# Living the Loopless Life

Aaron W. Hsu arcfide@sacrideo.us

LambdaConf 2024, Estes Park, CO

# Background

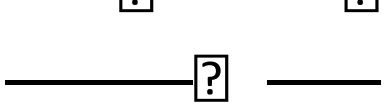
I write APL code,
primarily a compiler for APL written in APL.
Almost all of it is Loopless,
meaning no syntactic control flow syntax.

It is designed to be:
Data-parallel, GPU-friendly,
Fast,
Maintainable,
Portable.

APL is (one of) the best language(s) for this.
Why?

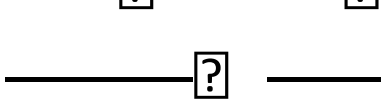
From an HCI/d perspective,
how does APL support
large-scale, data-parallel, loopless software?

What are its unique ergonomic affordances?



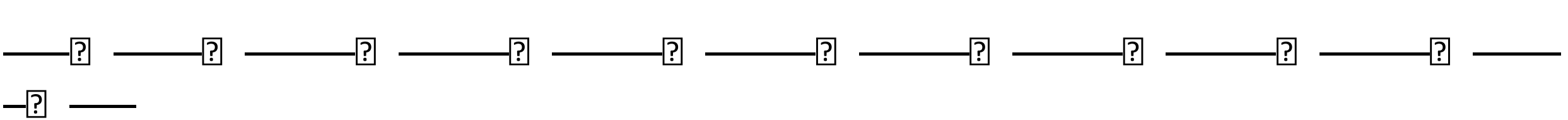
# Background

Most people don’t “get” APL.
They miss the big picture.



Baseline

This is the part everyone “gets”:



Baseline

| | Monadic | Dyadic |
|---|---|---|
| + | Conjugate | Plus |
| - | Negative | Minus |
| × | Signum | Times |
| ÷ | Reciprocal | Divide |
| \| | Magnitude | Residue |
| ⌈ | Ceiling | Maximum |
| ⌊ | Floor | Minimum |
| * | Exponent | Power |
| ⍟ | Nat Log | Logarithm |
| ! | Factorial | Binomial |

| | Dyadic | |
|---|---|---|
| | ∧ | And |
| | ∨ | Or |
| | ⍲ | Nand |
| | ⍱ | Nor |
| | < | Less than |
| | ≤ | Less than or Equal |
| | = | Equal |
| | ≥ | Greater than |
| | > | Greater than or Equal |
| | ≠ | Not Equal |

Baseline

$\mathbb{A}$: Array

---

Elements: $e_0\ e_1 \ldots e_{n-1} \in \mathbb{A}$ ,A | n $\longleftrightarrow \not\equiv$,A

Shape: $d_0 \ldots d_{k-1} \in \mathbb{N}$ ρA | k $\longleftrightarrow \not\equiv$ρA

$d_0 \longleftrightarrow \not\equiv$A

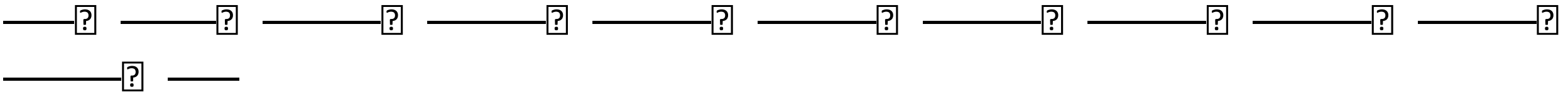
≡ Y Nesting level of A

X ≡ Y X same as Y

X ≢ Y X not same as Y

Scalars → Arrays

```
(make-array (array-shape A)
        (map s-prim (array-elems A)))
```



Baseline

## Dyadic Application

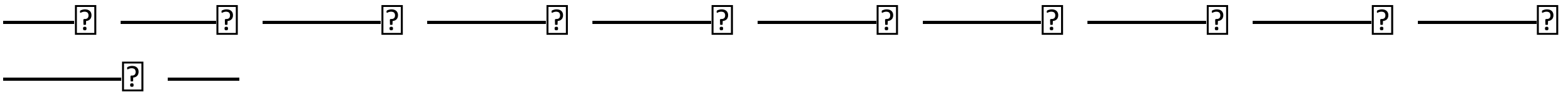
0 1 2 3 4 0 1 2 3 4 0 1 4 9 16

5 6 7 8 9 0 1 2 3 4 0 6 14 24 36

10 11 12 13 14 × 0 1 2 3 4 ⟷ 0 11 24 39 56

15 16 17 18 19 0 1 2 3 4 0 16 34 54 76

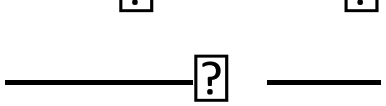
20 21 22 23 24 0 1 2 3 4 0 21 44 69 96



Baseline

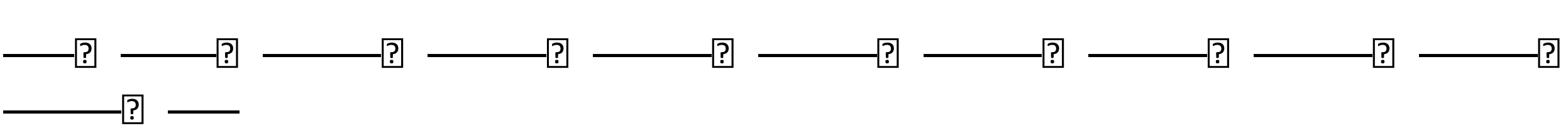
Scalar Extension

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | | 0 | 5 | 10 | 15 | 20 |
| 5 | 6 | 7 | 8 | 9 | | 25 | 30 | 35 | 40 | 45 |
| 10 | 11 | 12 | 13 | 14 | × 5 ⟷ | 50 | 55 | 60 | 65 | 70 |
| 15 | 16 | 17 | 18 | 19 | | 75 | 80 | 85 | 90 | 95 |
| 20 | 21 | 22 | 23 | 24 | | 100 | 105 | 110 | 115 | 120 |



Baseline

Rank Polymorphic Pointwise Lifting
Scalar Function Primitives



Baseline

Indexing/Assignment

$A[I_0;\ldots;I_{k-1}]\leftarrow X \quad A[I_0;\ldots;I_{k-1}] \quad X(f@g)Y$

$(f\ A)\leftarrow X \quad I_0\ \ldots\ I_{k-1}\square A$

Composition (Points-free/Tacit)

$X\ f\circ g\ Y \longleftrightarrow (X\ f\ (g\ Y)) \mid X\ f\ddot{\sim}\ Y \longleftrightarrow Y\ f\ X \mid X\ f\ddot{\circ} g\ Y \longleftrightarrow f\ (X\ g\ Y)$

$A\circ f\ Y \longleftrightarrow (A\ f\ Y) \quad \mid X\ A\ddot{\sim}\ Y \longleftrightarrow A \quad \mid X\ f\ddot{\bigcirc} g\ Y \longleftrightarrow (g\ X)\ f\ (g\ Y)$

$f\circ A\ Y \longleftrightarrow (Y\ f\ A)$

$X(f\ g\ h)Y \longleftrightarrow (X\ f\ Y)\ g\ (X\ h\ Y) \mid X\ (f\ g)\ Y \longleftrightarrow f\ (X\ g\ Y)$

# Iteration/Traversal

Pointwise isn't the only pattern…

{X} f¨ Y Each/Map f⌿ Y Reduce (First axis)
{X} f[axis] Y Along axis f/ Y Reduce (Last axis)
X ∘.f Y Outer product X f⌿ Y N-wise reduce (First)
X f.g Y Inner product X f/ Y N-wise reduce (Last)
{X} f⍣N Y Repeated iteration f⍀ Y Scan (First axis)
{X} f⍣g Y "Fixed Point" f\ Y Scan (Last axis)
{X} f⍤v Y Rank
{X} f⌸s Y Stencil
{X} f⌸ Y Key/Group by

# Manipulation

We need to change the structure of arrays…

X , Y  Catenate (Last)  ↑ Y  Lift depth  X ↑ Y  Take

 ⍪ Y  Table  ↓ Y  Push depth  X ↓ Y  Drop

X ⍪ Y  Catenate (First)

X ⍉ Y  Transpose

 ⌽ Y  Reverse (Last)

 ⊖ Y  Reverse (First)  ⊂ Y  Enclose  X ⊂ Y  Enc. Partition

X ⌽ Y  Rotate  (Last)  ⊆ Y  Nest  X ⊆ Y  Partition

X ⊖ Y  Rotate  (First)  ∈ Y  Flatten

# Manipulation

We need to select elements from arrays…

⊃ Y First
X ⊃ Y Pick
X / Y Replicate (Last)
X ⌿ Y Replicate (First)
X \ Y Expand (Last)
X ⍀ Y Expand (First)

X ~ Y Without
X ∪ Y Union
X ∩ Y Intersection
≠ Y Unique Mask
X ∈ Y Membership

X ⊢ Y Right
X ⊣ Y Left

# Theory

Basic CS theory concepts as algorithms (c.f. Conor)…

⍳ Y  Index space of shape Y
X ⍳ Y  Index of Y in X
⍸ Y  Indices of non-zeros
X ⍸ Y  Interval in X containing Y
⍋ Y  Grade up (V[⍋V] sorts V)
⍒ Y  Grade down
? Y  Roll dice
X ? Y  Deal cards
X ∈ Y  Find subarrays

X ⊥ Y  Decode Y as radix X
X ⊤ Y  Encode Y as radix X
⌹ Y  Matrix Inverse
X ⌹ Y  Matrix Divide

# Idioms

That’s the base…where everyone stops.
Solve your problem using only those primitives.

Idioms are the phrases that turn symbols into meaning.

Sub-indices given by X: ∊⍳¨X ⟷ ∊⍳¨1 2 3 ⟷ 0 0 1 0 1 2
Indices matching predicate: ⍸B ⟷ ⍸t=F ⟷ ⍸vb>0
Primes to N: (2=+⌿0=X∘.|X)/X←1+⍳N
Game of Life: ⊃1 ωV.∧3 4=+⌿,¯1 0 1∘.⊖¯1 0 1∘.⌽⊂ω
Depth to Parent: p⍳2{p[ω]←α[α⍸ω]}⍣⊢∘⊂⌸d⍳p←⍳≢d
RelU 3×3 Convolution: 0⌈(,[2+⍳3]{ω}⌺3 3⊢ω)+.×,[⍳3]α

## Equational Reasoning

Idioms can be expressed with other idioms,
Defined in terms of each other,
Inspire new paths.

They can connect the concept of one primitive to another.

Equational Reasoning

$$x \neq v \leftrightarrow \in x \rho \ddot{} v$$

Find j such that

$$(x \neq v)[j] \leftrightarrow v \leftrightarrow (x \neq v)[j+x-1]$$

when $\wedge \neq x>0$

Exclusive Scan:

$$(+ \not\backslash x)\text{-}x$$

# Encoding

It’s hard to use idioms if you hide everything behind abstraction.

Thus,
we must learn to encode the domain into arrays.

Maximize the degree to which idioms and primitives have a meaningful interpretation in the encoded domain.

# Encoding

## Trees

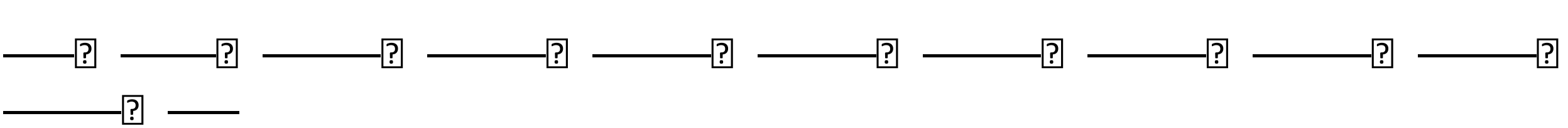
p  Parent  p[i] is the index into p of the parent of i
t  Type  t∈A F E P

$$\underline{\iota}(t=E)\wedge t[p]=F$$

"The nodes where the type is an expression and the type of its parent is a function."

iιp[i] ⟷ indices of i matching p[i]
⟷ independent tree matching subtree i

APL semantics mean something in the domain of trees now!
No new syntax/abstractions.



# Control Flow

Control flow/logic also needs to be encoded.

Booleans ⟷ Conditional Behavior/Branching
Indices ⟷ Pointers, Structures, Graphs
Shape ⟷ Partitions, groups

$$\underline{\iota}(t=E)\wedge t[p]=F$$

Booleans, Iteration, Traversal, Types, Pointers, Relationships, Object IDs

# Dependency

Architecture can ruin simplicity.

Simplify architecture by modeling APL data-flow in the large.

Minimize dependencies (edges) in the architecture graph,
aiming for linear > tree > single cycle > [crazy].

Compile: TK → PS → TT → GC → CC → LK

TK: $T_0 \rightarrow T_1 \rightarrow \ldots \rightarrow T_n$

PS: $P_0 \rightarrow P_1 \rightarrow \ldots \rightarrow P_n$

TT: $C_0 \rightarrow C_1 \rightarrow \ldots \rightarrow C_n$

# Secret Sauce

## Example: Function Specialization

Parsing APL expressions requires type information.
Recursively specialize functions on different types of inputs.

| | | |
|---|---|---|
| Namespaces: | 1 | () |
| Niladic: | 1 | () |
| 1st Order (Ambi): | 2 | (M D) |
| 2nd Order Monadic: | 4 | (AM AD FM FD) |
| 2nd Order Dyadic: | 8 | (AAM AAD AFM AFD FAM FAD FFM FFD) |

Secret Sauce

Replication Count

```
rc←1 1 2 4 8[k[i]]@(i←⍳t=F)⊢(≢p)⍴1
```

Secret Sauce

Encoding the recursion

```
rc←1 1 2 4 8[k[i]]@(i←⍳t=F)⊢(≢p)⍴1
_←{r[⍵]⊣x×←rc[⍵]}⍣≡r⊣x←rc
```

Secret Sauce

Compute group edges and ids (offset)

```
rc←1 1 2 4 8[k[i]]@(i←⍸t=F)⊢(≢p)⍴1
_←{r[⍵]⊣x×←rc[⍵]}⍣≡r⊣x←rc
j←(+⌿x)-x ⋄ ro←∈⍳¨x
```

Secret Sauce

Do the replication

```
rc←1 1 2 4 8[k[i]]@(i←⍸t=F)⊢(≢p)⍴1
_←{r[⍵]⊣x×←rc[⍵]}⍣≡r⊣x←rc
j←(+⌿x)-x ⋄ ro←∈⍳¨x
p t k n r lx vb rc pos end⌿⍨←⊂x
```

Secret Sauce

Update the tree edges

```
rc←1 1 2 4 8[k[i]]@(i←⍸t=F)⊢(≢p)⍴1
_←{r[⍵]⊣x×←rc[⍵]}⍣≡r⊣x←rc
j←(+⍀x)-x ⋄ ro←∊⍳¨x
p t k n r lx vb rc pos end⌿⍨←⊂x
p r{j[⍺]+⍵}←⊂⌊ro÷rc
```

Secret Sauce

Update lexical definition references

```
rc←1 1 2 4 8[k[i]]@(i←⍳t=F)⊢(≢p)⍴1
_←{r[⍵]⊣x×←rc[⍵]}⍣≡r⊣x←rc
j←(+⌿x)-x ⋄ ro←∈⍳¨x
p t k n r lx vb rc pos end⌿⍨←⊂x
p r{j[⍺]+⍵}←⊂⌊ro÷rc
vb[i]←j[vb[i]]+⌊ro[i]÷(x⌿x)[i]÷x[vb[i←⍳vb>0]]
```

Secret Sauce

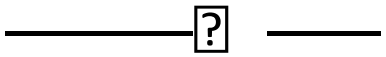
Tag functions with the appropriate specialization kind

```
rc←1 1 2 4 8[k[i]]@(i←⍳⍨t=F)⊢(≢p)⍴1
_←{r[⍵]⊣x×←rc[⍵]}⍣≡r⊣x←rc
j←(+⌿x)-x ⋄ ro←∊⍳¨x
p t k n r lx vb rc pos end⌿⍨←⊂x
p r{j[⍺]+⍵}←⊂⌊ro÷rc
vb[i]←j[vb[i]]+⌊ro[i]÷(x⌿x)[i]÷x[vb[i←⍳⍨vb>0]]
k[i]←0 1 2 4 8[k[i]](⊣+|)ro[i←⍳⍨t=F]
```

# Secret Sauce

⍝ Specialize functions to specific formal binding types

```
rc←1 1 2 4 8[k[i]]@(i←⍸t=F)⊢(≢p)⍴1
_←{r[⍵]⊣x×←rc[⍵]}⍣★≡r⊣x←rc
j←(+⌿x)-x ⋄ ro←∈⍳¨x
p t k n r lx vb rc pos end⌿⍨←⊂x
p r{j[⍺]+⍵}←⊂⌊ro÷rc
vb[i]←j[vb[i]]+⌊ro[i]÷(x⌿x)[i]÷x[vb[i←⍸vb>0]]
k[i]←0 1 2 4 8[k[i]](⊣+|)ro[i←⍸t=F]
```



# Secret Sauce

APL's economy and concision are keys,
they are not incident elements of the value proposition.

This unified economic simplicity and tersity of form
are the first things that people remove from the language
when they try to "improve" the language or borrow from it.

When you make the code small,
many things suddenly get easier.

## Secret Sauce

## Iverson’s Principles of Good Notation

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

**Thank you. Questions?**