PSD3 Principled AST Design v2

Status: Proposed **Date**: 2026-01-26 **Branch**: `feature/recursive-join` **Supersedes**: PRINCIPLED-AST-DESIGN.md (v1)

Executive Summary

This document refines the PSD3 AST design based on deeper analysis through the lens of recursion schemes. The key insight is that data-to-visual transformation involves **two orthogonal choices**:

1. **Enumeration**: How to traverse/extract elements from the input structure
2. **Assembly**: How to structure the output DOM

These choices are independent. Any enumeration strategy can combine with any assembly strategy. This gives us an N×M matrix of possibilities from a single primitive.

The Core Abstraction

What We're Doing (Categorically)

A visualization is a **transduction** - a structure-transforming fold:

```
Input Structure  ──(enumeration)──▶  Elements  ──(template)──▶  Visual
Specs  ──(assembly)──▶  DOM
     F a              coalgebra         [a]                        [Tree]
algebra        G DOM
```

This is a **hylomorphism with heterogeneous functors**:

- The input functor F (Array, Tree, Graph, etc.)
- The output functor G (DOM tree, but with varying nesting)
- The template: `a -> Tree a` (how to visualize one element)

Why "Join" Was Wrong

D3's "join" terminology conflates:

- Binding data to elements (the template)
- Iterating over collections (enumeration)
- Handling enter/update/exit (differential updates)
- DOM structure (assembly)

These are separate concerns. Our new vocabulary:

Concern	New Term	What It Does
Iteration	**Enumeration**	Extract elements from input structure

Concern	New Term	What It Does
DOM structure	**Assembly**	Combine visual specs into output
Element creation	**Template**	Map one datum to visual spec
Differential updates	**GUP**	Handle enter/update/exit transitions

The AST

Core Types

```
data Tree a
  -- | A DOM element with attributes, children, and behaviors
  = Elem
      { elemType :: ElementType
      , attrs :: Array (Attribute a)
      , children :: Array (Tree a)
      , behaviors :: Array (Behavior a)
      }

  -- | The universal fold: enumerate, transform, assemble
  | Fold
      { name :: String                    -- Selection name for retrieval
      , enumerate :: Enumeration a        -- How to get elements from
input
      , assemble :: Assembly              -- How to structure output
      , template :: a -> Tree a           -- How to visualize one element
      , gup :: Maybe (GUPSpec a)          -- Optional differential updates
      }

  -- | Empty (for conditional rendering)
  | Empty
```

Enumeration Strategies

```
-- | How to extract elements from a data structure
data Enumeration a
  -- | From a flat collection (Array, List, etc.)
  = FromArray (Array a)

  -- | From a tree structure with traversal order
  | FromTree
      { root :: a
      , children :: a -> Array a          -- Coalgebra
      , order :: TraversalOrder
      , includeInternal :: Boolean        -- Include non-leaf nodes?
      }

  -- | From a graph structure
```

```
  | FromGraph
      { graph :: Graph a
      , traversal :: GraphTraversal a
      }

  -- | Already enumerated with context (depth, index, parent, etc.)
  | WithContext (Array { datum :: a, context :: Context })

data TraversalOrder
  = DepthFirst
  | BreadthFirst
  | PreOrder
  | PostOrder

data GraphTraversal a
  = AllNodes
  | AllEdges
  | SpanningTreeFrom a                    -- BFS/DFS from root
  | TopologicalOrder                      -- For DAGs

data Context = Context
  { depth :: Int
  , index :: Int
  , parentKey :: Maybe String
  }
```

Assembly Strategies

```
-- | How to structure the output DOM
data Assembly
  -- | All elements as siblings (flat)
  = Siblings

  -- | Preserve input structure in output (nested)
  | Nested

  -- | Group elements by key before assembly
  | GroupedBy (forall a. a -> String)

  -- | Custom nesting based on context
  | ByDepth (Int -> ElementType)          -- Different wrapper per depth
```

GUP (Differential Updates)

```
data GUPSpec a = GUPSpec
  { enter :: Maybe (PhaseSpec a)
  , update :: Maybe (PhaseSpec a)
  , exit :: Maybe (PhaseSpec a)
  }
```

```
data PhaseSpec a = PhaseSpec
  { attrs :: Array (Attribute a)
  , transition :: Maybe TransitionConfig
  }

data TransitionConfig = TransitionConfig
  { duration :: Milliseconds
  , delay :: Maybe Milliseconds
  , ease :: EaseFunction
  }
```

Composition Patterns

Pattern 1: Simple Array (was: Join)

```
Fold
  { name: "circles"
  , enumerate: FromArray points
  , assemble: Siblings
  , template: \pt -> Elem Circle [ cx pt.x, cy pt.y, r 5.0 ]
  , gup: Nothing
  }
```

Pattern 2: Tree to Flat DOM (force layout of hierarchy)

```
Fold
  { name: "nodes"
  , enumerate: FromTree
      { root: flareData
      , children: _.children
      , order: DepthFirst
      , includeInternal: true
      }
  , assemble: Siblings                    -- Flat output!
  , template: \node ->
      Elem Circle [ r (if isLeaf node then node.value else 5.0) ]
  , gup: Nothing
  }
```

Pattern 3: Tree to Nested DOM (treemap)

```
Fold
  { name: "cells"
  , enumerate: FromTree { root: flareData, children: _.children, ... }
  , assemble: Nested                      -- Preserve hierarchy!
```

```
  , template: \node ->
      Elem Group
        [ Elem Rect [ width node.w, height node.h ]
        , Elem Text [ text node.name ]
        ]
  , gup: Nothing
  }
```

Pattern 4: Graph to Nodes + Edges

```
Elem SVG []
  [ Fold
      { name: "edges"
      , enumerate: FromGraph { graph: g, traversal: AllEdges }
      , assemble: Siblings
      , template: \edge -> Elem Line [ x1 edge.source.x, y1 edge.source.y,
... ]
      , gup: Nothing
      }
  , Fold
      { name: "nodes"
      , enumerate: FromGraph { graph: g, traversal: AllNodes }
      , assemble: Siblings
      , template: \node -> Elem Circle [ cx node.x, cy node.y ]
      , gup: Nothing
      }
  ]
```

Pattern 5: Nested Arrays to Nested DOM (was: NestedJoin)

Data: `Array (Array (Array Int))` → DOM: `<g><table><tr><td>`

```
Fold
  { name: "groups"
  , enumerate: FromArray groups
  , assemble: Siblings
  , template: \tables ->                  -- :: Array (Array Int)
      Elem Group []
        [ Fold
            { name: "tables"
            , enumerate: FromArray tables
            , assemble: Siblings
            , template: \rows ->          -- :: Array Int
                Elem Table []
                  [ Fold
                      { name: "rows"
                      , enumerate: FromArray rows
                      , assemble: Siblings
                      , template: \cells ->
```

```
                            Elem Tr []
                              [ Fold
                                  { name: "cells"
                                  , enumerate: FromArray cells
                                  , assemble: Siblings
                                  , template: \n ->
                                      Elem Td [] [ text (show n) ]
                                  , gup: Nothing
                                  }
                              ]
                        , gup: Nothing
                        }
                    ]
              , gup: Nothing
              }
        ]
  , gup: Nothing
  }
```

Pattern 6: Flat Data to Nested DOM (grouping)

```
-- Group flat items by category, then by subcategory
Fold
  { name: "categories"
  , enumerate: FromArray (groupBy _.category items)
  , assemble: Siblings
  , template: \catItems ->
      Elem Group [ class_ "category" ]
        [ Fold
            { enumerate: FromArray (groupBy _.subcat catItems)
            , assemble: Siblings
            , template: \subItems ->
                Elem Group [ class_ "subcategory" ]
                  [ Fold
                      { enumerate: FromArray subItems
                      , assemble: Siblings
                      , template: itemTemplate
                      , ...
                      }
                  ]
            , ...
            }
        ]
  , ...
  }
```

Pattern 7: With GUP (animated transitions)

```
Fold
  { name: "bars"
  , enumerate: FromArray data
  , assemble: Siblings
  , template: barTemplate
  , gup: Just $ GUPSpec
      { enter: Just $ PhaseSpec
          { attrs: [ height 0, opacity 0 ]
          , transition: Just { duration: 500, ease: EaseCubicOut }
          }
      , update: Just $ PhaseSpec
          { attrs: []
          , transition: Just { duration: 300, ease: EaseLinear }
          }
      , exit: Just $ PhaseSpec
          { attrs: [ height 0, opacity 0 ]
          , transition: Just { duration: 300, ease: EaseLinear }
          }
      }
  }
```

The Enumeration × Assembly Matrix

Every combination is valid:

Enumeration	Assembly	Use Case
Array	Siblings	Standard bar/scatter chart
Array	Nested	Grouped chart
Tree (flat)	Siblings	Force layout of hierarchy
Tree (preserve)	Nested	Treemap, sunburst
Tree (with depth)	ByDepth	Indented tree, org chart
Graph (nodes)	Siblings	Network nodes
Graph (spanning)	Nested	Hierarchical graph view
Nested Array	Nested	Table from nested data
Flat + groupBy	Nested	Table from flat data

Convenience Combinators

Built on top of `Fold`, not primitives:

```
-- Simple array iteration
forEach :: String -> Array a -> (a -> Tree a) -> Tree a
forEach name items template = Fold
```

```
  { name, enumerate: FromArray items, assemble: Siblings, template, gup:
Nothing }

-- Tree with flat output
flattenTree :: String -> Tree a -> (a -> Tree a) -> Tree a
flattenTree name tree template = Fold
  { name
  , enumerate: FromTree { root: tree, children: treeChildren, order:
DepthFirst, includeInternal: true }
  , assemble: Siblings
  , template
  , gup: Nothing
  }

-- Tree with nested output
preserveTree :: String -> Tree a -> (a -> Tree a) -> Tree a
preserveTree name tree template = Fold
  { name
  , enumerate: FromTree { root: tree, children: treeChildren, ... }
  , assemble: Nested
  , template
  , gup: Nothing
  }

-- Nested arrays with element types per level
nestedArrays :: Array ElementType -> (a -> Tree a) -> NestedArray a ->
Tree a
nestedArrays elemTypes leafTemplate data =
  -- Recursively builds nested Folds
  ...
```

Interpreter Sketch

```
interpret :: forall a. Selection -> Tree a -> Effect (SelectionMap a)

interpret parent (Elem spec) = do
  el <- createElement spec.elemType parent
  applyAttrs el spec.attrs Nothing
  applyBehaviors el spec.behaviors
  childMaps <- traverse (interpret (asSelection el)) spec.children
  pure $ unions childMaps

interpret parent (Fold spec) = do
  -- 1. Enumerate elements from input
  let elements = runEnumeration spec.enumerate

  -- 2. Apply template to each, getting visual specs
  let specs = map spec.template elements

  -- 3. Assemble into output structure
  case spec.assemble of
```

```
    Siblings -> do
      results <- forWithIndex elements \i (datum, spec) -> do
        el <- createElement elemTypeFromSpec parent
        childMaps <- interpret (asSelection el) spec
        bindDatum el datum
        pure { el, datum, childMaps }
      let selection = mkSelection (map _.el results) (map _.datum results)
      pure $ Map.singleton spec.name selection <> unions (map _.childMaps
results)

    Nested -> do
      -- Recursively create nested structure
      ...

    GroupedBy keyFn -> do
      -- Group then recurse
      ...

interpret _ Empty = pure Map.empty
```

Why This Design

Principled

- Based on recursion schemes (coalgebras, algebras, hylomorphisms)
- Enumeration is a coalgebra (how to unfold input)
- Assembly is an algebra (how to fold into output)
- Template is a natural transformation

Minimal

- One iteration primitive: `Fold`
- Three constructors total: `Elem`, `Fold`, `Empty`
- Complex patterns from composition, not special cases

Orthogonal

- Enumeration strategy independent of assembly strategy
- GUP independent of both
- Template independent of iteration

Composable

- Nested folds for nested data
- Combinators built on the primitive
- Closures for parent context (no special "hierarchical" mode)

Extensible

- New enumeration strategies (DAG, spatial index, etc.)

- New assembly strategies (virtual scroll, LOD, etc.)
- Same core primitive

Migration from V1

V1 Construct	V2 Equivalent
`Join`	`Fold { enumerate: FromArray, assemble: Siblings }`
`NestedJoin`	Nested `Fold`s or flatten in enumeration
`UpdateJoin`	`Fold { ..., gup: Just ... }`
`HierarchicalJoin`	Nested `Fold`s with closure
`RecursiveJoin`	`Fold { enumerate: FromTree, assemble: Nested }`

Open Questions

1. Key Function Location

Should `keyFn` be part of enumeration, assembly, or both?

- For diffing (GUP): needed at assembly
- For DOM keys: needed at assembly
- **Recommendation**: Part of `Fold`, not enumeration

2. Element Type in Fold

Should `Fold` specify the element type, or delegate to template?

- Current: template returns full `Elem`
- Alternative: `Fold { key: Circle, ... }` with template returning just attrs
- **Recommendation**: Template returns full `Elem` for flexibility

3. Selection Granularity

How do nested folds affect selection retrieval?

- Each `Fold` has a name
- Nested folds have nested names?
- **Recommendation**: Flat namespace, user ensures uniqueness

4. Context Propagation

How does a nested template access parent data?

- Current: closures (works!)
- Alternative: explicit context parameter
- **Recommendation**: Closures are sufficient and idiomatic

Conclusion

The V2 design recognizes that **enumeration** and **assembly** are orthogonal choices in a data-to-visual transduction. By parameterizing `Fold` with both, we get:

- Full combinatorial flexibility (any input shape × any output shape)
- One primitive instead of six
- Principled foundation in recursion schemes
- Natural composition for complex patterns

The cost is a more explicit specification of intent. The benefit is a system that's smaller, more powerful, and theoretically grounded.

Appendix: Recursion Schemes Glossary

Term	Definition	In PSD3
Algebra	`F a -> a` - combines structure	Assembly
Coalgebra	`a -> F a` - reveals structure	Enumeration
Catamorphism	Fold using an algebra	Fold with assembly
Anamorphism	Unfold using a coalgebra	Enumeration
Hylomorphism	Unfold then fold	Full Fold operation
Natural transformation	`F a -> G a`	Template (sort of)
Transduction	Structure-changing transformation	The whole thing