

Temperamento: Canonical Text–Score Translation and Transposition Invariance in an Executable MusicXML Notation

Tomas Laurenzo
Department of Critical Media Practices
University of Colorado Boulder
Boulder, Colorado, USA
tomas@laurenzo.net

Valeria Rocha
Independent Researcher
Montevideo, Uruguay
vale.rocha@proton.me

Abstract—*Temperamento* is a programming language in which the same program can be written as text or as a two-stave musical score encoded in MusicXML. Its design addresses two problems: translating reliably between textual and notated forms, and preserving program meaning when a score is transposed. On the Base stave, ordered pairs of major and minor triads select stack-machine operations through circle-of-fifths distance and endpoint mode. On the Voice stave, duration-framed notes encode non-negative operands as base-twelve values relative to the first triad root of each command pair. Because operations and operands are both defined relationally, global chromatic transposition preserves the complete compiled program. The paper defines canonical translation contracts among TOScript+, TOScript Core, and MusicXML, and evaluates the implementation through 204 assigned-cell MusicXML round trips, 36 whole-program transpositions, three source-direction packages, exhaustive harmonic-cell checks, cross-platform tests, and checksum-pinned MuseScore verification. The contribution is a programming-language design and artistic-research proposition about the boundary between musical and computational meaning, in which conventional notation serves as executable source, supported by deterministic text–score translation, relational encoding, and whole-program transposition invariance.

Keywords—programming languages, executable notation, MusicXML, program translation, transposition invariance, computer music

I. INTRODUCTION

Musical programming languages have long treated notes, intervals, chords, and other musical structures as executable syntax. Their implementations, however, read MIDI events or chord-symbol streams rather than a structured score representation. This leaves a distinct programming-language problem: whether a digital score, with staves, voices, durations, and reconstructed score time, can serve as an executable source representation while remaining translatable to and from textual code.

Temperamento addresses this problem through three executable representations. TOScript+ provides an ergonomic textual language; TOScript Core is the canonical executable form; and a restricted two-stave MusicXML score provides the notated form. TOScript+ lowers to Core, Core lifts to canonical TOScript+, a score compiles to Core, and

Core can be realized as a deterministic score. These translations preserve executable semantics rather than comments, symbolic names, voicing, spelling, engraving, or layout. The score is therefore a source representation of the program, not a visual interface layered over hidden code.

This design creates two preservation obligations. First, translating a program to another representation and back should recover the same canonical Core. Second, transposing a score should not change the program it encodes. That second requirement is stronger than preserving chord relations alone: any operation, operand, label, or addressing channel that depends on absolute pitch can change the resulting program.

Temperamento separates those channels across two staves. Exact major or minor triads on the Base stave are read in ordered pairs and encode operations. Duration-framed notes on the Voice stave encode non-negative operands relative to the root of the first triad of the pair. Both channels compile to Core and execute on a deterministic stack machine. Conversely, every supported Core program can be realized as a canonical two-stave score, while TOScript+ supplies comments, constants, named labels, branch syntax, and output conveniences that lower completely to Core.

This paper presents *Temperamento* as a programming-language design in which executable semantics are distributed across textual code and conventional musical notation. Its first contribution is the language architecture itself: paired harmonic relations encode operations, a separate root-relative melodic channel encodes operands, and computational and noncomputational musical material coexist within the same score. Its second contribution is a canonical translation model connecting TOScript+, TOScript Core, and MusicXML. Its third contribution is a whole-program transposition-invariance result, supported by an implementation and claim-directed verification. The design also functions as an artistic-research proposition about the relation between notation, interpretation, and computation, although the present paper evaluates its computational properties rather than its compositional or performative reception.

II. EXECUTABLE MUSIC AND SYMBOLIC PROGRAM TRANSLATION

Programs have long used musical materials as syntax. Fugue represents Prelude programs through polyphonic intervals, Velato interprets pitches relative to a mutable command root, and Schoenberg reads successive pitch-class distance, velocity, and overlap [1]–[3]. Polyphony, Tone-deaf and C Flat assign operations to chords or chord sequences

[4]–[6], and Chalc assigns them to triad quality [7]. Two properties of this body of work matter here. First, each implementation reads MIDI or chord-symbol text rather than MusicXML as executable source, so none reconstructs score time, staves, or voices from an interchange score. C Flat is explicit that its programs may be notated in other ways, but its interpreter reads MIDI. Second, these languages differ in how instructions and literals respond to transposition.

Fugue is the closest relative-literal antecedent: both instructions and pushed numbers are defined by intervals between successive notes, so global transposition preserves the decoded program. Schoenberg and Chalc are also invariant, but their data channels are not pitch-borne: Schoenberg derives operand magnitude from MIDI velocity, while Chalc uses non-harmonic input and output tokens. Polyphony and C Flat recognize operations relationally but read numeric literals from absolute pitch, so transposition can change data. Velato places commands and values relative to a mutable command root, but its documented exception for variable names leaves an absolute residue and prevents exact canonical-program identity under transposition. *Temperamento* therefore claims no priority for relative pitch-borne literals or transposition invariance. Its distinction is to make every pitch-bearing channel relative within a structured two-stave score, connect that score canonically to two textual forms, and verify exact whole-program recovery after transformation.

Within harmonic encodings, Chalc is the closest precedent. *Temperamento* gives the progression a different role. The ordered relation between two independently recognized triads is itself the instruction: the distance between their roots and the modes of both endpoints select one cell, while operands are written on a separate stave. Three chords in Chalc contribute three chord operations and two pointer movements; in *Temperamento*, recognized triads are consumed in non-overlapping pairs, producing one instruction per pair. Chalc admits only major and minor triads as source tokens, whereas *Temperamento* admits any well-formed sonority and treats non-triadic material as musical but non-computational.

Research also treats code as score and symbolic music as program. Magnusson examines code as musical notation [8]. Martinez Nieto's Algorithmic Score Language extends common Western notation with additional symbols so that one score can be read by performers and executed during live performance [9]; *Temperamento* takes a different route, adding no notational vocabulary and compiling ordinary pitches and durations to a stack machine. Temkin analyzes esolangs as conceptual and artistic work [10], and Singer and Draper argue that esolangs expose assumptions of language design [11]. Decomposer learns MIDI-to-Strudel decompilation, balancing reconstruction fidelity against code readability [12]. *Temperamento* addresses a narrower deterministic problem: canonical translation among explicitly defined representations, not inference from arbitrary symbolic music. It claims no priority for executable music or for transposition-invariant interval syntax. Its contribution is the combination of paired-chord operations, a separate root-relative operand channel, canonical translation between text and engraved notation, and claim-directed verification of the complete program.

III. WRITING A PROGRAM IN A SCORE

A. The two staves

A *Temperamento* score contains one notated part with two computational staves. Staff 1, Voice, carries operands; staff 2, Base, carries the harmonic events from which operations are read. The implemented score source is a deliberately restricted subset of score-partwise MusicXML, an XML-based interchange format for digital sheet music [13]. It declares exactly these two staves and uses at most one MusicXML voice on each. Event order follows rational score time reconstructed from notes, forward elements, and backup elements, rather than the order in which XML elements happen to appear.

The score may contain more music than the program. A Base onset is computational only when its simultaneous pitch classes form exactly a major or minor triad. Inversion, register, note order, enharmonic pitch-class equivalence, and octave doubling do not change that identity. Every other well-formed Base sonority remains visible in the score and in the compiler's inspection output, but is removed before the computational triads are paired. A C–F–G collection, for example, may be heard or labeled as Csus4 or Fsus2/C; the compiler records the pitches without choosing between those readings. This is a refusal to confuse a small computational alphabet with a general theory of harmony.

B. Harmony as instruction

Let the chord roots be pitch classes r_1 and r_2 in $\mathbb{Z}_{12}$, and let m_1 and m_2 denote major or minor mode. Position on the circle of fifths and distance along it are

$$p(r) = 7r \bmod 12; \quad d(r_1, r_2) = 7(r_2 - r_1) \bmod 12 \quad (1)$$

Because p is linear, $d(r_1, r_2) = (p(r_2) - p(r_1)) \bmod 12$: the quantity is a directed clockwise distance along the circle of fifths. An instruction is the tuple $h = (d(r_1, r_2), m_1, m_2)$. Cells are written dXY , where X and Y are the initial and final modes (M for major, m for minor). Twelve distances and four ordered mode combinations produce forty-eight cells. Seventeen are assigned in the present core; the remaining thirty-one are reserved. A recognized pair that selects a reserved cell is rejected rather than given an accidental or provisional meaning.

The relation, rather than either chord name, determines the operation. If every pitch is transposed by k semitones, the added term cancels:

$$d(T_k(r_1), T_k(r_2)) = 7((r_2 + k) - (r_1 + k)) \bmod 12 = d(r_1, r_2) \quad (2)$$

The same transposition preserves whether a pitch-class set is a major triad, a minor triad, or outside the computational alphabet. It therefore preserves the selection of Base events, their temporal order, their pairing, both modes, and the harmonic cell chosen by each pair.

C. Melody as data

Voice notes belonging to an instruction lie strictly between the onsets of its two Base triads. Each operand begins with a header note whose duration, measured in quarter notes, is a positive integer n . Exactly n non-overlapping payload notes follow. Simultaneous Voice notes are musical but noncomputational. Let r_1 be the root of the first triad of the enclosing pair — not of the score — and let $v_0, \dots, v_{n-1}$ be the pitch classes of the payload notes in score-

time order. The corresponding base-twelve digits and integer are

$$q_i = (v_i - r_i) \bmod 12, \quad 0 \leq i < n$$

$$v = \sum_{i=0}^{n-1} q_i 12^{n-1-i} \quad (3)$$

The pitch of the header is irrelevant; its duration provides the frame. Each payload note is read relative to r_1 , so transposing the Voice note and its reference root by the same amount leaves the digit unchanged. An interval-relative opcode channel would not be enough if operands were still absolute: the chords could move together while the program's literals changed.

D. Invariance of the complete program

Let C be the compiler, and let T_k transpose every pitched event of a supported score s by k semitones without changing its timing or structure. T_k is closed over the supported source language: event classification, pairing, operand framing and digit decoding depend on pitch class alone, pitch class is closed under addition of k in $\mathbb{Z}_{12}$, and MusicXML represents any resulting spelling through its numeric alter attribute. Hence

$$C(T_k(s)) = C(s), \quad \text{for every } k \in \mathbb{Z}_{12} \quad (4)$$

This equality concerns canonical TOScript Core, including every operand, not merely a list of operation names. Base classification and pairing remain fixed; Voice framing and relative digits remain fixed; and the temporal order, labels, and control flow do not move. Every compiled instruction is therefore identical.

The argument generalizes beyond this encoding. A transformation of source code is semantics-preserving only when every stage that contributes meaning survives it: which events are read, how they are grouped and ordered, how operations are selected, and how values and addresses are decoded. Preserving a single coordinate proves only a local property; an absolute literal, label, or addressing channel can still alter the program, as Polyphony, C Flat and Velato's variable names illustrate. This observation also determines how the claim must be tested. Algebraic checks on chord pairs are necessary, but they cannot establish whole-program invariance on their own. Complete scores must be parsed after transformation, their operands decoded again, and their canonical programs compared. The implementation follows this stronger requirement.

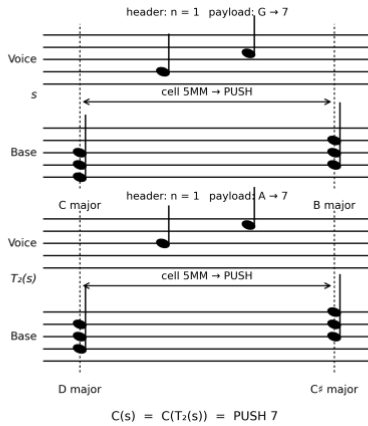


Fig. 1. One command window before and after transposition. Above, C major to B major selects cell 5MM (PUSH); a one-quarter header announces one payload digit, and G taken relative to C denotes 7. Below, T_2 yields D major, C-sharp major and A. Both compile to the identical canonical instruction.

IV. CANONICAL TRANSLATION AND EXECUTION

The implementation accepts TOScript Core, TOScript+, MusicXML, and compressed MXL as program sources. Native MuseScore files enter through MuseScore Studio's command-line conversion [14]; that documentation is inherited from the MuseScore 3 handbook, so the boundary is pinned by checksum rather than trusted by version. MusicXML parsing precedes harmonic interpretation and validates the container, score structure, timing values, staves, voices, chord membership, and computational note events. The standard MusicXML 4.0 partwise document type is accepted and stripped before parsing; entity declarations and all other DTDs, unsafe or duplicate archive entries, tied computational notes, changing staff counts, and unsupported part structures are rejected explicitly.

A. Translation contracts

Let L lower TOScript+ to Core, U lift Core to canonical TOScript+, N notate Core as MusicXML, and C compile a score. For every well-formed Core program P , v0.5.0 requires $L(U(P)) = P$ and $C(N(P)) = P$. The first property follows because U prints only Core operations and numeric labels in a canonical syntax that L parses directly. The second follows instruction by instruction: N chooses the assigned cell for each opcode and writes each operand as base-twelve digits relative to the first root of its pair; C applies the inverse cell lookup and digit decoding. One command measure is emitted per instruction, preserving order. These are canonical round trips, not source-preserving inverses: comments, names, voicing, spelling, engraving, ignored material, and layout are discarded. Corresponding to Fig. 1, the TOScript+ source `push 7` lowers to canonical Core `PUSH 7`.

B. The stack machine

Core executes on a deterministic stack machine over numeric values, with configuration $\langle pc, S, O, \Lambda \rangle$ for program counter, stack, output, and the label map, which resolves each label to the instruction following its declaration. Core operands are non-negative integers; arithmetic may yield negative and non-integer values, and any non-finite result is a runtime fault.

$$\text{PUSH } n: \langle pc, S, O, \Lambda \rangle \rightarrow \langle pc+1, S \cdot n, O, \Lambda \rangle$$

$$\text{ADD}: \langle pc, S \cdot x \cdot y, O, \Lambda \rangle \rightarrow \langle pc+1, S \cdot (x+y), O, \Lambda \rangle$$

$$\text{SWAP}: \langle pc, S \cdot x \cdot y, O, \Lambda \rangle \rightarrow \langle pc+1, S \cdot y \cdot x, O, \Lambda \rangle$$

$$\text{JMC } l: \langle pc, S \cdot c, O, \Lambda \rangle \rightarrow \langle pc', S, O, \Lambda \rangle; \quad pc' = \Lambda(l) \text{ if } c \neq 0, \\ \text{else } pc+1$$

SUB, MUL and DIV follow the shape of ADD. NOT, AND and OR treat zero as false and any non-zero value as true; EQ tests equality of values. All four push 0 or 1. Binary operations remove the right operand before the left. JMC consumes its condition, OUT consumes its value, and every committed jump has a defined target because labels are resolved before execution. Runtime faults include stack underflow, division by zero, non-finite arithmetic, malformed instructions, and exhaustion of configurable limits on steps, stack depth, output, integer magnitude, and captured trace.

Table I writes each cell as dXY, with M and m denoting major and minor endpoint modes.

TABLE I. ASSIGNED TOSCRIP Core CELLS

Cells	Operations	Operand
0-4MM	ADD, SUB, MUL, DIV, NOT	—
5-7MM	PUSH, POP, DUP	PUSH: n
8-10MM	AND, OR, EQ	—
11MM	LBL	label
0mm, 1mm	JMC, JMP	label
2mm	SWAP	—
4mm, 8mm	END, OUT	—

C. Implementation, inspection and availability

The Python implementation exposes TOScript Core and TOScript+ parsers, canonical TOScript+ lifting, Core-to-MusicXML notation, score compilation, execution, command-line tools, and inspection formats. Canonical serialization permits recovered programs and transformed scores to be compared exactly. Score-derived instructions retain measure, beat, rational time, harmonic cell, and contributing Voice onsets; ignored Base events retain their pitches and locations. Inspection can trace an instruction such as PUSH 7 to its two harmonies and melodic figure, while text, JSON, and local HTML views expose the mapping and bounded execution trace. The implementation, corpus, direction packages, and reproduction script for the v0.5.0 paper artifact are publicly available in Zenodo under DOI: 10.5281/zenodo.21763642.

These facilities establish an inspectable compiler and canonical notation path, not a general-purpose score editor. MuseScore remains the notation environment and external rendering boundary. Nor does canonical translation establish that composers or performers find the resulting scores useful. It proves consistency of the implemented representations, not their musical adequacy.

D. Reproducibility

The reproducibility procedure regenerates deterministic examples, direction packages, metadata, and documentation; runs formatting, strict typing, and coverage-gated tests; builds and clean-installs distributions; exercises the command-line interface; and compares two release-archive builds. Its final audit is read-only and verifies the exact committed tree. MuseScore lies outside the byte-deterministic core: a checksum-pinned MuseScore Studio 4.7.4 boundary renders every MusicXML example, recompiles the resulting MSCZ files, and checks canonical program and runtime equivalence. Presentation bytes are not claimed identical across versions, platforms, fonts, or sound profiles.

V. VERIFICATION AND EVALUATION

Verification separates the translation contracts from the transposition property, and separates evidence about the algebra from evidence about the pipeline. Table II reports both, pipeline evidence first.

A. Pipeline evidence

Every assigned cell is written as MusicXML in all twelve transpositions, with nontrivial operands where required. These 204 cases traverse parsing, score-time reconstruction, event classification, triad recognition, cell lookup, operand decoding, control-flow validation, and canonical rendering. Three complete programs — arithmetic, conditional, and iterative — are generated in every chromatic transposition,

with reversed voicing and doubled roots; all thirty-six variants compile to identical canonical Core and produce the same output. Three packages exercise the translation chain from different sources: MusicXML to Core to score to Core; Core to score to Core; and TOScript+ to Core to score to Core. Each records expected execution and a manifest. A separate checksum-pinned workflow renders every committed MusicXML example through MuseScore, recompiles the result, and compares canonical Core and runtime behavior. Further tests confirm that ignored Base sonorities and simultaneous Voice events do not alter execution.

B. Algebraic oracles

The expected circle of fifths is frozen in the tests as the literal sequence C, G, D, A, E, B, F-sharp, C-sharp, G-sharp, D-sharp, A-sharp, F, rather than generated with the production distance function. The operation map is likewise duplicated as a test-side reference. These oracles cover all forty-eight cells. Each cell is then transposed through all twelve pitch classes, and a further set of cases recognizes the triads from individual note events while varying note order, register, inversion, and root doubling. These enumerations are exhaustive rather than independent: each case restates (2) at one more pitch level. They are reported for completeness, not as the weight of the argument, which rests on the pipeline rows.

TABLE II. CLAIM-DIRECTED VERIFICATION

Family	Kind	Cases
Assigned-cell MusicXML round trips	pipeline	204
Whole-program transposition	pipeline	36
Canonical source-direction packages	pipeline	3
Harmonic-cell oracle	algebraic	48
Cell $\times$ chromatic transposition	algebraic	576
Recognition and voicing transformations	algebraic	576

C. Limits of the evidence

The broader suite covers text parsing and lifting, score generation, malformed containers, timing, operands, ignored events, reserved cells, labels, runtime limits, command-line behavior, source maps, archives, and the MuseScore boundary. At release v0.5.0, the suite comprises 1,713 tests and reports 98 percent branch coverage; CI exercises Python 3.11 and 3.13 under Linux, macOS, and Windows. The corpus contains five tutorials, four executable compositions, three conversion packages, and twelve committed transpositions, including Hello, World! output, conditional control, iteration, and a fixed three-disc Towers of Hanoi solution. These artifacts show that the pipeline can author, convert, inspect, render, and execute supported programs; they do not establish musical quality.

These observations bound the current results: The round-trip contract $C(N(P)) = P$ is verified exhaustively at the level of single instructions but only over a small hand-built corpus at the level of whole programs; property-based generation of random well-formed Core programs would close that gap, and will be implemented in future releases of the language. The tests share implementation structure with the system they test. The harmonic oracle duplicates the published mapping rather than calling production helpers, but the translation tests are not an independent compiler and the implementation is not mechanically verified, so a correlated opcode or notation

error could survive a canonical round trip. Finally, external validity is limited by the restricted grammars and the relatively small complete-program corpus, and the invariance property concerns symbolic pitch classes, not enharmonic spelling, acoustics, performance, or listener understanding.

VI. DISCUSSION

The system has two preservation obligations. Canonical translation preserves Core while normalizing surface information. Musical transposition preserves Core because every pitch-bearing semantic channel is relative. While the algebra is elementary—and would be uninteresting in a language without pitch-borne literals—it becomes relevant because the implementation has an event-selection stage and two distinct pitch-borne channels, one of which carries numbers. Preserving only chord distance would leave absolute Voice operands free to change, which is exactly how invariance fails in both Polyphony and C Flat. The proof therefore covers selection, pairing, operation, data, and order.

The treatment of nontriadic material is equally important to the musical premise. A suspended chord, dyad, cluster, or seventh chord is not malformed music; it is simply (and explicitly) outside the present computational alphabet. Reporting and ignoring such events avoids forcing a context-dependent harmonic analysis into a deterministic compiler, and it distinguishes *Temperamento* from harmonic languages that admit only the chords they can execute. The score can therefore carry material that matters to composition and performance without pretending that the machine understands it.

The language remains narrow-yet-extensible: one part, two staves, one MusicXML voice per staff, major and minor computational triads, non-negative Core operands, and a small TOScript+ subset. Canonical translation does not recover comments, names, spelling, voicing, layout, engraving, or ignored material. *Temperamento* is not a general harmonic analyzer or a claim about Western notation's universality. Compositional practice and perceived legibility require work with musicians; richer values, procedures, and alternative readings require longer treatment. The accompanying artifact documents a constructive Turing-completeness argument for idealized Core through a two-counter-machine translation; the present argument does not rely on that result.

VII. CONCLUSION

Temperamento provides canonical translations among TOScript+, TOScript Core, and a two-stave score. Core can be lifted to canonical TOScript+ and realized as MusicXML whose recompilation recovers identical canonical Core. Within score form, ordered pairs of major and minor triads encode operations through circle-of-fifths distance and endpoint mode, while Voice figures encode operands as base-twelve offsets from the first triad root of the pair. Global transposition preserves both relations and therefore the complete program.

The contribution is not transposition invariance or relative pitch literals in isolation: Fugue already combines both, and other languages preserve transposition through different channels. It is instead the auditable integration of paired-chord operations, a numeric channel written in pitch and read relatively so that no absolute residue remains, canonical translation between text and MusicXML score notation rather than a MIDI-only source pipeline, and whole-program

verification, immersed in a computational music and language design research program. Explicit grammars, canonical translators, a source-mapped compiler, deterministic interpreter, literal harmonic oracles, complete-score transformations, cross-platform tests, and checksum-pinned notation-tool verification make the claims inspectable and falsifiable.

REFERENCES

- [1] N. Ayzikovsky, “Fugue,” language specification and implementation, 2005. [Online]. Available: esolangs.org/wiki/Fugue [Accessed: Aug. 2, 2026].
- [2] D. Temkin, “Velato,” language documentation and compiler, 2009. [Online]. Available: github.com/rotytooth/Velato [Accessed: Aug. 2, 2026].
- [3] T. Aberbach, “Schoenberg: The MIDI esoteric programming language,” Jun. 2025. [Online]. Available: tomeraberba.ch/schoenberg [Accessed: Aug. 2, 2026].
- [4] A-jar-of-peanuts and H. Sun, “Polyphony,” language documentation and implementation, 2024. [Online]. Available: esolangs.org/wiki/Polyphony [Accessed: Aug. 2, 2026].
- [5] K. Robertson, “Tone-deaf,” reference interpreter, ver. 0.3.0, Mar. 2018. [Online]. Available: pypi.org/project/tonedeaf/ [Accessed: Aug. 2, 2026].
- [6] NicksterSand, “C Flat,” language documentation and interpreter, 2020. [Online]. Available: github.com/NicksterSand/Cflat [Accessed: Aug. 2, 2026].
- [7] Syuparn, “Cholc,” language specification and reference interpreter, 2025. [Online]. Available: esolangs.org/wiki/Cholc [Accessed: Aug. 2, 2026].
- [8] T. Magnusson, “Scoring with code: Composing with algorithmic notation,” *Organised Sound*, vol. 19, no. 3, pp. 268–275, 2014, doi: 10.1017/S1355771814000259.
- [9] J. C. Martinez Nieto, “The algorithmic score language: Extending common Western music notation for representing logical behaviors,” Ph.D. dissertation, Georgia Institute of Technology, Atlanta, GA, USA, 2018.
- [10] D. Temkin, “Language without code: Intentionally unusable, uncomputable, or conceptual programming languages,” *J. Science and Technology of the Arts*, vol. 9, no. 3, pp. 83–91, 2017, doi: 10.7559/citarj.v9i3.432.
- [11] J. Singer and S. Draper, “Let’s take esoteric programming languages seriously,” in *Proc. 2025 ACM SIGPLAN Int. Symp. New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! ’25)*, 2025, pp. 213–226, doi: 10.1145/3759429.3762632.
- [12] Y. Kim, A. Gandhi, D. Chung, G. Neubig, and C. Donahue, “Decompiler: Learning to decompile symbolic music to programs,” preprint, 2026, arXiv:2607.01849, doi: 10.48550/arXiv.2607.01849.
- [13] W3C Music Notation Community Group, “MusicXML 4.0,” Final Community Group Report, Jun. 2021. [Online]. Available: w3.org/2021/06/musicxml40 [Accessed: Aug. 2, 2026].
- [14] MuseScore Studio, “Command line usage,” MuseScore Studio Handbook. [Online]. Available: handbook.musescore.org/appendix/command-line-usage [Accessed: Aug. 2, 2026].