Music Interpreter Ideas

Context

Brainstorm session (Dec 2025) exploring music generation using the finally tagless / interpreter architecture.

Two Related Ideas

1. Music Interpreter for PSD3 (Near-term Demo)

Add a music interpreter to PSD3 to demonstrate the architecture's flexibility - same selection/data operations, interpreted as sound instead of DOM manipulation.

Sonification possibilities:

- Data values → pitch
- Selection size → volume/dynamics
- Hierarchical depth → octave
- Transitions → glissando/portamento
- Enter/update/exit → attack/sustain/release

This would be a powerful proof that the tagless architecture truly separates description from interpretation.

2. Standalone Generative Music DSL (Future Side Project)

A PureScript DSL for generative music, inspired by Tidal Cycles / Strudel, using the same architectural patterns as PSD3.

Core parallels:

PSD3	Music DSL
Selection	Pattern
`.selectAll()`, `.data()`, `.enter()`	`note()`, `sound()`, `stack()`
`.attr()`, `.style()`	`.gain()`, `.pan()`, `.lpf()`
Transforms (transition, etc.)	`fast()`, `slow()`, `rev()`, `every()`
Interpreters: DOM, English, Mermaid	Interpreters: Audio, Viz, English, Notation

Potential interpreters:

- **Audio** - Web Audio API or target Strudel directly
- **Notation** - Sheet music (SVG, VexFlow, ABC notation)
- **English** - "A quarter note C followed by an eighth note D..."
- **Visualization** - Piano roll, circular time, pattern structure trees
- **Analysis** - Chord recognition, harmonic analysis
- **MIDI export** - Standard interchange format

Key insight: Strudel already lives in JS/browser, so a PureScript DSL could target it as one interpreter (just like PSD3 targets D3).

Design questions to resolve:

- Where to draw the DSL boundary?
- Model just pattern combinators? Include synth/sampler layer? Effects/mixing?
- How to handle the mini-notation (parse into AST vs. generate as output)?

Why This Matters

Both ideas validate the core thesis: **finally tagless lets you write one program and interpret it many ways**. Music is a compelling domain because:

- Patterns are inherently compositional
- Multiple representations are natural (audio, notation, visualization)
- Generative/algorithmic approaches fit the functional paradigm
- Educational interpreters ("explain what this pattern does") are valuable

Implementation Update (Dec 2025)

Created `psd3-music` package as a new sub-repo to explore these ideas through prototyping.

Decision: Start with Sonification, Not Composition

Rather than jumping directly to a music composition DSL with mini-notation, we're starting with **data sonification**:

Rationale:

1. **Familiar territory** - Stay in PSD3's wheelhouse (data → output)
2. **Pure interpreter work** - No need to design new DSL syntax yet
3. **Immediate value** - Accessibility and multimodal data analysis
4. **Foundation for later** - Learn how temporal/spectral mappings work before tackling composition

Key insight: `audio` is a peer of `svg`, not a different interpretation of the same viz. This is for situations where visual attention is unavailable or impractical:

- Unsighted users analyzing data
- Pilots/drivers with vision occupied
- Monitoring systems (audio dashboards)
- Multimodal analysis (seeing + hearing data simultaneously)

Conceptual Mappings

D3 Domain	Music/Audio Domain
`select "svg"`	Create audio context
`selectAll "circle"`	Create array of sound events/oscillators
Data join	Bind data to sonic parameters

D3 Domain	Music/Audio Domain
`attr "cx"` (x position)	Time offset (when)
`attr "cy"` (y position)	Pitch (what frequency)
`attr "r"` (radius)	Duration/volume (how long/loud)
`attr "fill"` (color)	Timbre (waveform type)
Enter/Update/Exit	Sound onset/sustain/release (ADSR envelope)
Parent/child hierarchy	Sequential/parallel composition

Typed Joins in Music Context

The data join pattern could elegantly separate **temporal structure** from **musical content**:

```
-- Temporal structure (like a DOM skeleton)
rhythm = parse "x x [x x] x"

-- Musical content (like data to join)
melody = [C4, E4, G4, A4, C5]
chord = Maj7 C

-- Typed joins ensure compatibility
rhythm `joinNotes` melody   -- ✓ Rhythm + Notes → Melody
rhythm `joinChord` chord     -- ✓ Rhythm + Chord → Harmony
rhythm `joinChord` melody    -- ✗ Type error!
```

This mirrors modular synthesis philosophy:

- Structure generators = sequencers, clocks
- Content sources = oscillators, CV sources
- Typed joins = typed patch cables

Roundtripping Text ↔ Visual

The TreeBuilder3 UI inspired thinking about **visual mini-notation**:

- Type mini-notation when you know what you want: `"bd [sd sd] hh"`
- Drag/drop to explore structure visually
- See AST while editing text
- Edit tree and regenerate text

This could be a "provocative" PSD3 demo showing:

1. Same musical program, multiple interpreters (audio, notation, English, piano roll)
2. Built WITH PSD3 (the editor is itself a PSD3 viz - dogfooding)
3. Architecture isn't just for bar charts

Research Questions

From prototyping, we need to discover:

1. **What is MOAR ABSTRACT?** - Is there a deeper pattern that both D3 and music instantiate?
    - Both seem to do: `structure + data + mapping → interpreted output`
    - Structure: DOM skeleton vs. temporal pattern
    - Data: Arrays/records vs. musical materials
    - Mapping: Attributes vs. sonic parameters
    - Interpretation: DOM manipulation vs. audio scheduling
2. **What is "selection" in music?** - What gets selected/joined?
    - D3: DOM elements in spatial arrangement
    - Music: Event slots in temporal arrangement?
    - Both: Containers with positions that can hold data
3. **Where are the true parallels?** - By implementing both, we can see what's shared vs. domain-specific

Implementation Progress (Dec 20, 2025)

✓ Completed:

1. **Web Audio FFI** - Created `psd3-music/src/PSD3/Music/Internal/FFI.purs/.js`
    - `AudioContext` type and creation
    - `scheduleNote` function for Web Audio API scheduling
    - Clean separation: FFI for imperative browser APIs, pure PureScript for logic
2. **Music Interpreter** - Implemented `SelectionM` for audio output
    - `psd3-music/src/PSD3/Music/Interpreter/WebAudio.purs`
    - `appendData` schedules Web Audio notes instead of creating DOM elements
    - Proved Finally Tagless works: same code, completely different output medium
3. **Audio-Specific Attributes** - Created music equivalents of visual attributes
    - `psd3-music/src/PSD3/Music/Attributes.purs`
    - `time`, `pitch`, `duration`, `volume`, `timbre` (parallel to `cx`, `cy`, `r`, `fill`)
    - Same `IndexedAttr` infrastructure as D3 attributes
4. **Parabola Demo** - First working proof-of-concept
    - Standalone HTML (`parabola-audio-test.html`)
    - Same parabola data → SVG circles OR audio tones
    - Validated the architecture end-to-end
5. **Anscombe's Quartet Sonification** - Production demo
    - Four datasets with identical statistics but different sounds
    - Embedded in Tour page (`Component/Tour/TourSonification.purs`)
    - Interactive: click to start/stop looping arpeggiation for each dataset

    - Complete with data tables and visual feedback

Key Discovery: Arpeggiation for Audio Pattern Perception

Unlike vision (where you can scan back and forth), **audio needs repetition to perceive patterns**:

- Single playthrough: hard to grasp the "shape" of the data
- **Looping arpeggiation**: pattern becomes clear through repetition
- Timing: 175ms between notes, 400ms pause between cycles
- This is the auditory equivalent of how you visually scan a scatterplot

This insight directly informed the demo UX and is fundamental to data sonification.

Successful Attribute Mappings:

The following mappings proved effective in practice:

Data Dimension	Visual (D3)	Audio (Music)	Notes
x value	Horizontal position	**Time** (when to play)	✓ Works well - creates temporal ordering
y value	Vertical position	**Pitch** (frequency)	✓ Works well - higher values = higher tones
y value	Circle radius	**Volume** (amplitude)	✓ Works well - reinforces pitch information
-	-	Duration	0.15s worked well for 2x speed arpeggiation
-	-	Timbre	All datasets used 'sine' - differentiation via pattern, not timbre

Formula used:

- Frequency: `200 + (y * 50)` Hz (maps y ∈ [0, 15] to ~200-700 Hz range)
- Volume: `min(0.5, y / 15)` (normalized to avoid clipping)
- Duration: `0.15s` (short enough for fast arpeggiation, long enough to hear)

Architecture Validation:

The implementation proved the Finally Tagless thesis:

- Same `SelectionM` type class
- Same code structure (`appendData` with attributes)
- Completely different interpreters (DOM manipulation vs. Web Audio scheduling)
- Audio is truly a **peer of SVG**, not just a different visualization

Integration into Tour:

Created comprehensive Tour page at `/tour/sonification`:

- Explanation of Finally Tagless architecture

- Interactive demo with 4 dataset cards
- Audio mapping tables showing visual ↔ audio correspondences
- Implementation details for psd3-music package
- Future directions section

Next Steps:

1. ~~Implement basic Web Audio FFI~~ ✓
2. ~~Get simple tones playing from data~~ ✓
3. ~~Create Anscombe's Quartet sonification demo~~ ✓
4. ~~Document which attribute mappings work well~~ ✓
5. Compare Music interpreter with D3 interpreter to extract common patterns
6. Explore additional sonification examples (time series data, hierarchical data)
7. Consider ADSR envelopes for Enter/Update/Exit pattern

Future: Composition DSL

After learning from sonification work, consider a separate composition-focused DSL that:

- Uses mini-notation for temporal structure
- Supports typed joins for musical content
- Provides multiple interpreters (audio, notation, visualization)
- Enables live coding / algorithmic composition
- Could target Strudel as one interpreter (like PSD3 targets D3)

The sonification work is the research phase that will inform better design of the composition DSL.