Designing Your Data:
The bread and butter of APL

Aaron W. Hsu aaron@dyalog.com, Dyalog, Ltd.

Functional Conf 2025, Inside the Matrix

Background

APL

APL

The language with all those funny symbols.

Background

APL

```
(2=+⌿0=X∘.|X)⌿X←1+⍳N                  ⍝ Prime Numbers up to N
⊃1 ⍵∨.∧3 4=+⌿,¯1 0 1∘.⌽¯1 0 1⊖¨⊂⍵      ⍝ Game of Life
0⌈(,[2+⍳3]{⍵}⌺3 3⊢⍵)+.×,[⍳3]⍺          ⍝ RelU, 3×3 Convolution
```

Background

APL is:

APL is:

A rich, economical vocabulary over arrays...

APL is:

A rich, economical vocabulary over arrays...

with a **killer** syntax.

Challenge

Beginners focus on the symbols (the “verbs”).

Challenge

Beginners focus on the symbols (the “verbs”).

They get stuck when the problem “doesn’t fit” arrays.

Challenge

Experts focus on the data.

Challenge

Experts focus on the data.

Encode the data for simplicity and efficiency
using the Array Model.

Challenge

How do they do it?

What are some tactics for data encoding in APL?

The Relational Model

The Relational Model

Tuples organized into *Tables* with *Fields* and a *Header*;

The Relational Model

Tuples organized into *Tables* with *Fields* and a *Header*;
a language for manipulating relations.

Relational Variables correspond to named *Tables*.

The Relational Model

Tuples organized into *Tables* with *Fields* and a *Header*;
a language for manipulating relations.

Relational Variables correspond to named *Tables*.

APL is an excellent extended relational algebra.

The Array Model

Foundations

A: Array

Elements: $e_0\ e_1\ \ldots\ e_{n-1} \in A$,A

Shape: $d_0\ \ldots\ d_{k-1} \in \mathbb{N}$ ρA

Foundations

A: Array

Elements: $e_0\ e_1\ \ldots\ e_{n-1} \in A$

Shape: $d_0\ \ldots\ d_{k-1} \in \mathbb{N}$

$,A$	n	$\longleftrightarrow$	$\neq,A$ Count
ρA	k	$\longleftrightarrow$	$\neq\rho A$ Rank
	d_0	$\longleftrightarrow$	$\neq A$ Tally

Foundations

A: Array

Elements: $e_0\ e_1\ \ldots\ e_{n-1} \in A$

Shape: $d_0\ \ldots\ d_{k-1} \in \mathbb{N}$

$\equiv Y$ Nesting level of A

$,A$	n	$\longleftrightarrow$	$\neq,A$ Count
ρA	k	$\longleftrightarrow$	$\neq\rho A$ Rank
	d_0	$\longleftrightarrow$	$\neq A$ Tally

Foundations

A: Array

Elements: $e_0\ e_1\ \ldots\ e_{n-1} \in A$
Shape: $d_0\ \ldots\ d_{k-1} \in \mathbb{N}$

$\equiv Y$ Nesting level of A

,A	n	⟷ ≢,A	Count
ρA	k	⟷ ≢ρA	Rank
	d_0	⟷ ≢A	Tally

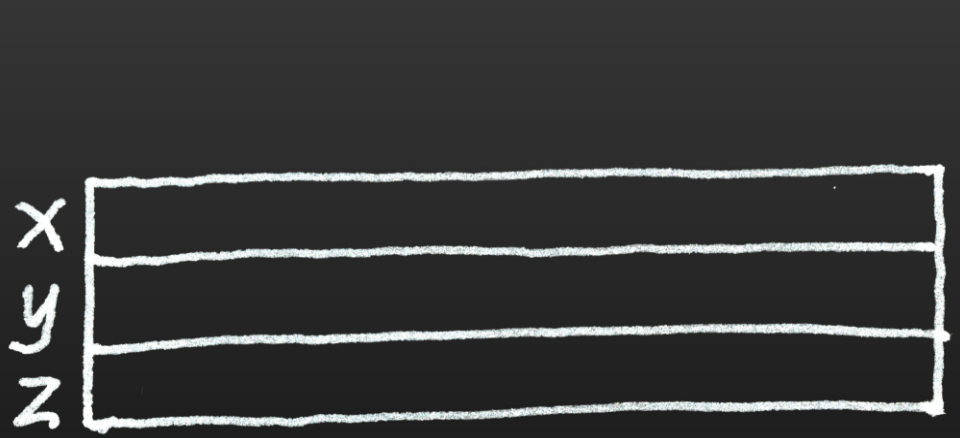
```
struct array {
      int rank;
      int shape[rank];
      struct array elements[n];
};
```

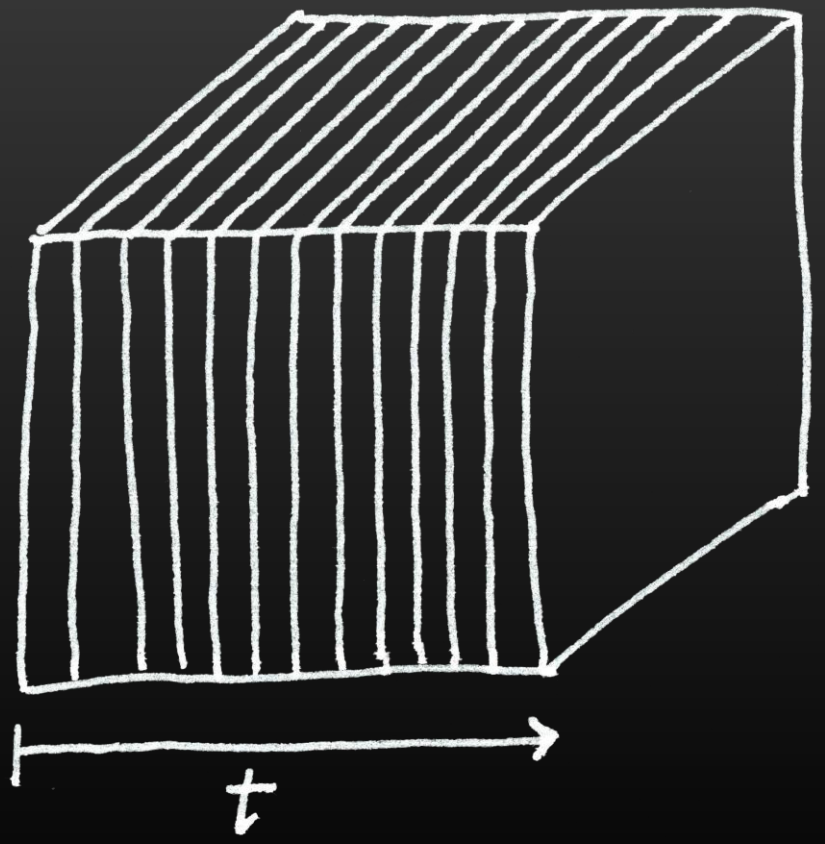
Choice #1: Leverage inherent dimensionality

Choice #1: Leverage inherent dimensionality

Slicing

Choice #1: Leverage inherent dimensionality



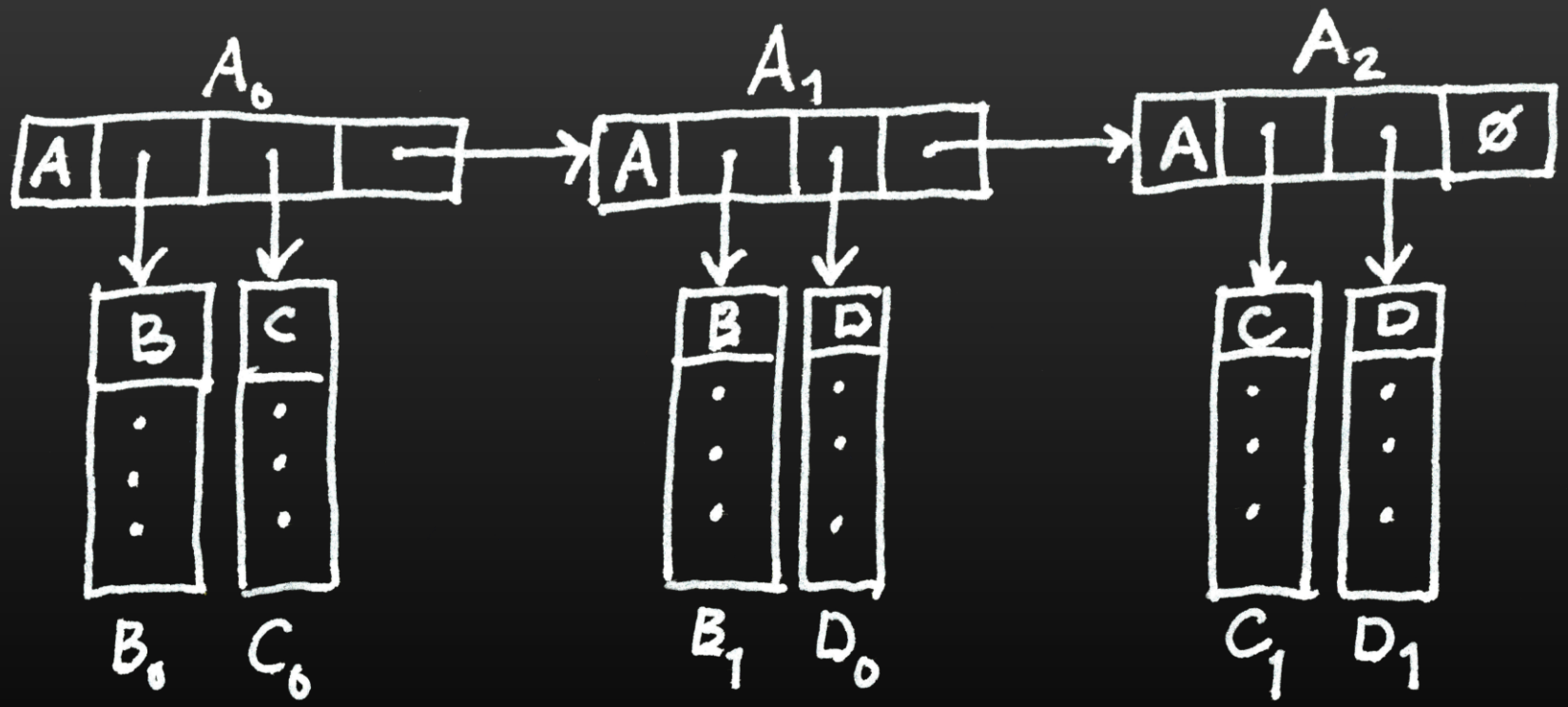


Aggregation

Aggregate objects instead of reified object references.

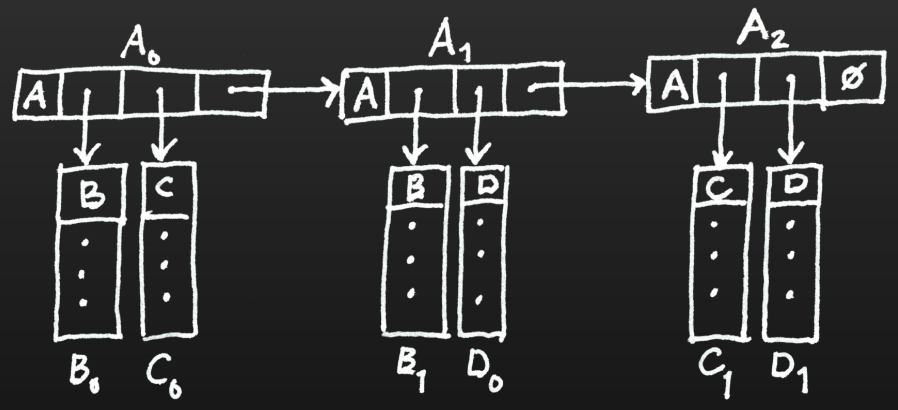
Aggregation

Aggregate objects instead of reified object references.



Aggregation

Aggregate objects instead of reified object references.



	A		
A_0	B_0	C_0	A_1
A_1	B_1	D_0	A_2
A_2	C_1	D_1	ø

B	
B_0	. . .
B_1	. . .

C	
C_0	. . .
C_1	. . .

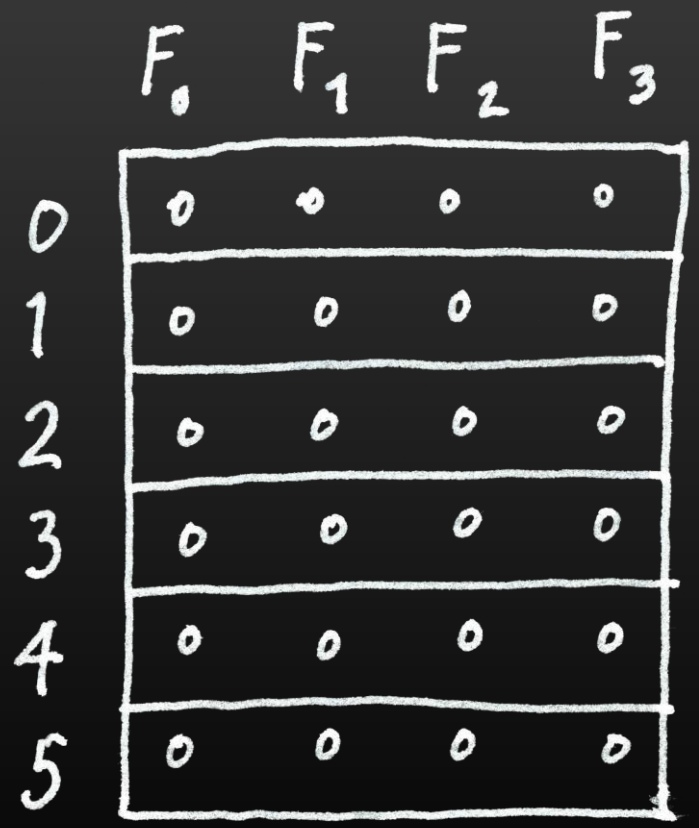
D	
D_0	. . .
D_1	. . .

Inverted Tables

Use inverted tables to represent relations, complex objects.

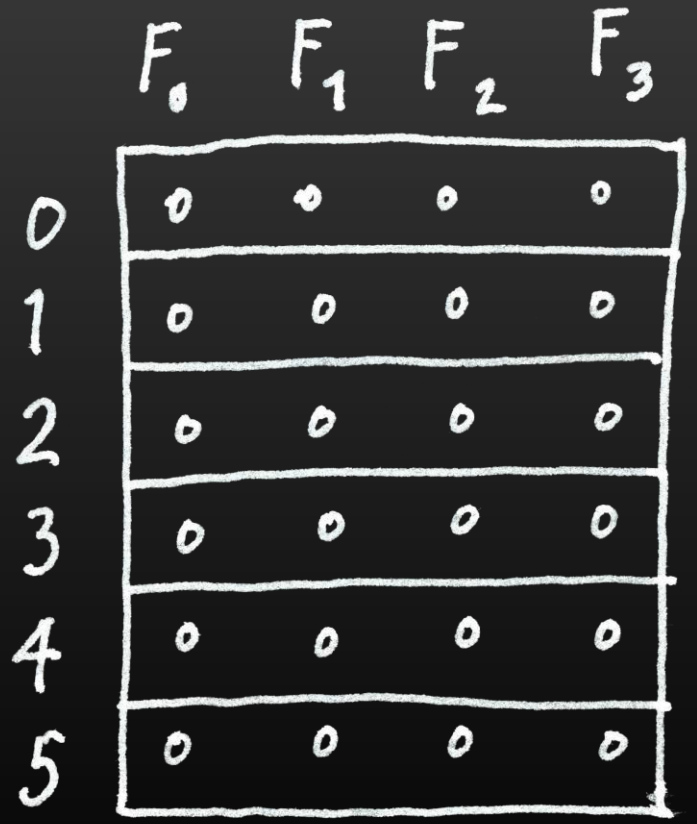
Inverted Tables

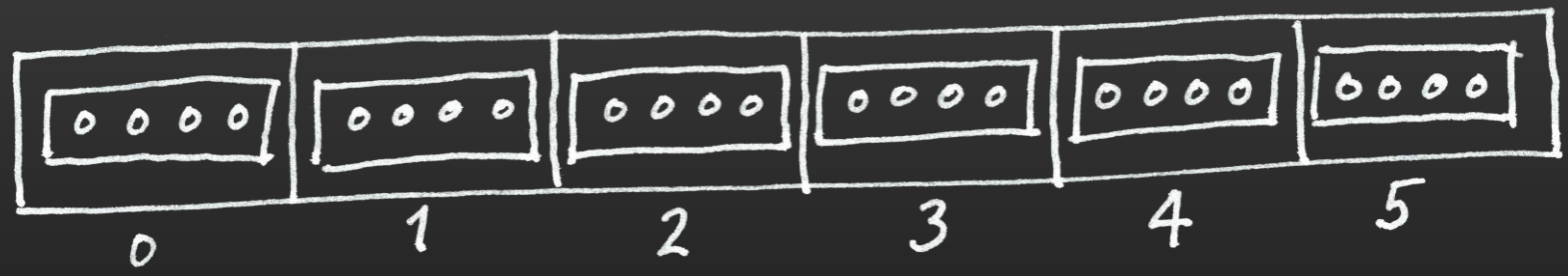
Use inverted tables to represent relations, complex objects.



Inverted Tables

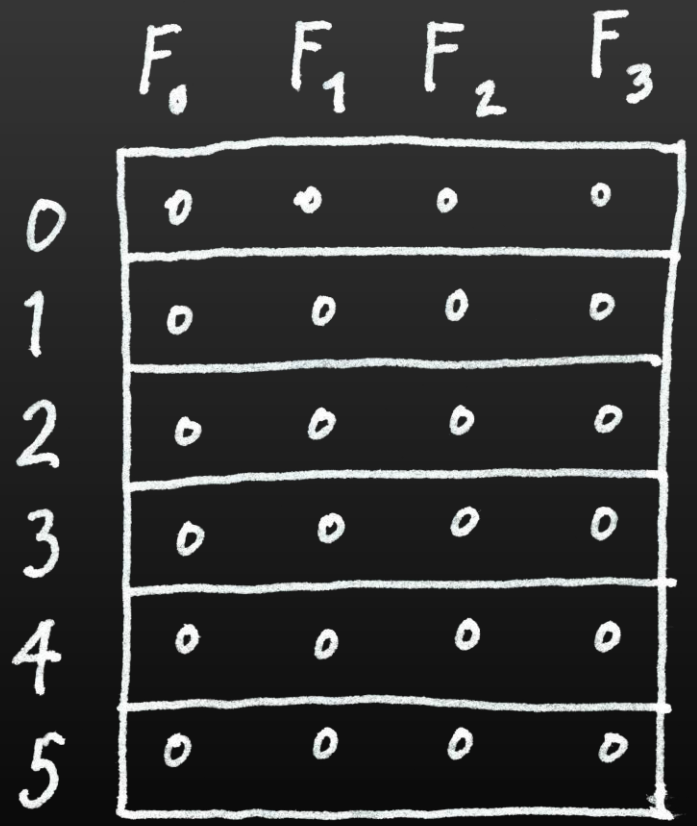
Use inverted tables to represent relations, complex objects.

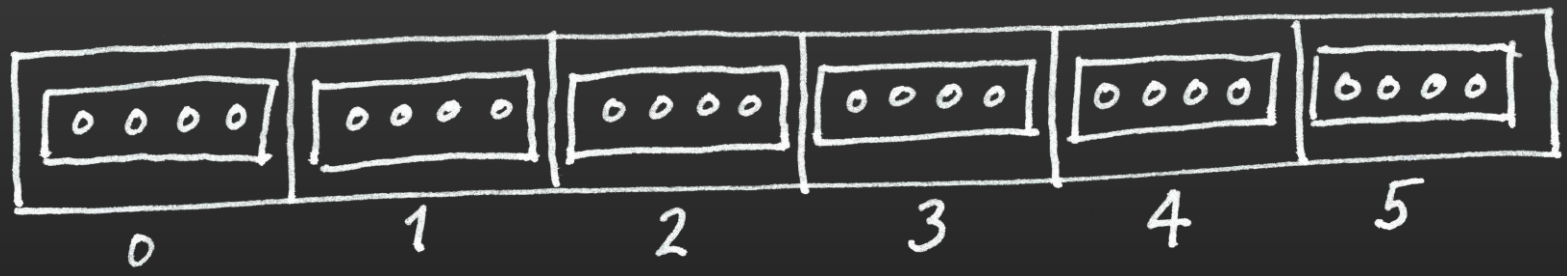


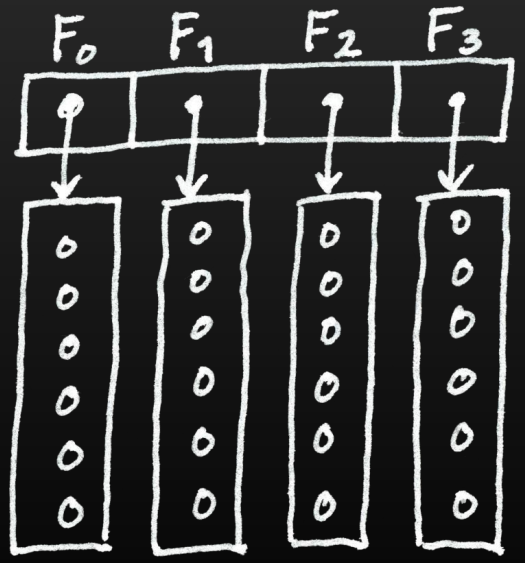


Inverted Tables

Use inverted tables to represent relations, complex objects.

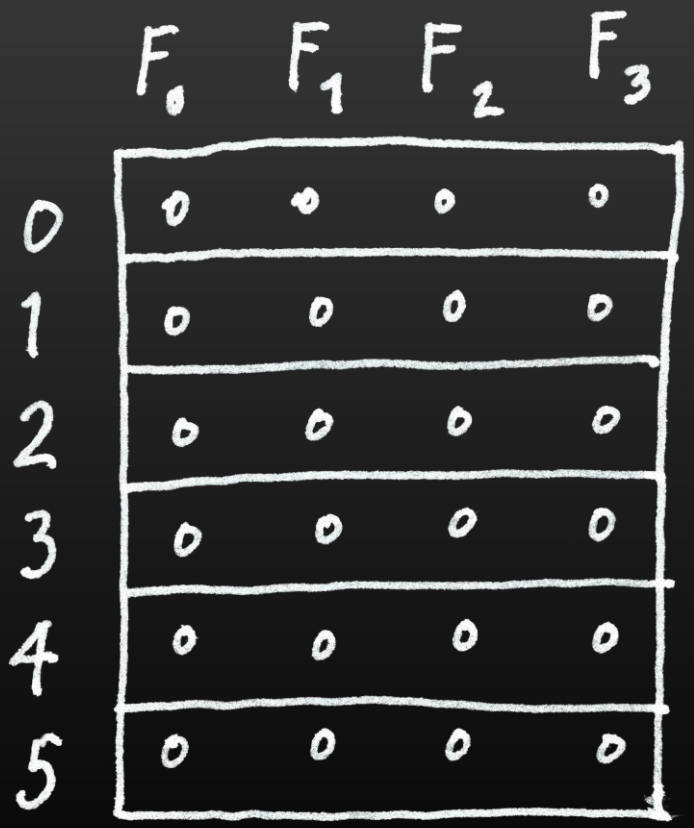


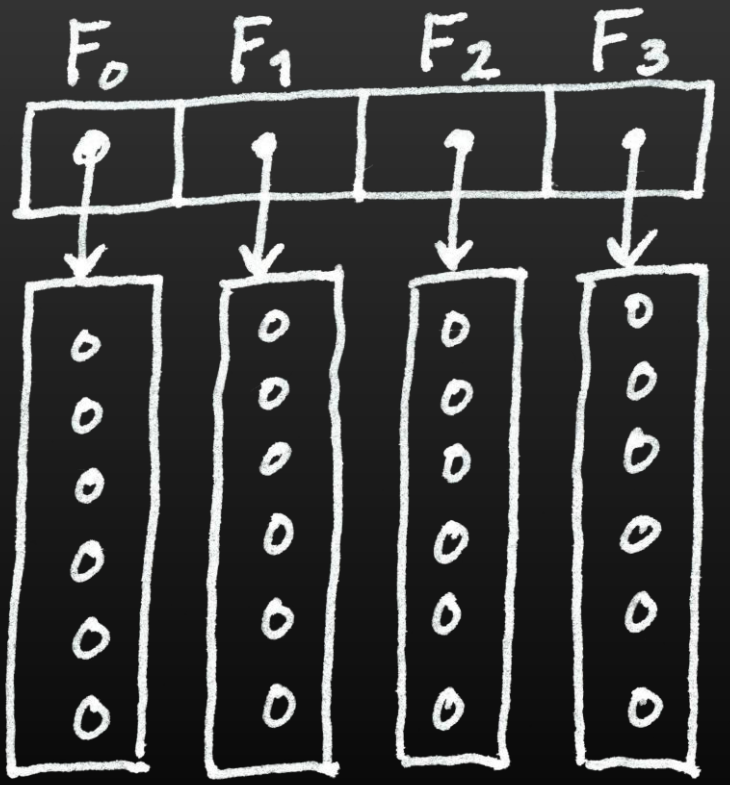




Inverted Tables

Use inverted tables to represent relations, complex objects.



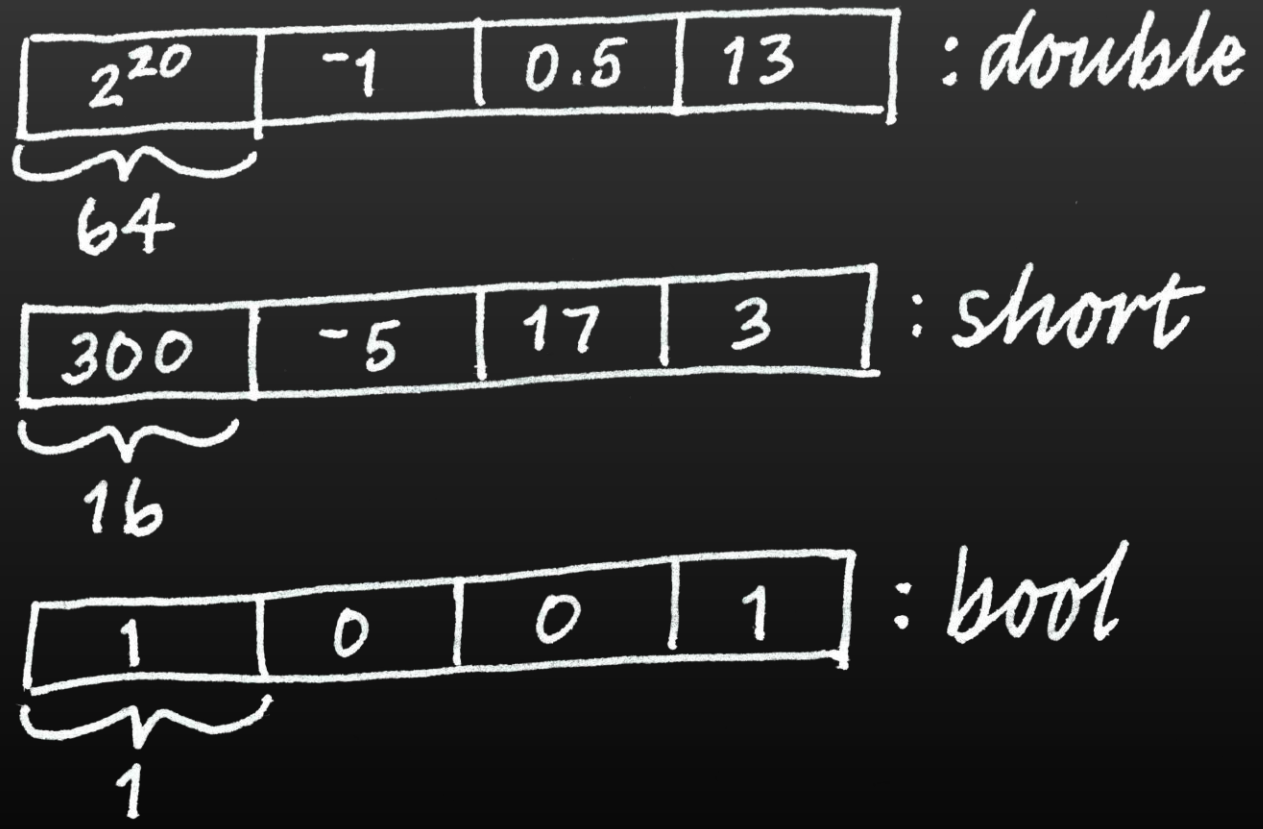


Implicit Structures

Leverage implicit data structures, explicit arrays.

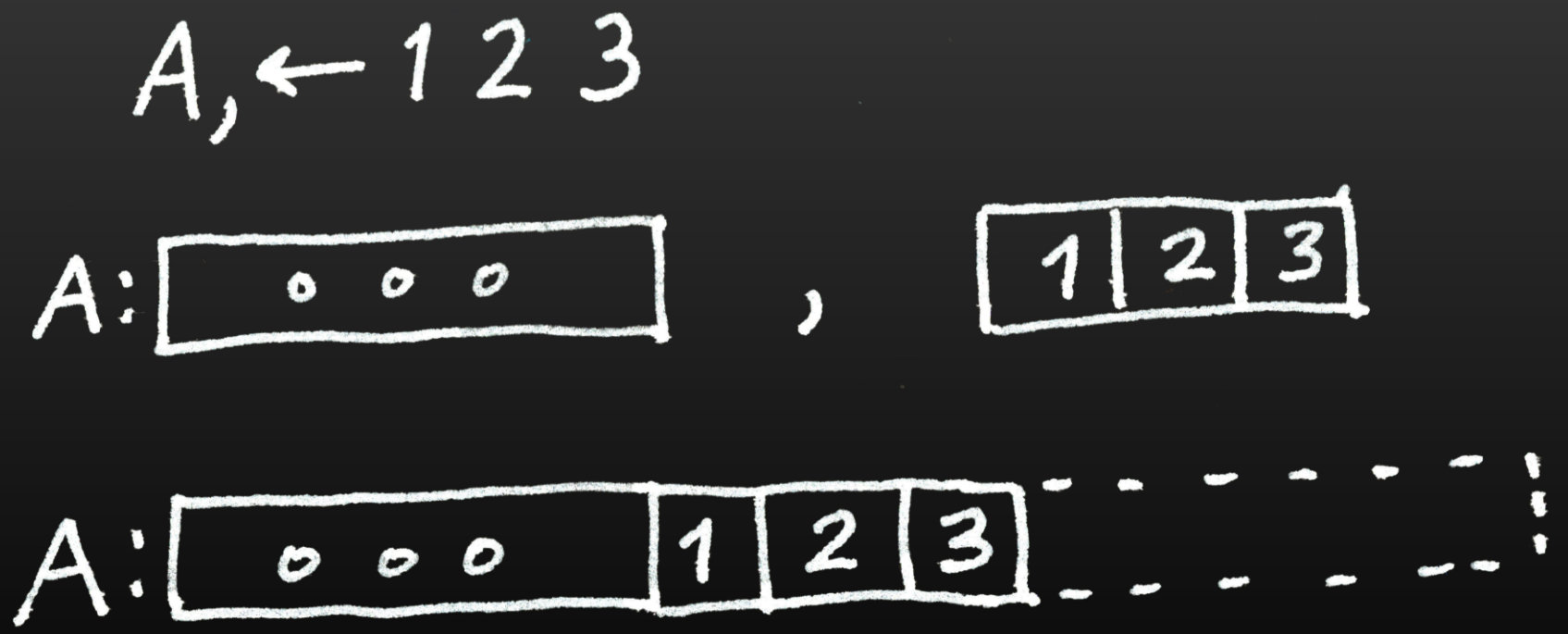
Implicit Structures

Leverage implicit data structures, explicit arrays.



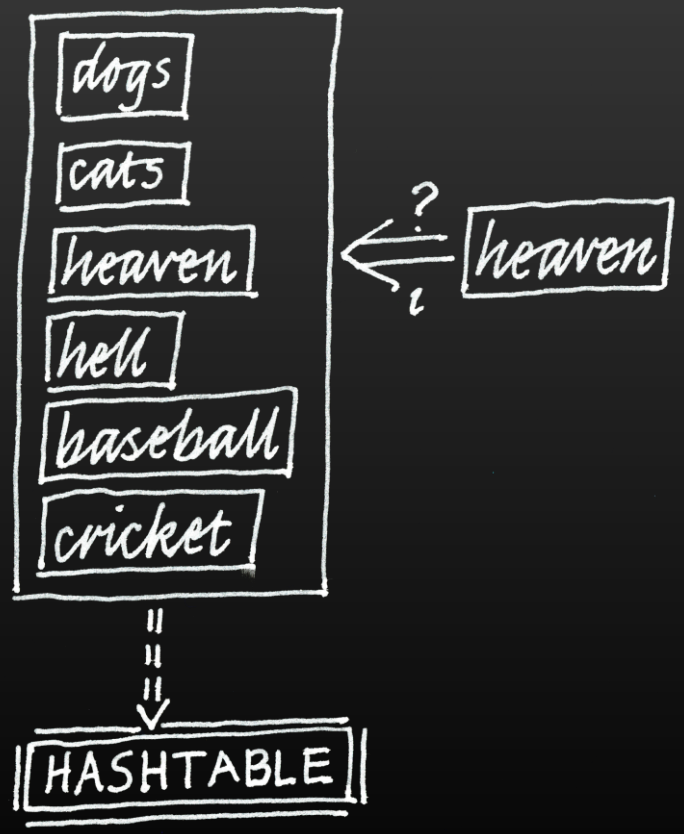
Implicit Structures

Leverage implicit data structures, explicit arrays.



Implicit Structures

Leverage implicit data structures, explicit arrays.

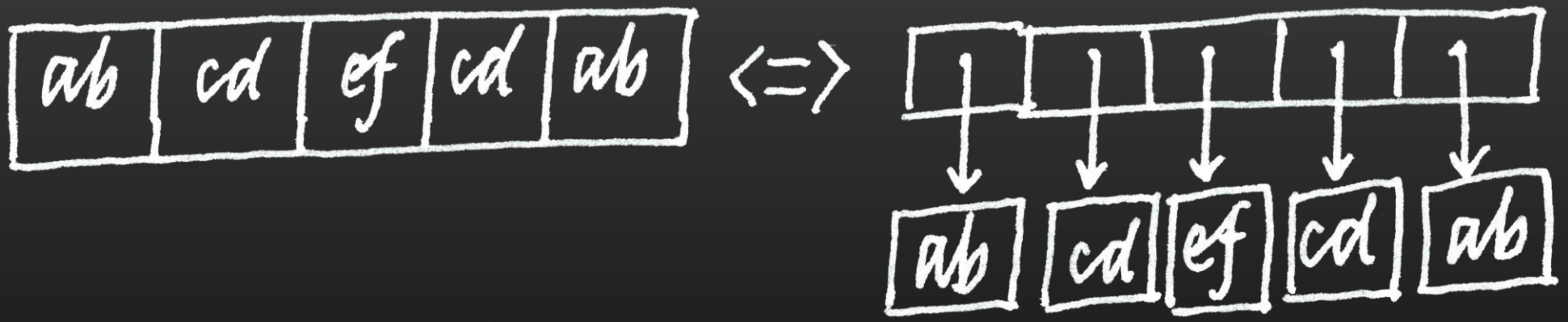


Symbol tables

Explicitly intern data using symbol tables.

Symbol tables

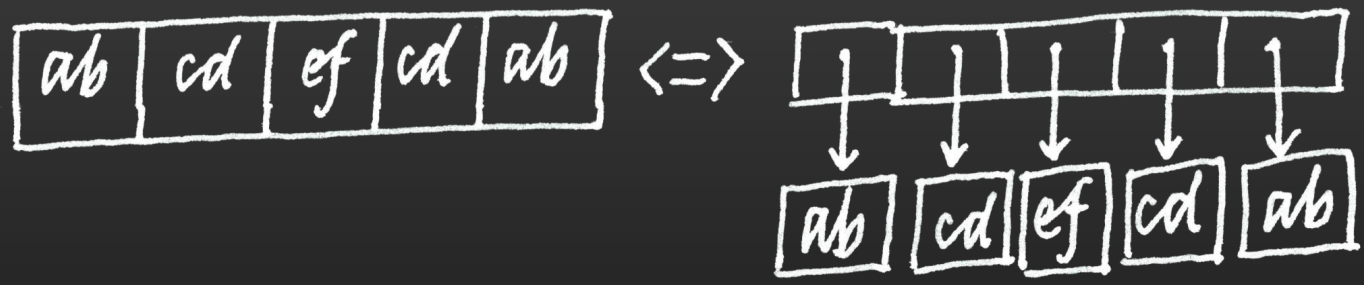
Explicitly intern data using symbol tables.



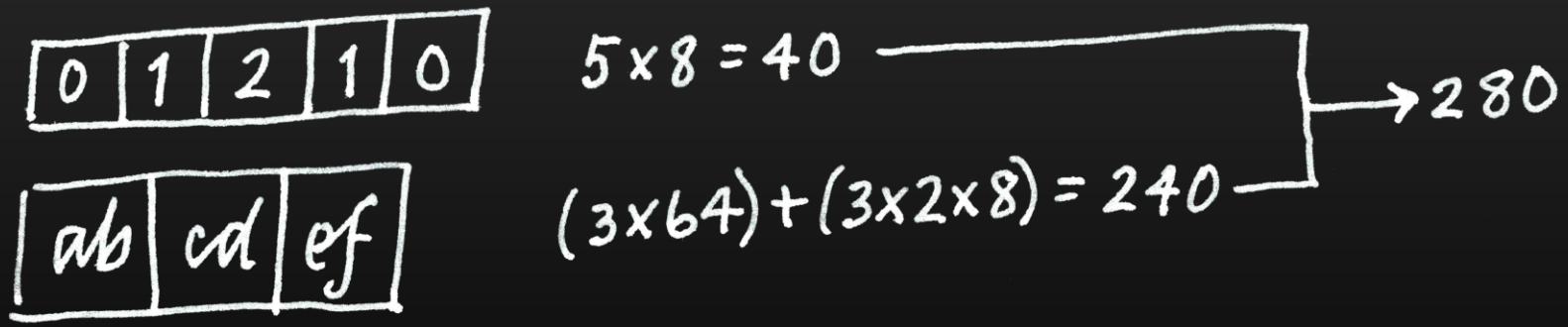
$(5 \times 64) + (5 \times 2 \times 8) \leftrightarrow 400$

Symbol tables

Explicitly intern data using symbol tables.



$(5 \times 64) + (5 \times 2 \times 8) \leftrightarrow 400$



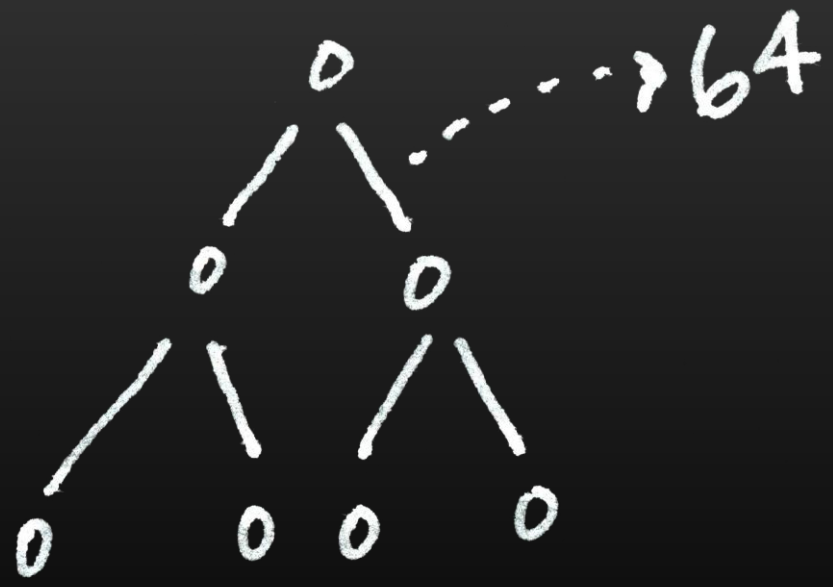
Symbol consumes 8-bits, not 64!

Pointers

Avoid generalized pointers,
use type-constrained explicit pointers with restricted range.

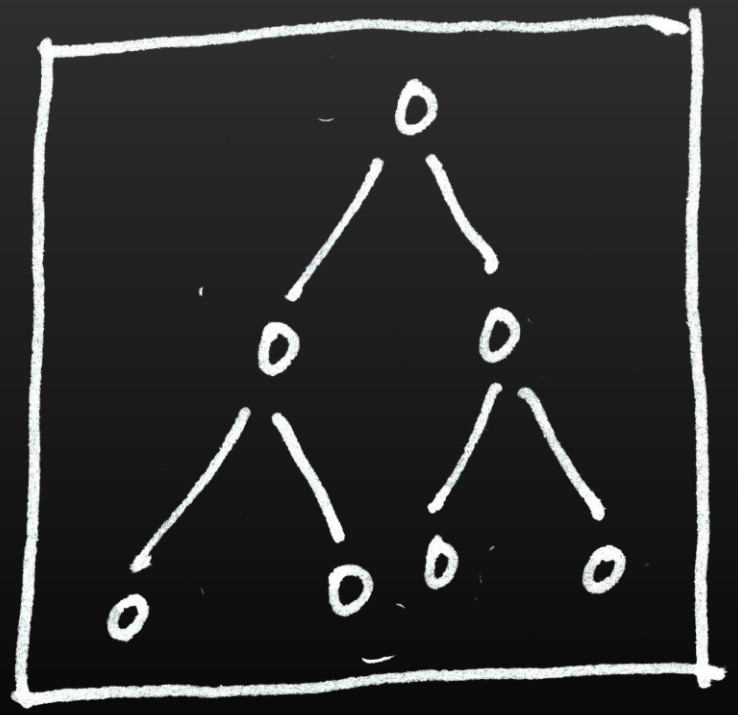
Pointers

Avoid generalized pointers,
use type-constrained explicit pointers with restricted range.



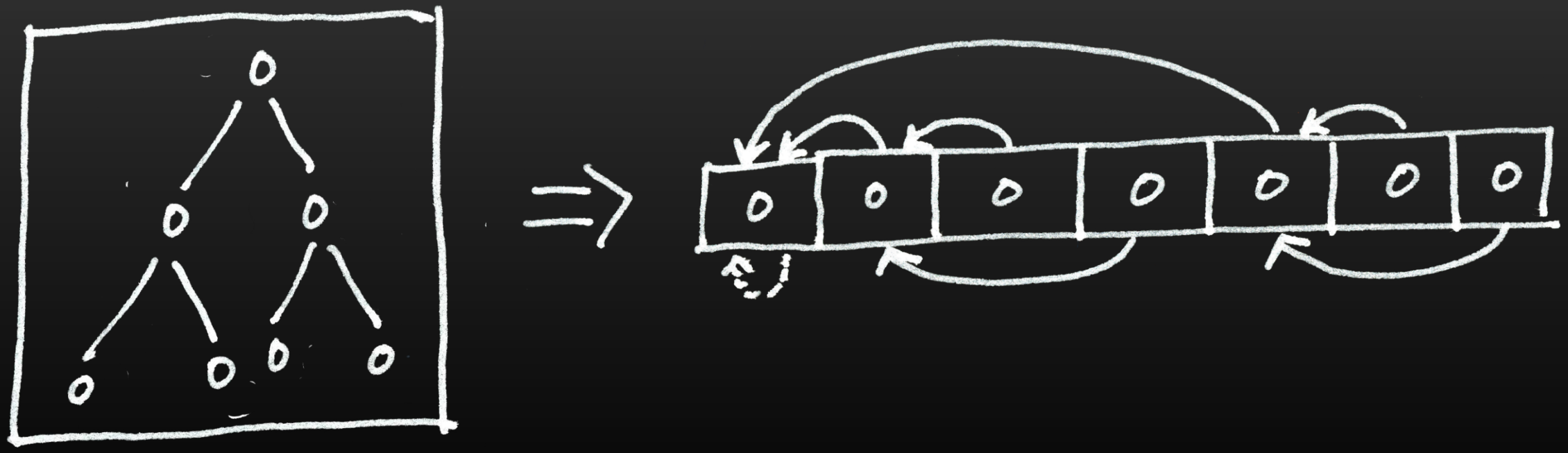
Pointers

Avoid generalized pointers,
use type-constrained explicit pointers with restricted range.



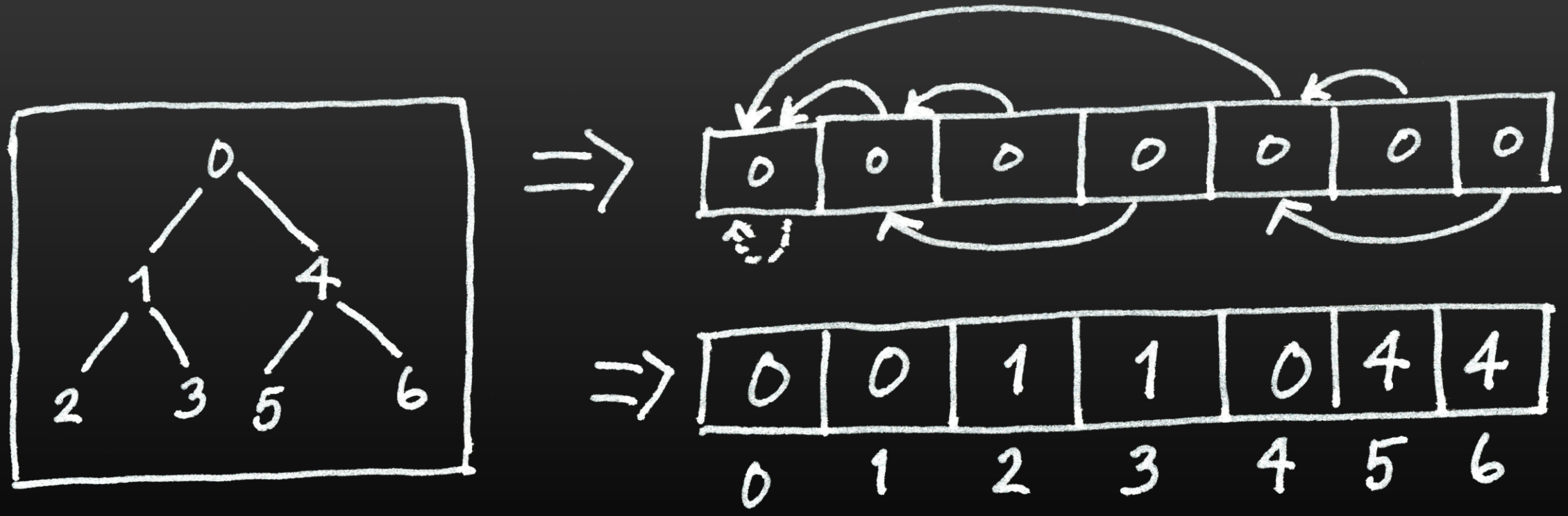
Pointers

Avoid generalized pointers,
use type-constrained explicit pointers with restricted range.



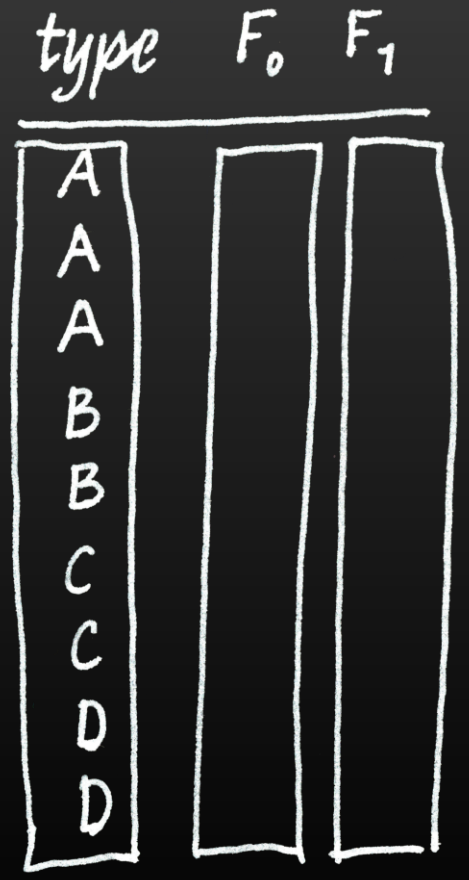
Pointers

Avoid generalized pointers,
use type-constrained explicit pointers with restricted range.



Think aggregately for tags, types, and classes.

Think aggregately for tags, types, and classes.



$type = A$

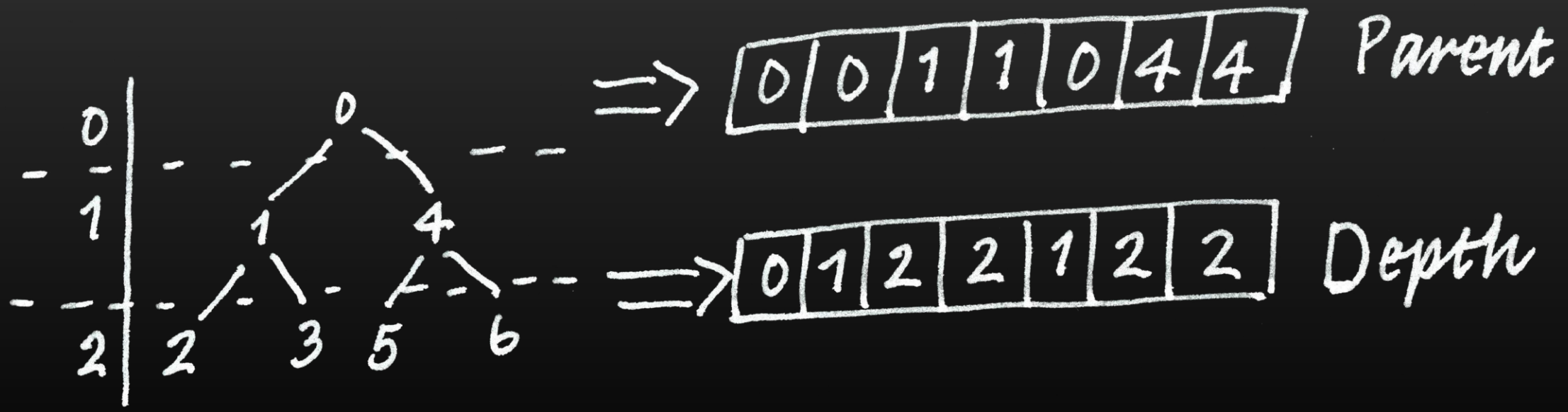
$(type = A) \vee (type = B)$

$type \in A\ B$

$\{\alpha (\neq \omega)\} \ni type$

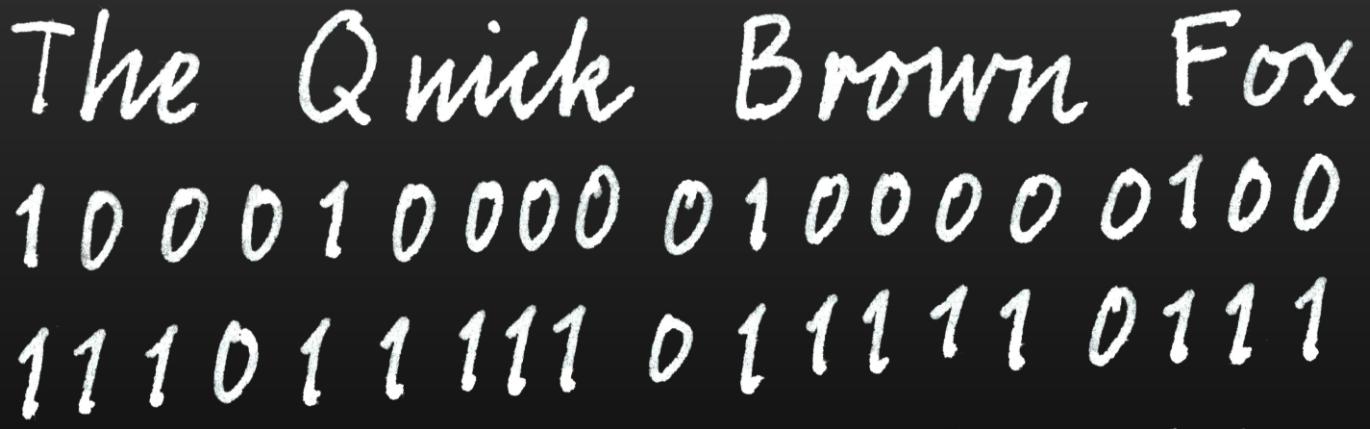
You aren’t restricted to a single representation,
don’t be afraid to switch representations to taste.

You aren’t restricted to a single representation,
don’t be afraid to switch representations to taste.



Take advantage of Bitvector masks.

Take advantage of Bitvector masks.



Boolean Masks

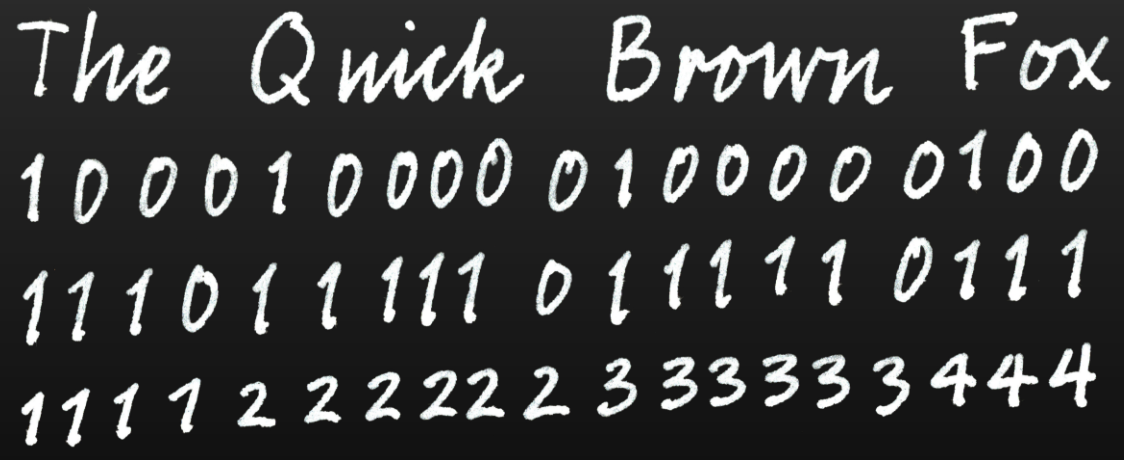
Take advantage of Bitvector masks.

```
(A×M)+(B×~M) ⍝ Select A if M, B otherwise
M⍀f(M⌿A)     ⍝ Masked A modified by f
```

Combine Enums, Views, and Masks
by using “Keys” either explicitly or implicitly.

Keys

Combine Enums, Views, and Masks
by using “Keys” either explicitly or implicitly.



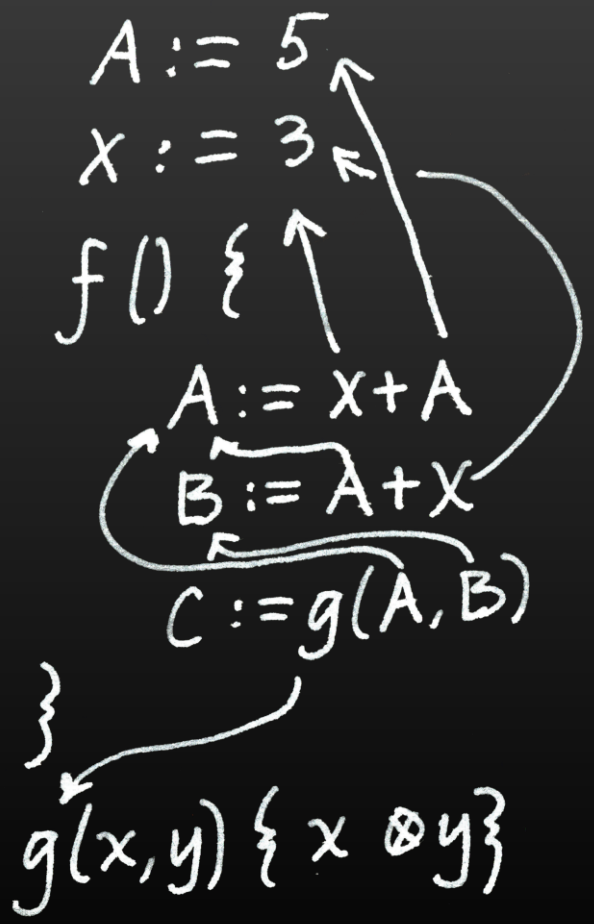
xoFnworBkciuQehT
444333322222111

ehTkciuQnworBxoF
1112222233333444

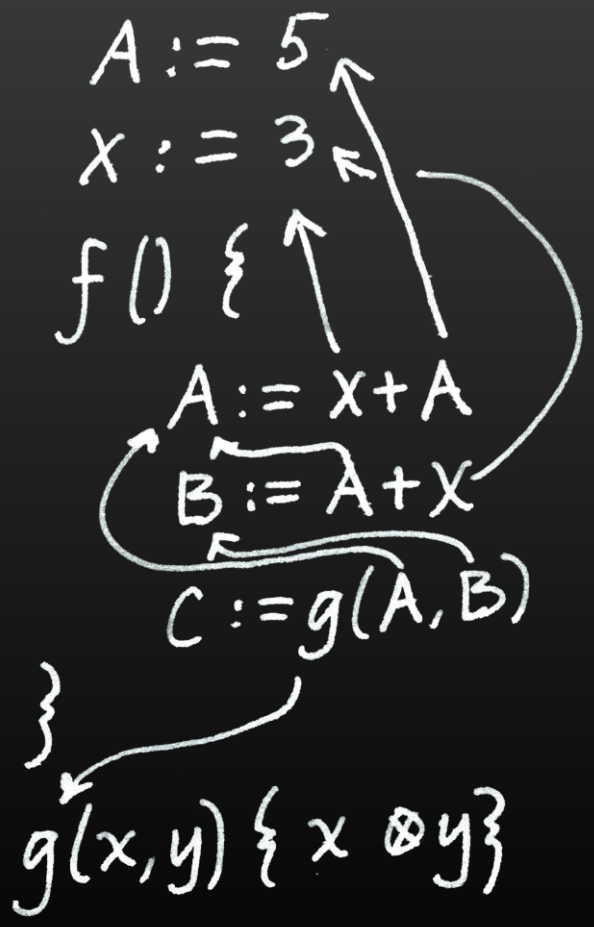
ehT kciuQ nworB xoF
11101111101111110111

Embrace the Total Array Ordering:
Leverage permutations and total array ordering for knowledge embedding.

Embrace the TAO:



Embrace the TAO:



A	X	f	g	A	X	A	X	A	B	B	A	g	C
1	1	1	1	0	0	1	0	0	1	0	0	0	1
0	3	6	17	9	8	7	12	11	10	16	15	14	13

Related Work

Moseley, Ben, and Peter Marks. "Out of the tar pit." Software Practice Advancement (SPA) 2006 (2006). https://blog.royalsloth.eu/archive/outOfTheTarPit.pdf

Lessons

Data hiding is a **myth**, so embrace more control.

Data hiding is a **myth**, so embrace more control.
Data encoding is critical to efficient array programming.

Data hiding is a **myth**, so embrace more control.
Data encoding is critical to efficient array programming.
You *can* regain control over your data without losing convenience.

Data hiding is a **myth**, so embrace more control.
Data encoding is critical to efficient array programming.
You *can* regain control over your data without losing convenience.
Data-driven code should fall out naturally.

Data hiding is a **myth**, so embrace more control.
Data encoding is critical to efficient array programming.
You *can* regain control over your data without losing convenience.
Data-driven code should fall out naturally.
Utilize implicit data structures under an array model.

Data hiding is a **myth**, so embrace more control.
Data encoding is critical to efficient array programming.
You *can* regain control over your data without losing convenience.
Data-driven code should fall out naturally.
Utilize implicit data structures under an array model.
Use low-level thinking with high-level language conveniences.

Data hiding is a **myth**, so embrace more control.
Data encoding is critical to efficient array programming.
You *can* regain control over your data without losing convenience.
Data-driven code should fall out naturally.
Utilize implicit data structures under an array model.
Use low-level thinking with high-level language conveniences.
Stick to global, avoid long-lived intermediate state.

Data hiding is a **myth**, so embrace more control.
Data encoding is critical to efficient array programming.
You *can* regain control over your data without losing convenience.
Data-driven code should fall out naturally.
Utilize implicit data structures under an array model.
Use low-level thinking with high-level language conveniences.
Stick to global, avoid long-lived intermediate state.
Take advantage of functional principles to manage global state.

Thank you. Questions?

aaron@dyalog.com

Data hiding is a **myth**, so embrace more control.
Data encoding is critical to efficient array programming.
You *can* regain control over your data without losing convenience.
Data-driven code should fall out naturally.
Utilize implicit data structures under an array model.
Use low-level thinking with high-level language conveniences.
Stick to global, avoid long-lived intermediate state.
Take advantage of functional principles to manage global state.