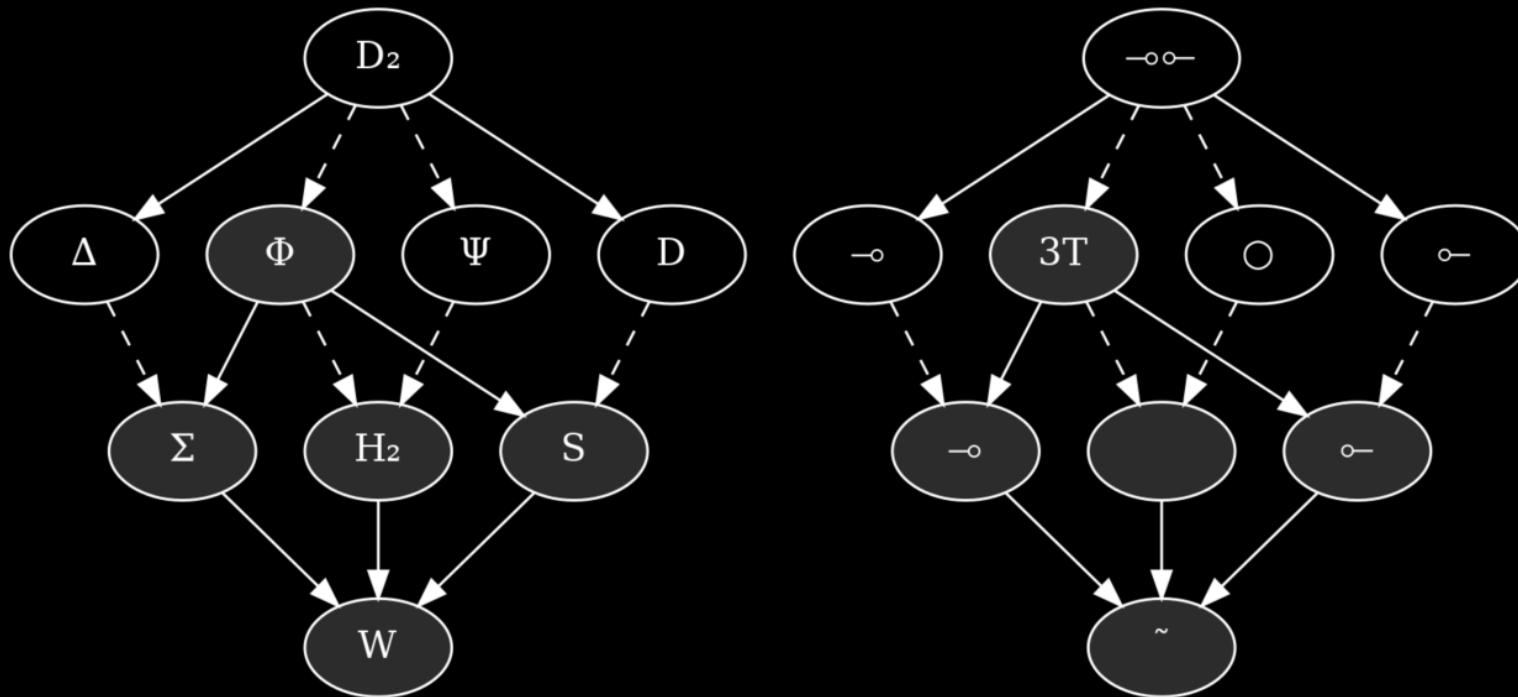


Composition Intuition



Conor Hoekstra



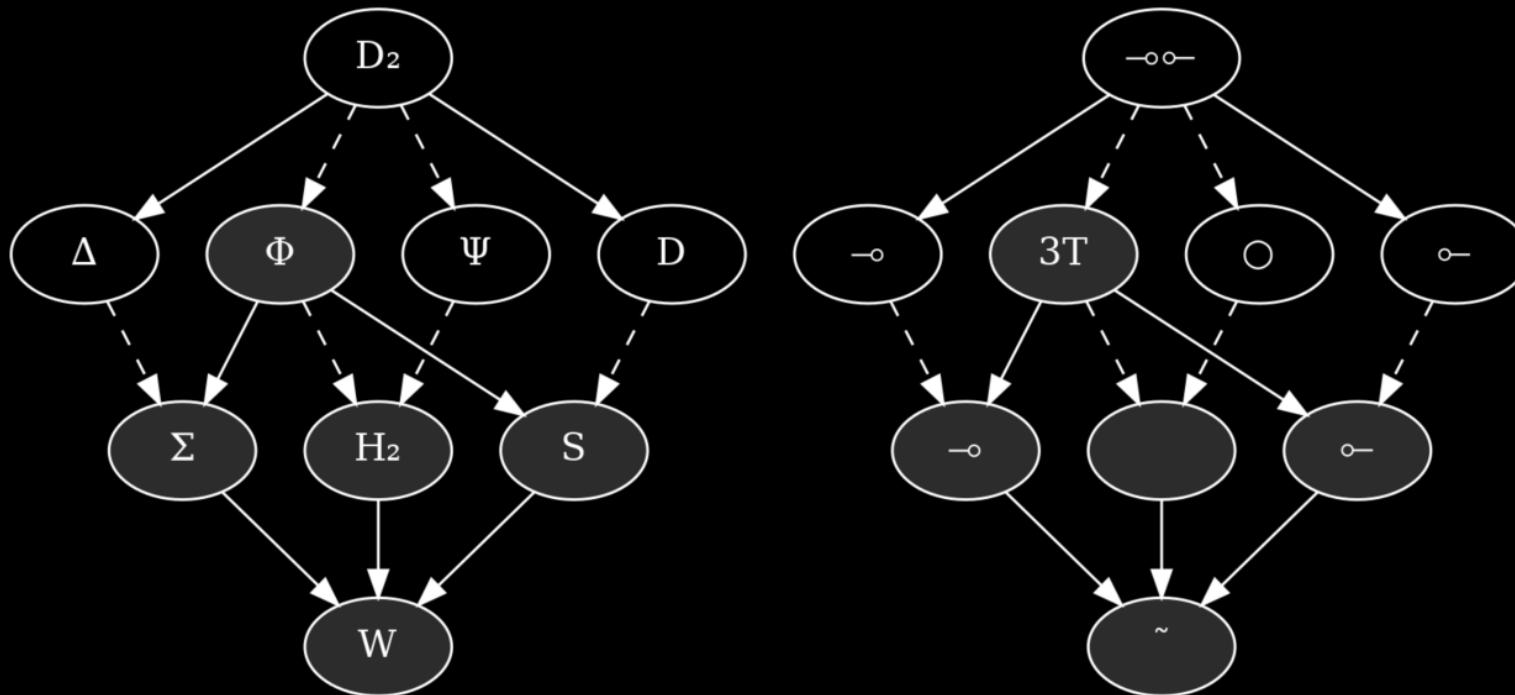
code_report



codereport

Composition Intuition

Combinators & Tacit Programming



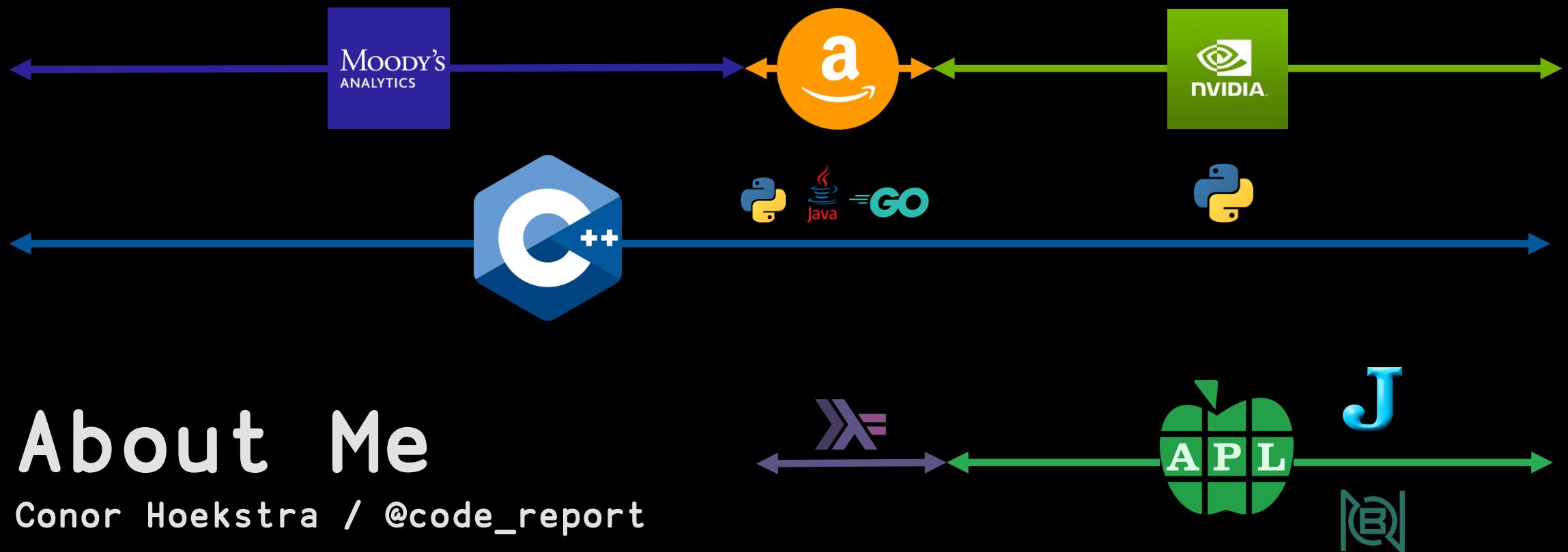
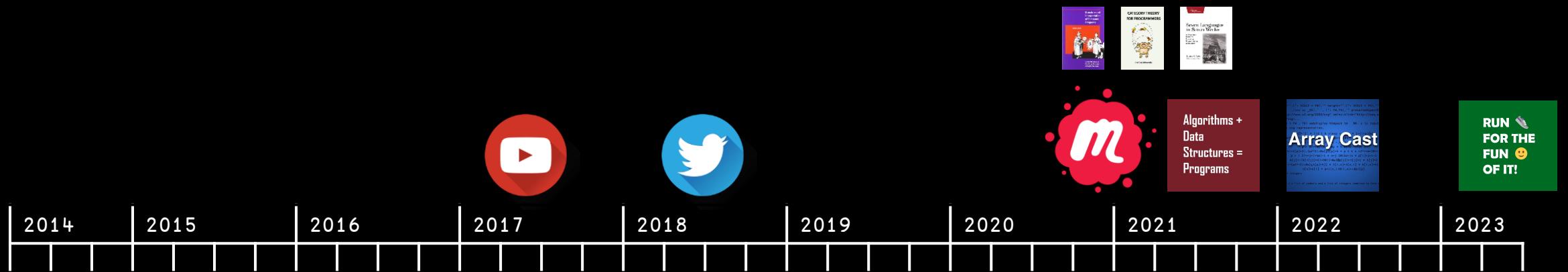
Conor Hoekstra



code_report



codereport



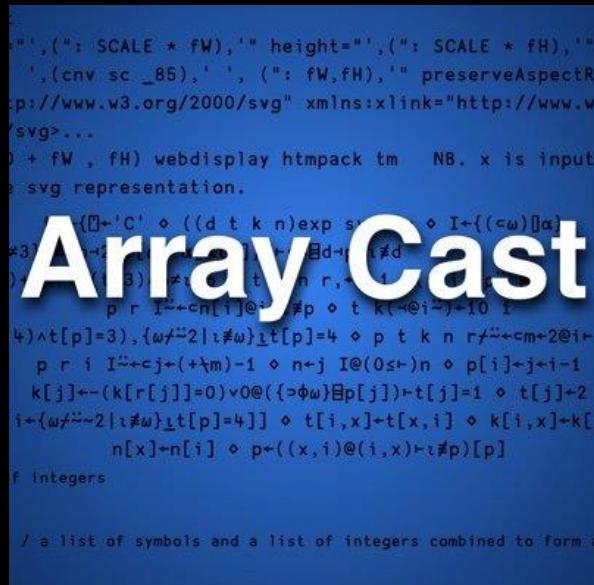


319 Videos

32 (20) Talks

Algorithms + Data Structures = Programs

133 Episodes
[@adspthepodcast](https://www.adspthepodcast.com)



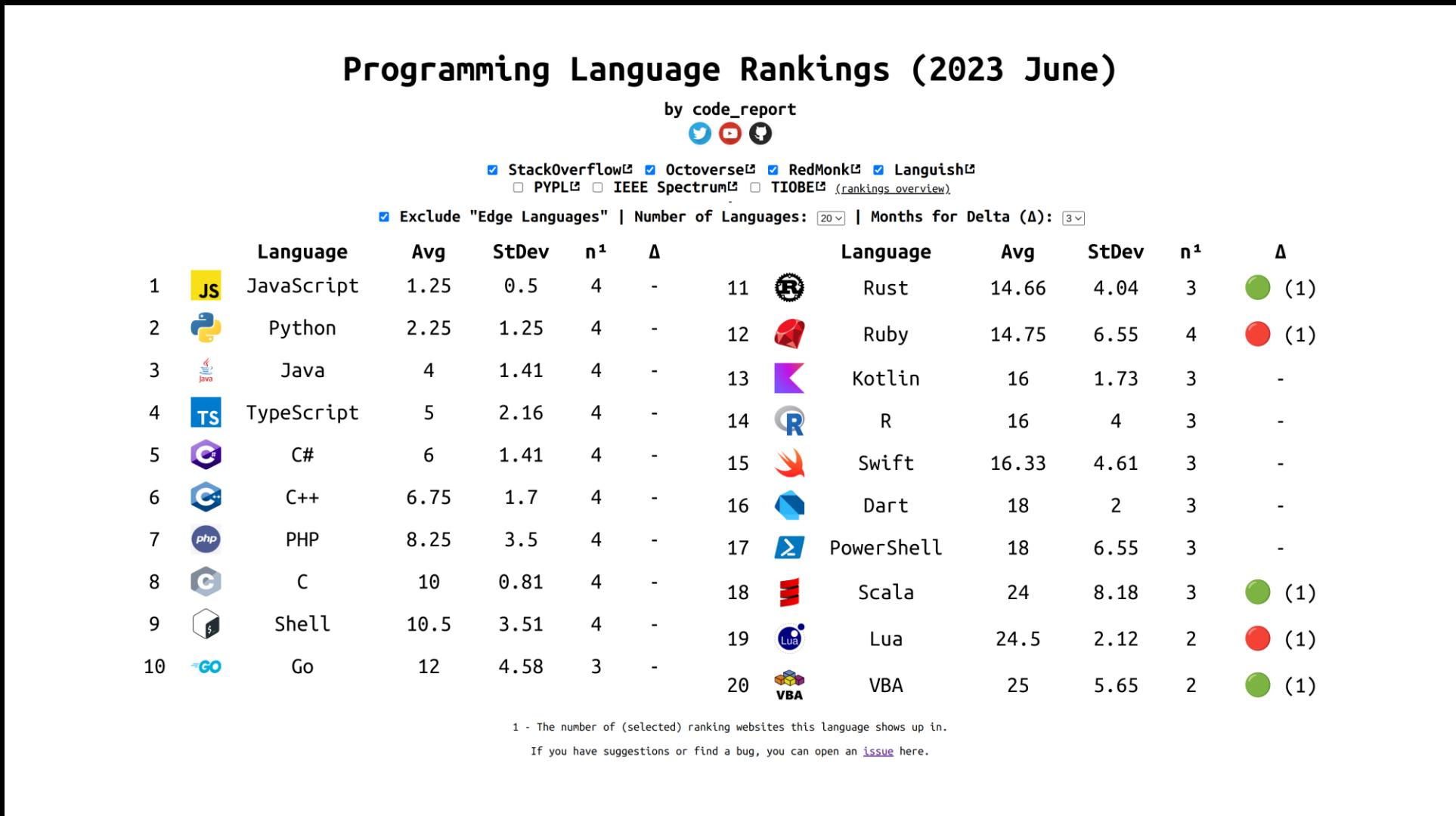
54 Episodes
[@arraycast](https://www.arraycast.com)



RUN FOR THE FUN OF IT!



9 Episodes
[@conorhoekstra](https://www.instagram.com/conorhoekstra)



<https://github.com/codereport/Content>

What is the best
combinator programming language?





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

the Twin Algorithms

Conor Hoekstra

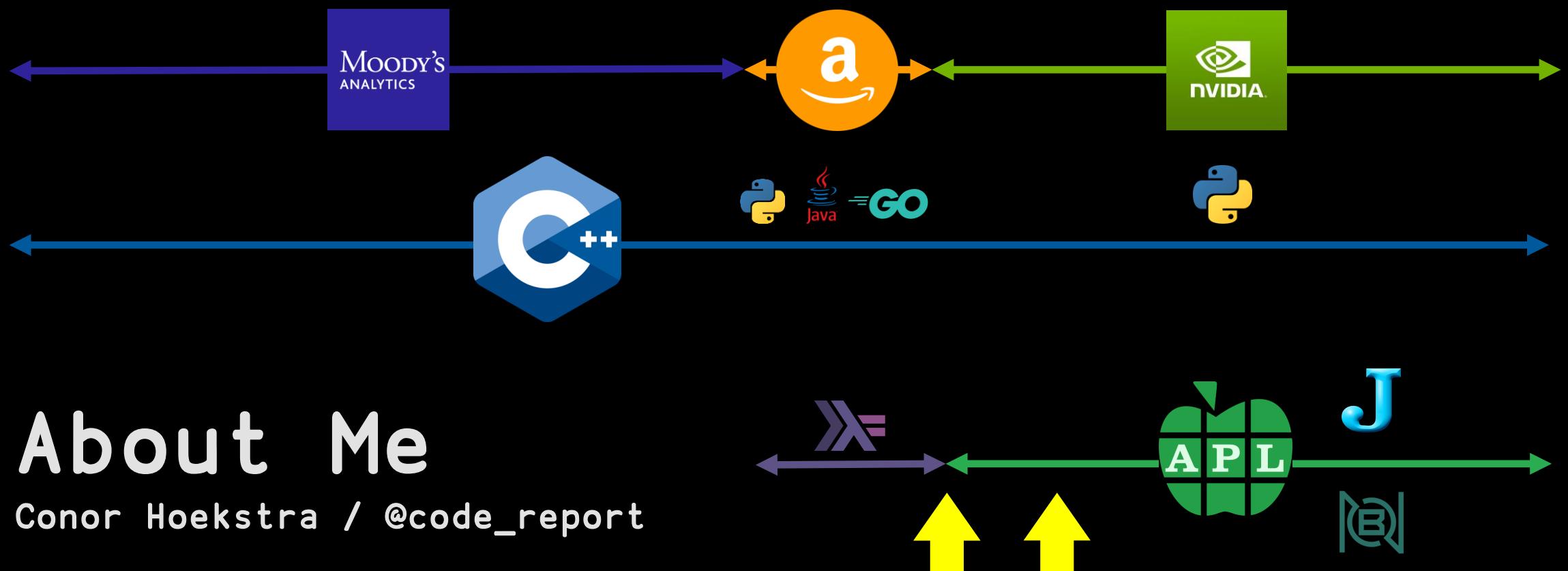
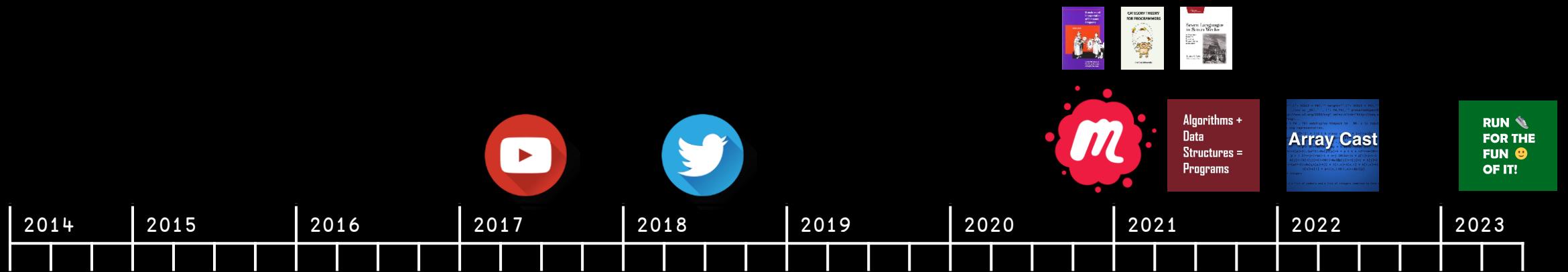


#include

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

Dec 2019

How did I discover combinatorics?



About Me

Conor Hoekstra / @code_report



Sept 15-17, 2016
strangeloop.com

36:13

"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

New C

Case convert

f ö g

Over

f ö g

Atop

≠ Y

Unique mask

Improved JSON

'P

 JSON

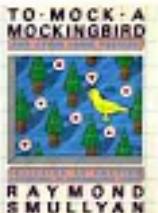
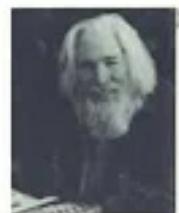
'D

 R/USE

'R

 INPUT

'N

Compositions**Dyalog 18.0 Webinar**
May 14, 2020Sept 15-17, 2016
thestrangeloop.com

36:13

"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



Sometime later in July

August 12



Combinatory logic

文 A 19 languages ▾

Contents [hide]

(Top)

In mathematics

In computing

Summary of lambda calculus

Combinatory calculi

Combinatory terms

Reduction in combinatory logic

Examples of combinators

Completeness of the S-K basis

Conversion of a lambda term to an equivalent combinatorial term

Explanation of the $T[\cdot]$ transformation

Simplifications of the transformation

η -reduction

One-point basis

Combinators B, C

CL_K versus CL_I calculus

Reverse conversion

Undecidability of combinatorial calculus

Applications

Compilation of functional languages

Logic

See also

References

Article Talk

Read Edit View history Tools ▾

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.



Combinatory calculi

Applications

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

$$\mathbf{AK}: A \rightarrow (B \rightarrow A),$$

$$\mathbf{AS}: (A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C)),$$

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

$$\mathbf{MP}: \text{from } A \text{ and } A \rightarrow B \text{ infer } B.$$

The calculus consisting of **AK**, **AS**, and **MP** is complete for the implicative 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 \rightarrow : on one hand, if $X \Vdash A \rightarrow B$, 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 $X \nvDash A \rightarrow B$, 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

$$\mathbf{AK}: A \rightarrow (B \rightarrow A),$$

$$\mathbf{AS}: (A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C)),$$

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

$$\mathbf{MP}: \text{from } A \text{ and } A \rightarrow B \text{ infer } B.$$

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.



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](#)

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 a pure **combinator** expression applied to a sequence of variables. Because there are great similarities between combinators and certain **APL** operators, a similar result obtains in ...

☆ 99 Cited by 3 Related articles All 4 versions

[\[PDF\]](#) acm.org[\[PDF\] History of lambda-calculus and combinatory logic](#)

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

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

☆ 99 Cited by 77 Related articles All 10 versions

[\[PDF\]](#) simons-rock.edu[APL trivia](#)

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 Dictionary **APL** function or operator, as also shown in Table 2, although ...

☆ 99 Cited by 3 Related articles All 4 versions

[\[PDF\]](#) acm.org[\[PDF\] Phrasal forms](#)

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

[\[PDF\]](#) acm.org

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

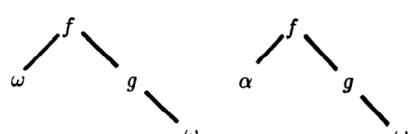
The notion of *verb rank*, first introduced by Iverson [Iv78], later elaborated by Keenan [Ke79], and further evolved by Whitney [Wh84], has been adopted by Iverson in his Dictionary [Iv87]. It refers to the rank of the subarrays of an argument which are the cells to which the verb applies. For example, the cells that negate applies to are items, and items are rank zero objects, and thus we say the rank of negate is zero. Similarly, the cells to which reverse applies are lists, or rank one 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 fg and is defined formally by identities and informally by hook-shaped diagrams as follows:

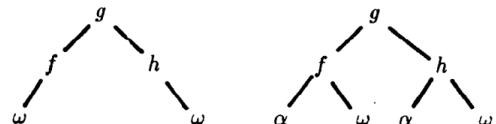
$$(fg)\omega \leftrightarrow \omega fg\omega; \quad \alpha(fg)\omega \leftrightarrow \alpha fg\omega$$



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

FORK. A *fork* is denoted by fgh and is defined formally by identities and informally by forking diagrams as follows:

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



The rank of the fork fgh 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 **S/gx** 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 cKx has the value c for all x .) Users of APL will appreciate the hook for the same reasons.

For example, $+/\div$ adds the reciprocal of the right argument to the left argument, a form used in describing continued fractions. Thus $(+/\div)\backslash 3\ 7\ 16\ \overline{294}$ 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 $(+\otimes)\omega$; and to decompose a number ω into numerator and denominator, one can write $(+\vee/\omega)$, 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,

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 $Sfgx$ 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 cKx 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 $(+÷)\backslash 3\ 7\ 16\ -294$ 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 $\Phi + \sin^2 \cos^2 \theta$, 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)

 Conor Hoekstra
@code_report

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

Sixteen

SK COMBINATORS

In this chapter we will learn about combinators. A combinator is a fixed set of functions that can be composed together to form more complex functions. In fact, combinators are supercombinators because they can be composed together to form even more complex functions.

The metaprogramming reduction rule for combinators needs no type annotations or implementations. This makes combinators easy to implement. We shall see how combinators can be used to implement the zipper pattern.

(Note: in this chapter, we use expressions in upper case.)

1 2 3

9:45 PM · Sep 16

View Tweet

1 2 3 | 6 7 8

1 Retweet 6 Likes

1 2 3 | 6 7 8

1 2 3 6 7 8

/ (+ [+])

1

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

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



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



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

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

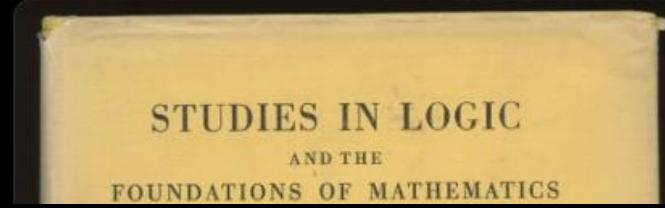
 Conor Hoekstra
@code_report

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)

PL is
S

APL is a Combinatory Logic programming language.



Conor Hoekstra @code_report · Sep 12

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



ALT



III

A Brief History of Combinatory Logic (CL)

A Brief History of Array Languages

A Brief History of CL in Array Languages

A Brief History of Combinatory Logic



Moses Schönfinkel
1988 - 1942

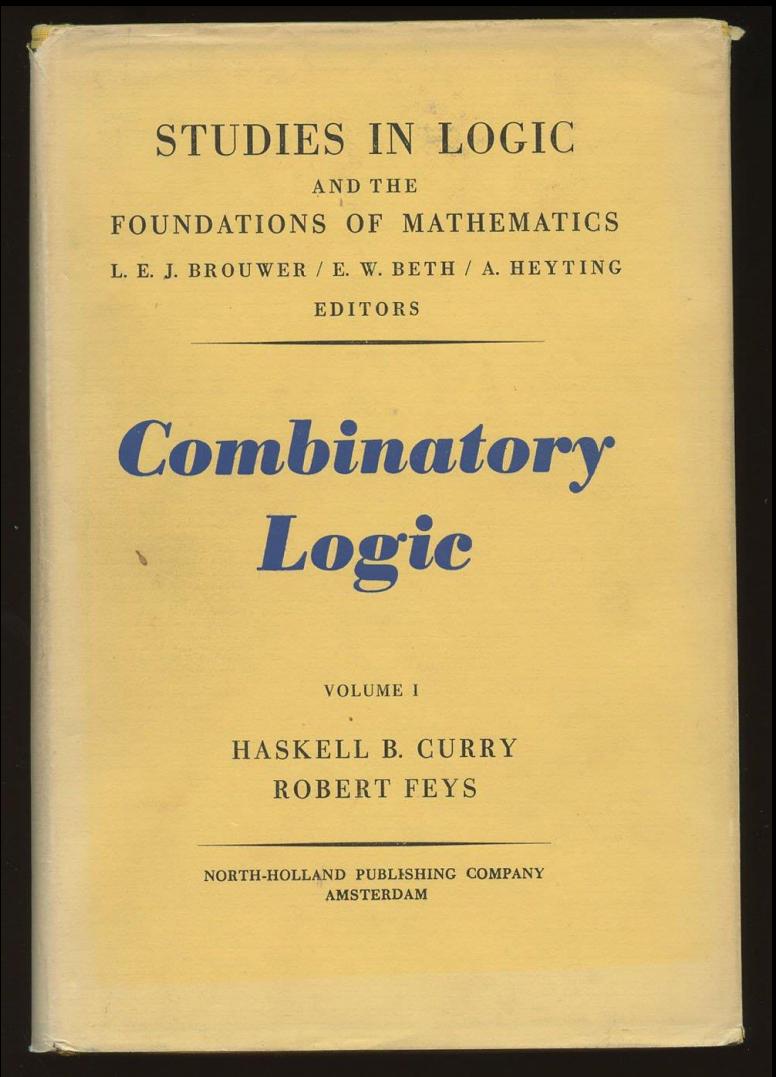


Moses Schönfinkel
1988 - 1942

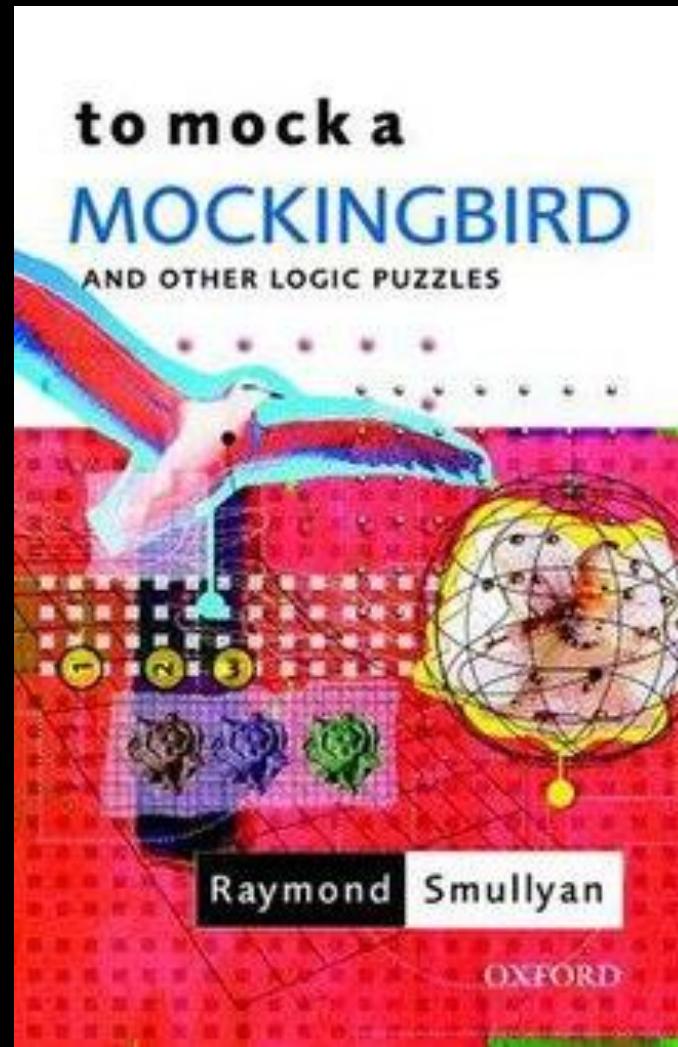


Haskell Curry
1900-1982

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, \dots)
1931	Curry	The Universal Quantifier in Combinatory Logic	Ψ, Φ_n (Φ, Φ_1, \dots)
1958	Curry and Feys	Combinatory Logic: Volume I	



1958

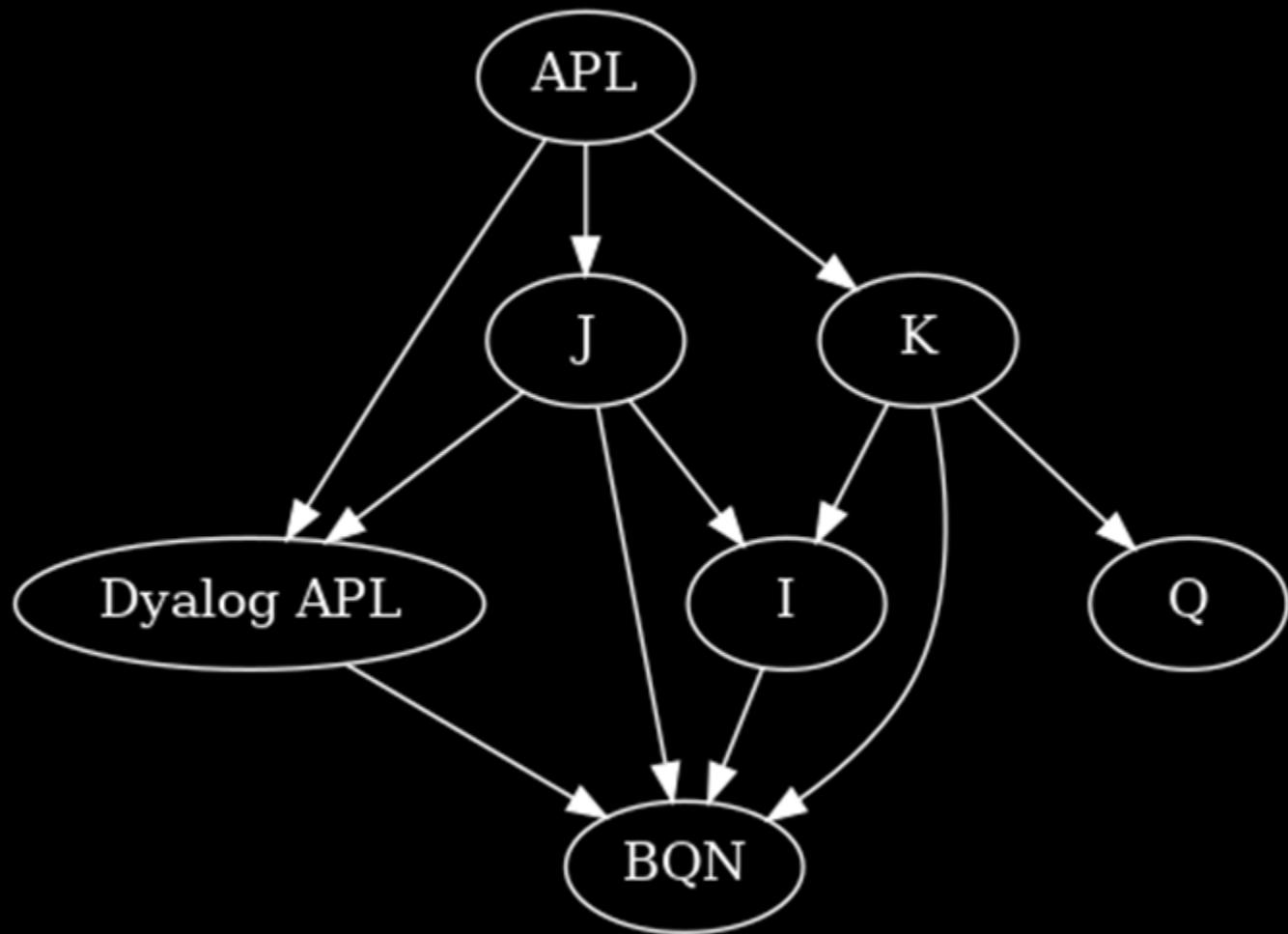


1985

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 Array Languages



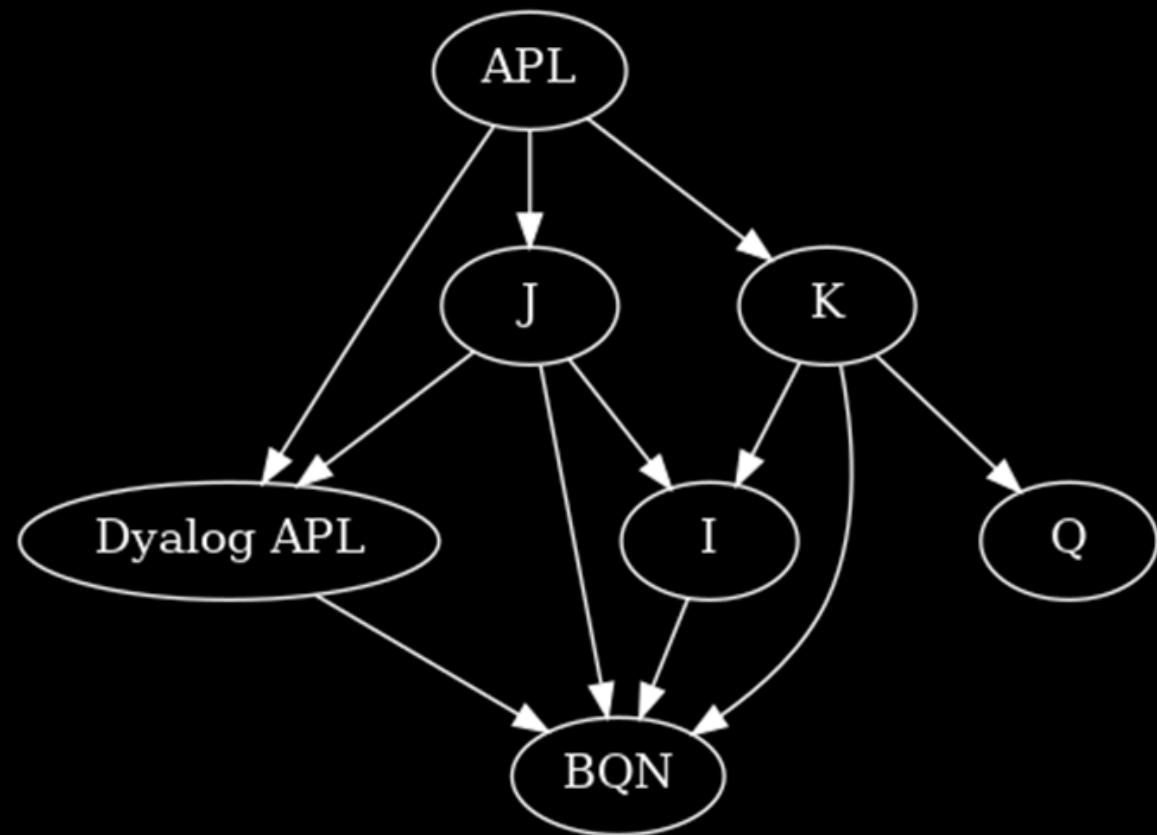


Table 1. Array languages timeline.

Language	Year
APL	1966
Dyalog APL 1.0	1983
J	1990
K	1994
Q	2003
I	2012
BQN	2020
Dyalog APL 18.0	2020

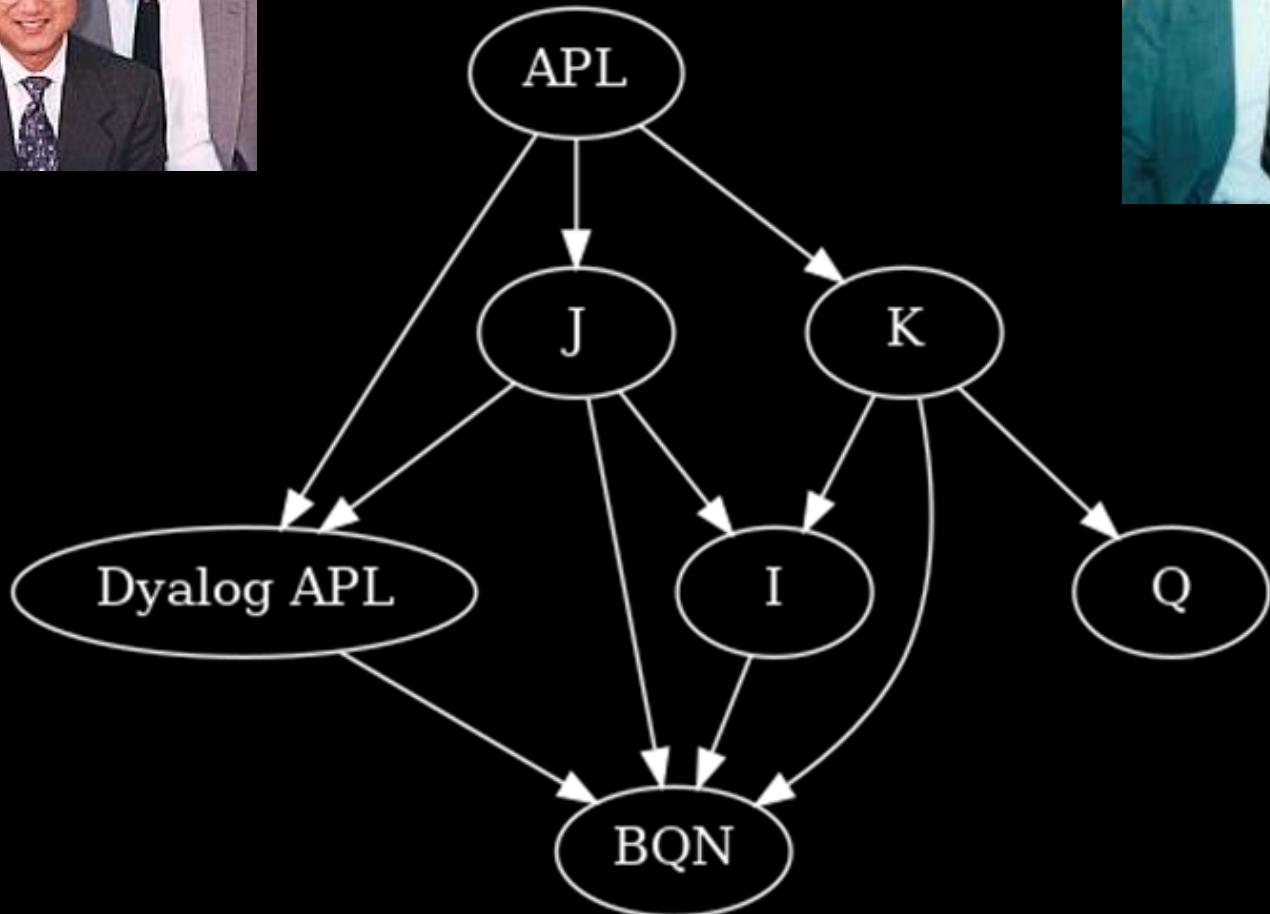


Table 1. Array languages timeline.

Language	Year
APL	1966
Dyalog APL 1.0	1983
J	1990
K	1994
Q	2003
I	2012
BQN	2020
Dyalog APL 18.0	2020

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	$\circ\tilde{\circ}$
2003	10.0	W	$\tilde{\circ}$
2013	13.0	K	\vdash
2014	14.0	B, B_1 , Φ , Φ_1	trains
2020	18.0	B, B_1 , Ψ , K	$\circ\ddot{\circ}\tilde{\circ}$

Combinator Table: Combinators, Birds, Spellings & More

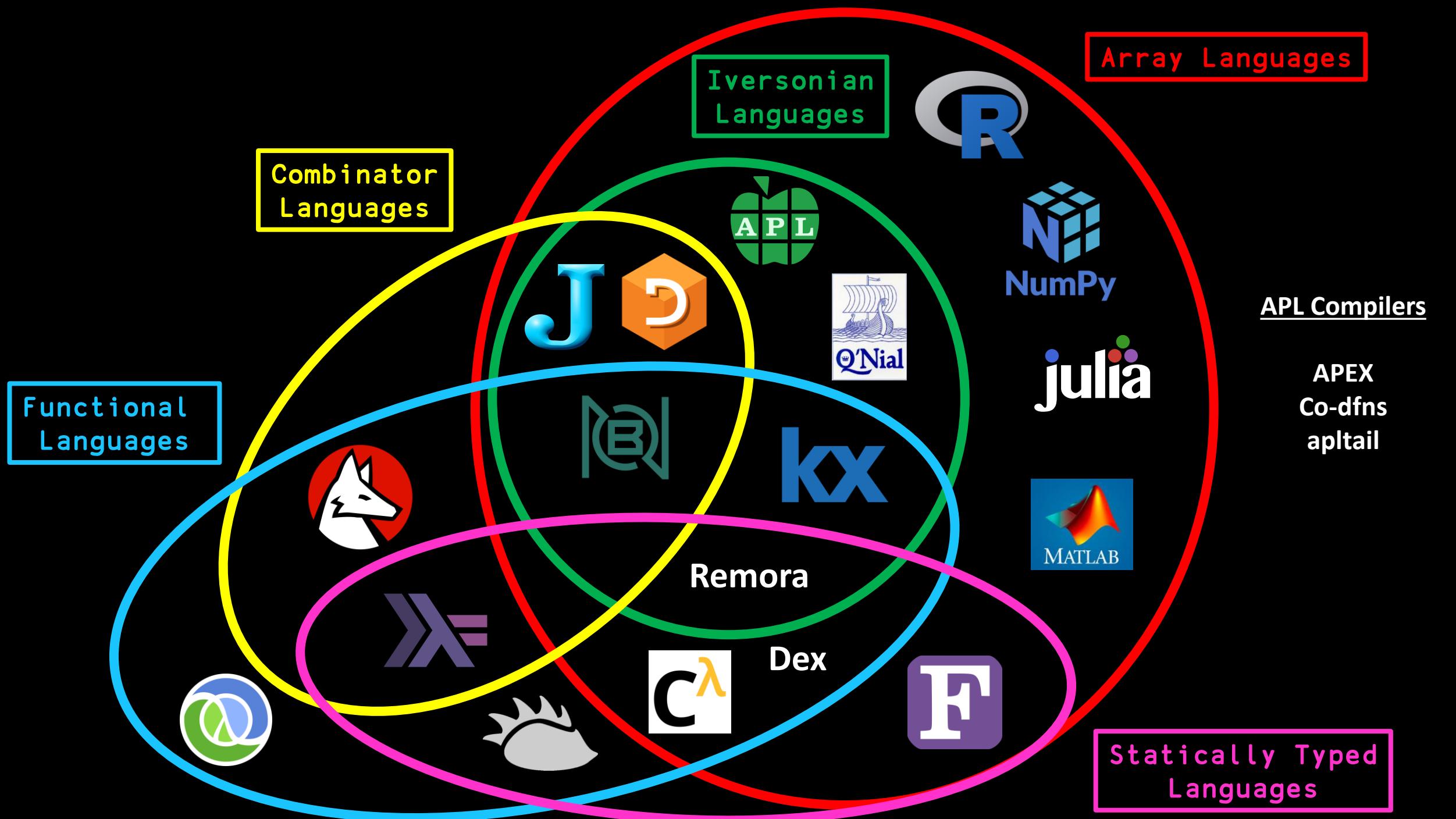
Function Abstraction	Symbol	Bird	CR ¹	CH ²	Elem ³	Haskell	APL	BQN	Intro
$\lambda abc.ac(bc)$	S	Starling	S	S		<*> / ap		○-	Sch24
$\lambda ab.a$	K	Kestrel	K	K	✓	const	⊤	⊤	Sch24
$\lambda a.a$	I	Identity	SKK	SKK	✓	id	-⊤-	-⊤-	Sch24
$\lambda ab.b$	K	Kite	KI	KI			⊤	⊤	
$\lambda ab.abb$	W	Warbler	C(BMR)	CSK	✓	join	~	~	Cur29
$\lambda abc.acb$	C	Cardinal	S(BBS)(KK)	B(ΦBS)KK	✓	flip	~	~	Sch24
$\lambda abc.a(bc)$	B	Bluebird	S(KS)K	S(KS)K	✓	.	○ö Ö 2T	○○ 2T	Sch24
$\lambda abcd.a(bcd)$	B ₁	Blackbird	BBB	DB		..	ö 2T	◦ 2T	Cur30
$\lambda abcde.a(bcde)$	B ₂	Bunting	B(BBB)B	DB ₁					Cur30
$\lambda abcd.a(b(cd))$	B ₃	Becard	B(BB)B	BDB					Cur30
$\lambda abcd.ab(cd)$	D	Dove	BB	BB			◦	○-	Smu85
$\lambda abcd.a(bd)(cd)$	Φ	Phoenix	-	B ₁ SB		liftA2	3T	3T	Cur31
$\lambda abcd.a(bc)(bd)$	Ψ	Psi	-	B(SΦCB)B		on	Ö	◦	Cur31
$\lambda abcde.abc(de)$	D ₁	Dickcissel	B(BB)	BD					
$\lambda abcde.a(bc)(de)$	D ₂	Dovekies	BB(BB)	DD					Smu85
$\lambda abcde.ab(cde)$	E	Eagle	B(BBB)	BB ₁					Smu85
$\lambda abcde.a(bde)(cde)$	Φ ₁	Pheasant	-	BΦΦ			3T	3T	Cur31
$\lambda abcdefg.a(bcd)(efg)$	Ê	Bald Eagle	B(BBB)(B(BBB))	D ₂ D ₂ D					Smu85

		⌚	J
I	⊣⊣	⊣⊣] [
K	⊣	⊣]
KI	⊣	⊣	[
S		-	2 Train
B	◦ ö ö 2T	◦ O 2T	@: &:
C	≈	~	~
W	≈	~	~
B ₁	ö 2T	◦ 2T	@:
D	◦	-	2 Train
Ψ	ö	O	&:
Φ	3 Train	3 Train	3 Train
Φ ₁	3 Train	3 Train	3 Train
D ₂		-oo-	

	 APL	⌚	J
I	⊣⊣	⊣⊣] [
K	⊣	⊣]
KI	⊣	⊣	[
S		-	2 Train
B	◦ ö ö 2T	◦ O 2T	@: &:
C	~	~	~
W	~	~	~
B ₁	ö 2T	◦ 2T	@:
D	◦	-	2 Train
Ψ	ö	O	&:
Φ	3 Train	3 Train	3 Train
Φ ₁	3 Train	3 Train	3 Train
D ₂		-oo-	

		⌚	J
I	⊣↑	⊣↑] [
K	↑	↑]
KI	⊣	⊣	[
S		-	2 Train
B	◦ ö ö 2T	◦ o 2T	@: &:
C	~	~	~
W	~	~	~
B ₁	ö 2T	◦ 2T	@:
D	◦	-	2 Train
Ψ	ö	o	&:
Φ	3 Train	3 Train	3 Train
Φ ₁	3 Train	3 Train	3 Train
D ₂		-oo-	

		⌚	J
I	⊣⊣	⊣⊣] [
K	⊣	⊣]
KI	⊣	⊣	[
S		⊖	2 Train
B	◦ ◦ ö 2T	◦ O 2T	@: &:
C	≈	~	~
W	≈	~	~
B ₁	ö 2T	◦ 2T	@:
D	◦	⊖	2 Train
Ψ	ö	O	&:
Φ	3 Train	3 Train	3 Train
Φ ₁	3 Train	3 Train	3 Train
D ₂		-oo-	
Σ		-o	
Δ		-o	



What is a combinator?

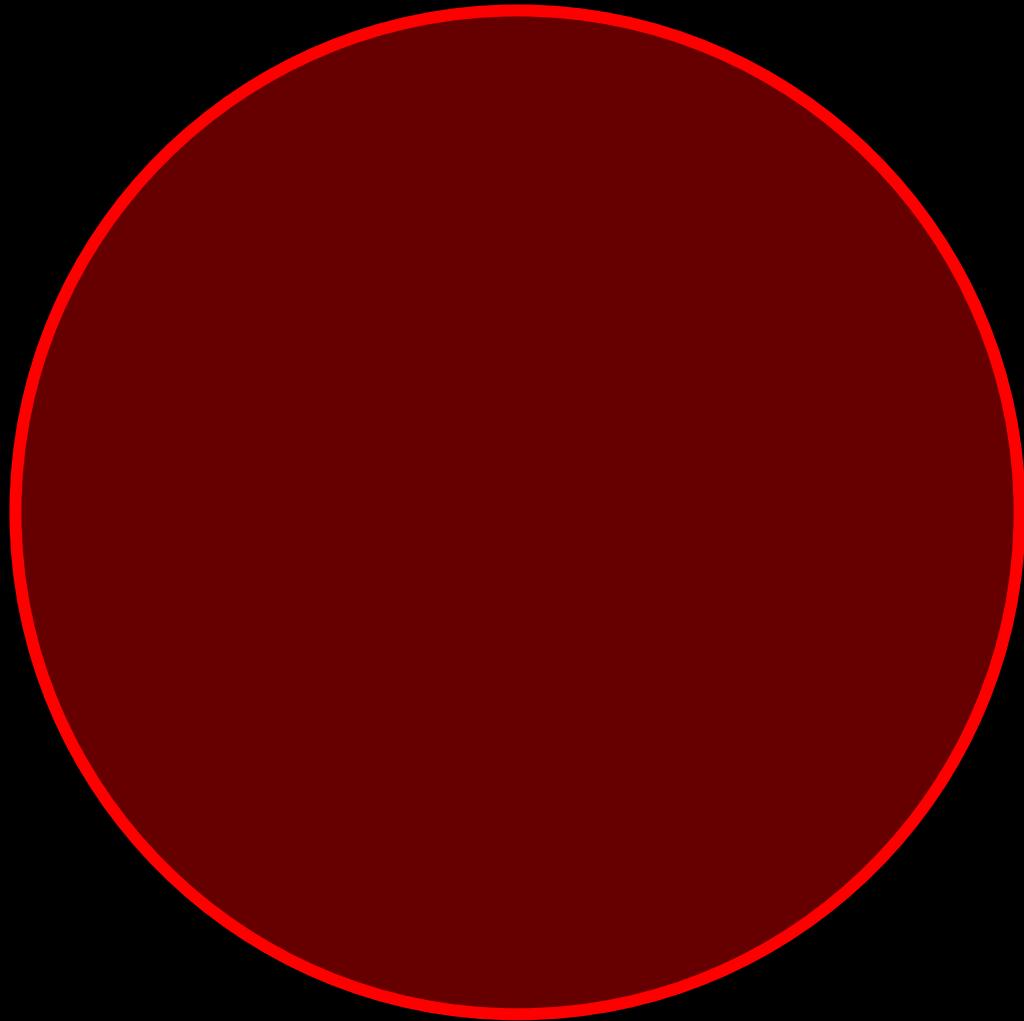
**combinator: a lambda expression
containing no free variables**

combinator: a function that deals
only in its arguments

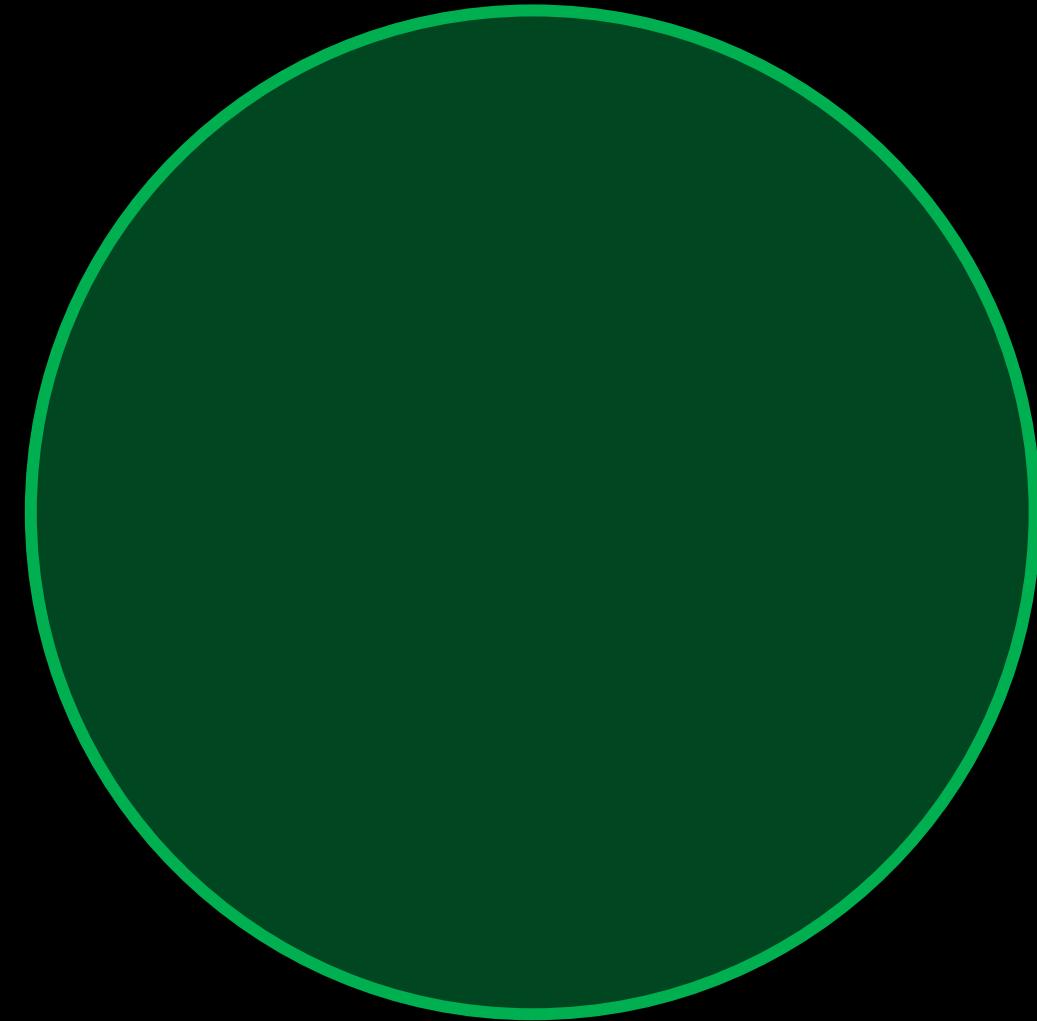
combinator = pure function?

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

pure function

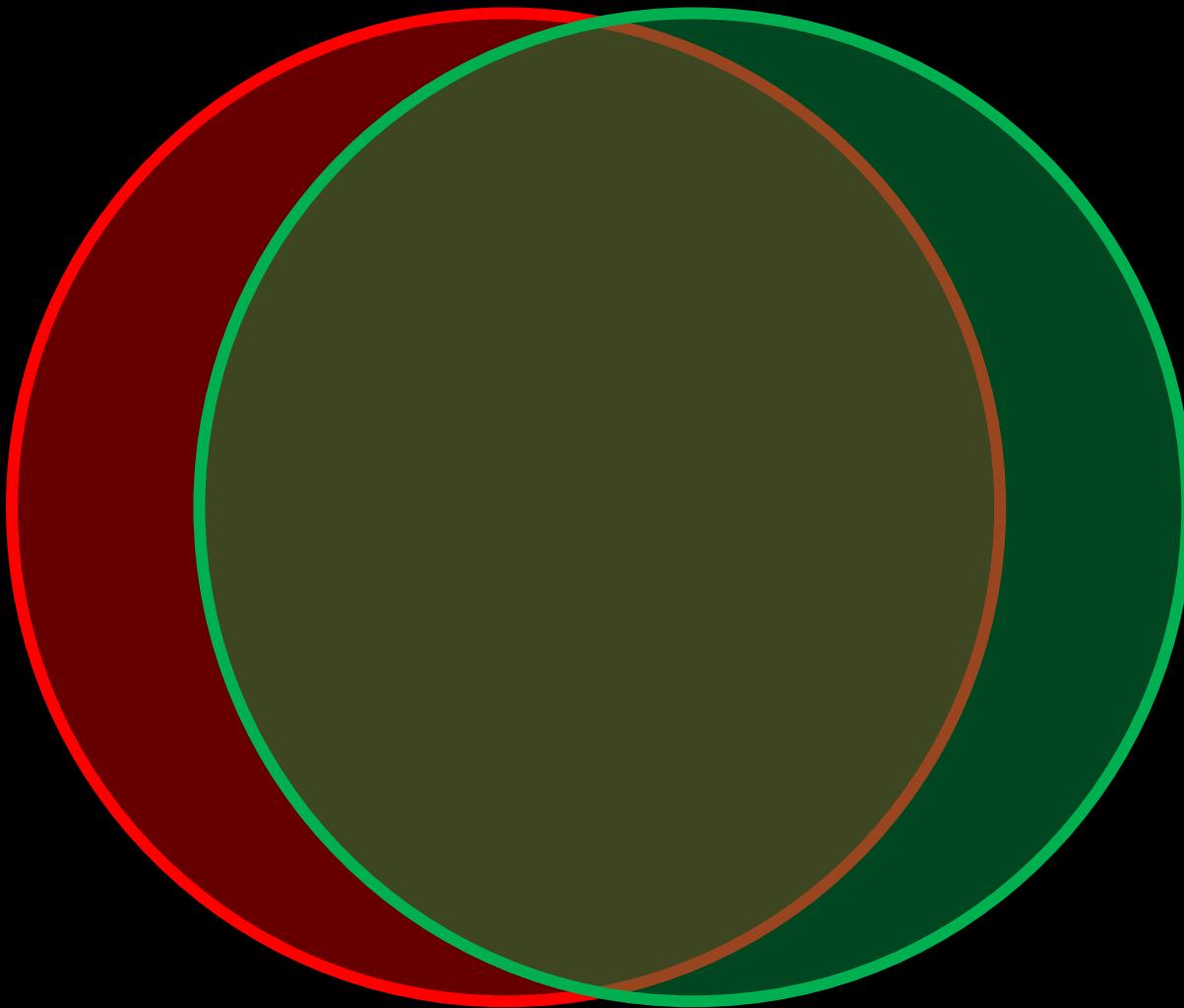


combinator



pure function

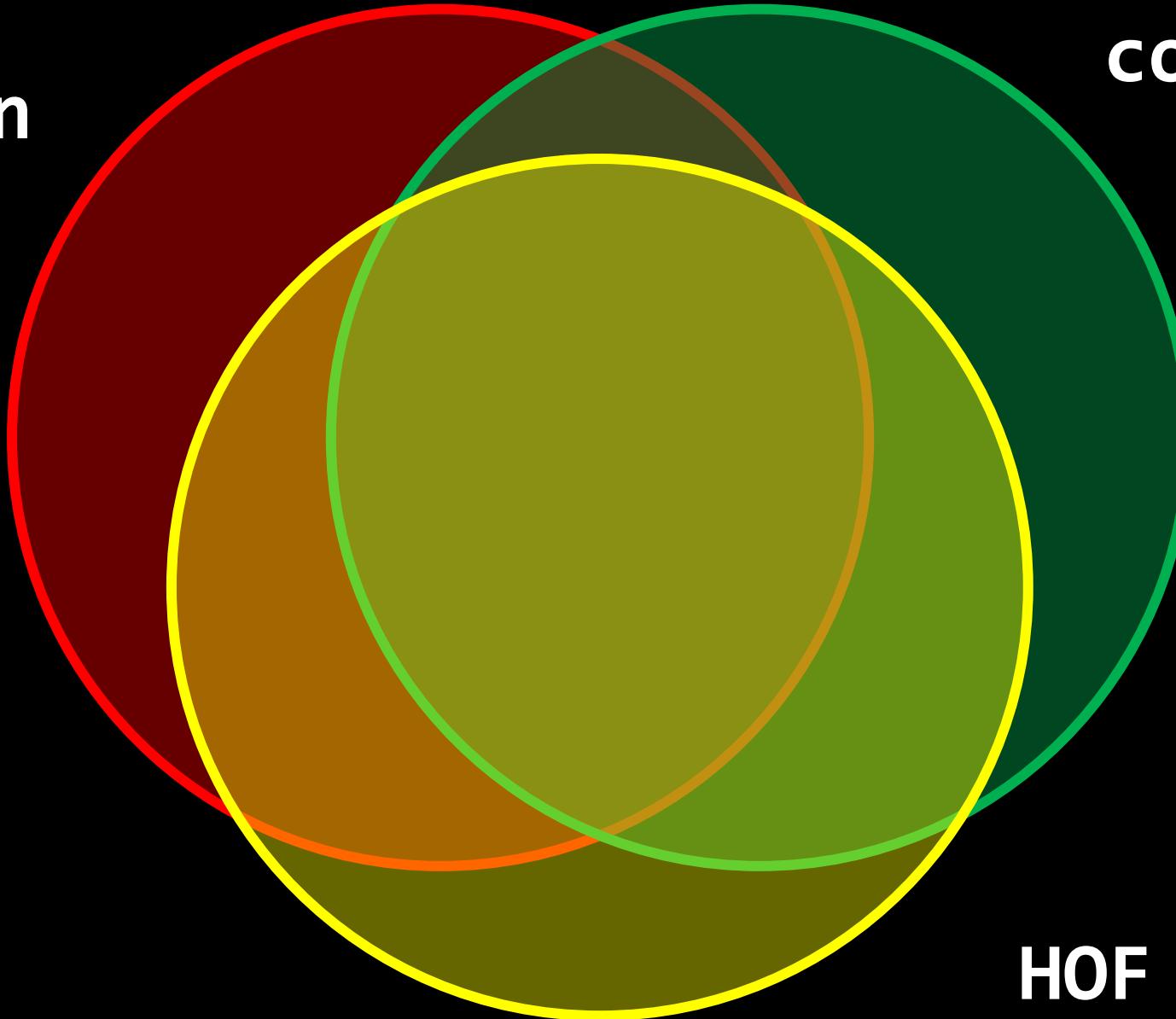
combinator



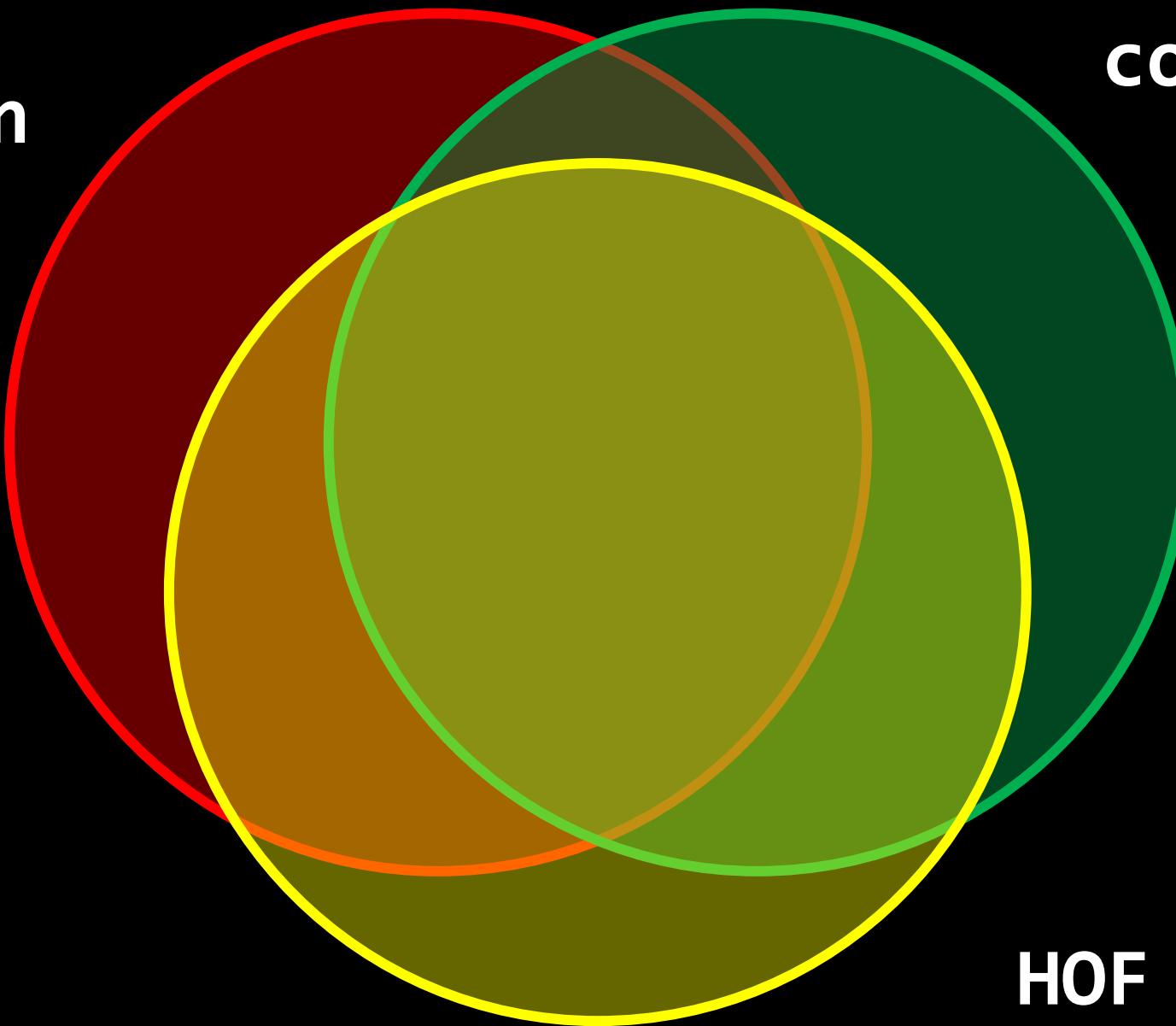
pure
function

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

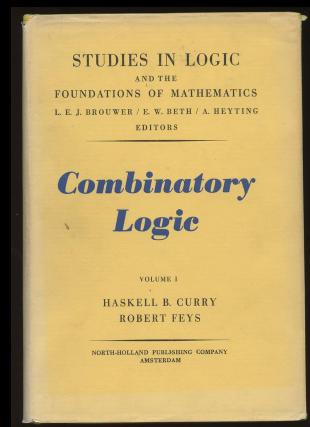
HOF: consumes OR produces a function

Show Us Some Combinators!

THE ELEMENTARY COMBINATORS

Table 2. The elementary combinators.

Combinator	Elementary Name
I	Elementary Identifier
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 \hat{E} combinator is "tetradic"

Unary/Monadic

Binary/Dyadic

Ternary/Triadic

Quaternary/Tetradic

Terminology [\[edit \]](#)

Latinate names are commonly used for specific arities, primarily based on [cardinal numbers](#) or [ordinal numbers](#). For example, 1-ary is based on

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



6



2



46



[[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))
```



2.00

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



Sept 15-17, 2016
thestrangeloop.com

"Point-Free or Die: Tacit Programming"

2.00

exactMatches =



Oct 15-17, 2016
onestrangeloop.com



▶ ▶ 🔍 23:08 / 36:12





2.00

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



Sept 15-17, 2016
thestrangeloop.com



```
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))

= exactMatches "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 .: (==))
```



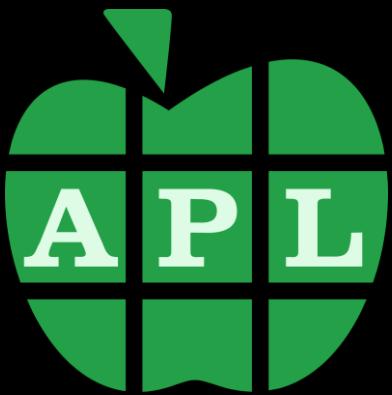
```
countElem True .: zipWith (==)
```

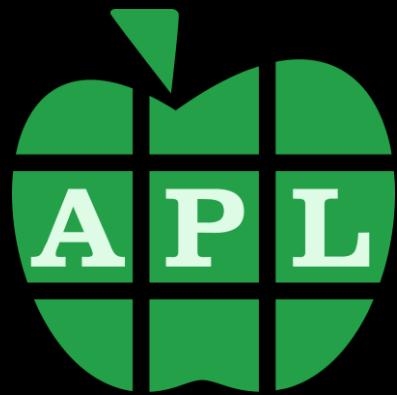


```
length . filter id .: zipWith (==)
sum . map fromEnum .: zipWith (==)
sum .: zipWith (fromEnum .: (==))
countElem True .: zipWith (==)
```



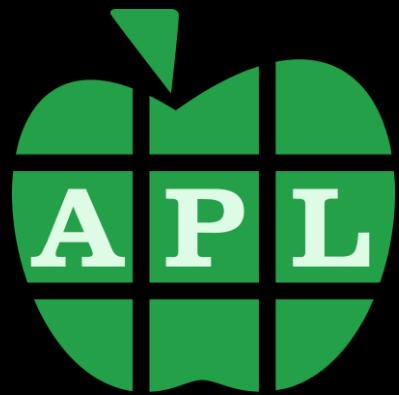
```
length . filter id .: zipWith (=)
sum . map fromEnum .: zipWith (=)
sum .: zipWith (fromEnum .: (=))
countElem True .: zipWith (=)
```





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



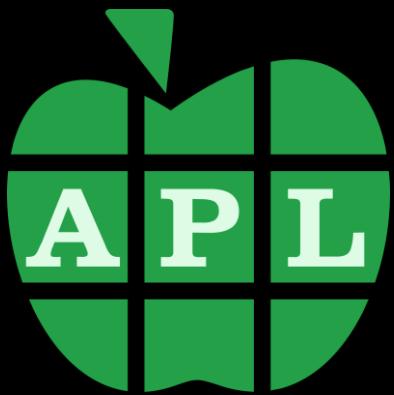


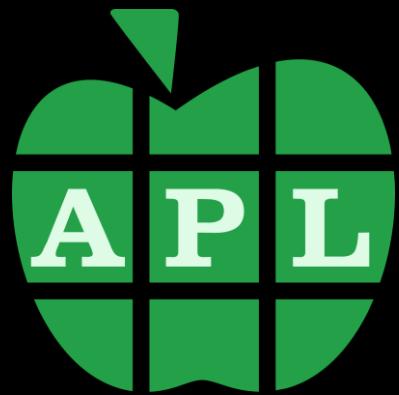
```
length . filter id .: zipWith (==)
sum . map fromEnum .: zipWith (==)
sum .: zipWith (fromEnum .: (==))
countElem True .: zipWith (==)
```

+ / =



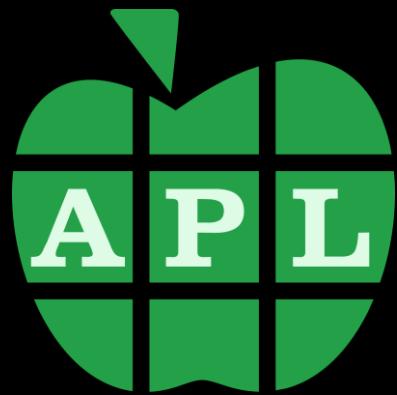
```
length . filter id .: zipWith (==)
sum . map fromEnum .: zipWith (==)
sum .: zipWith (fromEnum .: (==))
countElem True .: zipWith (==)
```


$$\{ + / \alpha = \omega \}$$



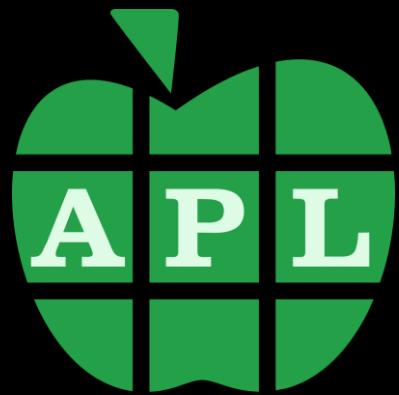
```
length . filter id .: zipWith (==)
sum . map fromEnum .: zipWith (==)
sum .: zipWith (fromEnum .: (==))
countElem True .: zipWith (==)
```

+ / =



```
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 (==)
```

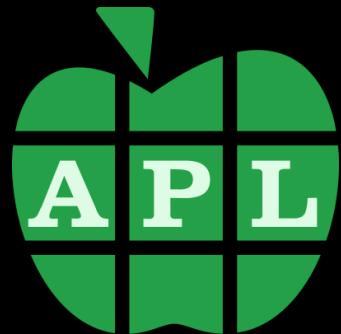
+ / =



```
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{});  
}
```



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



ExactMatches $\leftarrow +/_=$



Conor Hoekstra @code_report · Oct 21, 2020

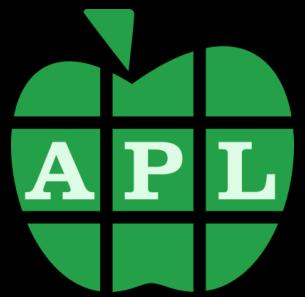
...

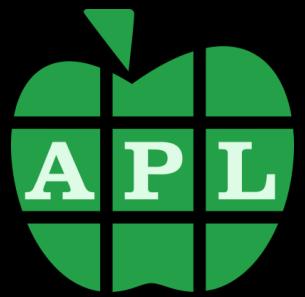
And for maybe my most amazing combinator discovery to date,
std::inner_product is the **Blackbird** 🐦 (which is what I was looking for
when I stumbled across the Phoenix) #combinators #cpp #algorithms





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

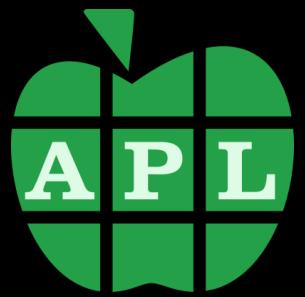

$$\{ (+/0>\omega), (+/0<\omega) \}$$



(+ / 0 > ⊢) , (+ / 0 < ⊢)



($0 > \top$), ($0 < \top$)
+ /



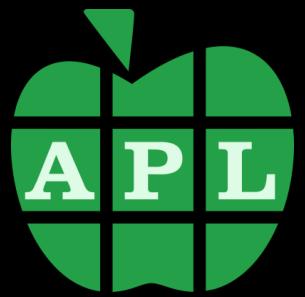
(0 > ⊢) , ö (+ /) (0 < ⊢)

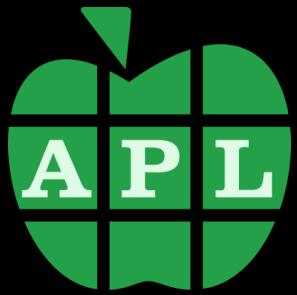


0 o (>, ö(+/)<)



0 o (>, ö (+/) <)
 |_____|
 ψ

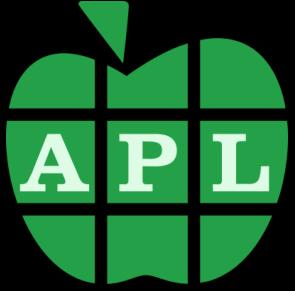

$$\Phi_1$$
$$0 \circ (>, \ddot{o} (+ /) <)$$
$$\Psi$$



```
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))
```

$$\Phi_1$$
$$0 \circ (>, \ddot{o} (+ /) <)$$
$$\Psi$$

The diagram illustrates the derivation of the APL function Ψ from Φ_1 . A curved brace originates from the symbol \circ in the expression $0 \circ (>, \ddot{o} (+ /) <)$ and points upwards to the symbol Φ_1 . Another curved brace originates from the entire expression $>, \ddot{o} (+ /) <$ and points downwards to the symbol Ψ .



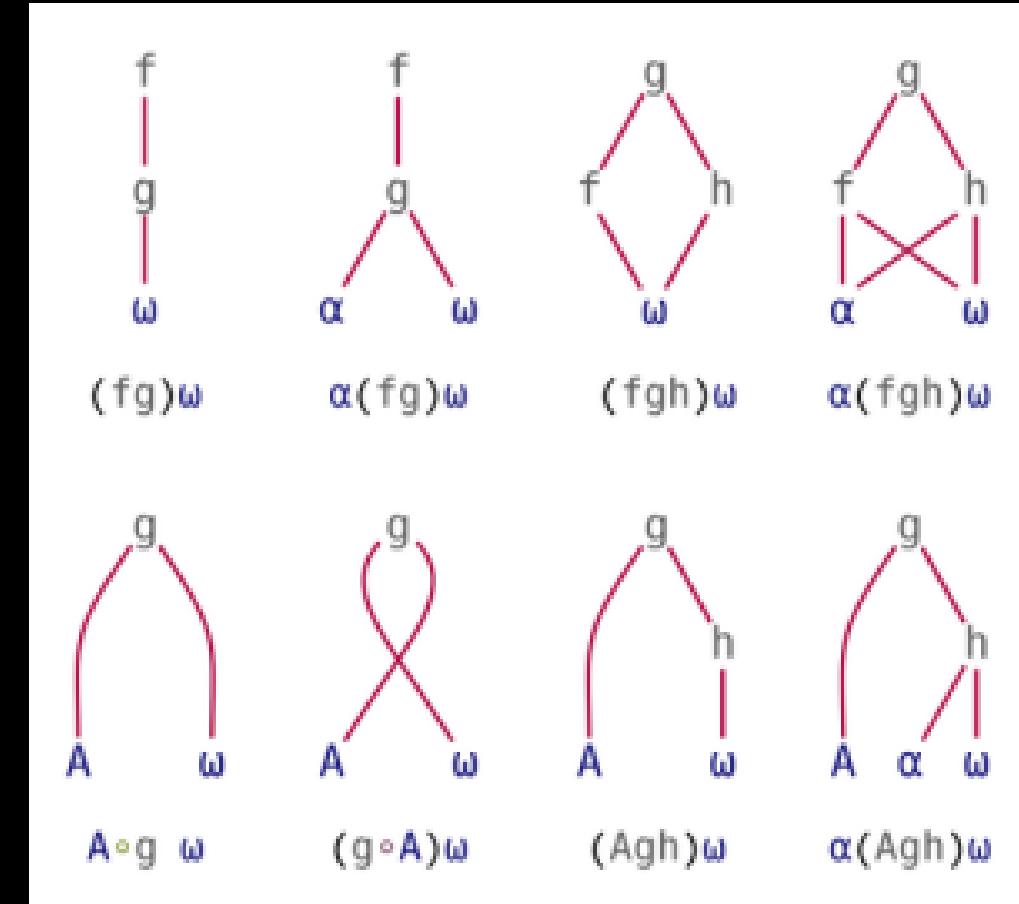
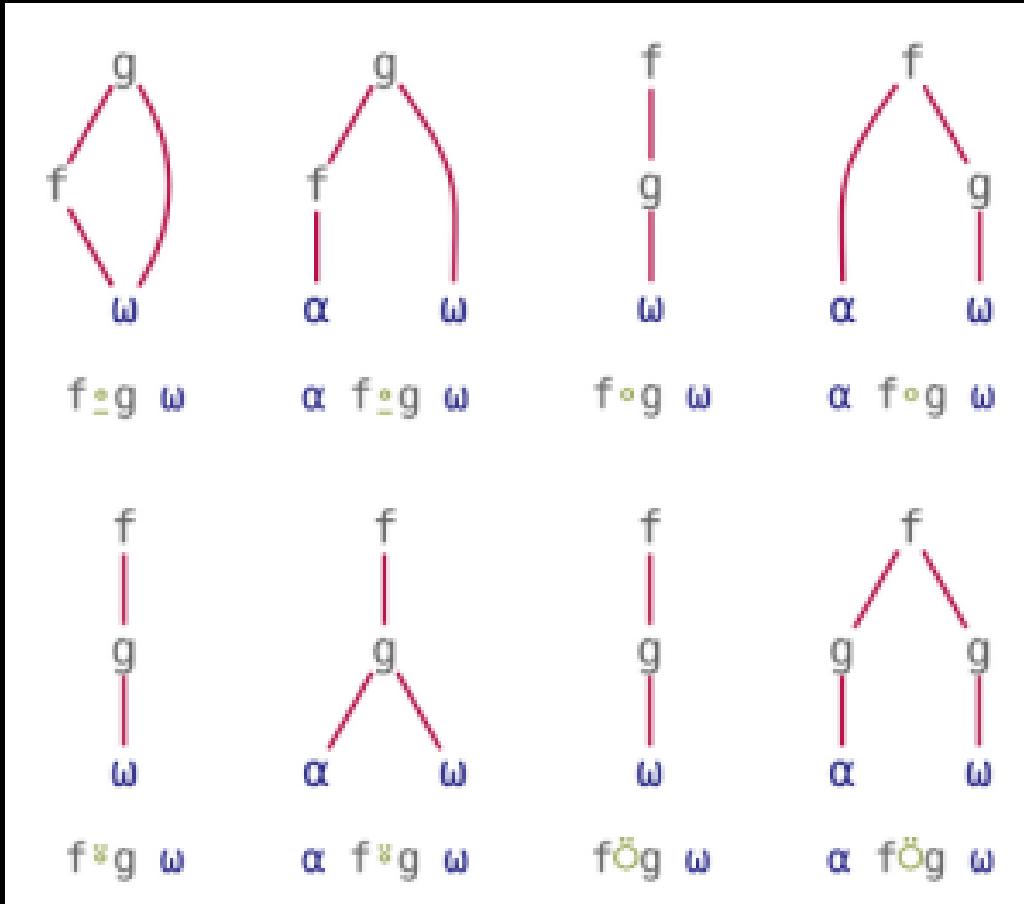
0 o (>, ö(+/)<)



```
liftA2 (on (,)  
        (length . filter id))  
        (map (>0))  
        (map (<0)))
```

Visualizing Combinators

https://aplwiki.com/wiki/Tacit_programming



Tacit Techniques in Dyalog 18.0

Composition operators

Beside

f
|
g
|
ω

$f \circ g \ \omega$

$(f \ g) \omega$

Compose

f
|
g
|
ω

$\alpha \ f \circ g \ \omega$

$\alpha(f \ g) \omega$

Bind

A
|
f
|
ω

$A \circ f \ \omega$

$f \circ A \ \omega$

Commute

f
|
ω

$f \tilde{\circ} \ \omega$

$\alpha \ f \tilde{\circ} \ \omega$

Atop

f
|
g
|
ω

$f \ddot{\circ} g \ \omega$

$(f \ g) \omega$

f
|
g
|
ω

$\alpha \ f \ddot{\circ} g \ \omega$

$\alpha(f \ g) \omega$

Over

f
|
g
|
ω

$f \ddot{\circ} g \ \omega$

$\alpha \ f \ddot{\circ} g \ \omega$

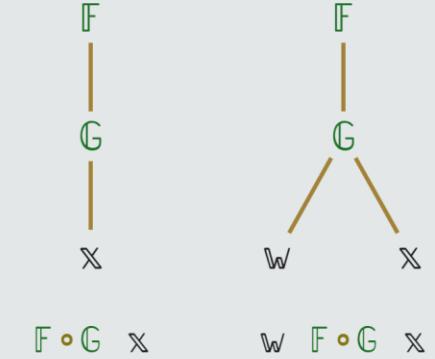
Constant

A
|
ω

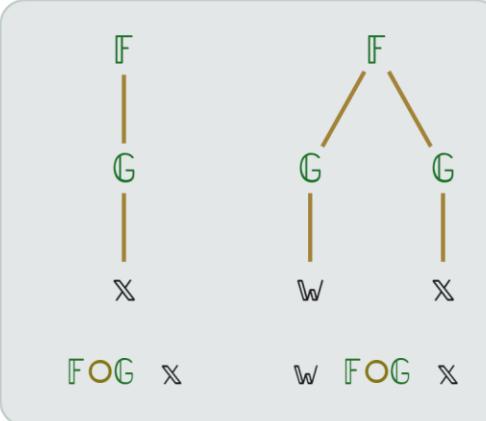
$(A \tilde{\circ}) \ \omega$

$\alpha \ (A \tilde{\circ}) \ \omega$

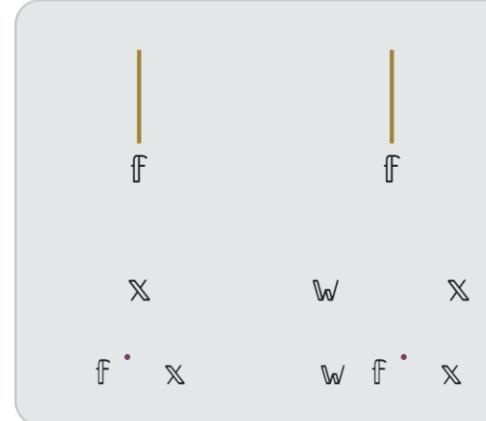
<https://mlochbaum.github.io/BQN/doc/tacit.html#combinators>



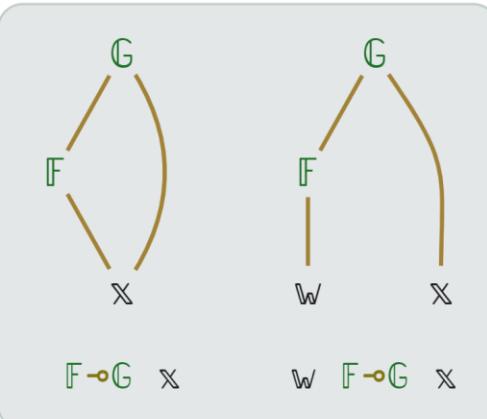
Atop



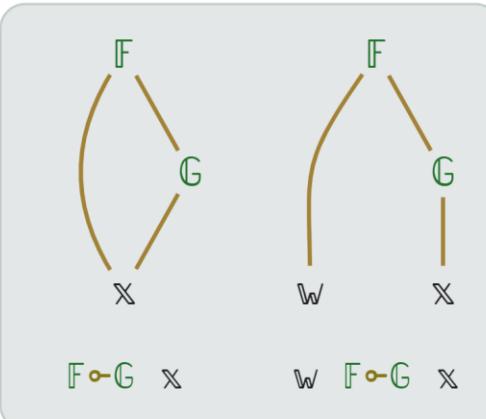
Over



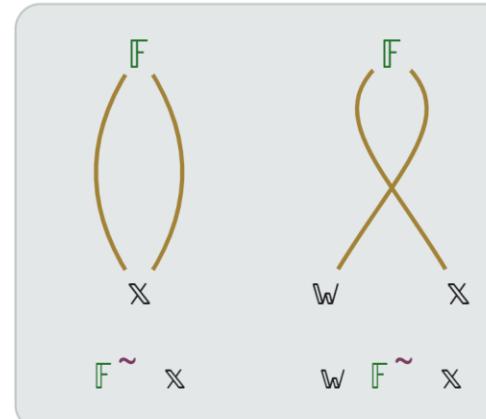
Constant



Before

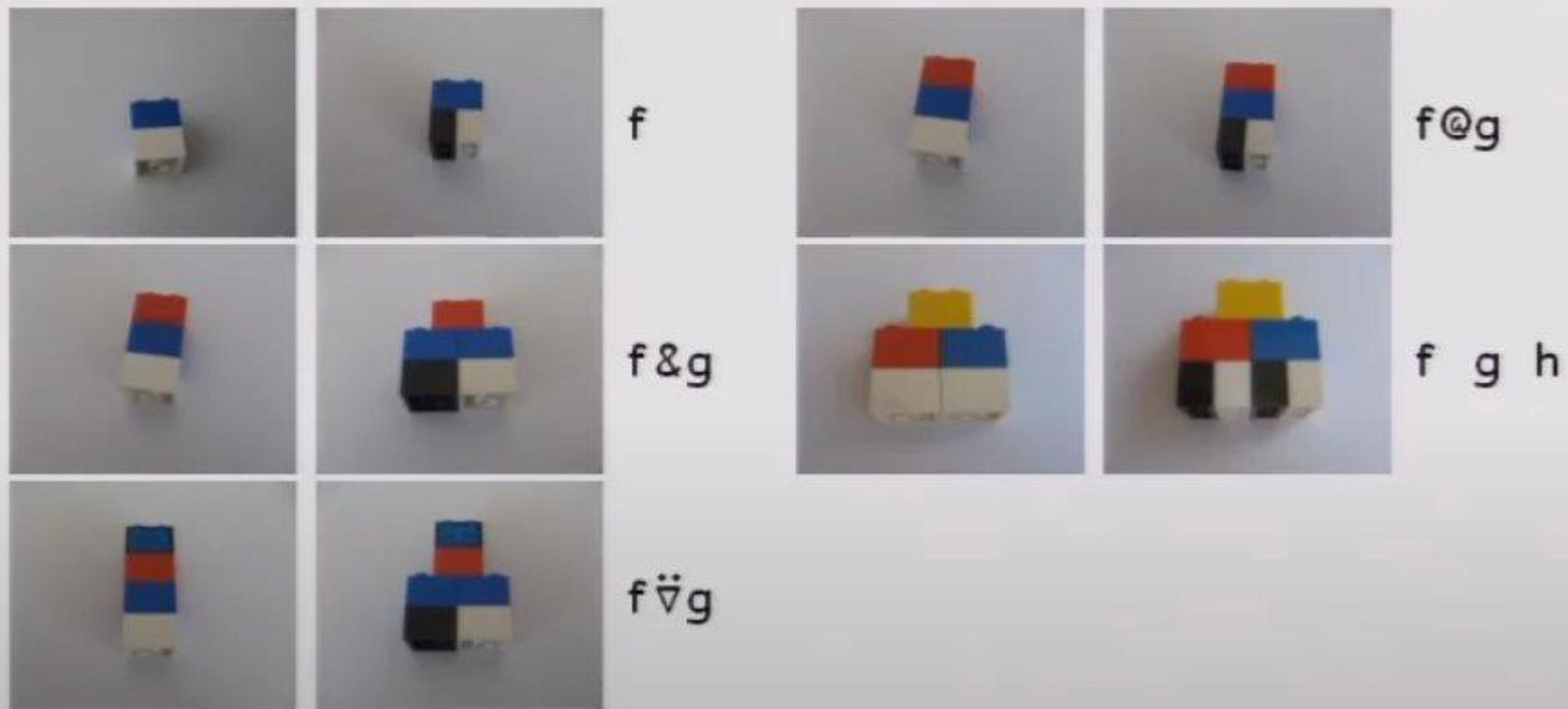


After



Self/Swap

Combinators: System Summary



What was the goal of this talk?

What was the goal of this talk?

Composition is more than just $f \circ g$

What was the goal of this talk?

Composition is more than just $f \circ g$

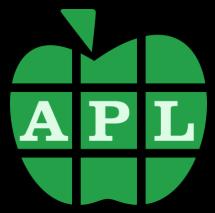
```
type state :  
    template <typename F1, typename F2>  
    auto compose(F1 f1, F2 f2)  
    {  
        return [=](auto const& x) { return f1(f2(x)); };  
    }
```

<https://www.youtube.com/watch?v=qL6zUn7iiLg>

What was the goal of this talk?

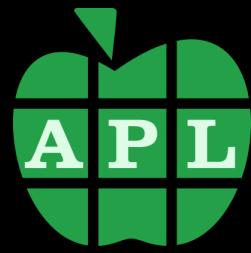
Composition is more than just $f \circ g$

- Sub goal 1: popularize array-combinator languages
- Sub goal 2: story of how I discovered combinators
- Sub goal 3: combinators are really important



o ö ö ~

+ - × ÷



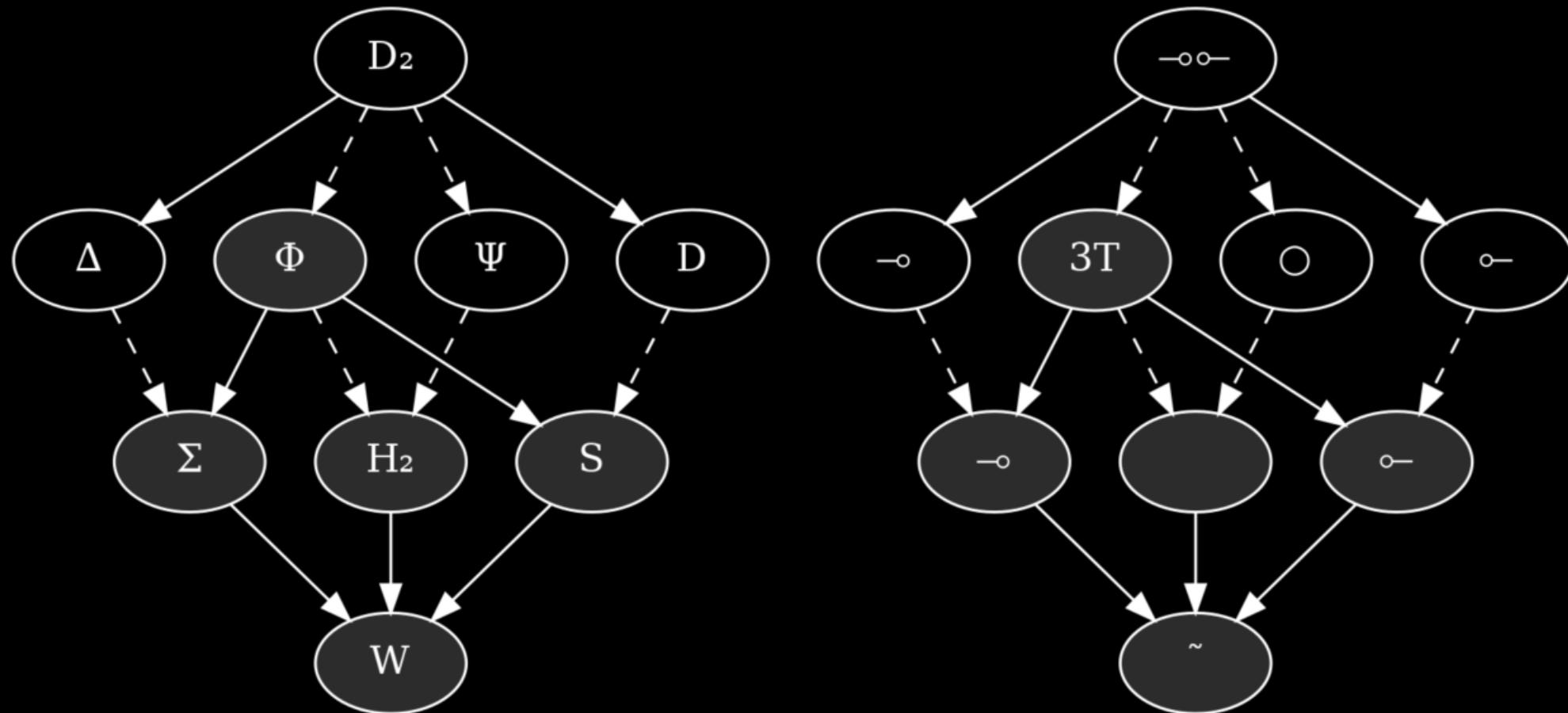
↑↑ o ö ö ~



↑↑ o ○○~

Things I didn't mention:

- Concatenative languages like FORTH, Joy
- FP & FL (John Backus & Alex Aiken)
- The implicit Φ combinator in The FL Project
- Function application & the curse of ()
- My discovery of the Ψ combinator
- The combinator hierarchy / specializations
- 4+-trains



www.combinatorylogic.com



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