

# Emmet-Style DSL Design for PSD3v2 Nested Elements

## The Problem

Code-explorer needs to create multi-level DOM structures:

```
Group → Circle + Text
```

Current PSD3v2 doesn't support nested builders, which is blocking the code-explorer port.

## The Insight

**The DOM structure IS the type signature of your visualization.**

Just like a type signature tells you what a function does, the DOM structure tells you what your visualization looks like:

```
-- Type signature tells you everything
map :: forall a b. (a -> b) -> Array a -> Array b

-- DOM signature should do the same
nodeViz :: "g > circle + text" -> Data -> Visualization
```

## Current Approaches (Rejected)

### Option 1: Callback Style

```
nodeGroup <- append Group [...] enter
withChildren nodeGroup \group -> do
  circle <- appendChild Circle [...] group
  text <- appendChild Text [...] group
  pure { circle, text }
```

**Problem:** Nesting, hard to see structure at a glance

### Option 2: Do-Notation (Current D3 Style)

```
groups <- append Group [transform ...] enter
circles <- appendChild Circle [radius ...] groups
texts <- appendChild Text [textContent ...] groups
pure { groups, circles, texts }
```

**Problem:** Imperative sequence, structure is scattered across multiple lines

### Option 3: Type-Level String Parsing

```
{ groups, circles, texts } <- append "g > circle + text" attrs enter
```

**Problem:** Would require dependent types, error messages would be incomprehensible

## The Sweet Spot: Emmet-Inspired Operators

Use **operator syntax** that reads like Emmet but uses real PureScript values:

```
-- Define structure as a value (not a string)
nodeStructure = group >: (circle +: text)

-- Single append creates the whole tree
nodes <- append nodeStructure enter

-- Attributes use type-safe record fields
attrs nodes.group [transform (\d -> translate d)]
attrs nodes.circle [radius (\d -> d.r), fill "blue"]
attrs nodes.text [textContent (\d -> d.name)]
```

Where:

- **>:** is the "child of" operator (like Emmet's **>**)
- **+:** is the "sibling of" operator (like Emmet's **+**)
- Return type is automatically a record with all the selections

## Implementation Approach

### Structure Builders (GADTs)

```
data Struct a where
  Elel :: ElementType -> Struct (Selection SBound Element datum)
  Child :: ElementType -> Struct child -> Struct { parent :: ..., child :: child }
  Sibling :: Struct a -> Struct b -> Struct (a & b) -- Row merge

-- Smart constructors
circle :: Struct (Selection SBound Element datum)
circle = Elel Circle

text :: Struct (Selection SBound Element datum)
text = Elel Text
```

```

group :: ElementType
group = Group

-- Operators
(>:) :: ElementType -> Struct child -> Struct { parent :: ..., child :: child }
(>:) = Child

(+:) :: Struct a -> Struct b -> Struct (a & b)
(+:) = Sibling

```

## Type-Level Magic (Using Row Types)

The return type is **automatically derived** from the structure:

```

-- circle alone
append circle enter
-- Returns: Selection SBound Element datum

-- group >: circle
append (group >: circle) enter
-- Returns: { group :: Selection ..., circle :: Selection ... }

-- group >: (circle +: text)
append (group >: (circle +: text)) enter
-- Returns: { group :: Selection ..., circle :: Selection ..., text :: Selection ... }

```

This uses PureScript's **row types** (which it already has!) without needing type-level string parsing.

## Append Implementation

```

class AppendStruct struct sel m where
  append :: struct -> Array (Attribute datum) -> sel SPending parent datum
-> m (ResultOf struct)

-- For single elements
instance AppendStruct ElementType sel m where
  append elem attrs selection = ... -- Current implementation

-- For nested structures
instance AppendStruct (Struct child) sel m where
  append (Child parent childStruct) attrs selection = do
    parentSel <- append parent attrs selection
    childResult <- appendChildren childStruct parentSel
    pure { parent: parentSel, child: childResult }

```

## Complete Example: Code-Explorer Nodes

## Current Approach (What We Have Now)

```
-- Scattered structure definition
groups <- append Group [transform (\d -> translate d)] enter
circles <- appendChild Circle [radius (\d -> d.r)] groups
texts <- appendChild Text [textContent (\d -> d.name)] groups
dragBehavior <- on (Drag $ simulationDrag "spago") groups
pure groups
```

## Emmet-Style Approach (What We Want)

```
-- Structure definition (visual!)
let structure = group >: (circle +: text)

-- Single append creates entire tree
nodes <- append structure enter

-- Attributes per element (type-safe!)
attrs nodes.group [transform (\d -> translate d)]
attrs nodes.circle [radius (\d -> d.r), fill "blue"]
attrs nodes.text [textContent (\d -> d.name), textAnchor "middle"]

-- Behaviors attach to specific elements
on (Drag $ simulationDrag "spago") nodes.group

pure nodes.group
```

## Why This Works

### Achieves All Goals

- **✓ Visual clarity** - `group >: (circle +: text)` reads like a tree
- **✓ Type safety** - Can't access `nodes.rect` if it doesn't exist
- **✓ Tagless final** - `append` is still in SelectionM typeclass
- **✓ Sane errors** - Just record field errors ("no field 'rect'"), not type-level gibberish
- **✓ Lightweight** - One structure def, then attribute calls

### Comparison to Alternatives

Approach	Clarity	Type Safety	Error Messages	Complexity
Do-notation	✗ Scattered	✓ Good	✓ Clear	✓ Low
Callbacks	✗ Nested	✓ Good	✓ Clear	⚠ Medium
String templates	✓ Very clear	✗ None	✗ Runtime	✓ Low
Type-level parsing	✓ Clear	✓ Perfect	✗ Incomprehensible	✗ Very high

Approach	Clarity	Type Safety	Error Messages	Complexity
Emmet operators	✓ Clear	✓ Good	✓ Clear	✓ Low

## Extended Ideas (Future)

### Attachment Points in Structure

Could we specify the attachment point too?

```
viz <- attach "div#vis" do
  svg <- elem SVG [width 800]
  zoomGroup <- child (group >: empty)
  nodes <- child (group >: (circle +: text)) `withData` nodeData
  pure { svg, zoomGroup, nodes }
```

### Data Join Integration

```
-- Specify join points in structure
let structure =
  svg >:
    (group.zoom >:
      (group.node `bindTo` nodes >: (circle +: text)))
-- Automatically handles enter/update/exit
viz <- visualize "div#vis" structure
```

### CSS-Style Selectors

```
-- Like Emmet with classes and IDs
let structure =
  group.zoom#zoomGroup >:
    (group.node >: (circle.primary + text.label))
```

## Implementation Plan

### Phase 1: Basic Operators ✓

1. Define **Struct** GADT
2. Implement **>:** (child) operator
3. Implement **+:** (sibling) operator
4. Create smart constructors for common elements

### Phase 2: Append Integration

1. Add `AppendStruct` typeclass
2. Implement for single elements (backward compatible)
3. Implement for nested structures
4. Handle attributes per element

### Phase 3: Test with NestedMatrix Example

1. Create simple 5x5 grid
2. Each cell: `group >: (circle +: text)`
3. Verify type inference works
4. Verify error messages are clear

### Phase 4: Code-Explorer Integration

1. Update RenderV2 to use nested structures
2. Replace simplified circles-only approach
3. Add text labels back in
4. Test full visualization

## Open Questions

1. **Attribute distribution:** How to specify different attributes for each element?
  - Current idea: `attrs nodes.circle [...]` (field access)
  - Alternative: Pass record of attributes to `append?`
2. **Naming convention:** How are fields named in the result record?
  - Use element type names? (`group, circle, text`)
  - Allow custom names? (`group.zoom, group.node`)
3. **Multiple children:** How to handle `group >: (circle +: circle)?`
  - Need unique names somehow
  - Index them? (`circle_1, circle_2`)
  - Require explicit naming?
4. **Update phase:** Do we need to access nested elements during update?
  - Currently only enter phase creates elements
  - Update might need to update children attributes too

## Why "First Idea, Best Idea" Applies Here

The Emmet syntax insight reveals something fundamental:

- **HTML/SVG is inherently tree-structured**
- **Emmet already solved this problem** for HTML
- **We're just adapting proven UX** to PureScript + D3

This isn't inventing something new, it's recognizing that the web platform already has an excellent DSL for describing nested structures, and bringing that clarity to our domain.

## Next Steps

1.  Document this design (this file)
  2. Prototype the `>:` and `:+` operators
  3. Create NestedMatrixV2 as proof-of-concept
  4. If it works well, integrate into code-explorer
  5. If not, at least we'll understand the tradeoffs better
- 

**Status:** Design phase complete, ready for implementation after compaction **Date:** 2025-01-15 **Context:** Code-explorer port blocked on nested elements, this would unblock it