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
```

APL is:

APL is:

A rich, economical vocabulary over arrays…

APL is:

A rich, economical vocabulary over arrays…

with a **killer** syntax.

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

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

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

Experts focus on the data.

Experts focus on the data.

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

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

$\mathbb{A}$: Array

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

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

Foundations

$\mathbb{A}$: Array

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

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

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

Foundations

$\mathbb{A}$: Array

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

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

≡ Y Nesting level of A

,A	|	n	⟷	≢,A	Count
⍴A	|	k	⟷	≢⍴A	Rank
		d_0	⟷	≢A	Tally

Foundations

$\bar{A}$: Array

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

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

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

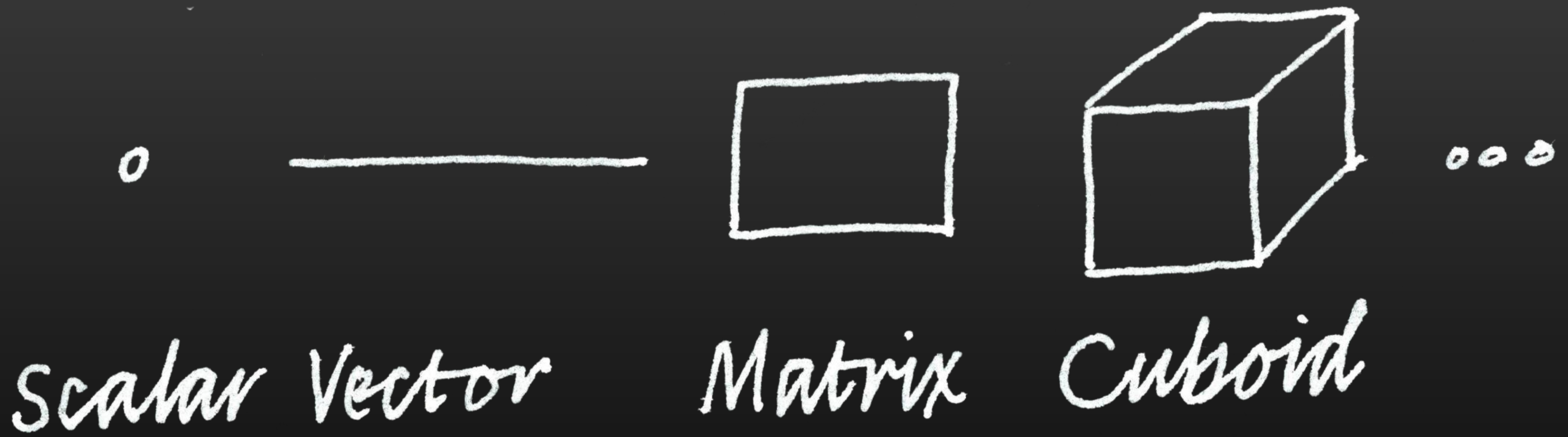
≡ Y Nesting level of A

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

Slicing

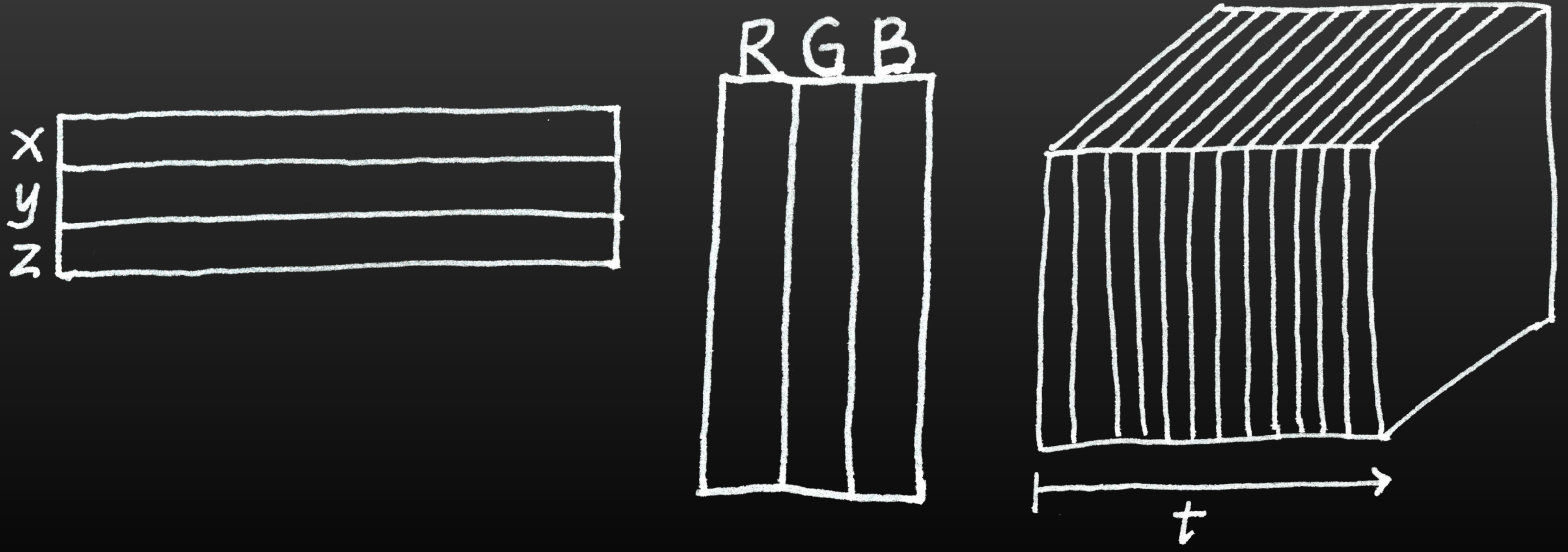
Choice #1: Leverage inherent dimensionality

Choice #1: Leverage inherent dimensionality



Slicing

Choice #1: Leverage inherent dimensionality

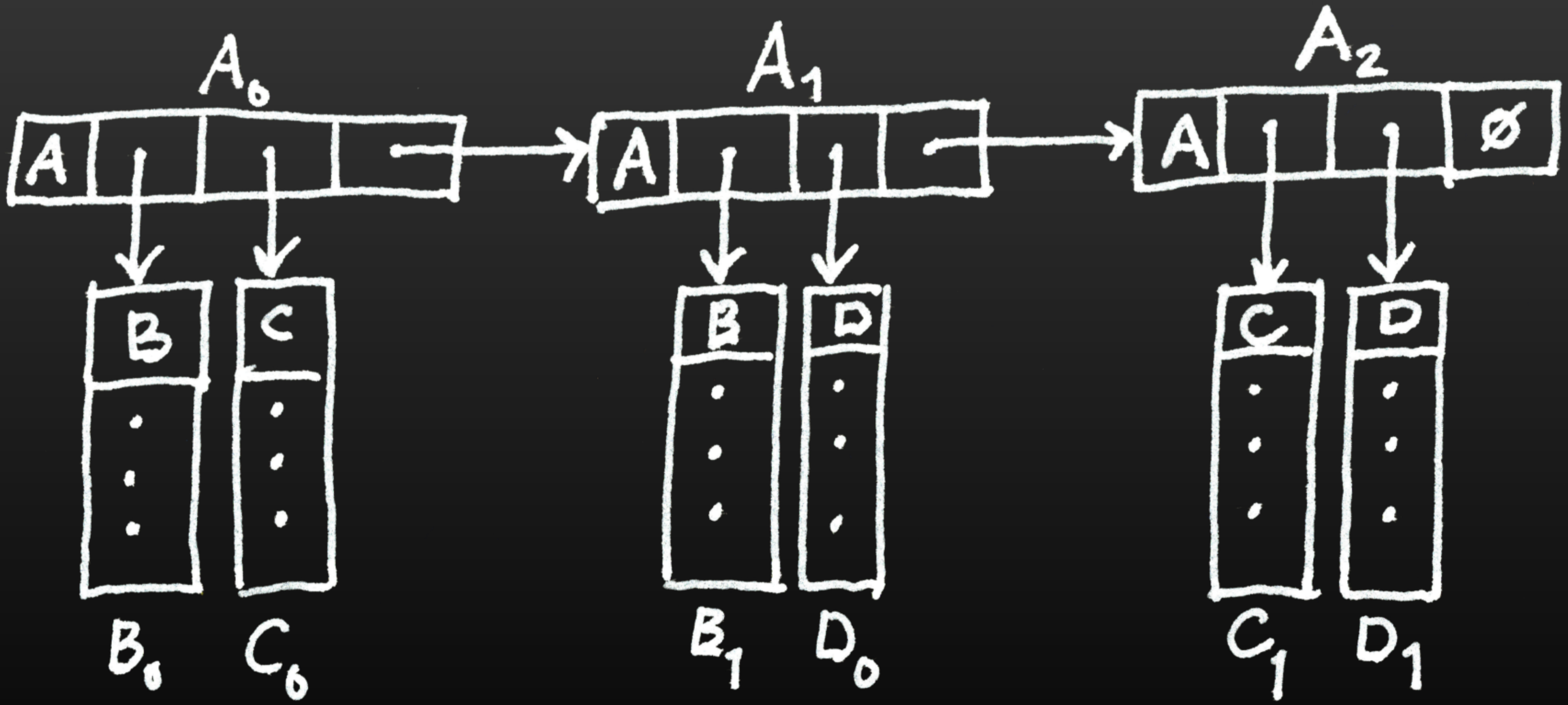


Aggregation

Aggregate objects instead of reified object references.

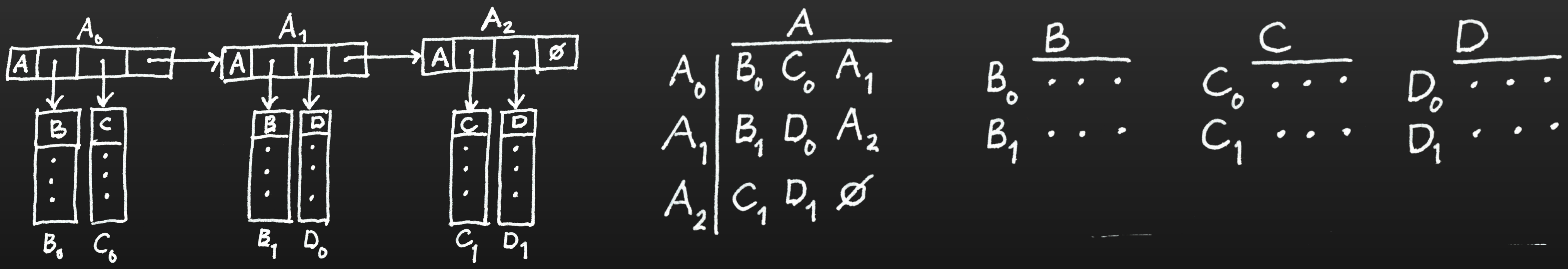
Aggregation

Aggregate objects instead of reified object references.



Aggregation

Aggregate objects instead of reified object references.

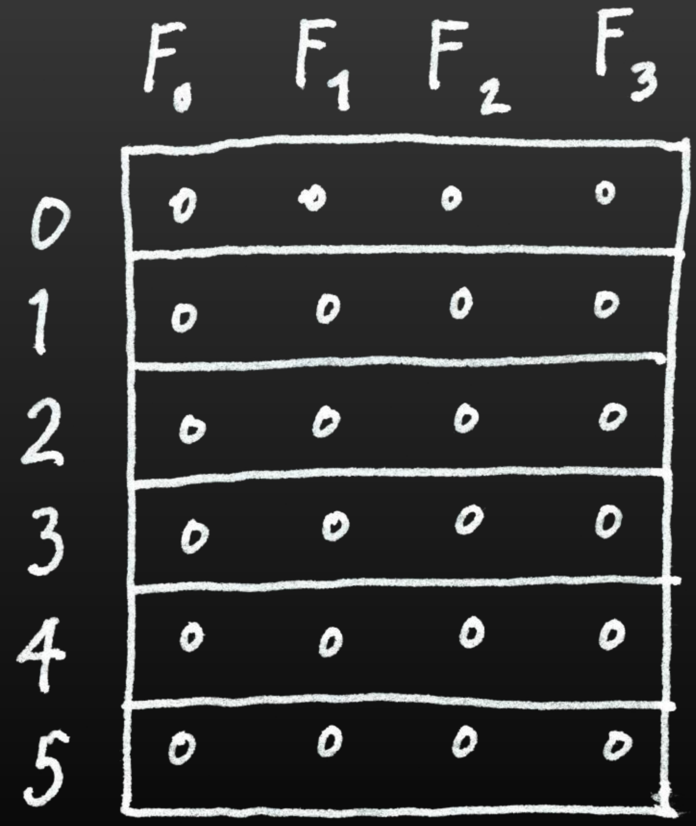


Inverted Tables

Use inverted tables to represent relations, complex objects.

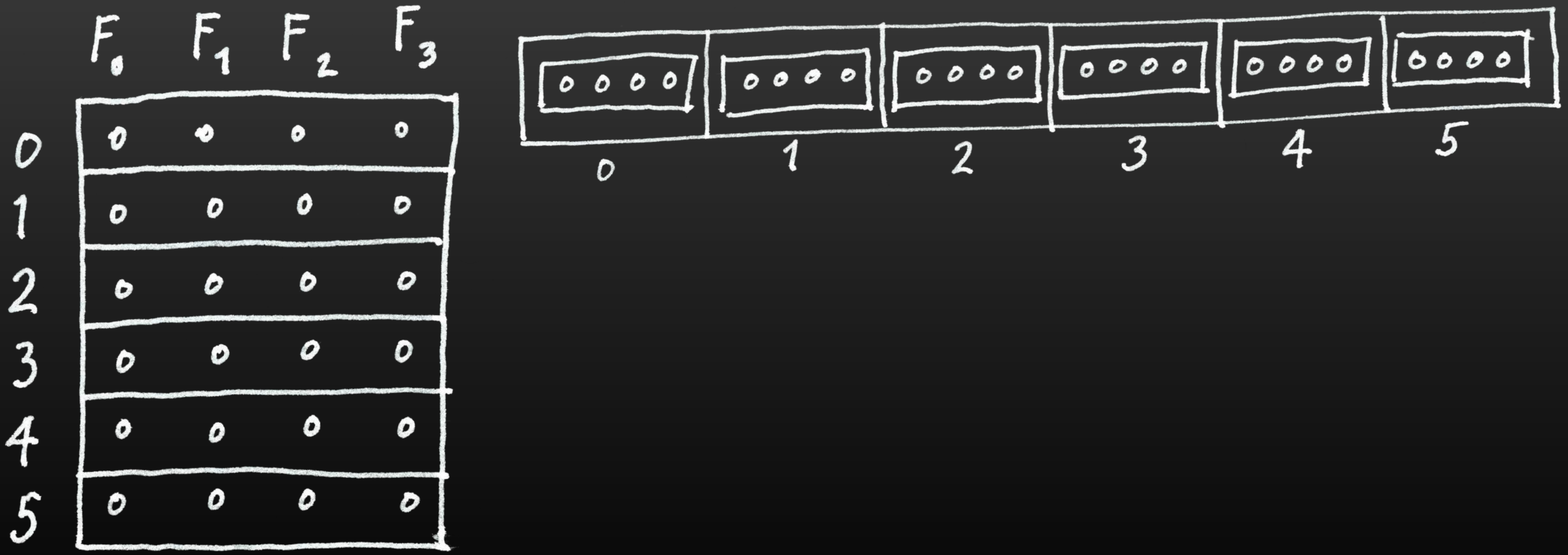
Inverted Tables

Use inverted tables to represent relations, complex objects.



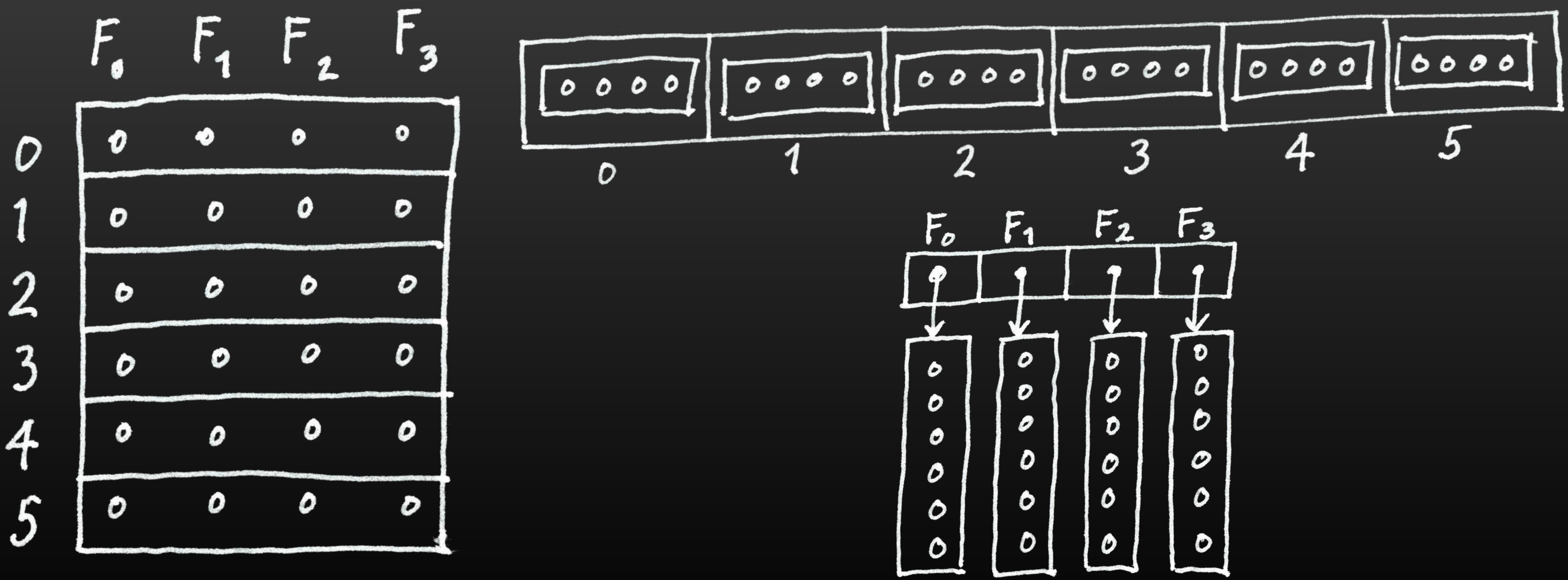
Inverted Tables

Use inverted tables to represent relations, complex objects.

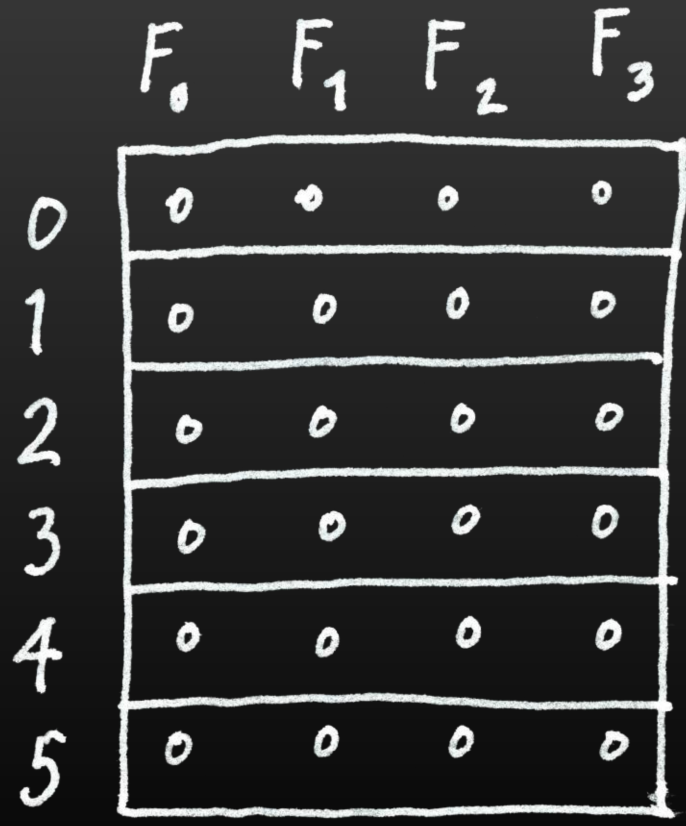


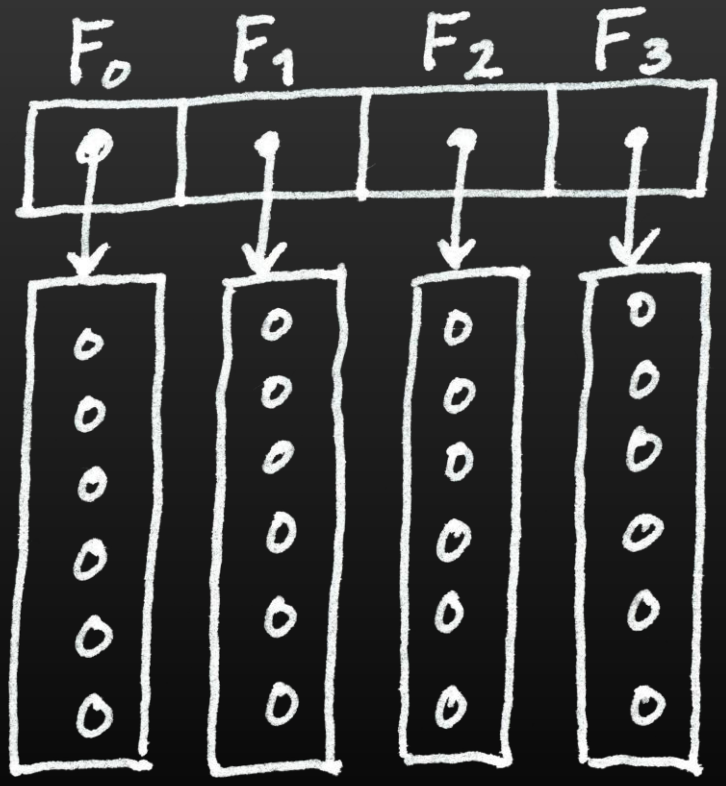
Inverted Tables

Use inverted tables to represent relations, complex objects.



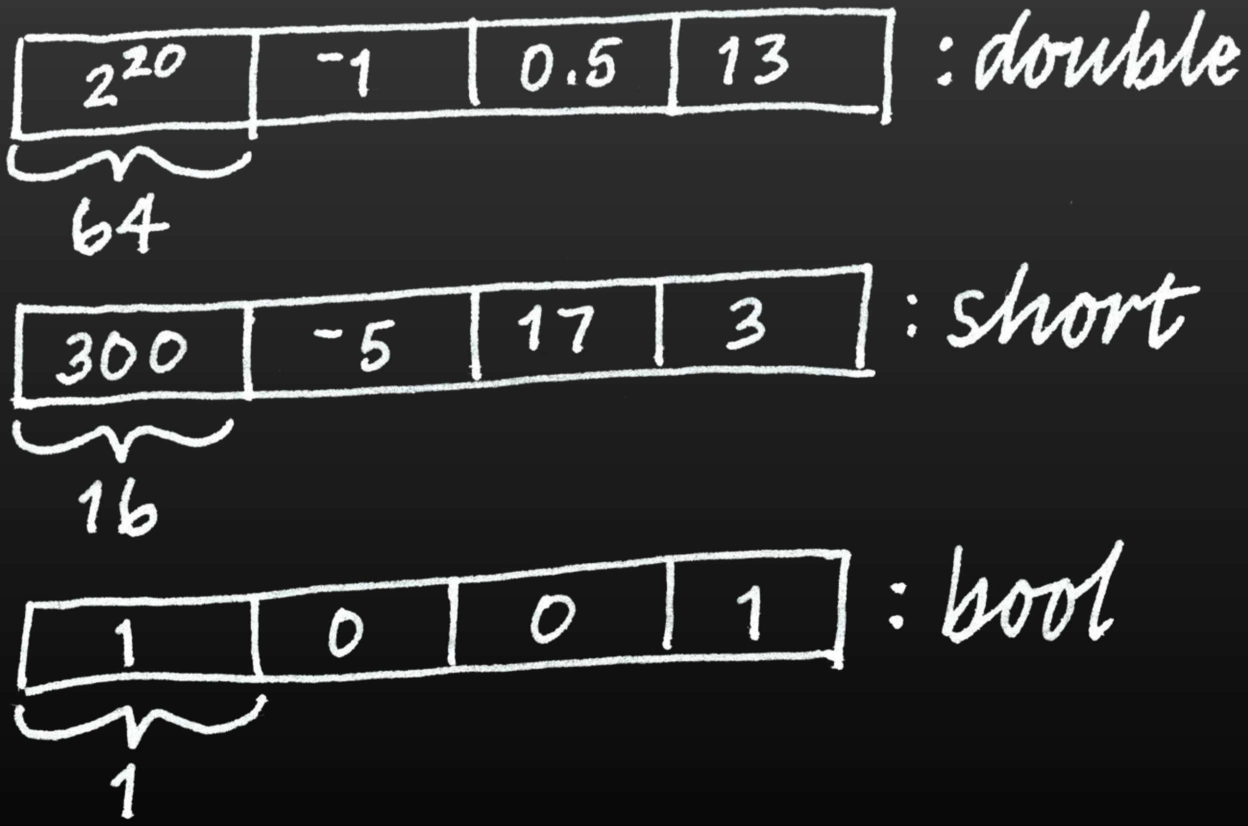
Use inverted tables to represent relations, complex objects.



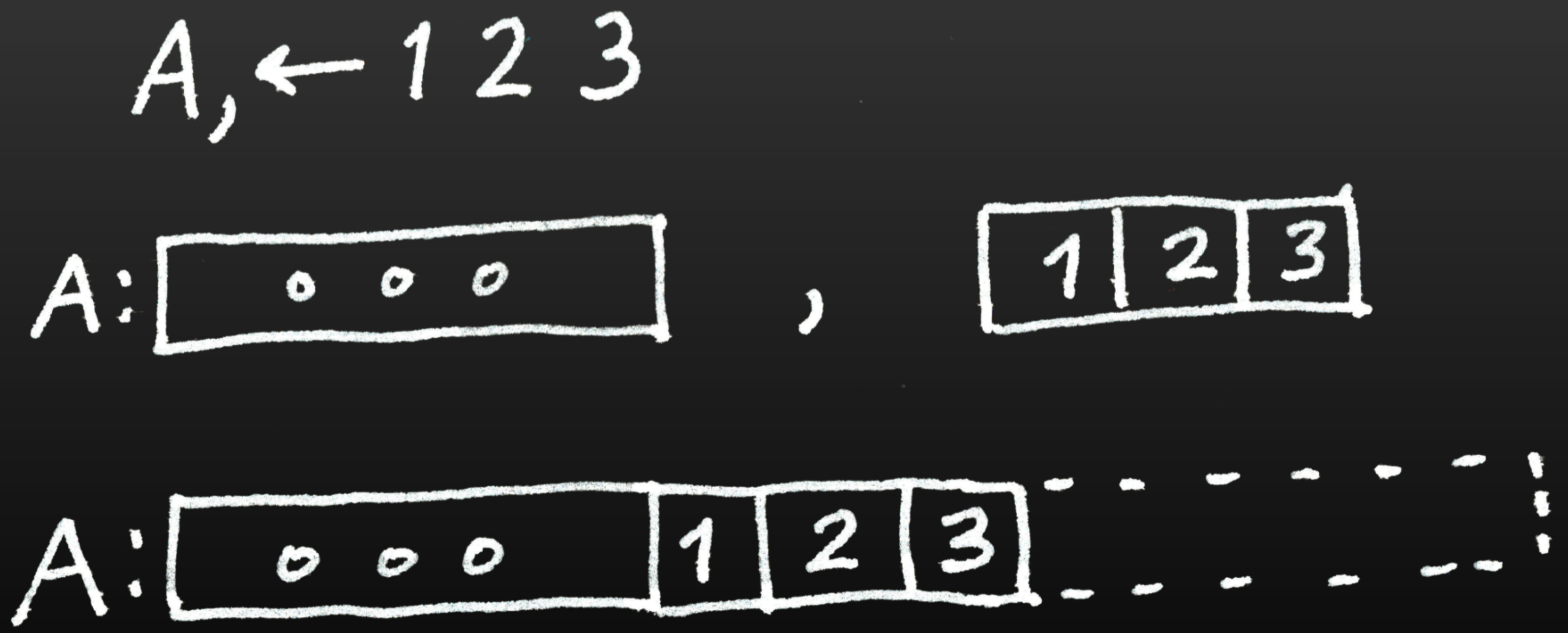


Leverage implicit data structures, explicit arrays.

Leverage implicit data structures, explicit arrays.

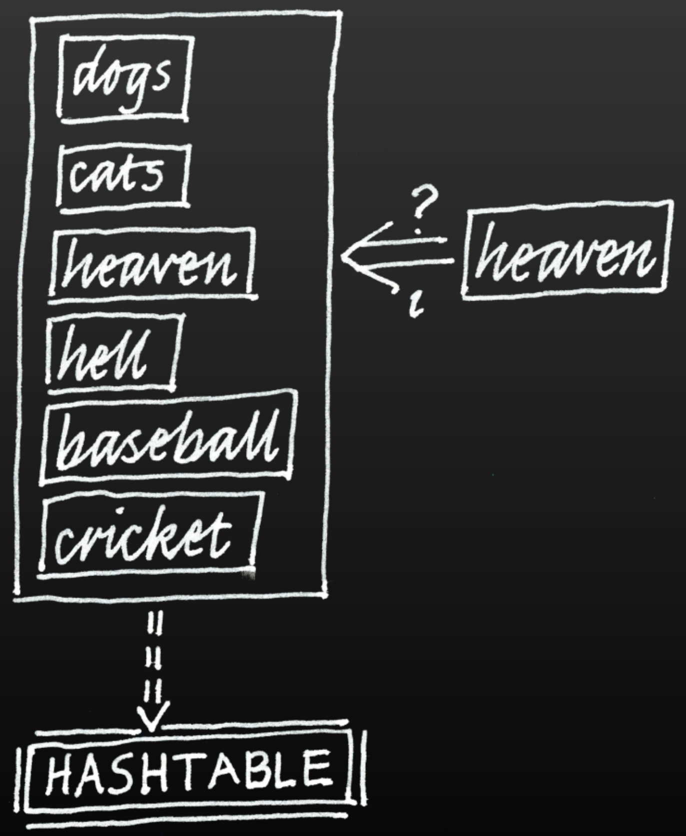


Leverage implicit data structures, explicit arrays.



Implicit Structures

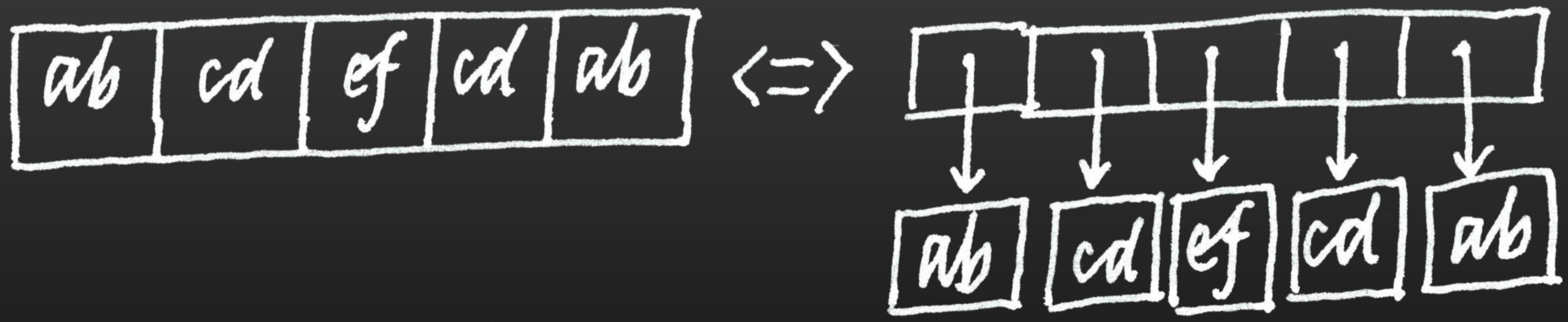
Leverage implicit data structures, explicit arrays.



Explicitly intern data using symbol tables.

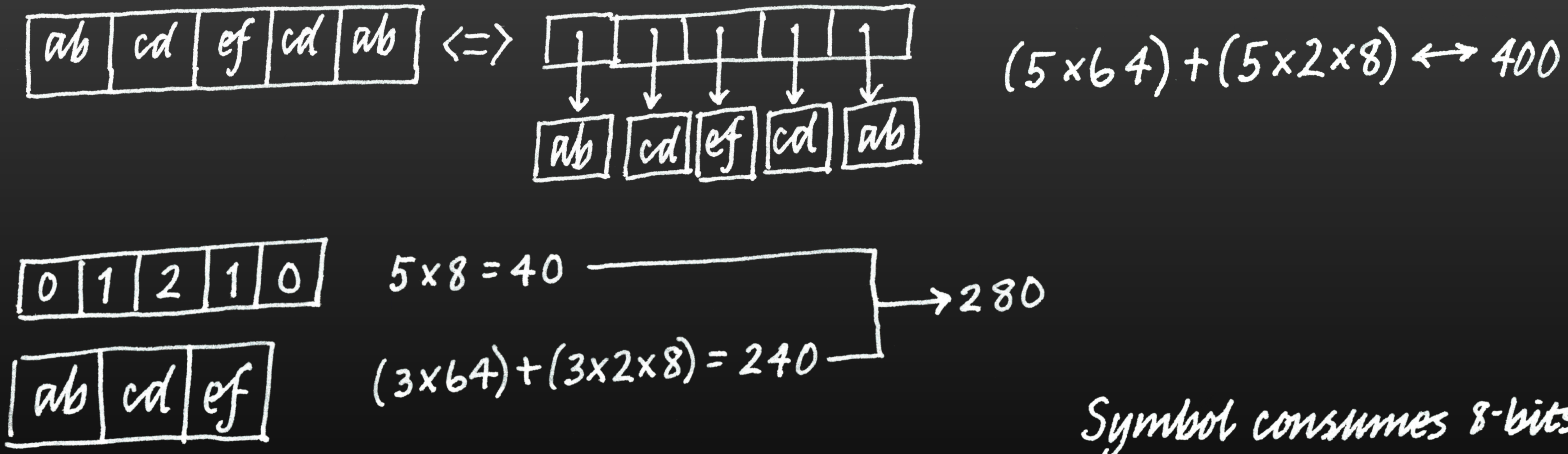
Symbol tables

Explicitly intern data using symbol tables.



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

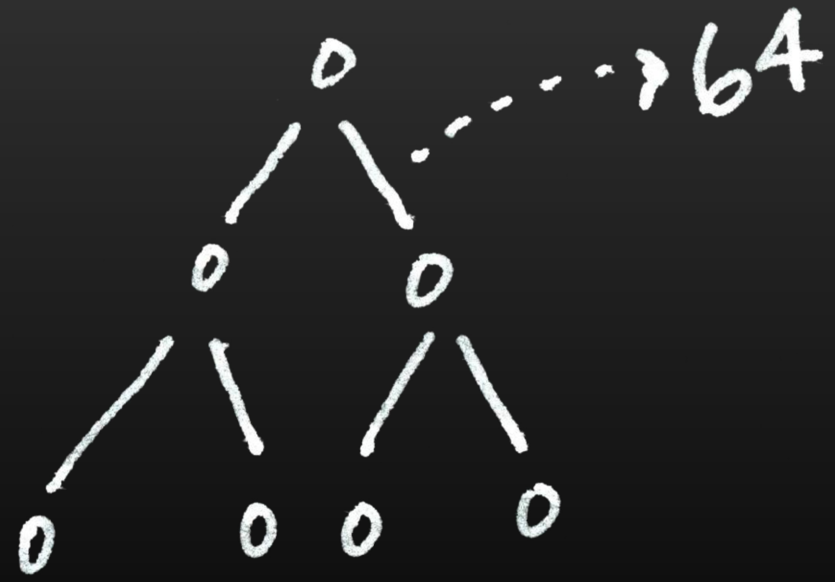
Explicitly intern data using symbol tables.



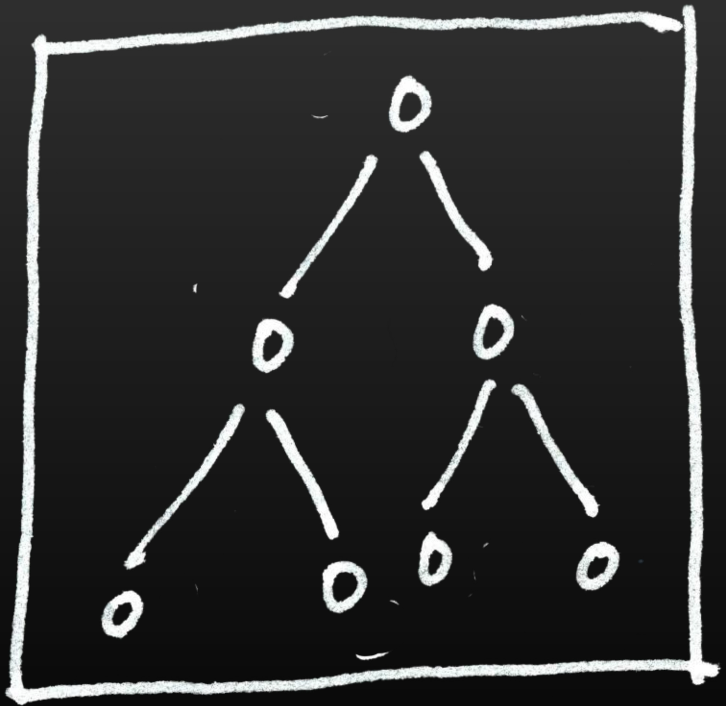
Symbol consumes 8-bits, not 64!

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

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

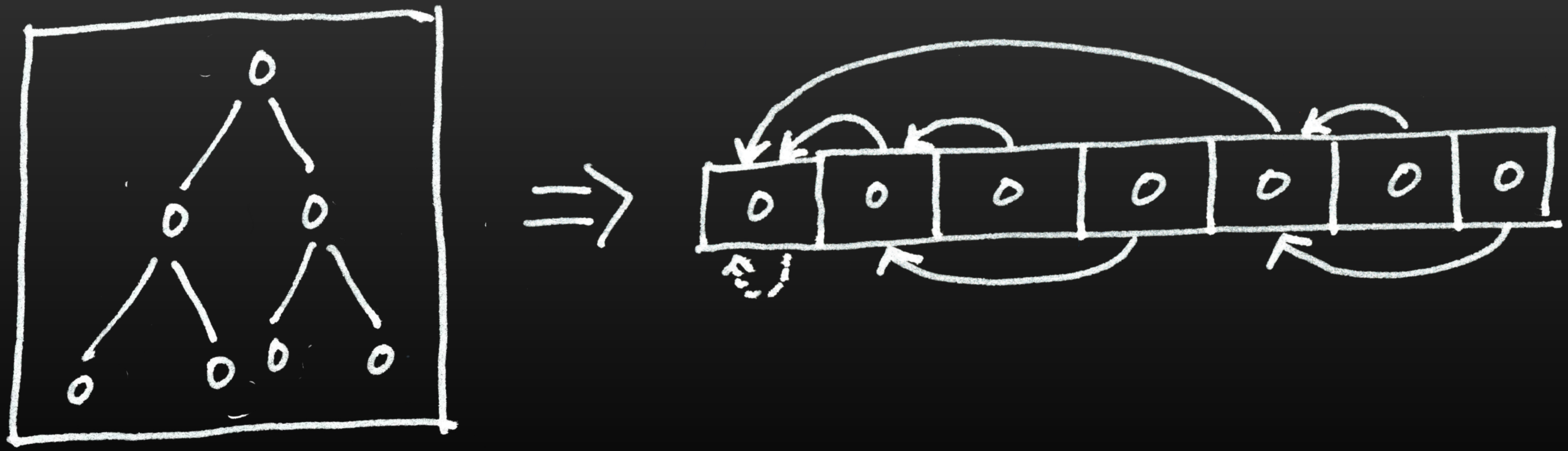


Avoid generalized pointers,
use type-constrained explicit pointers with restricted range.

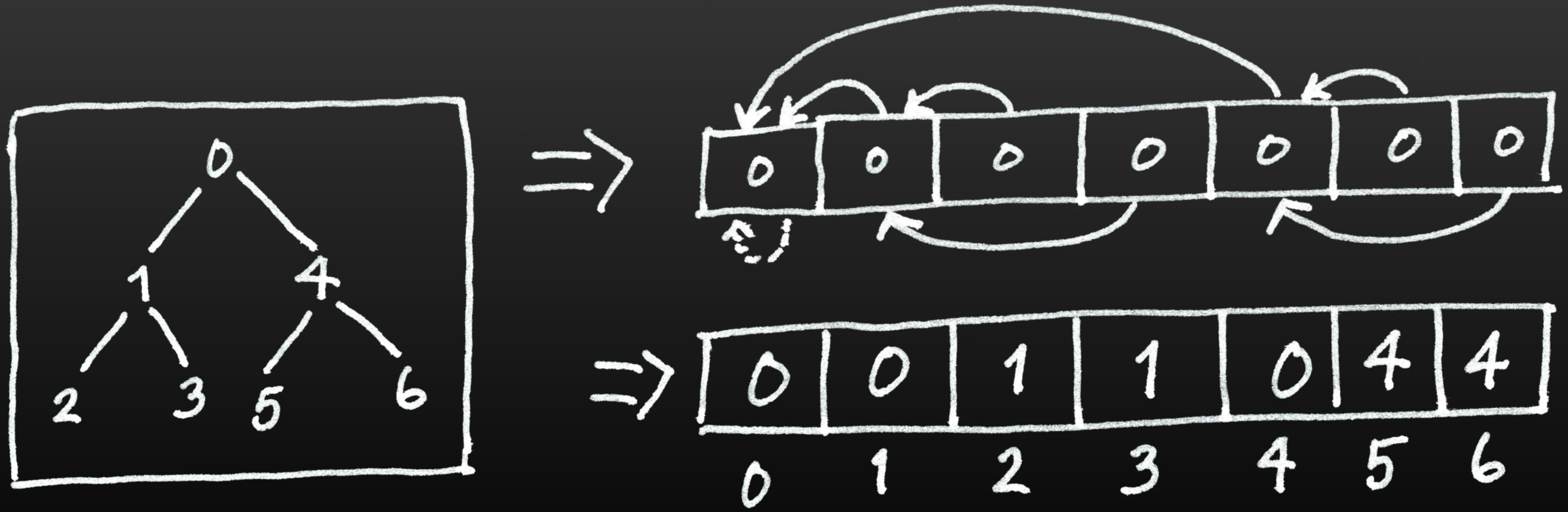


Pointers

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



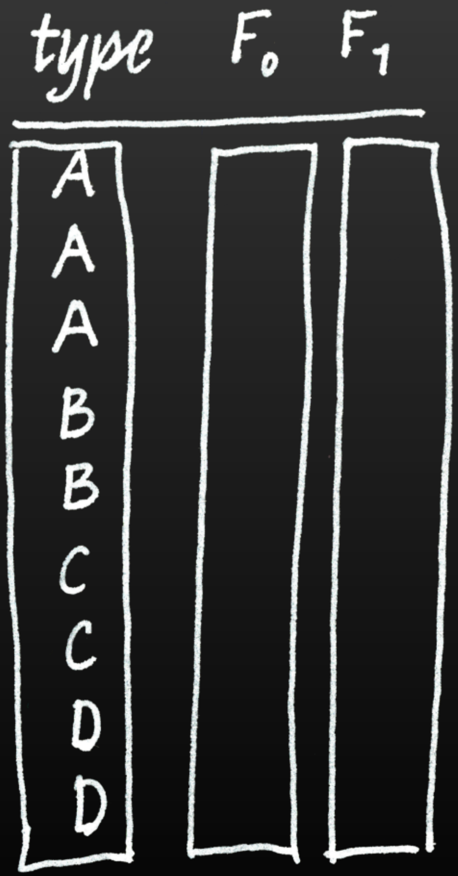
Avoid generalized pointers,
use type-constrained explicit pointers with restricted range.



Think aggregately for tags, types, and classes.

Enums/Types

Think aggregately for tags, types, and classes.



type=A

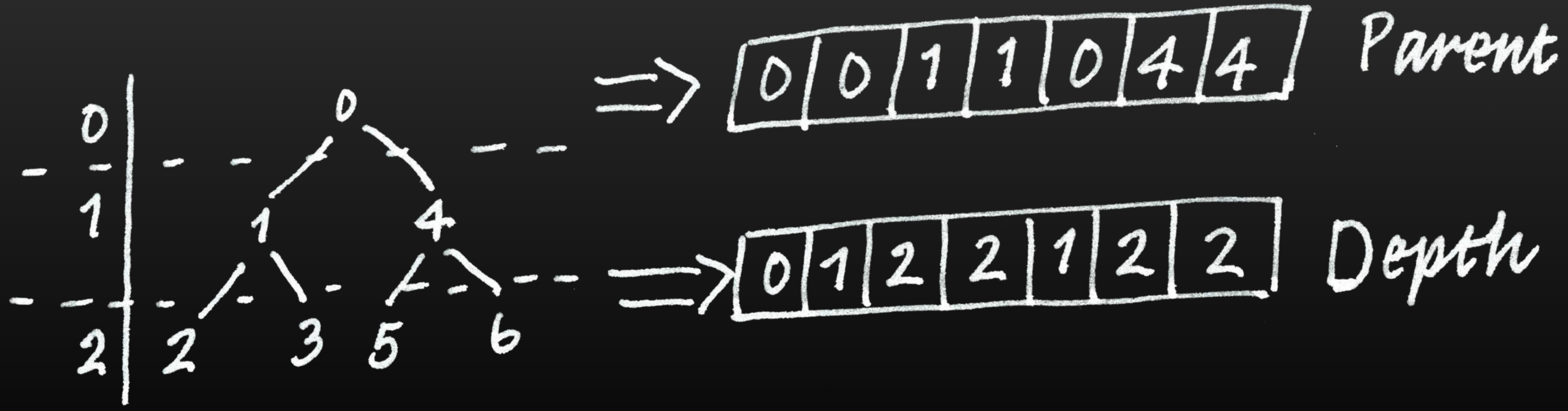
(type=A)∨(type=B)

type∊A B

{⍺(≢⍵)}⌸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.

The Quick Brown Fox

1000100000100000100

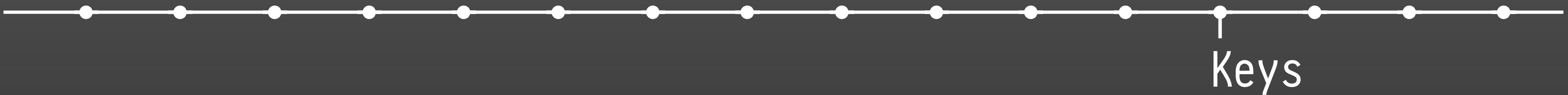
1110111110111110111

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.



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

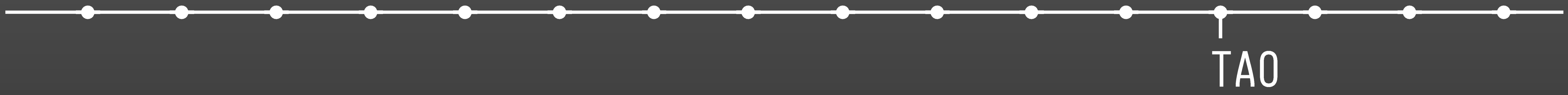
The Quick Brown Fox
1000100000100000100
1110111110111110111
1111222222333333444

xoFnworBkciuQehT
4443333322222111

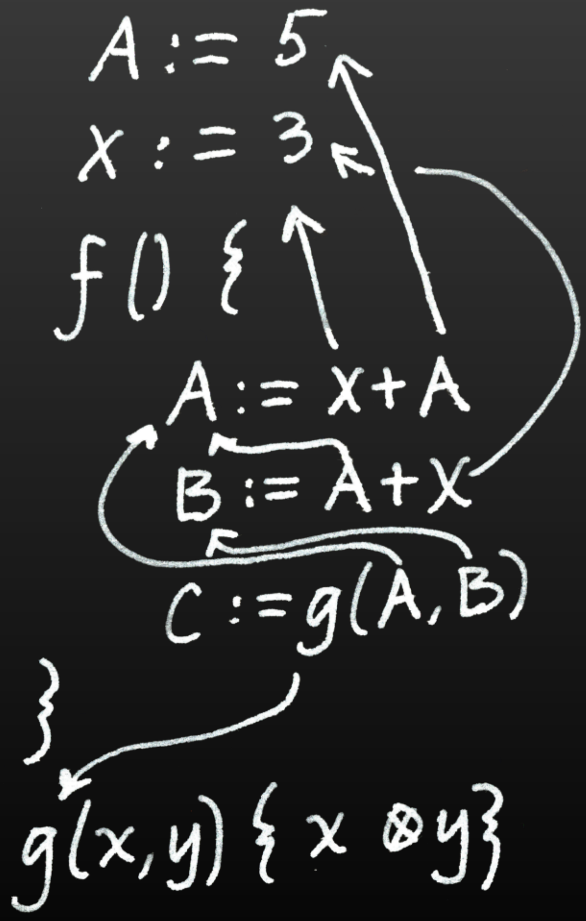
ehTkciuQnworBxoF
1112222233333444

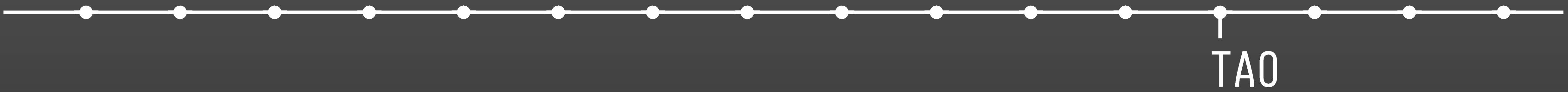
ehT kciuQ nworB xoF
1110111110111110111

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

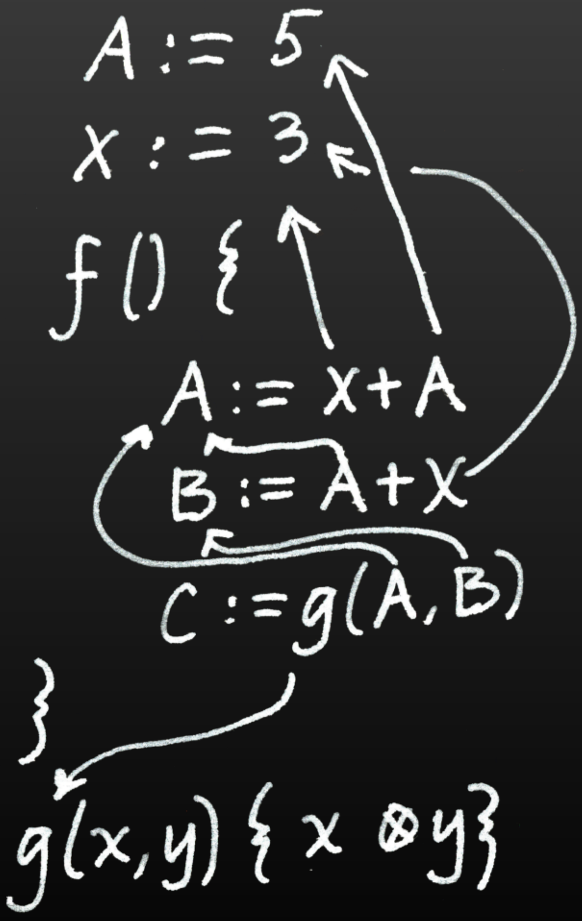


Embrace the TAO:





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

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.