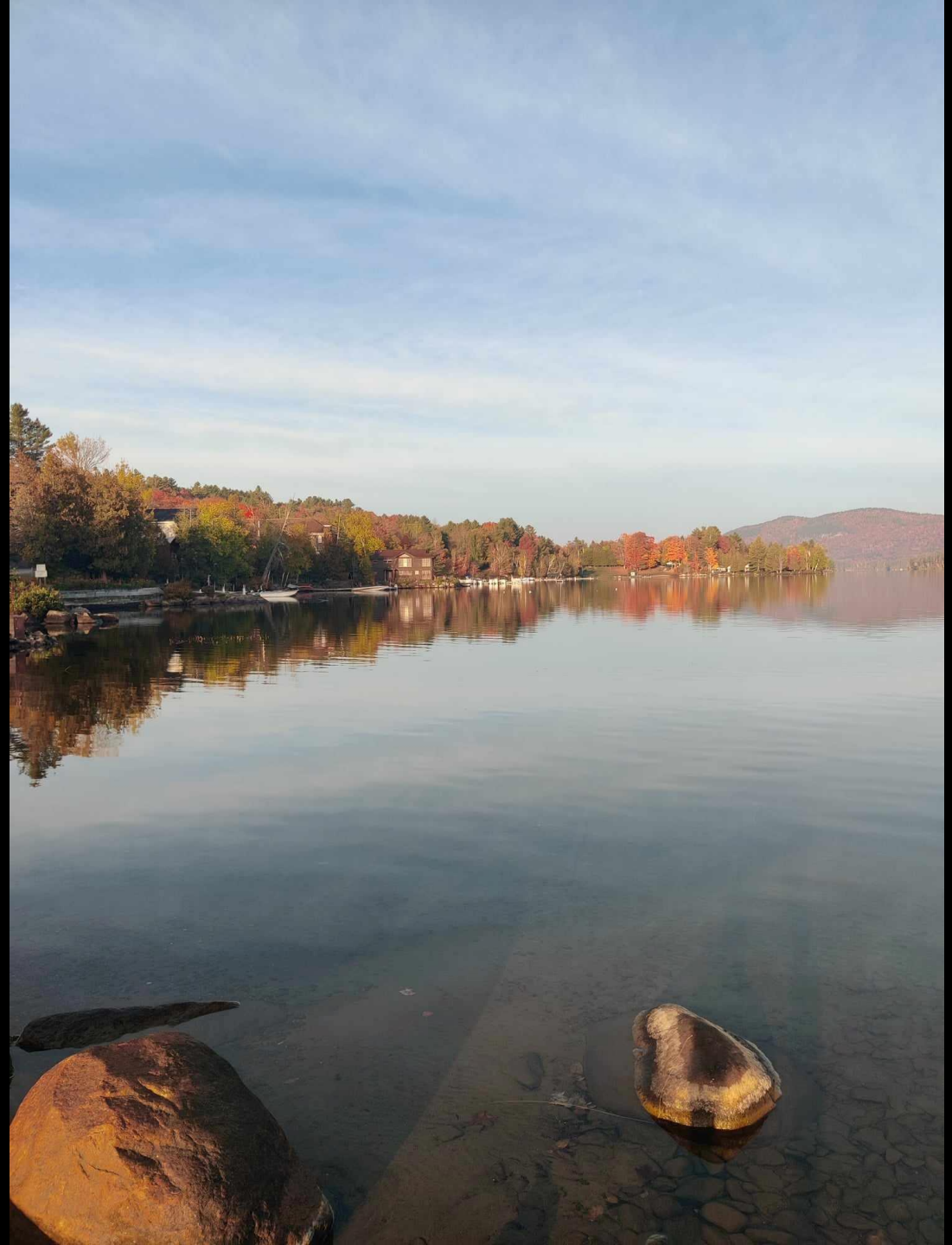
Easy Days Easy (Middle of Nowhere Edition us)

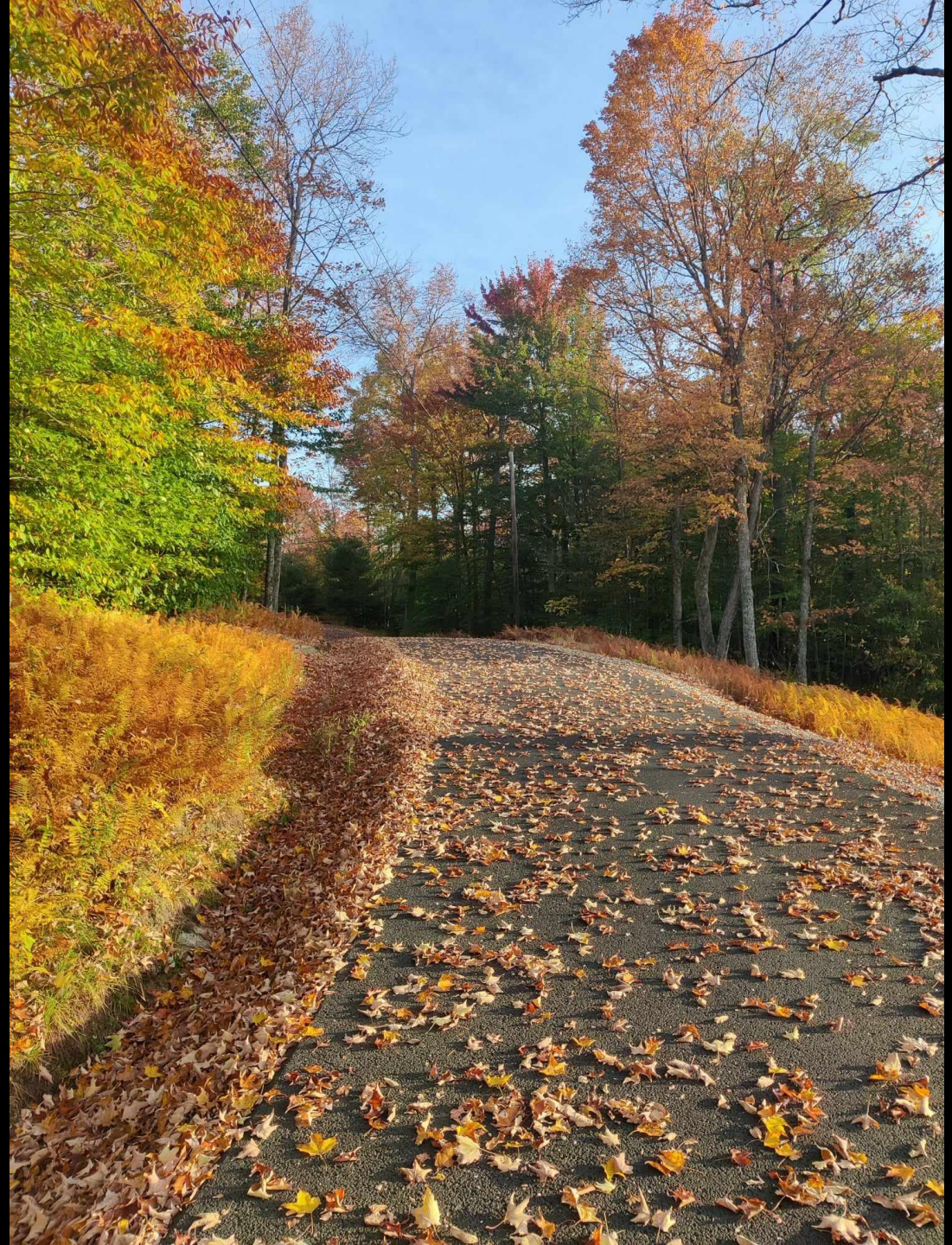
5 streets completed.

Distance	Pace	Time
14.19 km	4:47 /km	1h 7m









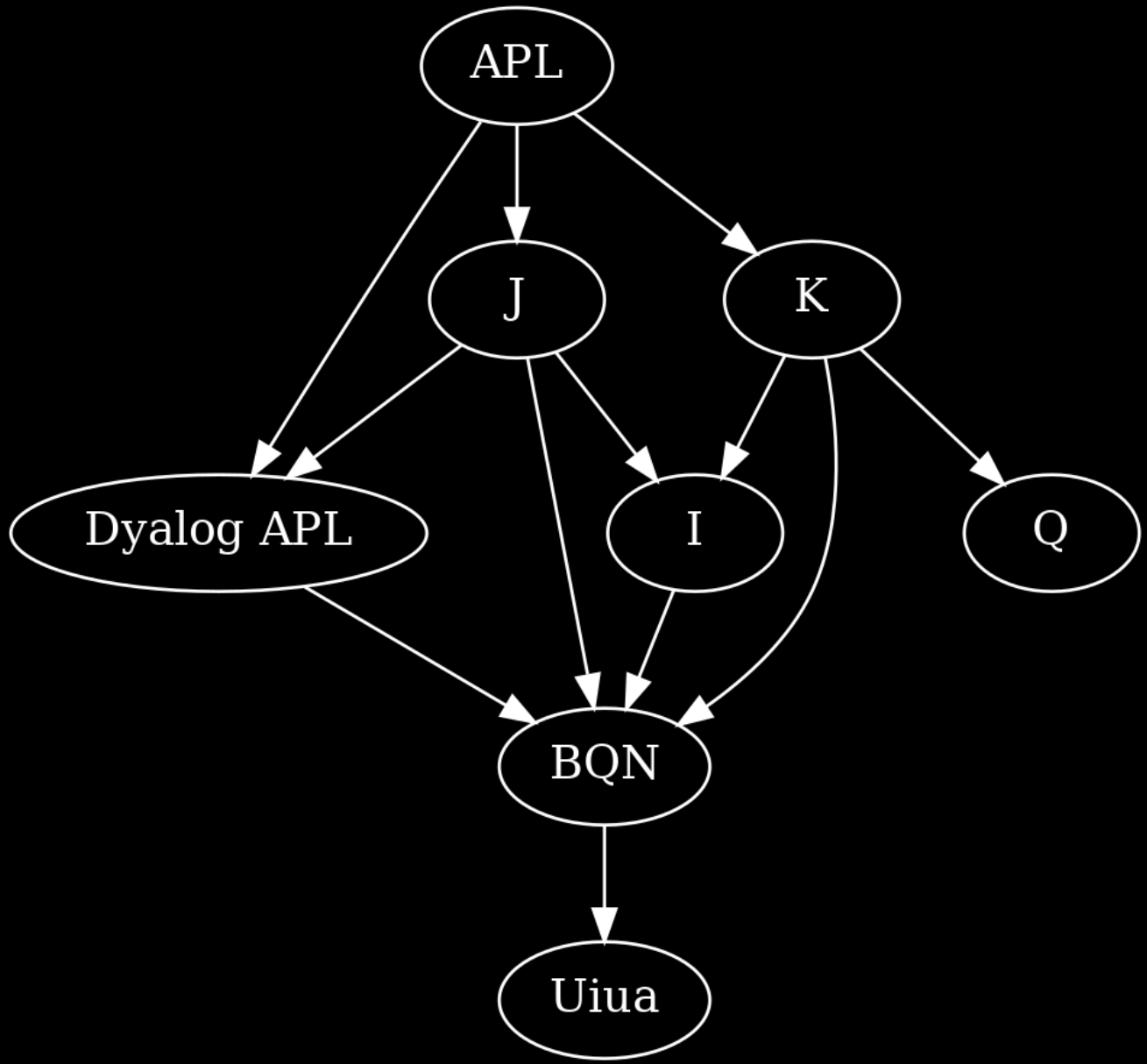
Why I ❤ APL

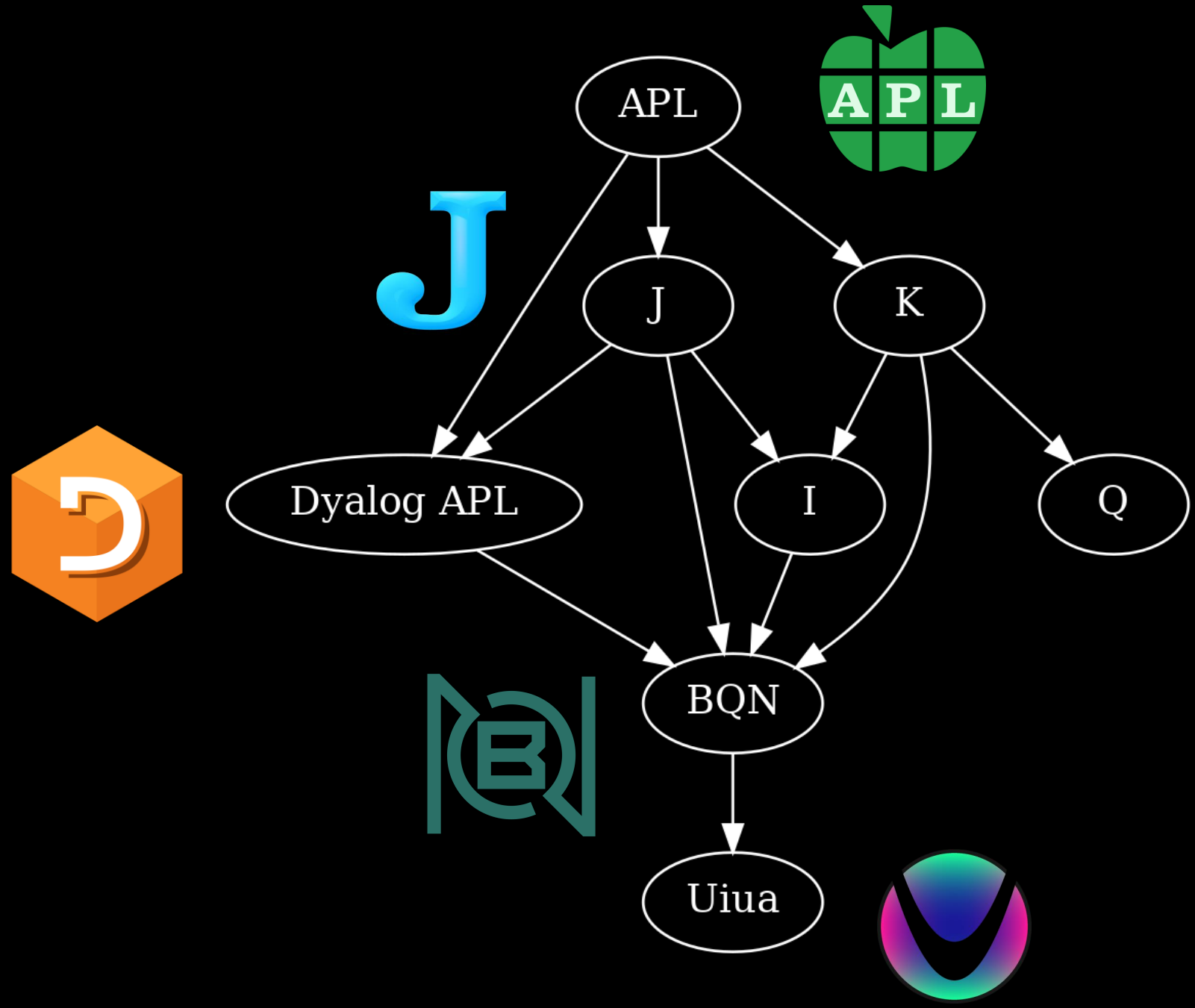
(and other array languages)

```cpp
#include <iostream>
#include <numeric>
#include <ranges>

auto main() -> int {

    auto nums = std::views::iota(1, 11)
        | std::views::transform([](auto i) { return 1 + 2 * i; });
    auto sum = std::accumulate(nums.begin(), nums.end(), 0);

    std::cout << sum << '\n';

    return 0;
}
```


```
+/1+2×⍳10
```

State of Modern Iversonian Array Languages


APL
J
K
Dyalog APL
I
Q
BQN
Uiua


APL
J
K
Dyalog APL
I
Q
BQN
Uiua


```
∘.=⍨⍳5
```

```
=/~i.5
```

```
=⌜~↕5
```

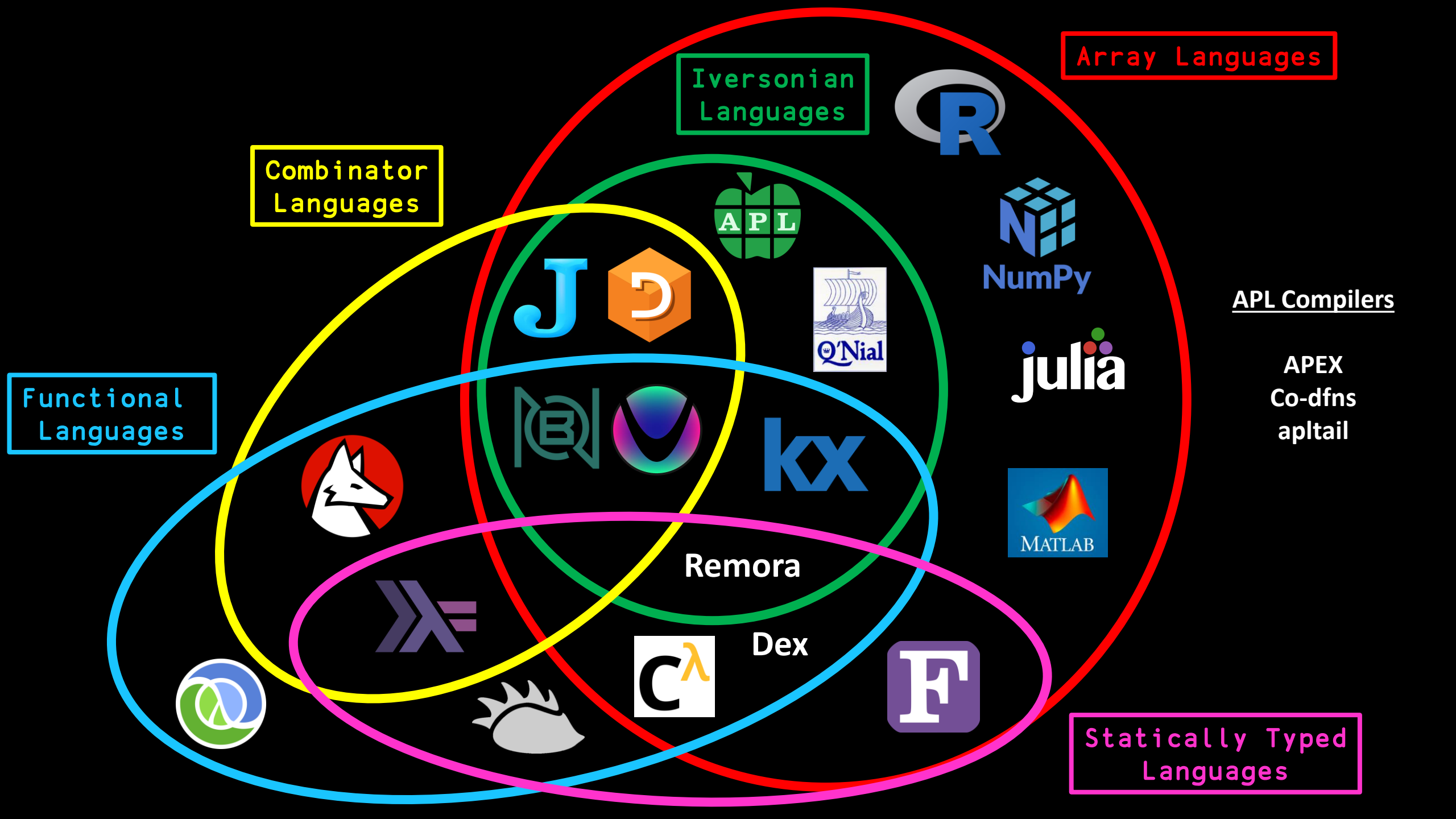
```
⊞=.↑5
```


```
∘.=⍨⍳5
```

```
=/~i.5
```

```
=⌜~↕5
```

```
⊞=.⍧5
```


Array Languages
Iversonian Languages
Combinator Languages
Functional Languages
Statically Typed Languages
APL
NumPy
Q'Nial
julia
kx
MATLAB
Remora
Dex
APL Compilers
APEX
Co-dfns
apltail

https://github.com/codereport/array-language-comparisons

README.md

Language / Library Websites

	Language	$	Main Website	Help / Docs	Online REPL
💚	Dyalog APL		dyalog.com	Dyalog Help	TryAPL
💚	J		jsoftware.com	J NuVoc	J Playground
💚	BQN		mlochbaum.github.io/BQN	BQN Docs	BQNPAD
💚	Uiua		uiua.org	Uiua Docs	UiuaPAD
💚	Q	$	code.kx.com/q	Q Ref	🚫
💙	Julia		julialang.org	Julia Docs	Replit
💙	MATLAB	$	mathworks.com/products/matlab.html	MATLAB Help	$
💙	NumPy*		numpy.org	NumPy Docs	Replit
💙	R		r-project.org	R Docs	JDoodle
💜	Nial		nial-array-language.org	Nial Dictionary	TIO
💜	Futhark		futhark-lang.org	Futhark Docs	🚫
💜	Dex		github.com/google-research/dex-lang	InDex	🚫
💜	Ivy		pkg.go.dev/robpike.io/ivy	Ivy Docs	🚫
💜❤️	SaC		sac-home.org	SaC Docs	🚫
❤️	ArrayFire*		arrayfire.com	ArrayFire Docs	🚫
❤️	MatX*		nvidia.github.io/MatX	MatX API Ref	🚫
❤️	ATen*		-	ATen Docs	🚫
❤️	Eigen*		eigen.tuxfamily.org	Eigen Dox	Godbolt

* Library, not an actual language

What is a combinator?

A function that deals only in functions

What are the combinators in APL?

Table 5. History of combinators in Dyalog APL.

Year	Version	Combinator	Spelling
1983	1.0	B, D, C	∘⍤
2003	10.0	W	⍨
2013	13.0	K	⊢
2014	14.0	B, B_1, Φ, $Φ_1$	trains
2020	18.0	B, B_1, Ψ, K	⍥⍤⍨

What is tacit (point free) programming?

Argument Free Code

```
Plus ← {α+ω}
```

Plus ← +

Some Examples

Combinatory Logic and Combinators in Array Languages

Conor Hoekstra
conorhoekstra@gmail.com
Toronto Metropolitan University
Toronto, ON, Canada

Abstract

The array language paradigm began in the 1960s when Ken Iverson created APL. After spending over 30 years working on APL, he moved on to his second array language J, a successor to APL which embeds a significant subset of combinatory logic in it. This paper will look at the existence of combinators in the modern array languages Dyalog APL, J and BQN and how the support for them differs between these languages. The paper will also discuss how combinators can be a powerful language feature as they exist in modern array languages.

CCS Concepts: • **Software and its engineering → Functional languages**.

Keywords: combinatory logic, combinators, array languages, apl, j, bqn

ACM Reference Format:
Conor Hoekstra. 2022. Combinatory Logic and Combinators in Array Languages. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming (ARRAY '22), June 13, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3520306.3534504

1 Introduction

The goal of this paper is to provide a brief history of both array languages and combinatory logic and then to more importantly show how a significant subset of combinatory logic exists in modern array languages in the form of combinators and why this is an extremely useful language feature.

Section 2 will briefly introduce array languages and how they differ from other programming languages. Section 3 will cover a brief history of array languages. Section 4 will cover a brief history of combinatory logic focusing on the papers and literature that first introduced individual combinators. Section 5 will enumerate the hierarchy of combinators and how certain combinators are specializations of other combinators. Section 6 will discuss the evolution of combinators in array languages and specifically how "trains" differ in modern array languages. Section 7 will enumerate each of the combinators from combinatory logic as they exist in each of Dyalog APL, J and BQN. Section 8 will discuss what makes combinators in modern array languages so powerful and expressive. Section 9 will list several additional examples. Finally, in Section 10 a summary of what has been discussed and shown in this paper will be provided.

ARRAY '22, June 13, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9269-3/22/06…$15.00
https://doi.org/10.1145/3520306.3534504

2 A Brief Introduction to Array Languages

The three main array languages that will be discussed in this paper are: Dyalog APL (1983), J (1990) and BQN (2020). These languages are notably different from other programming languages because APL and BQN use Unicode symbols for primitives and J uses ASCII symbols, digraphs, and trigraphs. Many have asserted that this makes the languages unreadable but that is the equivalent of saying Chinese is unreadable just because it doesn't use the Latin alphabet. As long as you have the requisite knowledge to read the symbols, not only is it extremely readable but it can also be extremely expressive and powerful. For example, the following snippet of APL code can be used to generate the first 10 odd numbers:

```
      ⍝ numbers from 1 to 10
      ⍳10
1 2 3 4 5 6 7 8 9 10
      ⍝ multiply by 2
      2×⍳10
2 4 6 8 10 12 14 16 18 20
      ⍝ add negative 1
      ¯1+2×⍳10
1 3 5 7 9 11 13 15 17 19
```

Other examples that demonstrate the expressivity and power of APL are examples using outer product. The following snippet of APL code can be used to create a multiplication table:

```
  9  3 11  0 14
  5 11  3 14  0
              ⌟

    # Deshape
    (↯|∘-⁼˜) vec
⟨ 0 6 2 9 5 6 0 8 3 11 ...

    # Deduplicate with B
    (⍷∘↯|∘-⁼˜) vec
⟨ 0 6 2 9 5 8 3 11 14 ⟩

    AllDiffs ← ⍷∘↯|∘-⁼˜
```

In the BQN code, no parentheses are required. The ˜ and ∘ have higher precedence than function application and no libraries are required.

9 Examples

Each of the examples are run with one or two test cases in only APL, but the output is the same for all of the APL, BQN and J functions.

9.1 average

```
// Translation (Tacit)
avg = phi(sum, divide, length)
// Translation (Explicit)
avg(x) = divide(sum(x), length(x))
```

```
⍝ APL
avg ← +/÷≢  ⍝ Φ
```

```
# BQN
Avg ← +´÷≠  # Φ
```

```
NB. J
avg =. +/%# NB. Φ
```

```
⍝ Test
      avg 1 2 3 4
2.5
```

9.2 plusOrMinus

```
// Translation (Tacit)
plusOrMinus = phi1(plus, concat, minus)
// Translation (Explicit)
plusOrMinus(x, y) =
    concat(plus(x,y), minus(x,y))
```

```
⍝ APL
plusOrMinus ← +,-  ⍝ Φ1
```

```
# BQN
PlusOrMinus ← +⋈-  # Φ1
```

```
NB. J
plusOrMinus =. +,- NB. Φ1
```

```
⍝ Test
      10 plusOrMinus 5
15 5
```

9.3 absoluteDifference

```
// Translation (Tacit)
absDiff = b1(abs, minus)
// Translation (Explicit)
absDiff(x, y) = abs(minus(x,y))
```

```
⍝ APL
absDiff ← |-    ⍝ B1
absDiff ← |⍨-   ⍝ B1
```

```
# BQN
AbsDiff ← |-    # B1
AbsDiff ← |∘-   # B1
```

```
NB. J
absDiff =. |@:- NB. B1
```

```
⍝ Tests
      10 absDiff 7
3
      7 absDiff 10
3
```

9.4 isPalindrome

```
// Translation (Tacit)
isPalindrome = phi(reverse, equals, i)
isPalindrome = s(equals, reverse)
// Translation (Explicit)
isPalindrome(x) = equals(reverse(x), i(x))
isPalindrome(x) = equals(x, reverse(x))
```

```
⍝ APL
isPalindrome ← ⍞≡⊢      ⍝ Φ I
isPalindrome ← ≡∘⍞⍨     ⍝ D W
```

```
# BQN
IsPalindrome ← ⍞≡⊢      # Φ I
IsPalindrome ← ≡⊸⍞      # S
IsPalindrome ← ≡⊸⍞˜     # D W
```

```
NB. J
isPalindrome =. |.-:]   NB. Φ I
isPalindrome =. -:|.    NB. S
isPalindrome =. (-:|.)~ NB. D W
```

```
⍝ Tests
      isPalindrome 'tacocat'
1
      isPalindrome 'tacodog'
0
```

9.5 isAnagram

```
// Translation (Tacit)
isAnagram = psi(equals, sort)
// Translation (Explicit)
isAnagram(x, y) = equals(sort(x), sort(y))
```

```
⍝ APL
sort      ← ⊂⍨⍋⌷⊢     ⍝ B1 Φ I
isAnagram ← ≡⍤sort    ⍝ Ψ
```

Each of the examples are run with one or two test cases in only APL, but the output is the same for all of the APL, BQN and J functions.

9.1 average

```
// Translation (Tacit)
avg = phi(sum, divide, length)
// Translation (Explicit)
avg(x) = divide(sum(x), length(x))
```

```
⍝ APL
avg ← +/÷≢   ⍝ ⌽
```

```
# BQN
Avg ← +´÷≠   # ⌽
```

```
NB. J
avg =. +/%# NB. ⌽
```

```
⍝ Test
      avg 1 2 3 4
2.5
```

9.2 plusOrMinus

```
// Translation (Tacit)
```

```
      7 absDiff
3
```

9.4 isPalindrome

```
// Translation (Ta
isPalindrome = phi
isPalindrome = s(e
// Translation (Ex
isPalindrome(x) =
isPalindrome(x) =
```

```
⍝ APL
isPalindrome ←
isPalindrome ←
```

```
# BQN
IsPalindrome ←
IsPalindrome ←
IsPalindrome ←
```

```
NB. J
isPalindrome =.
isPalindrome =.
isPalindrome =.
```

```
      avg 1 2 3 4
2.5
```

9.2 plusOrMinus

```
// Translation (Tacit)
plusOrMinus = phi1(plus, concat, minus)
// Translation (Explicit)
plusOrMinus(x, y) =
    concat(plus(x,y), minus(x,y))
```

```
⍝ APL
plusOrMinus ← +,-   ⍝ Φ1
```

```
# BQN
PlusOrMinus ← +⋈-   # Φ1
```

```
NB. J
plusOrMinus =. +,- NB. Φ1
```

```
⍝ Test
      10 plusOrMinus 5
15 5
```

```
NB. J
isPalindrome =.
isPalindrome =.
isPalindrome =.
```

```
⍝ Tests
      isPalindr
1
      isPalindr
0
```

9.5 isAnagram

```
// Translation (Tac
isAnagram = psi(equ
// Translation (Exp
isAnagram(x, y) = e
```

```
⍝ APL
sort      ← c⍥⍋
isAnagram ← ≡⍥s
```

```
3 11 ...

with B

 14 ⟩

|∘-⍨~
```

entheses are required. The ~ and
han function application and no

un with one or two test cases in
the same for all of the APL, BQN

9.3 absoluteDifference

```
// Translation (Tacit)
absDiff = b1(abs, minus)
// Translation (Explicit)
absDiff(x, y) = abs(minus(x,y))
```

```
⍝ APL
absDiff ← |-     ⍝ B1
absDiff ← |⍤-    ⍝ B1
```

```
# BQN
AbsDiff ← |-     # B1
AbsDiff ← |∘-    # B1
```

```
NB. J
absDiff =. |@:- NB. B1
```

```
⍝ Tests
      10 absDiff 7
3
      7 absDiff 10
3
```

9.4 isPalindrome

or all of the APL, BQN

x))

, minus)

9.4 isPalindrome

```
// Translation (Tacit)
isPalindrome = phi(reverse, equals, i)
isPalindrome = s(equals, reverse)
// Translation (Explicit)
isPalindrome(x) = equals(reverse(x), i(x))
isPalindrome(x) = equals(x, reverse(x))
```

```
⍝ APL
isPalindrome ← ⍱≡⊢       ⍝ Φ I
isPalindrome ← ≡∘⌽⍨      ⍝ D W
```

```
# BQN
IsPalindrome ← ⍱≡⊢       # Φ I
IsPalindrome ← ≡⊸⌽       # S
IsPalindrome ← ≡⊸⌽˜      # D W
```

```
NB. J
isPalindrome =. |.-:]    NB. Φ I
isPalindrome =. -:|.     NB. S
isPalindrome =. (-:|.)~  NB. D W
```

```
⍝ Tests
      isPalindrome 'tacocat'
```

```
NB. J
isPalindrome =. |.-:]    NB. Φ I
isPalindrome =. -:|.     NB. S
isPalindrome =. (-:|.)~ NB. D W

⍝ Tests
      isPalindrome 'tacocat'
1
      isPalindrome 'tacodog'
0
```

9.5 isAnagram

```
// Translation (Tacit)
isAnagram = psi(equals, sort)
// Translation (Explicit)
isAnagram(x, y) = equals(sort(x), sort(y))
```

```
⍝ APL
sort      ← ⊂⍤⍋⌷⊢      ⍝ B1 Φ I
isAnagram ← ≡⍤sort     ⍝ Ψ
```

```
# BQN
IsAnagram ← ≡○∧       # Ψ
# J
sort       =. /:~       NB. W
isAnagram =. -:&:sort NB. Ψ
A Tests
      'owls' isAnagram 'slow'
1
      'cats' isAnagram 'dogs'
0
```

9.6 isDisjoint

```
// Translation (Tacit)
isDisjoint = e(nil, equals, intersect)
// Translation (Explicit)
isDisjoint(x, y) = equals(nil, intersect(x, y))
A APL
isDisjoint ← ⍬≡∩    A E
# BQN
IsDisjoint ← ⟨⟩≡∈/⊣ # E Φ1
IsDisjoint ← ¬∘∨´∈  # B B1
NB. J
isDisjoint =. (i.0)-:e.#[ NB. E Φ1
A Tests
      1 2 isDisjoint 3 4 5
1
      2 3 isDisjoint 3 4 5
0
```

9.7 isPrefixOf

```
// Translation (Tacit)
isPrefixOf = b1(first, find)
// Translation (Explicit)
isPrefixOf(x, y) = first(find(x, y))
A APL
isPrefixOf ← ⊃⍷  A B1
isPrefixOf ← ⊃⍤⍷ A B1
# BQN
IsPrefixOf ← ⊑⍷  # B1
IsPrefixOf ← ⊑⍤⍷ # B1
NB. J
isPrefixOf =. {.@:E. NB. B1
A Tests
      'cat' isPrefixOf 'catch'
1
      'dog' isPrefixOf 'catch'
0
```

10 Summary

In this paper, a brief history of both array languages and combinatory logic was provided. The relationships between combinators were examined and captured visually by the combinator hierarchy in Figure 2. Combinators as they exist in the modern array languages Dyalog APL, J and BQN were enumerated and explained and differences were highlighted where they existed. Finally, the power and expressivity of combinators in modern array languages was examined.

Acknowledgments

Thank you to the following individuals who have provided reviews and meaningful feedback: Adám Brudzewsky, Computer Programmer at Dyalog Limited and creator of Extended Dyalog APL[10] and APLCart[11]; Marshall Lochbaum, creator of the I[23] and BQN[24] programming languages; Dr. Troels Henriksen, assistant professor at DIKU and creator of the Futhark[1] programming language; and Dr. David Mason, Chair of Computer Science at Toronto Metropolitan University.

References

[1] 2013. The Futhark Programming Language. https://futhark-lang.org/
[2] 2021. Family tree of array languages - APL Wiki. https://aplwiki.com/wiki/Family_tree_of_array_languages
[3] 2021. K - APL Wiki. https://aplwiki.com/wiki/K
[4] 2021. Timeline of APL dialects - APL Wiki. https://aplwiki.com/wiki/Timeline_of_APL_dialects
[5] 2022. Dyalog APL - APL Wiki. https://aplwiki.com/wiki/Dyalog_APL
[6] 2022. List of language developers - APL Wiki. https://aplwiki.com/wiki/List_of_language_developers
[7] Alexander Aiken, John H Williams, and Edward L Wimmers. 1994. *The FL project: The design of a functional language.* Citeseer.
[8] John Backus. 1978. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Commun. ACM* 21, 8 (1978), 613–641. Publisher: ACM New York, NY, USA.
[9] John Backus. 1985. From function level semantics to program transformation and optimization. In *Colloquium on Trees in Algebra and Programming*. Springer, 60–91.
[10] Adám Brudzewsky. 2018. dyalog-apl-extended: Dyalog APL Extended. https://github.com/abrudz/dyalog-apl-extended
[11] Adám Brudzewsky. 2019. APLcart - Find your way in APL. https://aplcart.info/
[12] Haskell B Curry. 1929. An analysis of logical substitution. *American journal of mathematics* 51, 3 (1929), 363–384. Publisher: JSTOR.
[13] Haskell Brooks Curry. 1930. Grundlagen der kombinatorischen Logik. *American journal of mathematics* 52, 4 (1930), 789–834. Publisher: JSTOR.
[14] Haskell Brooks Curry. 1931. The universal quantifier in combinatory logic. *Annals of Mathematics* (1931), 154–180. Publisher: JSTOR.
[15] Haskell Brooks Curry. 1932. Some additions to the theory of combinators. *American Journal of Mathematics* 54, 3 (1932), 551–558. Publisher: JSTOR.
[16] Haskell B. Curry. 1941. A Revision of the Fundamental Rules of Combinatory Logic. *The Journal of Symbolic Logic* 6, 2 (1941), 41–53. http://www.jstor.org/stable/2266655 Publisher: Association for Symbolic Logic.
[17] Haskell B Curry. 1942. The combinatory foundations of mathematical logic. *The Journal of Symbolic Logic* 7, 2 (1942), 49–64. Publisher: Cambridge University Press.
[18] Haskell B Curry. 1948. A simplification of the theory of combinators. *Synthese* (1948), 391–399. Publisher: JSTOR.

```
# BQN
IsAnagram ← ≡O∧        # Ψ
# J
sort       =. /:~      NB. W
isAnagram =. -:&:sort NB. Ψ
A Tests
      'owls' isAnagram 'slow'
1
      'cats' isAnagram 'dogs'
0
```

9.6 isDisjoint

```
// Translation (Tacit)
isDisjoint = e(nil, equals, intersect)
// Translation (Explicit)
isDisjoint(x, y) = equals(nil, intersect(x, y))
A APL
```

combinators were examined an
combinator hierarchy in Figure 2
in the modern array languages D
enumerated and explained and d
where they existed. Finally, the
combinators in modern array la

Acknowledgments

Thank you to the following i
vided reviews and meaningful fe
Computer Programmer at Dy
of Extended Dyalog APL[10] a
Lochbaum, creator of the I[23
ming languages; Dr. Troels Henr
DIKU and creator of the Futhark
and Dr. David Mason, Chair of C
Metropolitan University.

9.6 isDisjoint

```
// Translation (Tacit)
isDisjoint = e(nil, equals, intersect)
// Translation (Explicit)
isDisjoint(x, y) = equals(nil, intersect(x, y))
```

```
⍝ APL
isDisjoint ← ⍬≡∩     ⍝ E
```

```
# BQN
IsDisjoint ← ⟨⟩≡∊/¬ # E Φ1
IsDisjoint ← ¬·∨´∊  # B B1
```

```
NB. J
isDisjoint =. (i.0)-:e.#[ NB. E Φ1
```

```
⍝ Tests
      1 2 isDisjoint 3 4 5
1
      2 3 isDisjoint 3 4 5
0
```

9.7 isPrefixOf

```
// Translation (Tacit)
isPrefixOf = b1(first, find)
// Translation (Explicit)
```

Computer Programmer
of Extended Dyalog AP
Lochbaum, creator of
ming languages; Dr. Troe
DIKU and creator of the
and Dr. David Mason, Ch
Metropolitan University.

References

[1] 2013. The Futhark Progra
[2] 2021. Family tree of arra
com/wiki/Family_tree_of
[3] 2021. K - APL Wiki. http
[4] 2021. Timeline of APL dia
Timeline_of_APL_dialect
[5] 2022. Dyalog APL - APL W
[6] 2022. List of language de
wiki/List_of_language_de
[7] Alexander Aiken, John H
The FL project: The design
[8] John Backus. 1978. Can p
mann style? A functional
ACM 21, 8 (1978), 613–64
[9] John Backus. 1985. From
formation and optimizati
Programming. Springer, 6

```
0
```

9.7 isPrefixOf

```
// Translation (Tacit)
isPrefixOf = b1(first, find)
// Translation (Explicit)
isPrefixOf(x, y) = first(find(x, y))
```

```
⍝ APL
isPrefixOf ← ⊃⍷  ⍝ B1
isPrefixOf ← ⊃⍤⍷ ⍝ B1
```

```
# BQN
IsPrefixOf ← ⊏⍷  # B1
IsPrefixOf ← ⊏⊸⍷ # B1
```

```
NB. J
isPrefixOf =. {.@:E. NB. B1
```

```
⍝ Tests
      'cat' isPrefixOf 'catch'
1
      'dog' isPrefixOf 'catch'
0
```

10 Summary

In this paper, a brief history of both array languages and

The FL project: The design of a

[8] John Backus. 1978. Can prog
mann style? A functional sty
ACM 21, 8 (1978), 613–641.

[9] John Backus. 1985. From fu
formation and optimization.
Programming. Springer, 60–9

[10] Adám Brudzewsky. 2018. dya
https://github.com/abrudz/d

[11] Adám Brudzewsky. 2019. A
//aplcart.info/

[12] Haskell B Curry. 1929. An a
journal of mathematics 51, 3

[13] Haskell Brooks Curry. 1930.
American journal of mathem
JSTOR.

[14] Haskell Brooks Curry. 1931.
logic. *Annals of Mathematics*

[15] Haskell Brooks Curry. 1932.
tors. *American Journal of Ma*
JSTOR.

[16] Haskell B. Curry. 1941. A
Combinatory Logic. *The Jo*
53. http://www.jstor.org/st
Symbolic Logic.

[17] Haskell B Curry. 1942. The co
logic. *The Journal of Symbo*
Cambridge University Press.

[18] Haskell B Curry. 1948. A sim

[[end of  talks]]

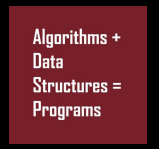
What is the best combinator programming language?

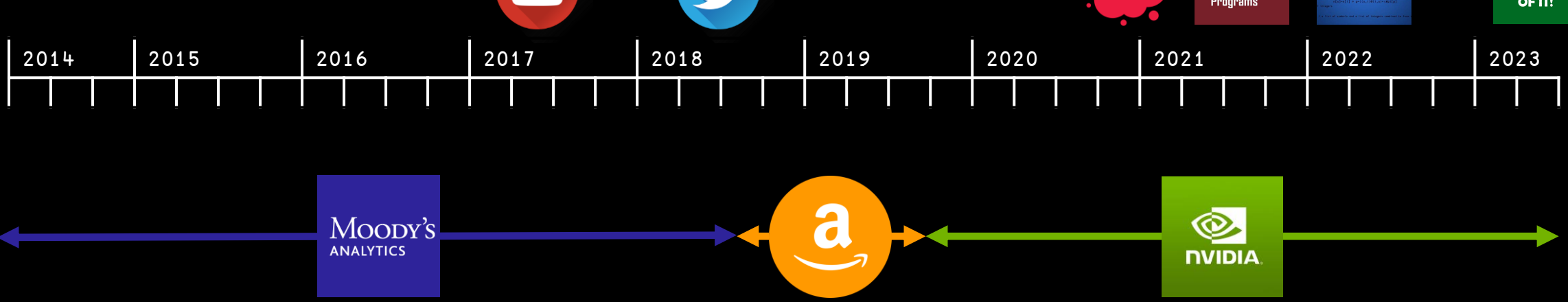
How did I discover array languages (and fall in love with them)?

https://www.youtube.com/watch?v=NiferfBvN3s

Dec 2019

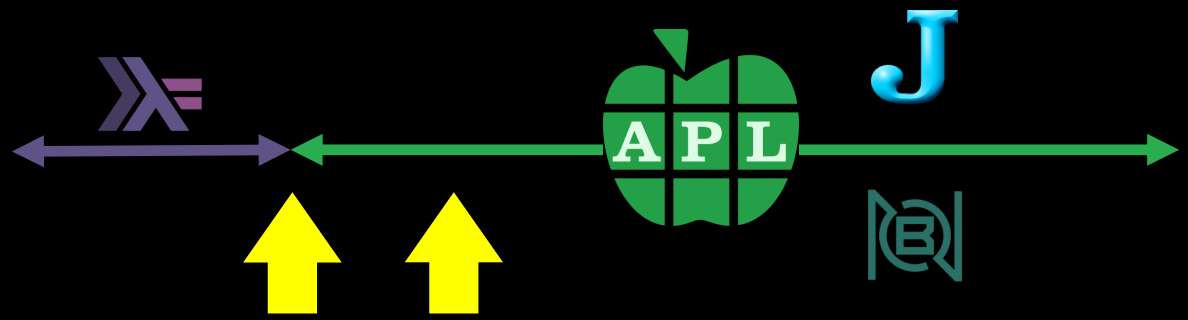
How did I discover combinators?

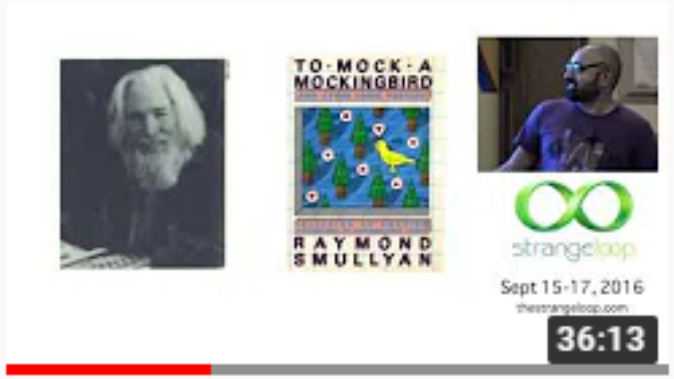




About Me

Conor Hoekstra / @code_report



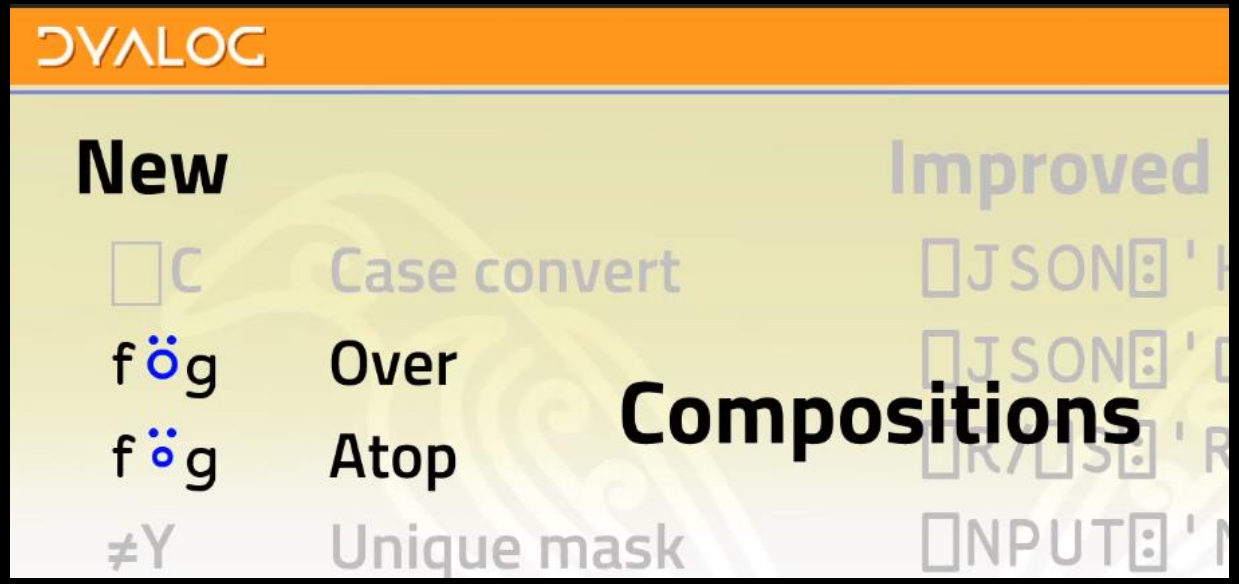


"Point-Free or Die: Tacit Programming in Haskell and Beyond" by Amar Shah

Strange Loop • 14K views • 3 years ago

Tacit programming, or programming in the "point-free" style, allows you to define a function without reference to one or more of its ...

July 5, 2020



Dyalog 18.0 Webinar
May 14, 2020

July 5, 2020

Sometime later in July

August 12

Combinatory logic

From Wikipedia, the free encyclopedia

Not to be confused with combinational logic, a topic in digital electronics.

Combinatory logic is a notation to eliminate the need for quantified variables in mathematical logic. It was introduced by Moses Schönfinkel[1] and Haskell Curry,[2] and has more recently been used in computer science as a theoretical model of computation and also as a basis for the design of functional programming languages. It is based on **combinators**, which were introduced by Schönfinkel in 1920 with the idea of providing an analogous way to build up functions—and to remove any mention of variables—particularly in predicate logic. A combinator is a higher-order function that uses only function application and earlier defined combinators to define a result from its arguments.

In mathematics [edit]

Combinatory logic was originally intended as a 'pre-logic' that would clarify the role of quantified variables in logic, essentially by eliminating them. Another way of eliminating quantified variables is Quine's predicate functor logic. While the expressive power of combinatory logic typically exceeds that of first-order logic, the expressive power of predicate functor logic is identical to that of first order logic (Quine 1960, 1966, 1976).

The original inventor of combinatory logic, Moses Schönfinkel, published nothing on combinatory logic after his original 1924 paper. Haskell Curry rediscovered the combinators while working as an instructor at Princeton University in late 1927.[3] In the late 1930s, Alonzo Church and his students at Princeton invented a rival formalism for functional abstraction, the lambda calculus, which proved more popular than combinatory logic. The upshot of these historical contingencies was that until theoretical computer science began taking an interest in combinatory logic in the 1960s and 1970s, nearly all work on the subject was by Haskell Curry and his students, or by Robert Feys in Belgium. Curry and Feys (1958), and Curry *et al.* (1972) survey the early history of combinatory logic. For a more modern treatment of combinatory logic and the lambda calculus together, see the book by Barendregt,[4] which reviews the models Dana Scott devised for combinatory logic in the 1960s and 1970s.

In computing [edit]

In computer science, combinatory logic is used as a simplified model of computation, used in computability theory and proof theory. Despite its simplicity, combinatory logic captures many essential features of computation.

Combinatory logic can be viewed as a variant of the lambda calculus, in which lambda expressions (representing functional abstraction) are replaced by a limited set of *combinators*, primitive functions without free variables. It is easy to transform lambda expressions into combinator expressions, and combinator reduction is much simpler than lambda reduction. Hence combinatory logic has been used to model some non-strict functional programming languages and hardware. The purest form of this view is the programming language Unlambda, whose sole primitives are the S and K combinators augmented with character input/output. Although not a practical programming language, Unlambda is of some theoretical interest.

Applications [edit]

Compilation of functional languages [edit]

David Turner used his combinators to implement the SASL programming language.

Kenneth E. Iverson used primitives based on Curry's combinators in his J programming language, a successor to APL. This enabled what Iverson called tacit programming, that is, programming in functional expressions containing no variables, along with powerful tools for working with such programs. It turns out that tacit programming is possible in any APL-like language with user-defined operators.[9]

Logic [edit]

The Curry–Howard isomorphism implies a connection between logic and programming: every proof of a theorem of intuitionistic logic corresponds to a reduction of a typed lambda term, and conversely. Moreover, theorems can be identified with function type signatures. Specifically, a typed combinatory logic corresponds to a Hilbert system in proof theory.

The **K** and **S** combinators correspond to the axioms

AK: $A \to (B \to A)$,
AS: $(A \to (B \to C)) \to ((A \to B) \to (A \to C))$,

and function application corresponds to the detachment (modus ponens) rule

MP: from A and $A \to B$ infer B.

The calculus consisting of **AK**, **AS**, and **MP** is complete for the implicational fragment of the intuitionistic logic, which can be seen as follows. Consider the set W of all deductively closed sets of formulas, ordered by inclusion. Then $\langle W, \subseteq \rangle$ is an intuitionistic Kripke frame, and we define a model $\Vdash$ in this frame by

$$X \Vdash A \iff A \in X.$$

This definition obeys the conditions on satisfaction of $\to$: on one hand, if $X \Vdash A \to B$, and $Y \in W$ is such that $Y \supseteq X$ and $Y \Vdash A$, then $Y \Vdash B$ by modus ponens. On the other hand, if $X \nVdash A \to B$, then $X, A \nvdash B$ by the deduction theorem, thus the deductive closure of $X \cup \{A\}$ is an element $Y \in W$ such that $Y \supseteq X$, $Y \Vdash A$, and $Y \nVdash B$.

Let A be any formula which is not provable in the calculus. Then A does not belong to the deductive closure X of the empty set, thus $X \nVdash A$, and A is not intuitionistically valid.

See also [edit]

- Applicative computing systems
- B, C, K, W system
- Categorical abstract machine
- Combinatory categorial grammar
- Explicit substitution

Applications [edit]

Compilation of functional languages [edit]

David Turner used his combinators to implement the SASL programming language.

Kenneth E. Iverson used primitives based on Curry's combinators in his J programming language, a successor to APL. This enabled what Iverson called tacit programming, that is, programming in functional expressions containing no variables, along with powerful tools for working with such programs. It turns out that tacit programming is possible in any APL-like language with user-defined operators.[9]

Logic [edit]

The Curry–Howard isomorphism implies a connection between logic and programming: every proof of a theorem of intuitionistic logic corresponds to a reduction of a typed lambda term, and conversely. Moreover, theorems can be identified with function type signatures. Specifically, a typed combinatory logic corresponds to a Hilbert system in proof theory.

The **K** and **S** combinators correspond to the axioms

AK: $A \to (B \to A)$,
AS: $(A \to (B \to C)) \to ((A \to B) \to (A \to C))$,

and function application corresponds to the detachment (modus ponens) rule

MP: from A and $A \to B$ infer B.

The calculus consisting of **AK**, **AS**, and **MP** is complete for the implicational fragment of the intuitionistic logic, which can be seen as follows. Consider the set W of all deductively closed sets of formulas, ordered by inclusion. Then $\langle W, \subseteq \rangle$ is an intuitionistic Kripke frame, and we define a model $\Vdash$ in this frame by

$$X \Vdash A \iff A \in X.$$

curry combinators and the "j programming language"

All Images News Videos Shopping More Settings Tools

About 1,380 results (0.37 seconds)

en.wikipedia.org › wiki › Combinatory_logic

Combinatory logic - Wikipedia

Combinatory logic is a notation to eliminate the need for quantified variables in mathematical ... Haskell **Curry** rediscovered the **combinators** while working as an instructor at Princeton University in late ... Kenneth E. Iverson used primitives based on **Curry's combinators** in his **J programming language**, a successor to APL.

You've visited this page 2 times. Last visit: 12/08/20

link.springer.com › chapter

Reduction languages and reduction systems | SpringerLink

May 28, 2005 - Backus, **J.: 'Programming Language** Semantics and Closed Applicative Languages'; San ... **Curry**, H.B.; Feys, R.: 'Combinatory Logic'; Vol. ... Hindley, J.R.; Seldin, J.P.: 'Introduction to **Combinators** and λ-Calculus'; London ...

by W Kluge - 1987 - Cited by 1 - Related articles

wiki.tcl-lang.org › page › Tacit+programming

Tacit programming - the Tcler's Wiki! - Tcl/Tk

Mar 25, 2014 - ... article [L2] about the **J programming language** (the "blessed successor" to APL, ... Only implicitly present is a powerful function **combinator** called "fork". ... to Schönfinkel/**Curry's** S **combinator** (see Hot **Curry** and **Combinator** ...

if 0 {As KBK pointed out in the Tcl chatroom, the "hook" pattern corresponds to Schönfinkel/Curry's S combinator (see Hot Curry and Combinator Engine), while "fork" is called S' there.

Google Scholar "combinatory logic" "combinator" "APL"

Articles About 61 results (0.03 sec)

Any time
Since 2020
Since 2019
Since 2016
Custom range...

Sort by relevance
Sort by date

include patents
include citations

Create alert

Pure functions in APL and J

[PDF] acm.org

E Cherlin - Proceedings of the international conference on APL'91, 1991 - dl.acm.org

... Any expression in **combinatory logic** made up of combinators and variables can be abstracted into apure **combinator** expres- sion applied to asequence of variables. Because there are great similarities between combinators and certain **APL** operators, a simihm result obtains in ...

☆ 99 Cited by 3 Related articles All 4 versions

[PDF] History of lambda-calculus and combinatory logic

[PDF] simons-rock.edu

F Cardone, JR Hindley - Handbook of the History of Logic, 2006 - hope.simons-rock.edu

... **Combinatory logic** was invented by Moses Ilyich Schönfinkel ... 2In today's notation (following [Curry and Feys, 1958, Ch.5]) their axiom-schemes are BXY Z = X(YZ), CXY Z = XZY , IX = X, KXY = X, SXY Z = XZ(YZ), and a **combinator** is any applicative combination of basic ...

☆ 99 Cited by 77 Related articles All 10 versions »

APL trivia

[PDF] acm.org

E Cherlin - ACM SIGAPL **APL** Quote Quad, 1990 - dl.acm.org

... The branch of mathematics that makes the most of trivia is Curry's **combinatory logic** [Cu58]. A **combinator** is, approximately, a trivial operator ... Each of these combinators corresponds somewhat to a Diction- ary **APL** function or operator, as also shown in Table 2, although ...

☆ 99 Cited by 3 Related articles All 4 versions

[PDF] Phrasal forms

[PDF] acm.org

EE McDonnell, KE Iverson - Conference proceedings on **APL** as a tool of ..., 1989 - dl.acm.org

... In **combinatory logic** one of the most useful primitive combinators is designated by S [Sch24] ... (The constancy **combinator** K is defined in infur notation so that cKz has the value c for all x.) Users of **APL** will appreciate the hook for the same reasons ...

☆ 99 Cited by 14 Related articles All 3 versions

Phrasal Forms

K. E. Iverson
Toronto

E. E. McDonnell
I. P. Sharp Associates
Palo Alto

NOTE: In this paper we use the linguistic terms *verb* and *pronoun* interchangeably with the mathematical terms *function* and *variable.*

INTRODUCTION

In standard APL [ISO88] certain forms are ungrammatical, and new definitions could be adopted for them without conflict. Such definitions we shall call *phrasal forms* [AHD76]. For example, if b and c are pronouns, the phrase $b\ c$ is meaningless, and in APL2 [IBM85] the definition $(\subset b), (\subset c)$, where $\subset$ is the APL2 *enclose* function, is adopted for it.

VERB RANK

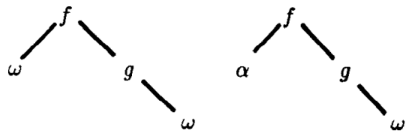
The notion of *verb rank*, first introduced by Iverson [Iv78], later elaborated by Keenan [Ke79], and further evolved by Whitney [Wh84], has been adopted by Iverson in his Dictionary [Iv87]. It refers to the rank of the subarrays of an argument which are the cells to which the verb aplies. For example, the cells that negate applies to are items, and items are ranlk zero objects, and thus we say the rank of negate is zero. Similarly, the cells to which reverse applies are lists, or rank ome objects, and thus we say the rank of reverse is one. Not only primitive verbs, but also derived and defined verbs have rank. The idea is a powerful one, producing great simplifications, and so we define the ranks of the new constructions we describe herein.

DEFINITIONS

In the following definitions, f, g, and h denote verbs and α and ω denote pronouns.

HOOK. A *hook* is denoted by $f\ g$ and is defined formally by identities and informally by hook-shaped diagrams as follows:

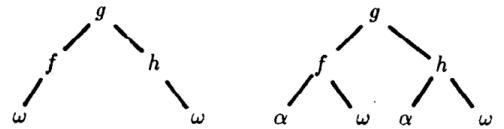
$$(fg)\omega \leftrightarrow \omega f g\omega; \qquad \alpha(fg)\omega \leftrightarrow \alpha f g\omega$$



The rank of the hook $f\ g$ is the maximum of the ranks of f and g. Note that the verb is used monadically.

FORK. A *fork* is denoted by $f\ g\ h$ and is defined formally by identities and informally by forking diagrams as follows:

$$(fgh)\omega \leftrightarrow (f\omega)g(h\omega); \qquad \alpha(fgh)\omega \leftrightarrow (\alpha f\omega)g(\alpha h\omega)$$



The rank of the fork $f\ g\ h$ is the maximum of the ranks of f and g. Note that the central verb is used dyadically or monadically according to whether the fork is applied to two arguments or one. Parenthesis are required around hook and fork forms only to avoid ambiguity.

DISCUSSION OF HOOK AND FORK

HOOK. In combinatory logic one of the most useful primitive combinators is designated by **S** [Sch24]. Curry defines $\mathbf{S}fgx$ in prefix notation to be $fx(gx)$[CuFeCr74]. In common mathematical infix notation this would be given by $(x)f(g(x))$, which one can write in APL as $xfgx$, and this is the hook form $(fg)x$. The combinatory logician appreciates this form because of its great expressiveness: it can be shown that **S**, along with **K**, the *constancy* combinator, suffice to define all other combinators of interest [Ro50]. (The constancy combinator **K** is defined in infix notation so that $c\mathbf{K}x$ has the value c for all x.) Users of APL will appreciate the hook for the same reasons.

For example, **+÷** adds the reciprocal of the right argument to the left argument, a form used in describing continued fractions. Thus $(+\div)\backslash 3\ 7\ 16\ {}^{-}294$ gives the first four convergents to pi, which, to nine decimals, are 3, 3.1412857143, 3.14159292, and 3.141592654. Further, $=\lfloor$ is a proposition that tests whether its argument is an integer, the number of primes less than positive integer ω is approximately $(\div\circledast)\omega$; and to decompose a number ω into numerator and denominator, one can write $(\div\vee/)\omega, 1$ (where $\vee$ is the *greatest common divisor).*

FORK. The forks $f + h$ and $f \times h$ and $f \div h$ provide formal treatment of the identical but informal phrases used in mathematics [e.g. Ef89] for the sum and product and quotient,

er evolved by
his Dictionary
an argument
example, the
are ranlk zero
ero. Similarly,
ome objects,
only primitive
ank. The idea
ns, and so we
escribe herein.

rbs and α and

d formally by
ns as follows:

$fg\omega$

and g. Note that the central verb is used dyadically or monadically according to whether the fork is applied to two arguments or one. Parenthesis are required around hook and fork forms only to avoid ambiguity.

DISCUSSION OF HOOK AND FORK

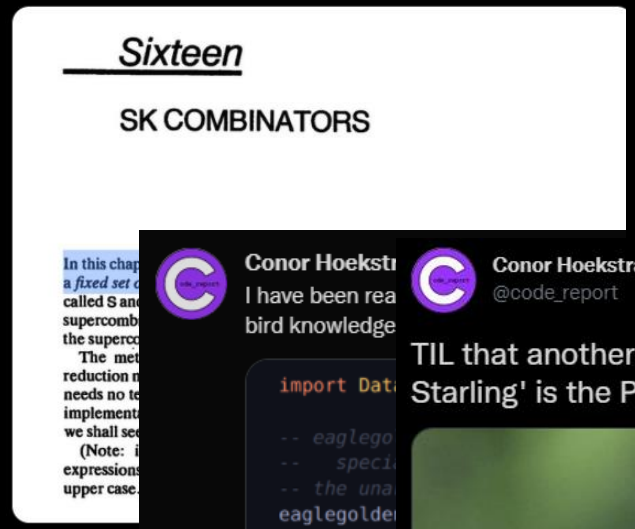
HOOK. In combinatory logic one of the most useful primitive combinators is designated by **S** [Sch24]. Curry defines $\mathbf{S}fgx$ in prefix notation to be $fx(gx)$[CuFeCr74]. In common mathematical infix notation this would be given by $(x)f(g(x))$, which one can write in APL as $xfgx$, and this is the hook form $(fg)x$. The combinatory logician appreciates this form because of its great expressiveness: it can be shown that **S**, along with **K**, the *constancy* combinator, suffice to define all other combinators of interest [Ro50]. (The constancy combinator **K** is defined in infix notation so that $c\mathbf{K}x$ has the value c for all x.) Users of APL will appreciate the hook for the same reasons.

For example, +÷ adds the reciprocal of the right argument to the left argument, a form used in describing continued fractions. Thus (+÷)\3 7 16 ¯294 gives the first four con-

Curry [Cu31] defines a *formalizing combinator*, Φ, in prefix notation, such that $\Phi fghx$ means $f(gx)(hx)$. In common mathematical infix notation this would be designated by $(g(x))f(h(x))$. An example of this form is $\Phi+sin^2cos^2\theta$, meaning $sin^2\theta + cos^2\theta$. The fork $(f\ g\ h)\omega$ has the same meaning, namely $(f\omega)g(h\omega)$. Curry named this the *formalizing combinator* because of its role in defining formal implication in terms of ordinary implication.

Conor Hoekstra
@code_report

For a second, I thought @simonpj0 had already implemented CEAF (Combinator-Enabled Algorithm Fusion) but alas the graph reduction he refers to is not that (it is reducing a program down to the S, K & I #combinators)



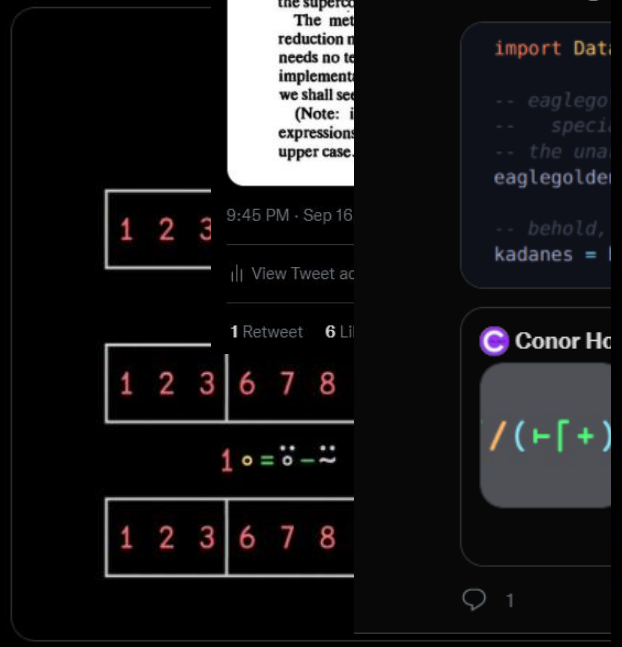
9:45 PM · Sep 16

View Tweet ac

1 Retweet 6 Li

Conor Ho
@code_rep

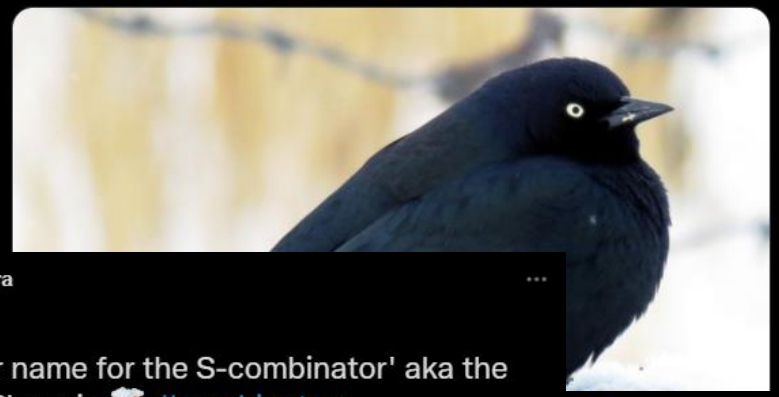
I am not sure
Combinatory
C, E, Ê, K an
Logic so muc



12:38 AM · Sep 24, 2021 · Twitter Web App

Conor Hoekstra
@code_report

And for maybe my most amazing combinator discovery to date, std::inner_product is the Blackbird 🤯 (which is what I was looking for when I stumbled across the Phoenix) #combinators #cpp #algorithms



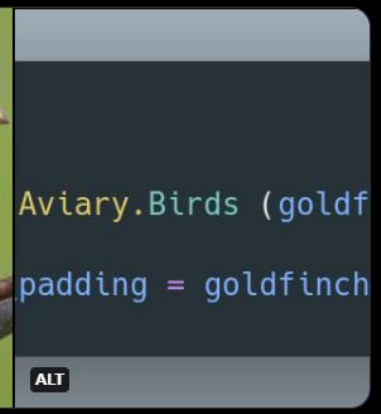
Conor Hoekstra
@code_report

TIL that another name for the S-combinator' aka the Starling' is the Phoenix 🤯 #combinators

9:06 PM · Oct 21, 2020 · Twitter Web App

Conor Hoekstra
@code_report

The Combinator Quest continues and I managed to use a new combinator today, the G-combinator, aka the Goldfinch 🙂



8:55 PM · Sep 10, 2020 · Twitter Web App

Conor Hoekstra
@code_report

In #APL, trains are ALL of the following:

- Bluebird (B combinator)
- Dove (D combinator)
- Phoenix (S' combinator)
- Blackbird (B1 combinator)
- Eagle (E combinator)
- Golden Eagle (E^ combinator specialization)

APL is a Combinatory Logic programming language.

2:56 AM · Sep 21, 2021 · Twitter Web App

A Brief History of Combinatory Logic (CL)
A Brief History of CL in Array Languages

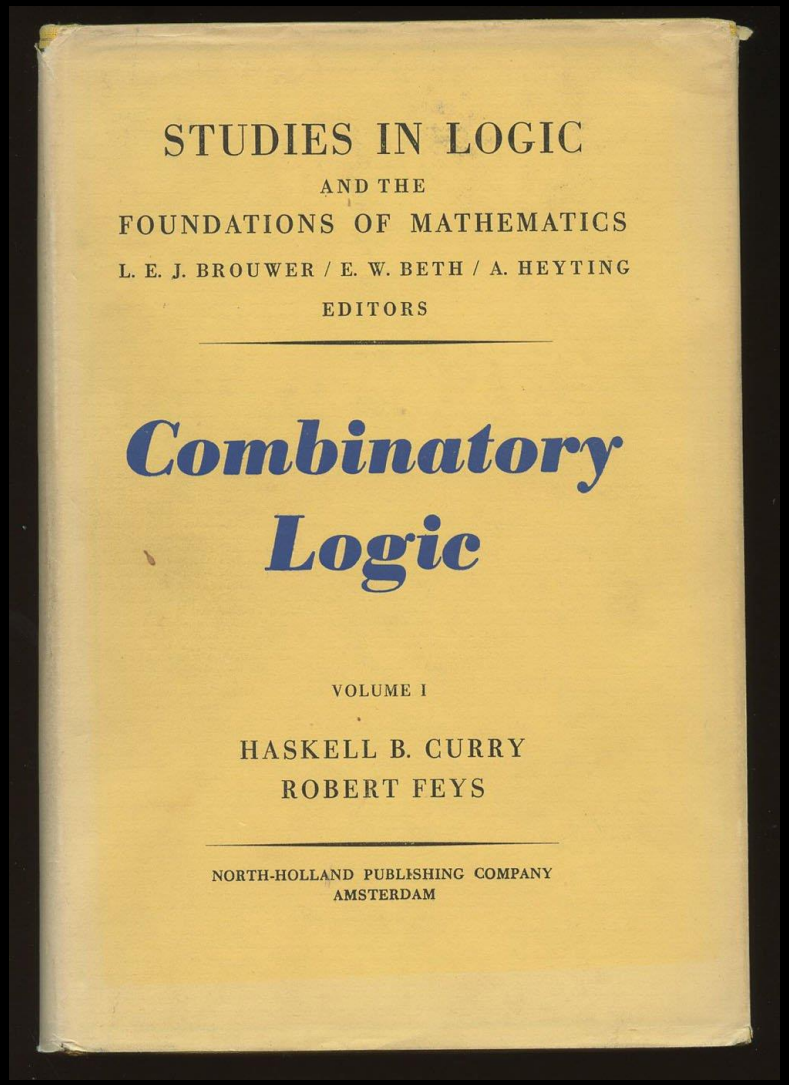
A Brief History of Combinatory Logic

Moses Schönfinkel
1888 - 1942

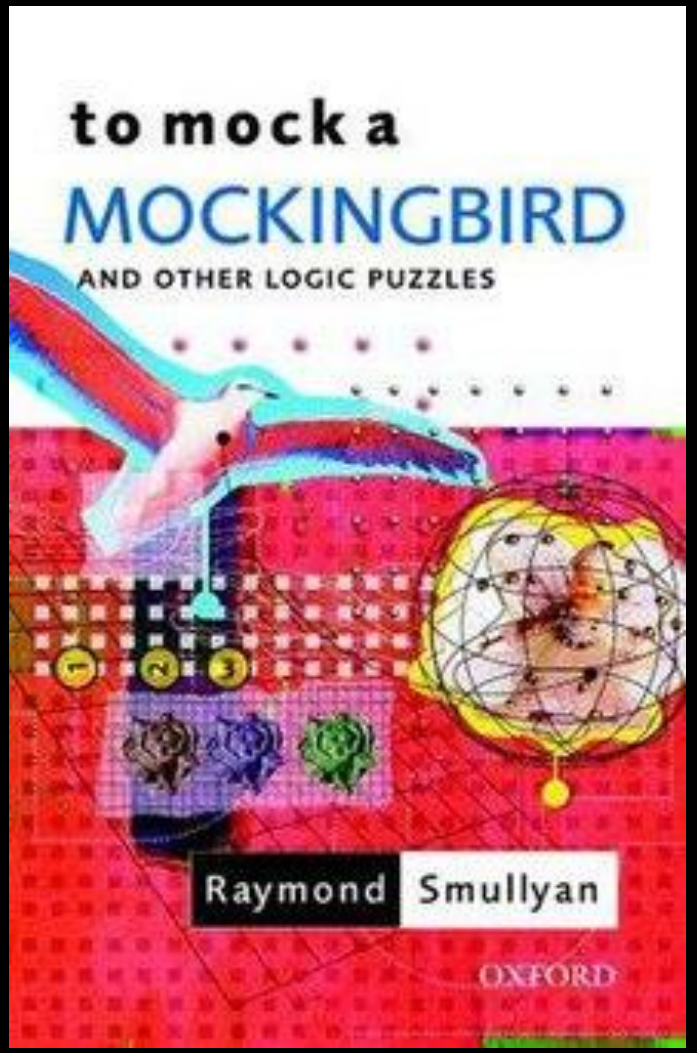
Moses Schönfinkel
1888 - 1942

Haskell Curry
1900-1982

Year	Author(s)	Title	Introduced
1924	Schönfinkel	On the building blocks of mathematical logic	S K I B C
1929	Curry	An Analysis of Logical Substitution	W
1930	Curry	The Foundations of Combinatory Logic	B_n (B_1, B_2, …)
1931	Curry	The Universal Quantifier in Combinatory Logic	Ψ, Φ_n (Φ, Φ_1, …)
1958	Curry and Feys	Combinatory Logic: Volume I	


STUDIES IN LOGIC
AND THE
FOUNDATIONS OF MATHEMATICS
L. E. J. BROUWER / E. W. BETH / A. HEYTING
EDITORS
Combinatory Logic
VOLUME I
HASKELL B. CURRY
ROBERT FEYS
NORTH-HOLLAND PUBLISHING COMPANY
AMSTERDAM

1958


to mock a
MOCKINGBIRD
AND OTHER LOGIC PUZZLES
Raymond Smullyan
OXFORD

1985

www.combinatorylogic.com/table.html

	Author	Year	Paper
Sch24	Moses Schönfinkel	1924	On the building blocks of mathematical logic
Cur29	Haskell Curry	1929	An Analysis of Logical Substitution
Cur30	Haskell Curry	1931	Grundlagen der Kombinatorischen Logik (The Foundations of Combinatory Logic)
Cur31	Haskell Curry	1931	The universal quantifier in combinatory logic
Cur48	Haskell Curry	1948	A Simplification of the Theory of Combinators
Cur58	H. Curry & R. Feys	1958	Combinatory Logic: Volume I
Tur78	David Turner	1979	Another algorithm for bracket abstraction
Smu85	Raymond Smullyan	1985	To Mock a Mockingbird
Iv89	K. Iverson & E. McDonnell	1989	Phrasal Forms
Loc12	Marshall Lochbaum	2012	Added hook, backhook

A Brief History of CL in Array Languages

Table 4. 2 and 3-trains in APL, BQN and J.

Year	Language	2-Train	3-Train
1990	J	S and D	Φ and Φ_1
2014	Dyalog APL	B and B_1	Φ and Φ_1
2020	BQN	B and B_1	Φ and Φ_1

Table 5. History of combinators in Dyalog APL.

Year	Version	Combinator	Spelling
1983	1.0	B, D, C	∘⍤
2003	10.0	W	⍨
2013	13.0	K	⊢
2014	14.0	B, B_1, Φ, $Φ_1$	trains
2020	18.0	B, B_1, Ψ, K	⍥⍤⍨

Combinator Table: Combinators, Birds, Spellings & More

Function Abstraction	Symbol	Bird	CR[1]	CH[2]	Elem[3]	Haskell	APL	BQN	Intro
λabc.ac(bc)	S	Starling	S	S		<*> / ap		∘⊢	Sch24
λab.a	K	Kestrel	K	K	✅	const	⊢	⊢	Sch24
λa.a	I	Identity	SKK	SKK	✅	id	⊣⊢	⊣⊢	Sch24
λab.b	κ	Kite	KI	KI			⊣	⊣	
λab.abb	W	Warbler	C(BMR)	CSK	✅	join	⍨	˜	Cur29
λabc.acb	C	Cardinal	S(BBS)(KK)	B(ΦBS)KK	✅	flip	⍨	˜	Sch24
λabc.a(bc)	B	Bluebird	S(KS)K	S(KS)K	✅	.	∘⍤2T	∘○ 2T	Sch24
λabcd.a(bcd)	B_1	Blackbird	BBB	DB		.:	⍤2T	∘ 2T	Cur30
λabcde.a(bcde)	B_2	Bunting	B(BBB)B	DB_1					Cur30
λabcd.a(b(cd))	B_3	Becard	B(BB)B	BDB					Cur30
λabcd.ab(cd)	D	Dove	BB	BB			∘	∘⊢	Smu85
λabcd.a(bd)(cd)	Φ	Phoenix	-	B_1SB		liftA2	3T	3T	Cur31
λabcd.a(bc)(bd)	Ψ	Psi	-	B(SΦCB)B		on	⍤	○	Cur31
λabcde.abc(de)	D_1	Dickcissel	B(BB)	BD					
λabcde.a(bc)(de)	D_2	Dovekies	BB(BB)	DD					Smu85
λabcde.ab(cde)	E	Eagle	B(BBB)	BB_1					Smu85
λabcde.a(bde)(cde)	$Φ_1$	Pheasant	-	BΦΦ			3T	3T	Cur31
λabcdefg.a(bcd)(efg)	Ê	Bald Eagle	B(BBB)(B(BBB))	D_2D_2D					Smu85

	APL	BQN	J
I	⊣⊢	⊣⊢	][
K	⊢	⊢	]
KI	⊣	⊣	[
S		⍥	2 Train
B	∘⍤2	∘○ 2	@: &:
C	⍨	˜	~
W	⍨	˜	~
B_1	⍤2	∘ 2	@:
D	∘	⍥	2 Train
Ψ	⍤	○	&:
Φ	3 Train	3 Train	3 Train
$Φ_1$	3 Train	3 Train	3 Train
D_2		⍥⍥	

	APL	BQN	J
I	⊣⊢	⊣⊢	][
K	⊢	⊢	]
KI	⊣	⊣	[
S		⍥	2 Train
B	∘⍤⍤ 2T	∘○ 2T	@: &:
C	⍨	˜	~
W	⍨	˜	~
B_1	⍤ 2T	∘ 2T	@:
D	∘	⍥	2 Train
Ψ	⍤	○	&:
Φ	3 Train	3 Train	3 Train
$Φ_1$	3 Train	3 Train	3 Train
D_2		⍥⍥	

	APL	BQN	J
I	⊣⊢	⊣⊢	][
K	⊢	⊢	]
KI	⊣	⊣	[
S		⍥	2 Train
B	∘⍤⍤ 2T	∘○ 2T	@: &:
C	⍨	˜	~
W	⍨	˜	~
B_1	⍤ 2T	∘ 2T	@:
D	∘	⍥	2 Train
Ψ	⍤	○	&:
Φ	3 Train	3 Train	3 Train
$Φ_1$	3 Train	3 Train	3 Train
D_2		⍥⍥	

	APL	BQN	J
I	⊣⊢	⊣⊢	][
K	⊢	⊢	]
KI	⊣	⊣	[
S		⍥	2 Train
B	∘⍤⍤ 2T	∘○ 2T	@: &:
C	⍨	˜	~
W	⍨	˜	~
B_1	⍤ 2T	∘ 2T	@:
D	∘	⍥	2 Train
Ψ	⍤	○	&:
Φ	3 Train	3 Train	3 Train
$Φ_1$	3 Train	3 Train	3 Train
D_2		⍥⍥	
Σ		⍥	
Δ		⍥	

What is a combinator?

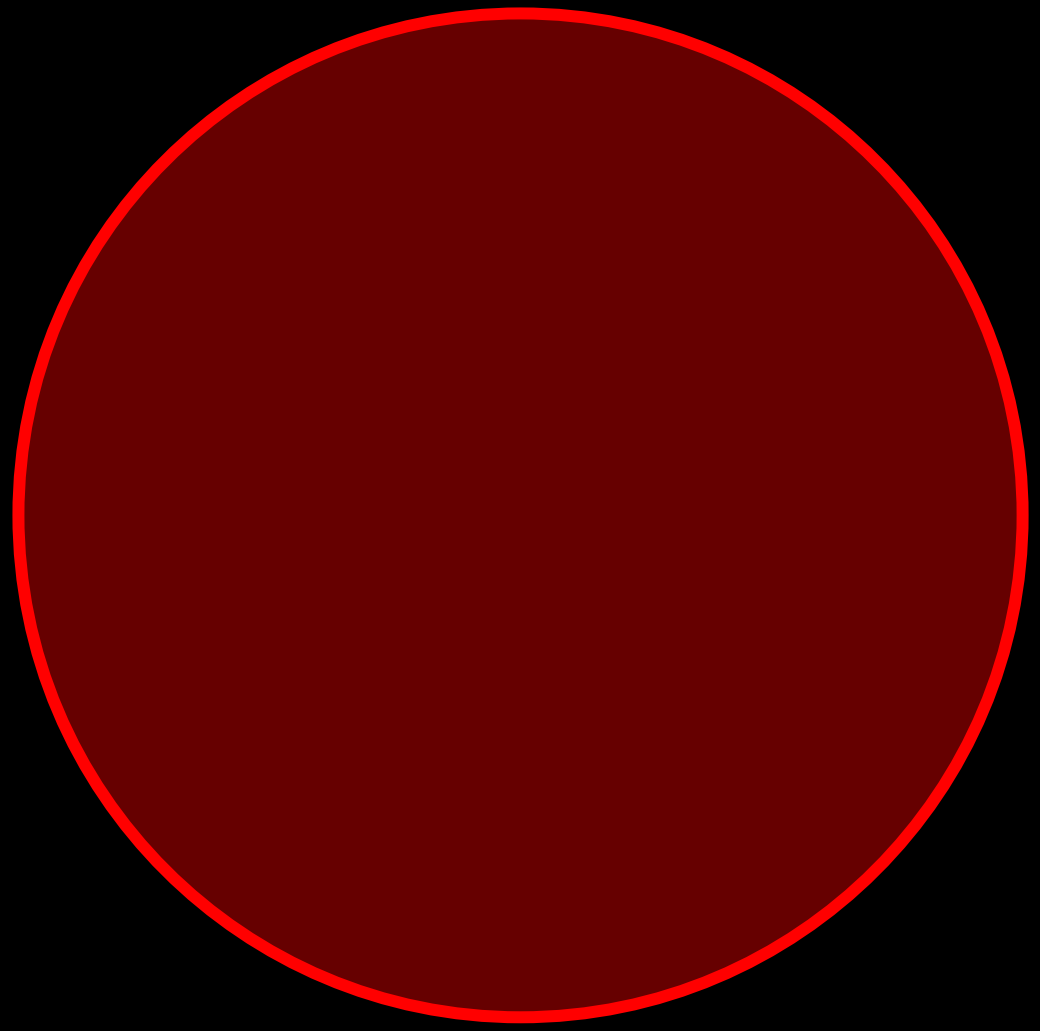
combinator: a lambda expression
containing no free variables

combinator: a function that deals only in its arguments

combinator = pure function?

pure function: same input = same output / no side effects

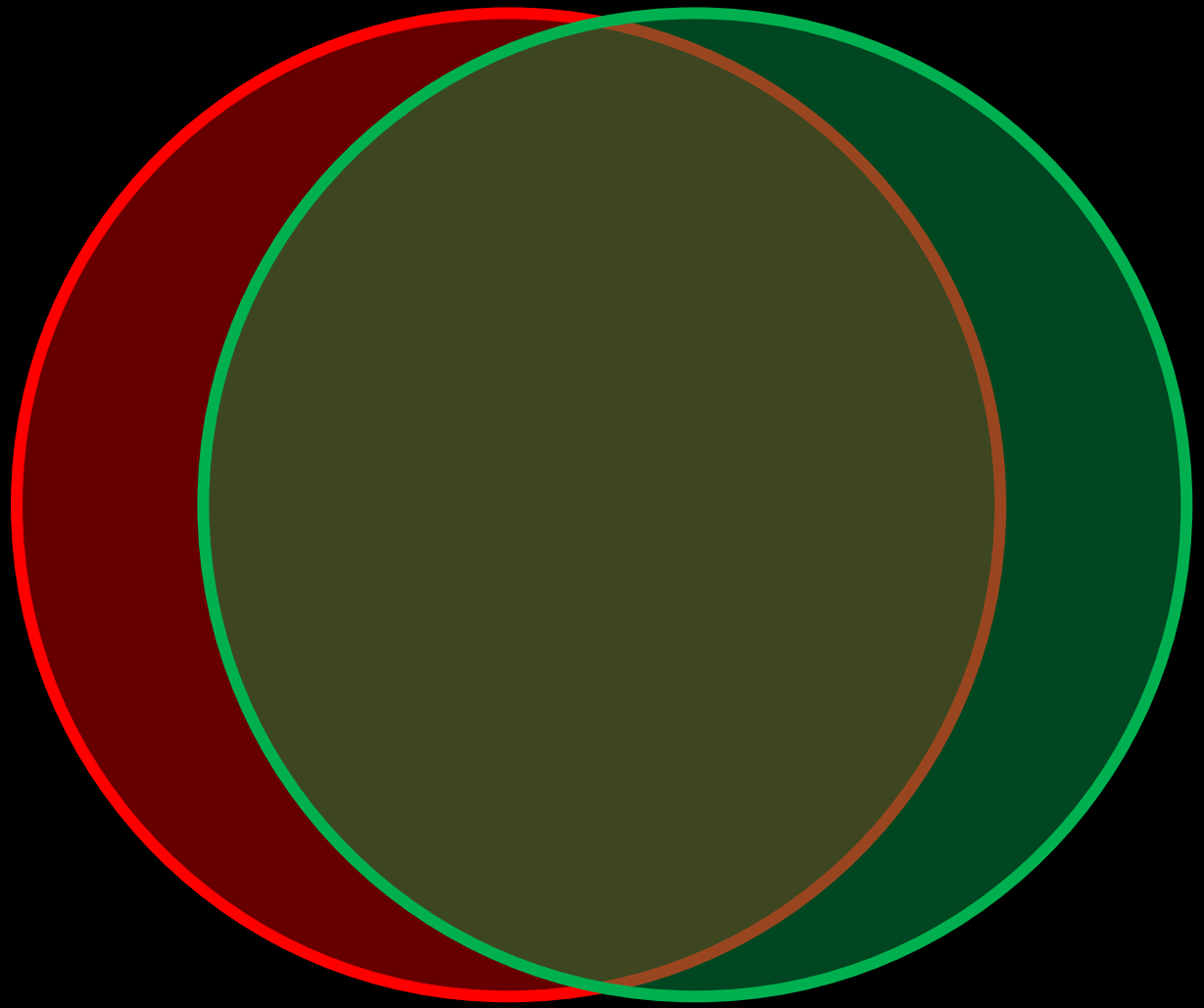
pure function



combinator

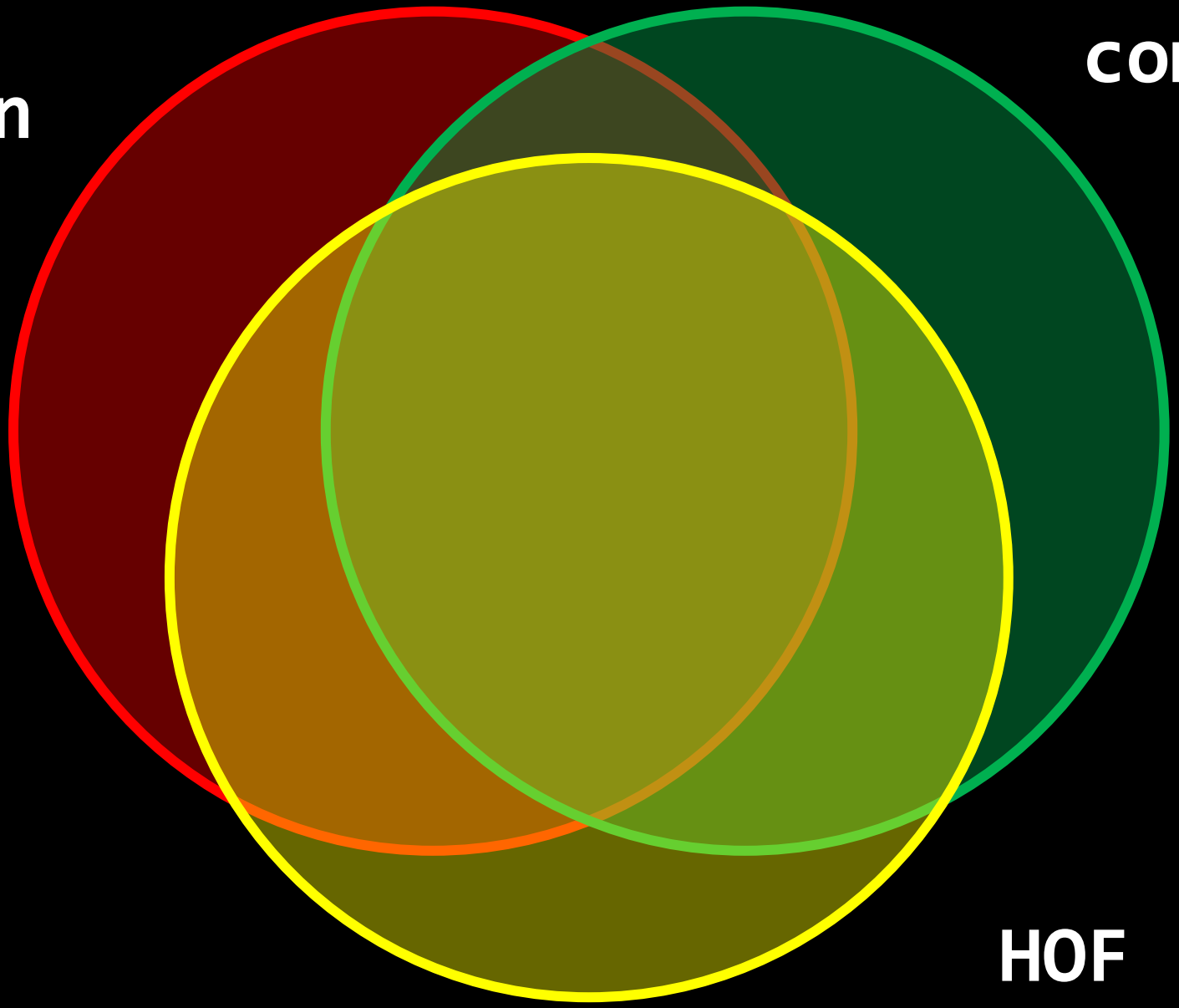
pure function

combinator



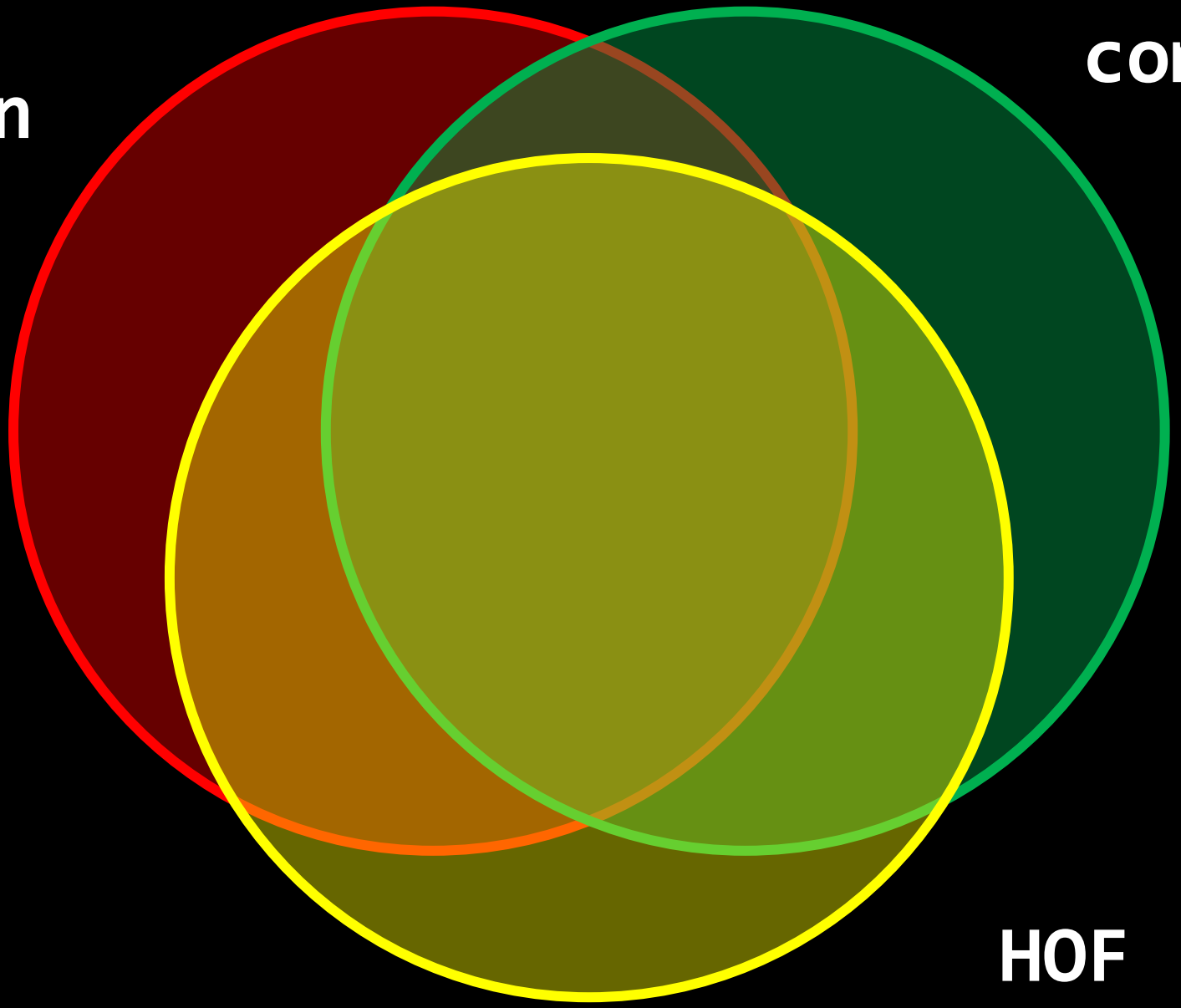
pure
function

combinator



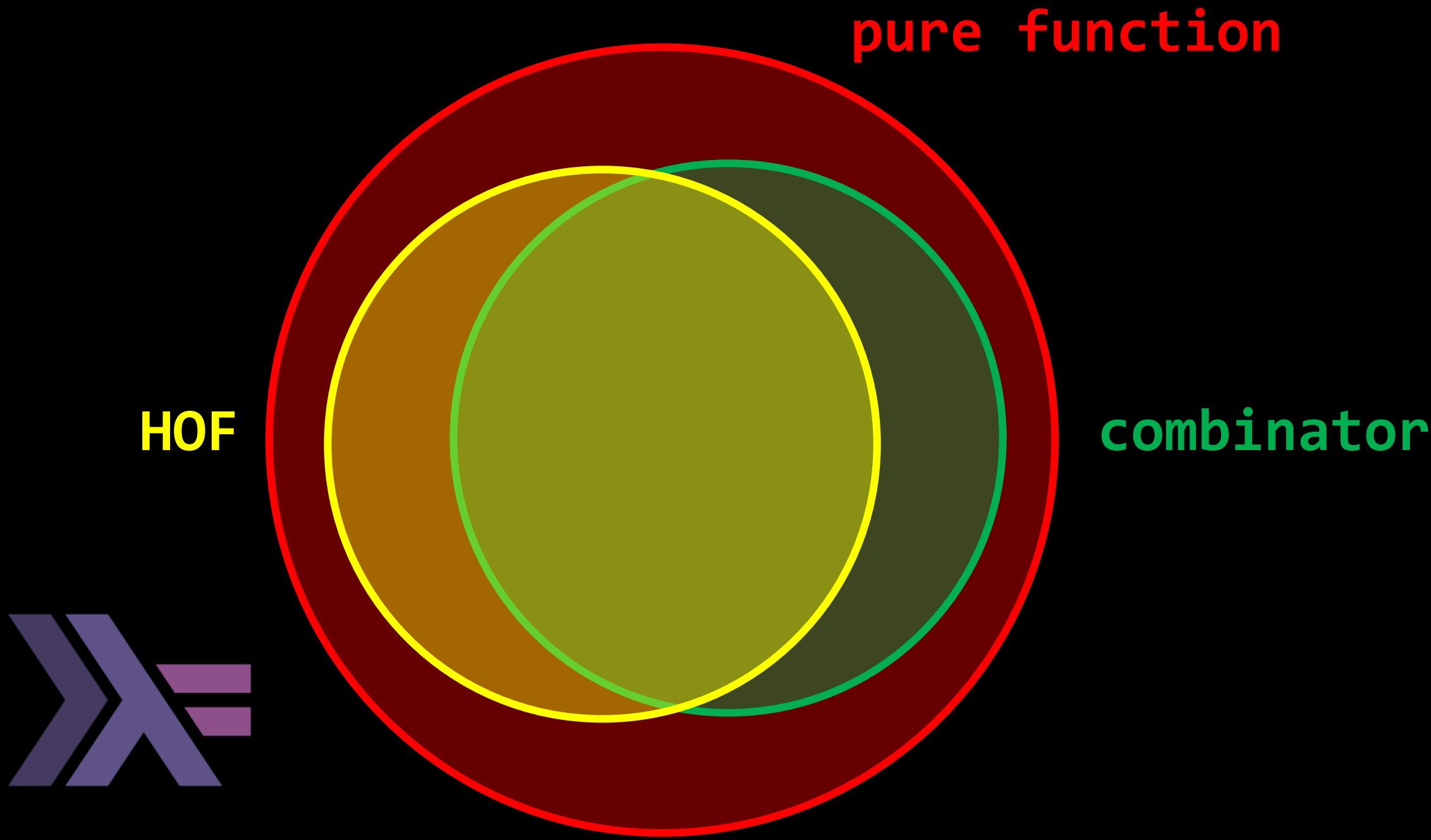
HOF

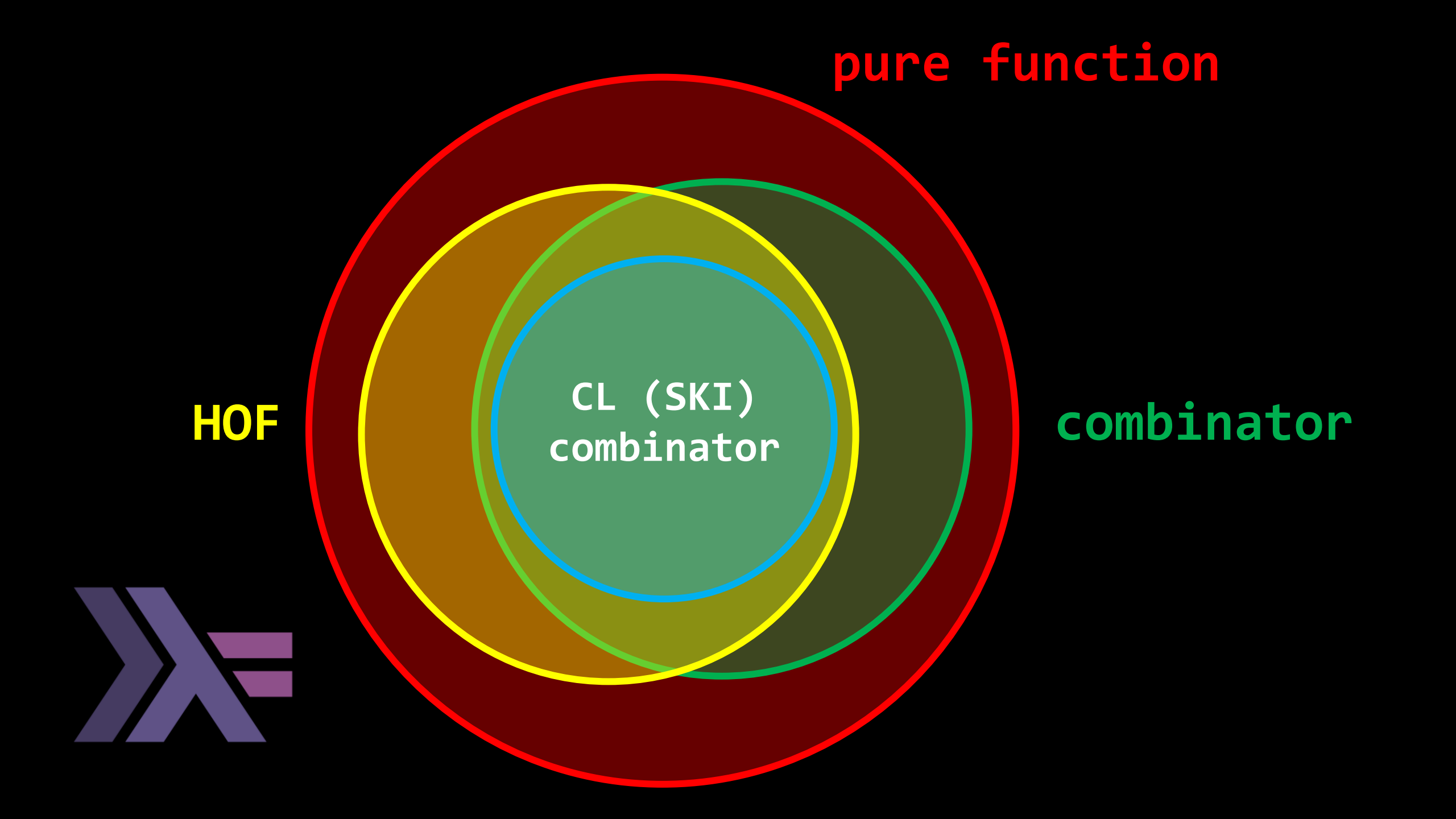
pure
function

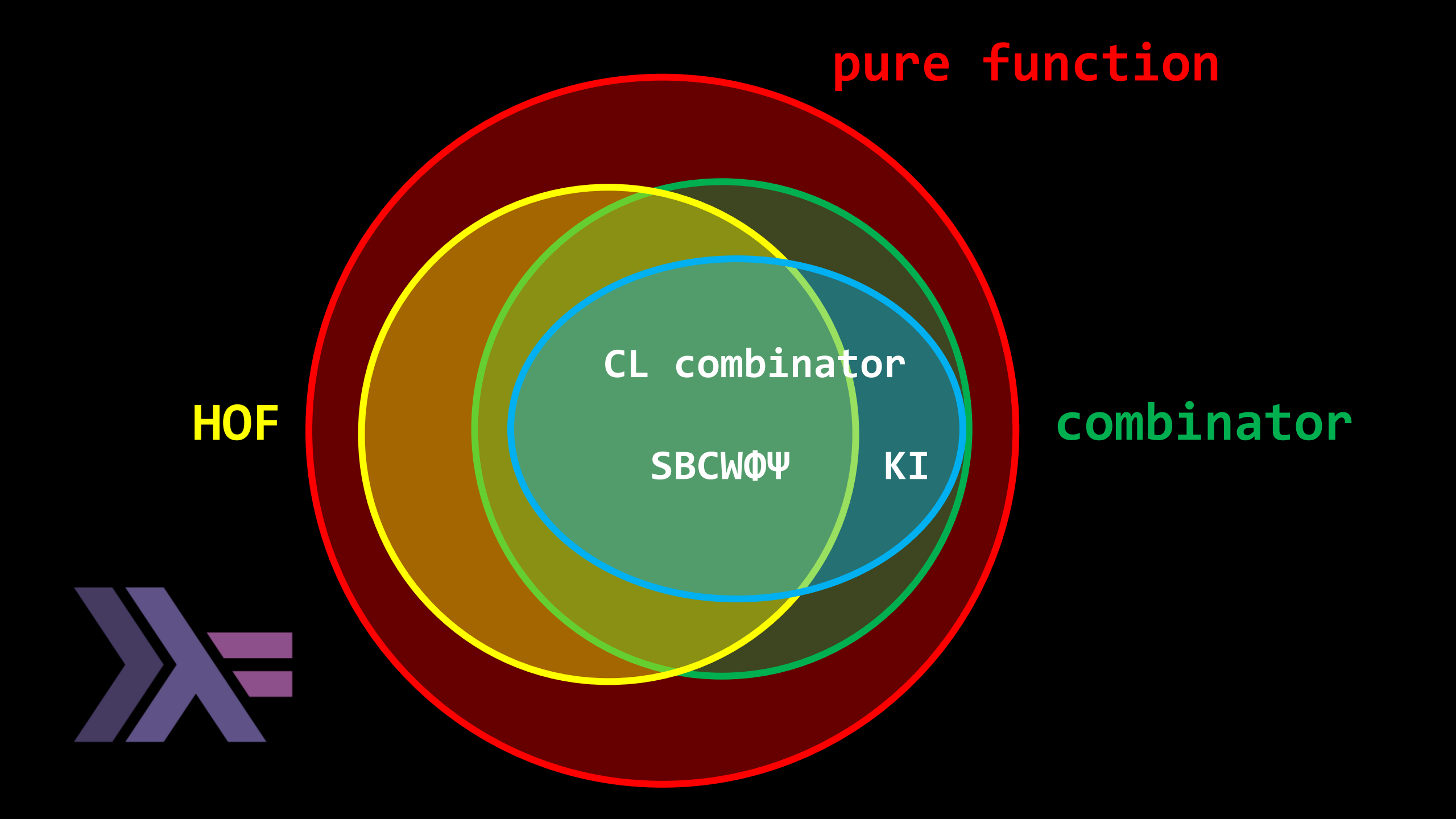


combinator

HOF


pure function
HOF
combinator


pure function
HOF
combinator
CL (SKI)
combinator


pure function
HOF
combinator
CL combinator
SBCWΦΨ
KI

combinator: a function that deals only in its arguments

CL combinator: a combinator that only consumes AND produces functions*

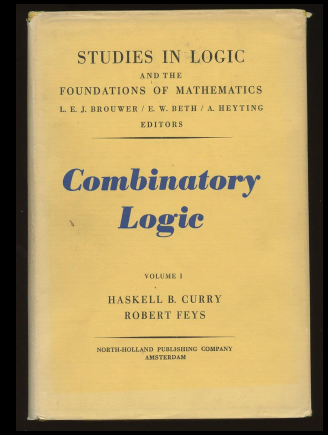
pure function: same input = same output / no side effects

HOF: consumes OR produces a function

THE ELEMENTARY COMBINATORS

Table 2. The elementary combinators.

Combinator	Elementary Name
I	Elementary Identificator
C	Elementary Permutator
W	Elementary Duplicator
B	Elementary Compositor
K	Elementary Cancellator



```
def i(x):
    return x
```

```
def k(x, y):
    return x
```

```
def w(f):
    return lambda x: f(x, x)
```

[[digression]]

Conor Hoekstra @code_report · Jan 8, 2022

Also, I apologize for my above average number of tweets 🐦 today, but this table of Greek/Latin words for describing function **arity** will be necessary for a future talk.

The Ê combinator is "tetradic"

Unary/Monadic
Binary/Dyadic
Ternary/Triadic
Quaternary/Tetradic

Terminology [edit]

Latinate names are commonly used for specific arities, primarily bas
cardinal numbers or ordinal numbers. For example, 1-ary is based o

x-ary	Arity (Latin based)	Adicity (Greek based)
0-ary	*Nullary* (from *nūllus*)	*Niladic*
1-ary	*Unary*	*Monadic*
2-ary	*Binary*	*Dyadic*
3-ary	*Ternary*	*Triadic*
4-ary	*Quaternary*	*Tetradic*

[[end of digression]]

```
def w(f):
    return lambda x: f(x, x)
```

```
def b(f, g):
    return lambda x: f(g(x))
```

```
def c(f):
    return lambda x, y: f(y, x)
```

```
def s(f, g):
    return lambda x: f(x, g(x))
```

```
def i   (x):          return x
def k   (x, y):       return x
def ki  (x, y):       return y
def s   (f, g):       return lambda x:       f(x, g(x))
def b   (f, g):       return lambda x:       f(g(x))
def c   (f):          return lambda x, y:    f(y, x)
def w   (f):          return lambda x:       f(x, x)
def d   (f, g):       return lambda x, y:    f(x, g(y))
def b1  (f, g):       return lambda x, y:    f(g(x, y))
def psi (f, g):       return lambda x, y:    f(g(x), g(y))
def phi (f, g, h):    return lambda x:       g(f(x), h(x))
```

```
def b  (f, g):    return lambda x:    f(g(x))

def b1 (f, g):    return lambda x, y: f(g(x, y))
```

```
def b  (f, g):    return lambda x:    f(g(x))
def b1 (f, g):    return lambda x, y: f(g(x, y))
```

"Point-Free or Die: Tacit Programming in Haskell and Beyond" by Amar Shah

2.00

```
exactMatches = length . filter id ... zipWith (==)
```

strangeloop

Sept 15-17, 2016

thestrangeloop.com

23:08 / 36:12

"Point-Free or Die: Tacit Programm

2.00

exactMatches =

trangeloop

ot 15-17, 2016

nestrangeloop.com

23:08 / 36:12

"Point-Free or Die: Tacit Programming in Haskell and Beyond" by Amar Shah

2.00

```
exactMatches = length . filter id ... zipWith (==)
```

strangeloop

Sept 15-17, 2016

thestrangeloop.com

23:08 / 36:12

```
exactMatches = length . filter id .: zipWith (==)
```

```
exactMatches     = length . filter id .: zipWith (==)
exactMatches c g = length (filter id (zipWith (==) c g))
```

```
exactMatches     = length . filter id .: zipWith (==)
exactMatches c g = length (filter id (zipWith (==) c g))

= exactMathes "RRGG" "RYGB"
= length (filter id (zipWith (==) "RRGG" "RYGB"))
= length (filter id [True, False, True, False])
= length [True, True]
= 2
```

```
length . filter id .: zipWith (==)
```

```
sum .: zipWith (==)
```

```
sum . map fromEnum .: zipWith (==)
```

```
sum .: zipWith (fromEnum .: (==))
```

```
countElem True .: zipWith (==)
```

```
length . filter id .: zipWith (==)
sum . map fromEnum .: zipWith (==)
 sum .: zipWith (fromEnum .: (==))
   countElem True .: zipWith (==)
```

```
length . filter id .: zipWith (==)
sum . map fromEnum .: zipWith (==)
 sum .: zipWith (fromEnum .: (==))
   countElem True .: zipWith (==)
```

```
length . filter id .: zipWith (==)
sum . map fromEnum .: zipWith (==)
 sum .: zipWith (fromEnum .: (==))
   countElem True .: zipWith (==)
```

=

```
length . filter id .: zipWith (==)
sum . map fromEnum .: zipWith (==)
 sum .: zipWith (fromEnum .: (==))
   countElem True .: zipWith (==)
```


```
+/=
```

```
length . filter id .: zipWith (==)
sum . map fromEnum .: zipWith (==)
 sum .: zipWith (fromEnum .: (==))
   countElem True .: zipWith (==)
```


```
{+/⍺=⍵}
```

```
length . filter id .: zipWith (==)
sum . map fromEnum .: zipWith (==)
 sum .: zipWith (fromEnum .: (==))
   countElem True .: zipWith (==)
```


```
+/=
```

```
length . filter id .: zipWith (==)
sum . map fromEnum .: zipWith (==)
 sum .: zipWith (fromEnum .: (==))
   countElem True .: zipWith (==)
```


```
+ . =
```

```
length . filter id .: zipWith (==)
sum . map fromEnum .: zipWith (==)
 sum .: zipWith (fromEnum .: (==))
   countElem True .: zipWith (==)
```


```
+/=
```

```cpp
auto exact_matches(std::string_view code,
                   std::string_view guess) -> int {
    return std::transform_reduce(
        code.begin(), code.end(), guess.begin(), 0,
        std::plus{},
        std::equal_to{});
}
```

```haskell
exactMatches =  length
             .  filter id
             .: zipWith (==)
```


```apl
ExactMatches ← +/=
```

Conor Hoekstra @code_report · Oct 21, 2020

And for maybe my most amazing combinator discovery to date, std::inner_product is the **Blackbird** 🤯 (which is what I was looking for when I stumbled across the Phoenix) #combinators #cpp #algorithms

https://www.youtube.com/watch?v=U6I-Kwj-AvY


```
{(+/0>⍵),(+/0<⍵)}
```


```
(+/0>⊢),(+/0<⊢)
```

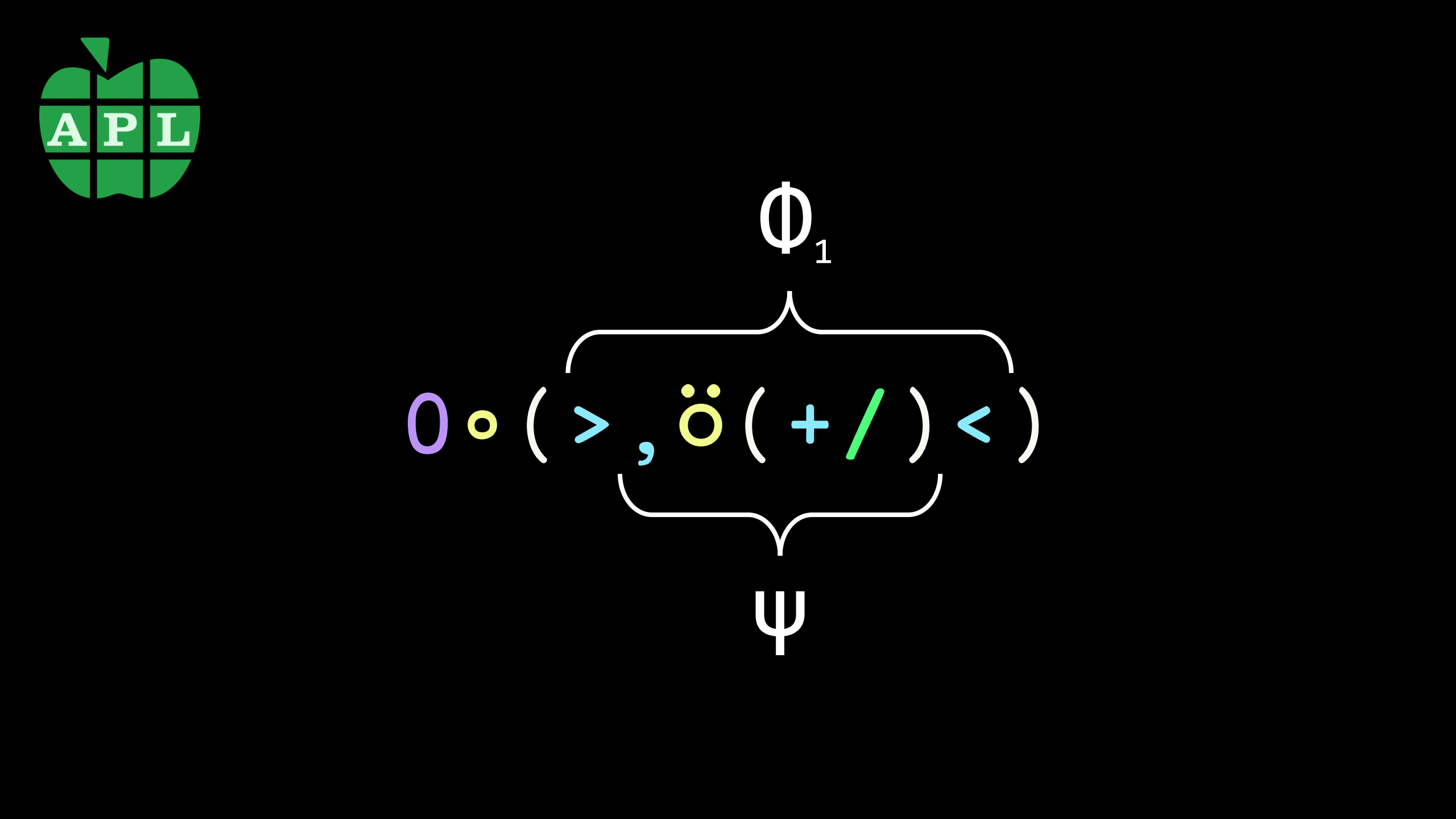

```
(   0>⊢),(   0<⊢)
      +/
```


```
(0>⊢),⍤(+/)(0<⊢)
```


```
0∘(>,⍤(+/)<)
```


```
0∘(>,⍤(+/)<)
```

ψ


Φ1
0∘(>,ö(+/)<)
ψ


```
def psi (f, g):    return lambda x, y: f(g(x), g(y))
def phi1(f, g, h): return lambda x, y: g(f(x, y), h(x, y))
```

Φ₁

```
0∘(>,⍤(+/)<)
```

Ψ


APL

```
{(+/⍵>⍺)⌈(+/⍵<⍺)}
⍵∘(>⌈⍤(+/)<)
```


```
{(+/⍵>0)⌈(+/⍵<0)}
0∘(>⌈⍤(+/)<)
```

```
/↥⊜⌿.⌵±
```


APL

```
{(+/⍵>⍵)⌈(+/⍵<⍵)}
⍵∘(>⌈⍤(+/)<)
⌈/(≠⍨(⊆⍨|∘×))
```

```
/↥⊜⌿.⌵±
```


```
{(+/0>ω)⌈(+/0<ω)}
0∘(>⌈⍤(+/)<)
⌈/(≠⍨(⊆⍨|∘×))
⌈/⍤≠⍨⍤⊆⍨|∘×
```

```
/↥⊜⌿.∨±
```


```
{⌈/≠⍤⊆⍨|×⍵}
⌈/(≠⍤(⊆⍨)|∘×))
```

```
{{>./(#;._1)0,|*y}}
([:>./&(#;._1)0,|&*)
```

```
{⌈´+´⍤⊆⍨|⊐⊸∪×𝕩}
⌈´∘+´⍤⊆⍨∘|∘⊐⊸∪×
```

```
/↥⊜卌.∨±
```


```
{(+/⍵>ω)⌈(+/⍵<ω)}
⍵∘(>⌈⍤(+/)<)
```

```
{{(+/⍵>y)>.(+/⍵<y)}}
⍵&(>>.&:(+/)<)
```

```
{(+´⍵>𝕩)⌈(+´⍵<𝕩)}
⍵⊸(>⌈○(+´)<)
```

```
↥::/+⊐><⍵
```

Why Combinators?

Why Combinators?

Programming with combinators can lead to extremely elegant code

Why Combinators?

Programming with combinators can lead to extremely elegant code if you have the right set with the right primitives.

Why Combinators?

Programming with combinators can lead to extremely elegant code if you have the right set with the right primitives. And it sees like there is a threshold.

				
2-Train is B/B_1	✓	✗	✓	B/B_1 for free
Cap / Nothing	🤔	✓	✓	No need
Full set (S, Σ, etc)	🤔	🤔	✓	No need?
Elegant Repeated B* *Tacit	✗	✗	✗	✓
Split Primitive	⊆ Partition	;._X Cut	⊔ Group	⊜ Partition

A PROGRAMMING TECHNIQUE FOR NON-RECTANGULAR DATA

Bob Smith
APL Application Analyst
Scientific Time Sharing Corporation
21243 Ventura Blvd., Suite 240
Woodland Hills, CA 91364
(213) 347-1633

Abstract

A programming technique is developed to deal with certain common operations on non-rectangular (ragged) data. A representation of such data in current *APL* is defined; a notation for applying primitive functions to this data is developed; and user-defined functions that simulate these operations for certain primitive monadic functions are illustrated.

Motivation

The coordinates of matrices and higher-dimensional arrays serve to delimit the values of the rectangular data into distinct "lines" (e.g., the rows of a matrix). Primitive functions (e.g., reduction and scan) then may be applied independently to each distinct line to produce various results.

Occasionally, however, it becomes necessary to apply a primitive function independently to successive parts of a vector (called the argument vector). These parts are analogous to the lines of a higher-dimensional array. For example, the problem may be to plus-reduce certain parts of a numeric vector. If the argument vector were ⍳10 and if it were broken into several parts as follows:

```
1 2 3  4  5 6 7 8  9 10
```

then the desired result would be 6 4 26 19.

In this case, the non-rectangular data is represented as an argument vector along with a vector of the lengths which partition the argument vector (here, the implicit lengths are 3 1 4 2). One might think of the argument vector as a vector of vectors.

Previously, this kind of problem has been solved by first expanding and reshaping the argument vector into a matrix. This transformation allows the programmer to take advantage of the delimiting property of the coordinates of the matrix and apply the primitive function to the entire matrix to obtain the result. If the argument vector were stored in *B* and the lengths of the successive parts stored in *A*, a solution is

```
+/((⍴A),⌈/A)⍴(,A∘.≥⍳⌈/A)\B .
```

As a general method, the argument vector-to-matrix approach has several drawbacks.

First, the expansion result may not fit in the workspace. Because the length of the expansion result is the number of parts times the length of the longest part, it only takes one long part and many small parts to create a nuisance. *WS FULL* is most programmers' least favorite error message.

Second, the expansion process may pad each part (on the way to creating a row of the matrix) with an inappropriate value. The 0's that expansion uses as fillers for numeric structures won't bother +/, but ×/ and ∧/ won't like them. This problem can be circumvented; however, it's one more thing with which the programmer must be bothered.

Third, this approach can be expensive because of all the extra work involved in inserting fill elements via expansion.

The following sections discuss an alternate approach to solving this class of problems. Essentially, these problems involve representing non-rectangular data as a vector of vectors along with a partition, and applying a primitive function to each independent part. First, we must characterize the concept of "parts", and then combine that with the primitive function and the argument vector to produce the result.

Partition Vectors

A simple and unique characterization of any partitioning of an argument vector is as a Boolean vector of the same length (called a partition vector). Each 1 in the partition vector corresponds to the beginning of a part and the following 0's correspond to the remaining elements in that part.

The partition of ⍳10 used above would be characterized as

```
1 2 3   4   5 6 7 8   9 10
1 0 0   1   1 0 0 0   1  0
```

It is easy to see that this characterization is unique and that every Boolean vector of the same length as the argument vector and whose first element is 1 represents a partition of the argument vector. There are 2*N-1 such partitions of length *N*. Note that the number of parts represented by the partition vector *P* is +/*P*.

The characterization as a Boolean vector is chosen over that of a vector of successive lengths of the parts for several reasons. The Boolean vector often is easily computed from and stored with the argument vector (as in blanks in a character vector); has a simpler conformability test (*P*[1]∧(⍴*P*)=⍴*B* vs. ((+/*P*)=⍴*B*)∧∧/*P*≥0); often uses less storage than an integer vector; and combines more naturally with the user-defined functions illustrated later.

Although a representation as a vector of lengths would allow zero lengths, there is no real added capability. The zero lengths have no effect in conjunction with shape-preserving functions like scan and grade. For reduction, presumably one fills in the result with the appropriate identity element. I prefer to leave that job to the expansion function.

The Partition Operator

We now need to combine the partition vector, the primitive function, and the argument vector. One method of doing this is to combine the partition vector and the primitive function into a derived function via an operator. This operator (called the partition operator) is then dyadic, with a partition vector on the left and a primitive function on the right. The result is a derived function (a partition function) which has the same valence as the original primitive function. That is, the resulting function is dyadic if the original primitive function is dyadic; monadic otherwise.

For simplicity, a new symbol is used for this operator. It is not the goal of this paper to suggest that this operator or symbol be implemented; it is introduced only as notation to simplify the presentation.

Definition: partition operator ⍿

(which Jim Ryan calls the "dagger" symbol)

Let ⍺ be any primitive monadic function, and *P* and *B* any vectors of equal length where *P* is a Boolean vector whose first element is 1.

```
R ← P⍿⍺ B
```

is the result of applying ⍺ to each part of *B*, where *P* defines the partition. Specifically,

```
R←⍳0
I←0
L1:→((+/P)<I←I+1)/0
R←R, ⍺ (I=+\P)/B
→L1
```

The shape of *R* depends upon ⍺. If ⍺ is a reduction function, the shape is +/*P* (one value per part since reduction reduces vectors to scalars); if ⍺ is a shape-preserving function, the shape is ⍴*P* (e.g., scan preserves shape). If ⍺ is defined for higher-dimensional arrays, then so is *P*⍿⍺, and in the same way. Notationally, this would be written as *P*⍿(⍺[*K*]) *B*. For example, *P*⍿(+\[2]) *B*.

If ⍺ is a dyadic function, the syntax of the partition operator is

```
R ← A (P⍿⍺) B
```

and the result is calculated as

```
R←⍳0
I←0
L1:→((+/P)<I←I+1)/0
R←R,((I=+\P)/A) ⍺ (I=+\P)/B
→L1
```

See Iverson [1] for a discussion of operators which take data as arguments.

The above example of a partitioned plus reduction of ⍳10 would be written as

```
      1 0 0 1 1 0 0 0 1 0 ⍿(+/) ⍳10
6 4 26 19
```

or as a function of *A* and *B* with *B*←⍳10 and *A*←3 1 4 2 (the partition lengths) as

```
      ((⍳+/A)∈1++\¯1↓0,A)⍿(+/) B
6 4 26 19
```

Applications

I. In accounting applications, one encounters the problem of accumulating amounts of money into bins. For example, *AMT* might be a vector of amounts, and *ACCT* might be a vector of corresponding account codes. Potentially, there are amounts in

expansion function.

The Partition Operator

We now need to combine the partition vector, the primitive function, and the argument vector. One method of doing this is to combine the partition vector and the primitive function into a derived function via an operator. This operator (called the partition operator) is then dyadic, with a partition vector on the left and a primitive function on the right. The result is a derived function (a partition function) which has the same valence as the original primitive function. That is, the resulting function is dyadic if the original primitive function is dyadic; monadic otherwise.

For simplicity, a new symbol is used for this operator. It is not the goal of this paper to suggest that this operator or symbol be implemented; it is introduced only as notation to simplify the presentation.

See Iverson [1] for a
operators which take d

The above example of
reduction of ⍳10 would

```
      1 0 0 1 1 0 0 0
6 4 26 19
```

or as a function of *A*
A←3 1 4 2 (the partiti

```
      ((⍳+/A)∈1++\¯1↓0
6 4 26 19
```

Applications

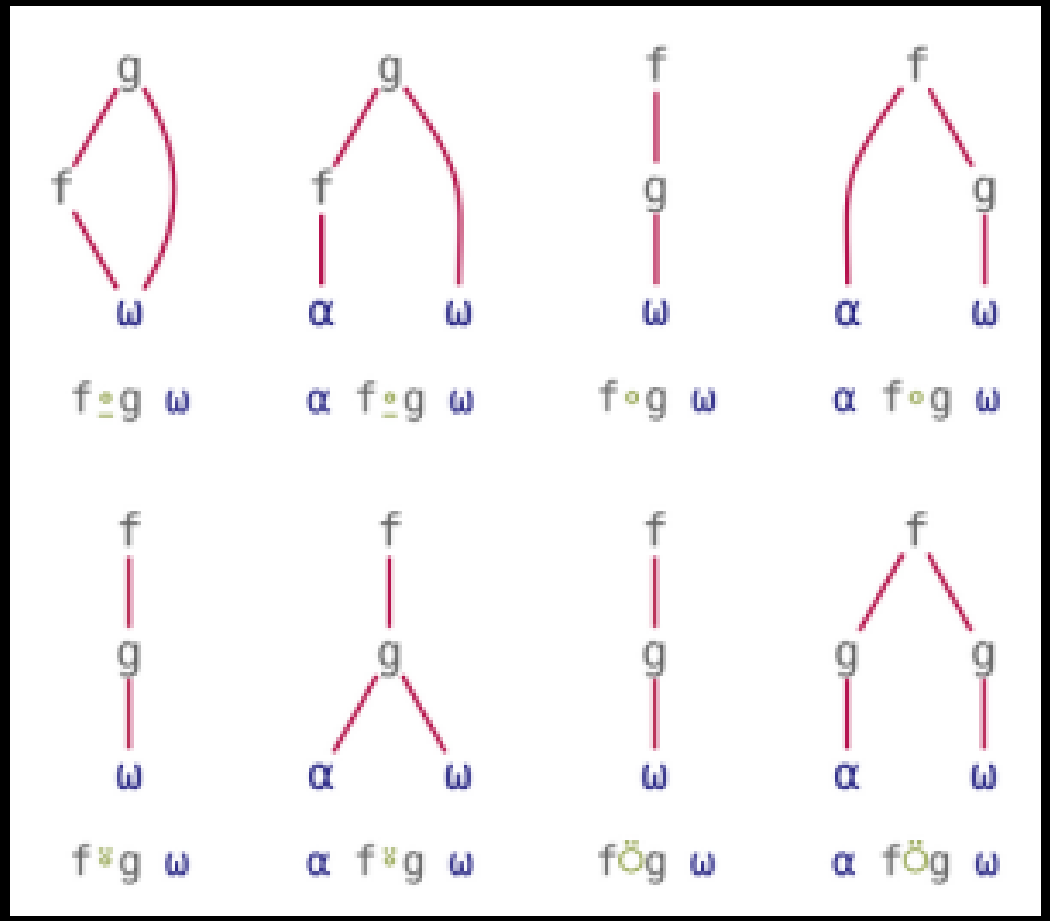
I. In accounting app
encounters the problem
amounts of money into
AMT might be a vector
might be a vector of c
codes. Potentially, t

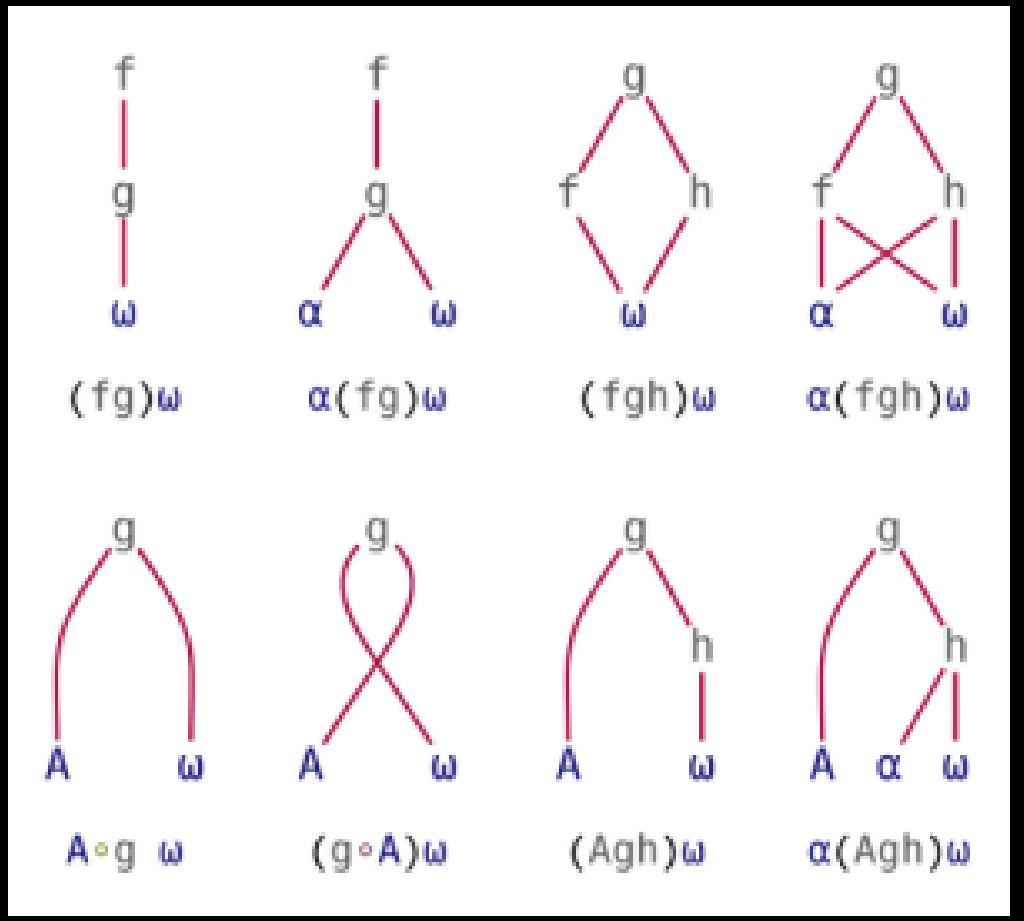
				
2-Train is B/B_1	✓	✗	✓	B/B_1 for free
Cap / Nothing	🤔	✓	✓	No need
Full set (S, Σ, etc)	🤔	🤔	✓	No need?
Elegant Repeated B* *Tacit	✗	✗	✗	✓
Split Primitive	⊆ Partition	;._X Cut	⊔ Group	⊜ Partition

				
2-Train is B/B_1	✓	✗	✓	B/B_1 for free
Cap / Nothing	🤔	✓	✓	No need
Full set (S, Σ, etc)	🤔	🤔	✓	No need?
Elegant Repeated B* *Tacit	✗	✗	✗	✓
Ambivalence	✓	✓	✓	✗
Split Primitive	⊆ Partition	;._X Cut	⊔ Group	⊜ Partition

Visualizing Combinators

https://aplwiki.com/wiki/Tacit_programming



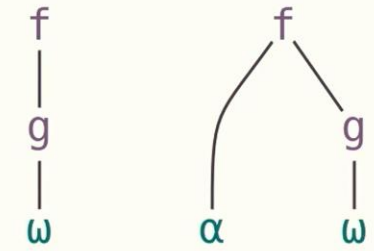


Tacit Techniques in Dyalog 18.0

Composition operators

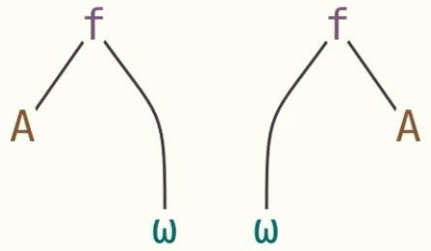
~~Compose~~

Beside



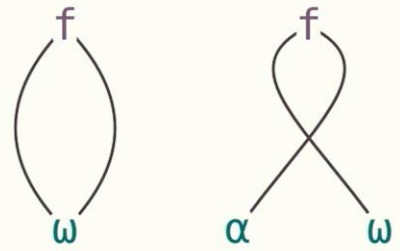
f∘g ω α f∘g ω

Bind



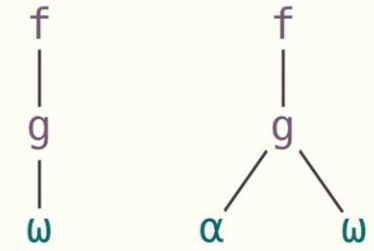
A∘f ω f∘A ω

Commute



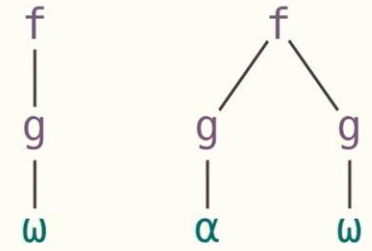
f⍨ ω α f⍨ ω

Atop



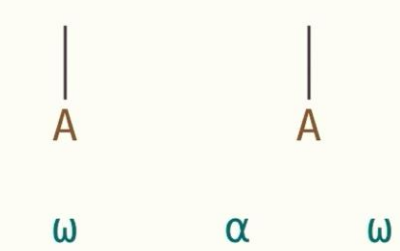
f⍤g ω α f⍤g ω
(f g)ω α(f g)ω

Over



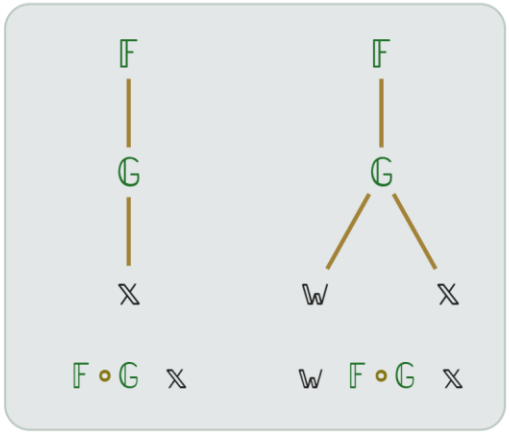
f⍥g ω α f⍥g ω

Constant

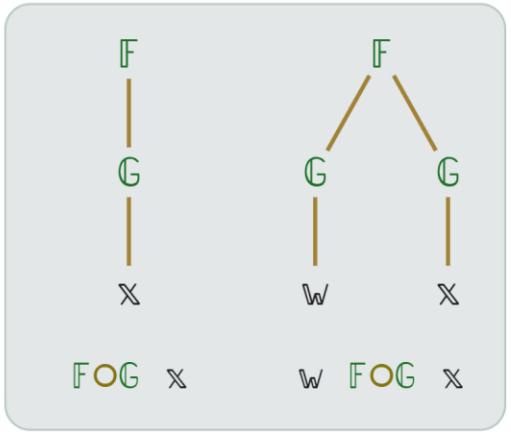


(A⍨) ω α (A⍨) ω

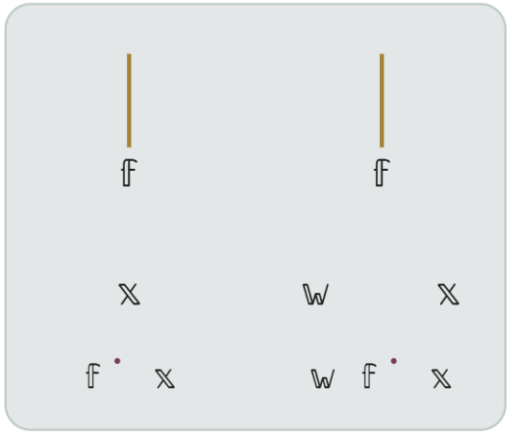
https://mlochbaum.github.io/BQN/doc/tacit.html#combinators



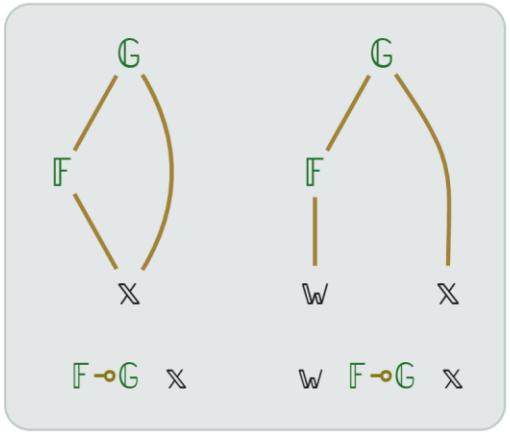
Atop



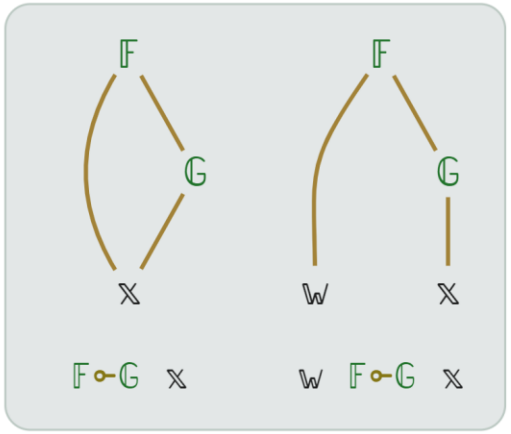
Over



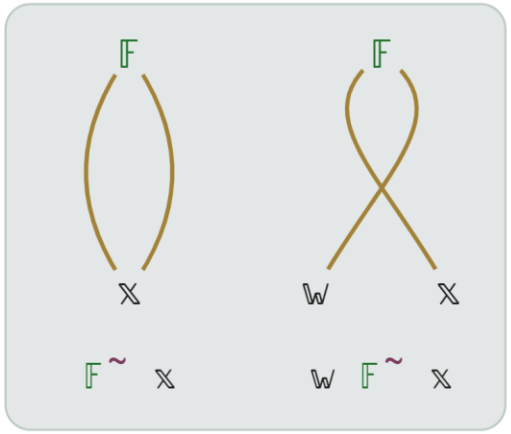
Constant



Before



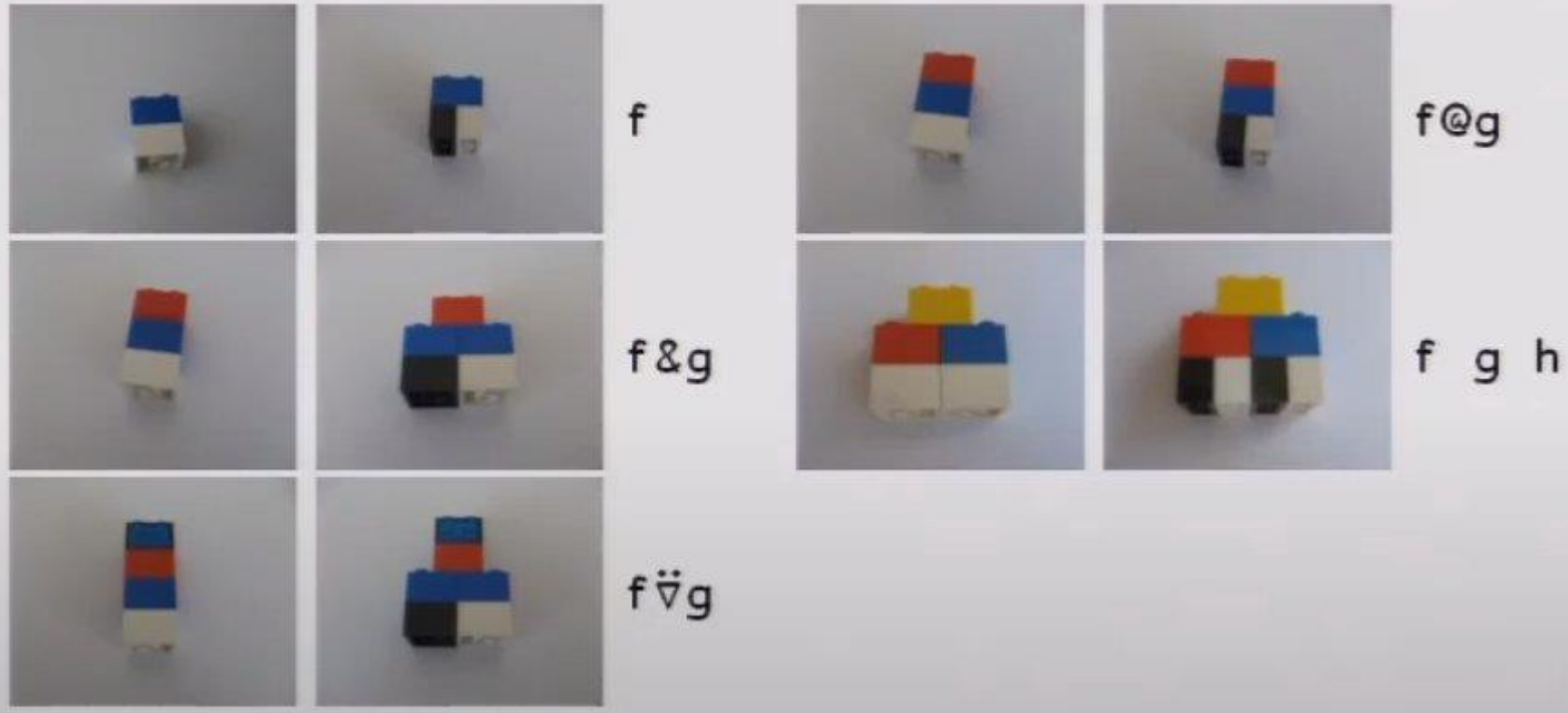
After



Self/Swap

Roger Hui – What is Functional Programming?

Combinators: System Summary

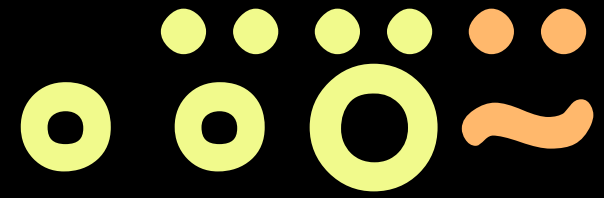


What was the goal of this talk?

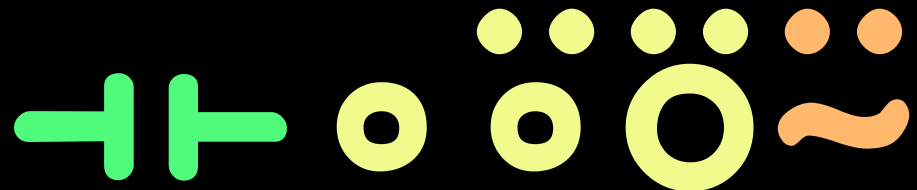
What was the goal of this talk?

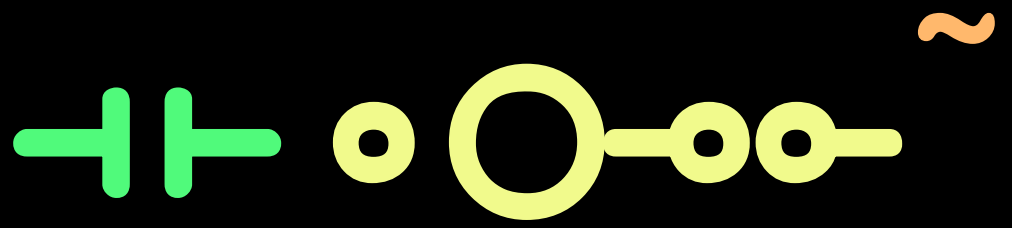
APL (and friends) have combinators
What is missing?


APL




APL





Thank You

https://github.com/codereport/Content/Talks

Conor Hoekstra

code_report

codereport

Questions?

https://github.com/codereport/Content/Talks

Conor Hoekstra

code_report

codereport