# WesterParse: Software for Evaluating Westergaardian Species Counterpoint

Documentation 2020


**Abstract**

This report describes WesterParse, a software program designed to evaluate tonal species counterpoint exercises in the version developed by Peter Westergaard (1975). The program is written in Python and relies on the `music21` toolkit. The core component is a parser that interprets the pitch-syntactic structure of contrapuntal lines. The parser compiles a set of possible syntactic interpretations and reports whether the line is valid. If asked, the program can display the interpretations in a notation program such as MuseScore. A separate component of WesterParse is a voice-leading evaluator that can test the counterpoint of simple species for compliance with Westergaard's rules of voice leading. A companion web application allows WesterParse to be used as a pedagogical tool.

## 1 Westergaardian Species Counterpoint

In the early 1970s Peter Westergaard developed an approach to understanding tonal music centered on the cognition of linear syntax and counterpoint rather than chords and harmony. The goal of the enterprise was to give students "the ability to understand the complex and varied voice-leading patterns of actual eighteenth- and nineteenth-century music in terms of the simpler patterns available under the artificial constraints of species counterpoint" (Westergaard 1975, vii). What distinguishes Westergaard's approach to species counterpoint from traditional forms is the rigorous fashion in which the individual line is regarded as (and constructed to be) "an entity with its own structure, unfolding in time" (29).[1] In Westergaard's formulation, each line in the contrapuntal exercise must be constructed by applying rules that generate notes with certain order-dependent functions (see Ex. 1).[2] The generative rules constitute a syntax for lines that prolong a single triad. Although formulated as rules for composing new lines, the rules can be understood as rules that guide a listener who attributes structure to lines that have already been composed, and therein lies an important link to understanding classically tonal music.[3]

The forms of simple linear structure articulated in the A rules are just those that Heinrich Schenker posited as the components of the Ursatz: a primary upper line and a bass line. The kernel structure of a primary upper line consists of three functions: a tonic pitch that acts as the final element of the structure; a tonic-triad pitch that lies above the final element and acts as the initial structural element; and pitches in the scale that fill the span between the initial and final elements with a complete stepwise motion. Bass lines have a kernel structure consisting of an arpeggiation from the tonic degree to the dominant and on to another tonic degree. (Westergaard adds a third, less constrained type of line

1. Westergaard also replaced the Fuxian modal systems with the systems of classical major-minor tonality.

2. In this article, I follow Westergaard's text in referring to rules with the letters A and B; WesterParse, however, uses the letters S and E.

3. The simple lines of species counterpoint are rhythmically uniform and relatively brief, so the basic syntactic system is likewise simple. The syntactic system for complex, rhythmically differentiated lines, on the other hand, involves rhythmically and contrapuntally sensitive rules. Parsing such lines lies well outside the current scope of the WesterParse project.

**Primary upper lines: the basic step motion**

A1. The final pitch in the basic step motion must be a tonic.

A2. The first pitch in the basic step motion must be a tonic triad member a third, fifth, or an octave above the final pitch.

A3. These two pitches must be joined by inserting the pitches of intervening diatonic degrees to form a descending step motion.

**Bass lines: the basic arpeggiation**

A1. The final pitch of the basic arpeggiation must be a tonic.

A2. The first pitch of the basic step arpeggiation must be a tonic.

A3. The middle pitch of the basic step arpeggiation must be a dominant either a fifth above or a fourth below the final tonic.

**Secondary structures**

B1. Any triad pitch may be repeated.

B2. A neighbor may be inserted between consecutive notes with the same pitch.

B3. Any triad pitch may precede the first pitch [of the basic step motion] or may be inserted between any two consecutive pitches so long as no dissonant skip and no skip larger than an octave is created.

B4. Any two consecutive notes forming a skip may be joined by a step motion.

Example 1: Rules for composing lines (Westergaard 1975, 56–58).

that begins and ends on a tonic-triad pitch; I call this a generic line.)

The kernel structures can be elaborated by the addition of syntactically dependent elements (the B rules): tonic-triad pitches may be repeated, new tonic-triad pitches may be inserted (subject to some constraints), and stepwise transitions may be created between consecutions of identical pitches (neighboring motions) or nonadjacent pitches (passing motions).[4]

In Westergaardian species counterpoint, there are two domains of rule: (1) the generative syntactic rules shown in Example 1, which govern the construction of the individual melodic line, and (2) a group of fairly traditional voice-leading rules, which govern the moment-to-moment contrapuntal interaction of melodies that unfold simultaneously. Composing counterpoint involves negotiating competing demands in these two domains. Because the choice of each and every note in a species counterpoint exercise alters the overall set of compositional contingencies, the student faces a complex process of adjudicating the consequences of every choice, both retrospectively and prospectively, as well as vertically and horizontally. Learning to follow and negotiate the rules of species counterpoint brings the student toward the goal of being able to understand and articulate the linear-contrapuntal perception of classically tonal music first described by Schenker.[5]

4. The A and B rules can be derived from two fundamental operations, which I call INSERT and STEPTO. The INSERT operation ranges over tonic-triad pitches, and STEPTO ranges over scale pitches. Since all tonic-triad pitches are included in the scale, those pitches may have either an INSERT or STEPTO function, depending upon the context. Many types of melodic event can be derived from these two operations: arpeggiation, anticipation, and voice exchange, for example, are species of INSERT, while incomplete neighbors and appoggiaturas are species of STEPTO.

5. See Peles 1997 for an excellent introduction to Westergaard's ambitious theory of classical tonality.

## 2 WesterParse

WesterParse is a software program that evaluates the pitch-syntactic structure of the simple tonal lines found in Westergaardian species counterpoint exercises, three samples of which are provided below in Example 2. WesterParse also includes a module for evaluating voice leading. In addition to determining whether the lines of an exercise conform to the rules of linear syntax and voice leading, WesterParse also constructs a set of legitimate interpretations for every line that it evaluates. The program is written in Python and relies on the `music21` toolkit developed at MIT by Michael Cuthbert and his colleagues.[6] The input to the program is a MusicXML file containing a counterpoint exercise. The output of the program comes in three forms: messages displayed onscreen to student users, interpretations of lines displayed in a simplified Schenkerian notation, and arrays of data.

In this article, I first explain the motivations for devising a computer program that evaluates species counterpoint exercises for conformity with the two domains of rules. I then provide links to the program's code and documentation, as well as the pedagogical website; I also display some samples from a corpus website. In the next section, I describe how WesterParse prepares musical input for evaluation. The main section of the article describes how WesterParse evaluates lines for conformity with the line-writing rules in Example 1, and this is followed by a shorter section that describes WesterParse's evaluation of voice leading (counterpoint). The article concludes with comments on testing WesterParse results for conformity with Westergaard's interpretations, some thoughts about further development of the project, and acknowledgement of contributions to the project.

### 2.1 The Purpose and Scope of WesterParse

Why write a program that can analyze and evaluate species counterpoint? Is it not, after all, the point of species counterpoint to inculcate that capacity in students? When I set out to write this program, it was partly out of curiosity. Westergaard's rules are so simple and clear that I had long thought it possible to write a program capable of analyzing contrapuntal lines in terms of them. It turned out to be more difficult than I thought, because many aspects of cognition and interpretation that are implicit in Westergaard's approach needed to be made explicit, such as the internal structure of syntactic units, the nature of syntactic dependency, and particulary the role of memory in maintaining a grasp of head tones and transitions in progress.[7] So, in retrospect, one purpose in writing WesterParse was to refine the Schenkerian/Westergaardian theory of musical cognition by articulating essential components of what it means to hear linear prolongation.

The second purpose was pedagogical. I have been teaching Westergaard's species counterpoint to undergraduates for over twenty-five years. As in traditional lessons, students compose exercises and bring them to class for evaluation and feedback. There is often a time lag of several days between the act of composition and the reception of feedback, and days or even weeks may elapse between receipt of feedback and work on revising the composition. To give students more frequent and more timely feedback, I thought

6. Cuthbert and Ariza 2010; see the Music21 website.

7. Readers familiar with Schenker will know, of course, that he frequently mentions the mental retention of the linear progression's head tone.

it would be useful if evaluation and feedback could be built into a software program and made available at the click of a button, while the student is still in the flow of composing. The project required two components: a software program that could evaluate lines and counterpoint, and a web application in which students could compose counterpoint and submit it for evaluation.

In the frontend web application, the student begins by deciding how many measures of counterpoint to write, how many parts, and what key signature to use. The webpage then presents the student with a blank set of staves. The student enters notes as desired, and a separate edit mode allows the student to go back and change notes and add or delete measures. When the exercise is complete, the student can ask the program to evaluate the lines or evaluate the voice leading. Clicking "Evaluate Lines" sends a representation of the line in MusicXML to the WesterParse program on the server, which then evaluates the construction of the lines and returns a report that is displayed on the web page. The report indicates whether the lines are generable according to Westergaard's rules and, if not, where errors were encountered. A similar report is issued when the student clicks "Evaluate Voice Leading." Finally, the student can click a button to download the exercise as a MusicXML file, open that file in, say, MuseScore, and add analytical annotations (slurs, ties, rule names).

In its current form, WesterParse can evaluate first and second species in two or three parts, third and fourth species in two parts, and a version of harmonic second species in two parts. The version of third species implemented in WesterParse allows repetition and insertion of consonant non–tonic-triad pitches within a measure, which is a deviation from Westergaard's version. The version of harmonic second species has a similar extension, allowing the repetition and insertion of pitches within the governing triad of a harmonic span (I, V, or II); harmonic species also has requirements for coordinating the kernel structures of the bass and primary lines, corresponding roughly to Schenker's Ursatz forms.[8]

### 2.2 The Code, Documentation, and Web Implementation

The full code for WesterParse is available on GitHub. Documentation of the software is available at ReadTheDocs. Readers interested in testing the pedagogical implementation may visit Westergaardian Species Counterpoint Online, where there is also a link to the complete sets of rules for linear construction and voice leading.

### 2.3 The WesterParse Corpus Site

To test and demonstrate the reliability of the WesterParse parsing algorithms, I assembled a corpus of 145 examples. About half are drawn from Westergaard's text: 48 single lines and 29 complete examples of species counterpoint. The corpus also includes an additional collection of 27 lines and 41 counterpoint compositions, some of my own invention and others written by students. Readers interested in exploring the corpus on their own are welcome to use the WesterParse Corpus Viewer [under development].

8. The post-Westergaard extensions were originally developed by Fred Everett Maus and Marion Guck, respectively, in the early 1990s.

Example 2: WesterParse's final-state interpretation of three sample exercises.

To illustrate the parsing capabilities of WesterParse, I have selected three samples from the corpus, shown in Example 2. The first of these is taken from Westergaard's text, where it is intended to exemplfy various issues involving similar motion to a perfect fifth. The upper line is interesting for the way in which the arrival of B♭ in bar 7 requires the parser to reinterpret the syntax; having initially decided that the E in bar 2 passed to the F in bar 6, it must then reject that interpretation in order to connect B♭ to the A in bar 3, effectively postponing the resolution of E until the line arrives on F in bar 11. The bass line is notable for the long descent from D to A, interrupted by a number of secondary structures.

The bass line of the second example also requires the parser to revise its interpretation in midstream. The B in bar 4 initially seems to resolve the passing C in bar 2, but the low D♯ puts that into question, requiring that the preceding B be demoted to an insertion and that C be returned to the list of open transitions, where it will remain until the arrival of B in bar 6. Shown here is one of the two interpretations that WesterParse generates for the upper line; the other takes the initial G as A2.

The upper line of the third example illustrates some of the complexities of third species, where the rules allow for the elaboration of local harmonies, as can be seen in bars 5 and 7.

## 3 Preparing a Source for Evaluation

How, then, does WesterParse operate? Since there are two domains of rule (linear syntax and voice leading), there are also two evaluative functions: EvaluateLines and Evaluate-Counterpoint. Both functions take as input a MusicXML file and convert it to a `music21`

Score object. After conversion, WesterParse can access the content of the source file in a variety of ways: parts, measures, notes, simultaneities, and so forth. A significant advantage of using `music21` is that its already robust collection of musical objects and relations is easily extensible and customizable. And since MusicXML is the standard exchange format for programs such as Finale and MuseScore, WesterParse is not wedded to a particular notation program.

### 3.1 Creating the Global Context Object

WesterParse creates a Global Context object to hold the `music21` Score object and its evaluations. It automatically takes several steps to prepare the score for evaluation. First, the score is checked to ensure that it has at least one Part object and that each part contains notes. If the score validation fails, an error is recorded and the program is exited. Next, some recordkeeping tasks are performed: an id number is assigned to each part in the context and the rhythmic species of each part is identified; a list is created to collect general errors; and a dictionary is created to collect specific errors that arise when the individual parts are parsed. Then, some setup operations are performed on the individual notes in each part. A position index is assigned to each note; this is the primary note reference used during parsing. A Rule object is attached to each note to hold a reference to the A or B rule used to generate the note. Each note is also assigned a Dependency object, which stores information about the note's syntactic relationships, such as left and right heads as well as references to other notes that are dependent upon it. A passing tone in a third-progression, for example, stores references to the notes that initiate and conclude the passing motion, and those terminal notes both contain a reference to the intervening passing tone. And, finally, each note is provided with a Consecutions object, which stores information pertinent to how the note is approached and left, including the size and direction of the intervals to the left and right of the target note as well as the category of the consecution (same, step, skip).

Interpreting the syntactic structure of a Westergaardian line and the voice leading of counterpoint requires two frames of reference: a tonic triad and its associated scale. So the next step in preparing a context for evaluation is to determine a key for the context. If the user has provided the key (keyname and mode), it must be validated; if not, the key must be inferred from the composition itself. These tasks are handled by the KeyFinder module (see below). If no valid key is established, an error is returned and the program is exited. If a key is successfully validated or inferred, a referential tonic scale degree is determined for each part and then a Concrete Scale Degree object is assigned to each note in the part.[9] The scale and tonic triad are also provided to the part for use in parsing.

If one of the parts is in third species, the harmonic content of each downbeat is calculated and stored in a dictionary in the Global Context. If the input is harmonic

9. Unlike the usual scale degree class, a concrete scale degree (CSD) is the position of a pitch relative to a specific tonic pitch in a scale. The referential tonic has a value of 0. Hence, the value of the fifth above the tonic is 4 and the fourth below is -3. In addition to a value, a CSD has a representation called degree, which returns the traditional name of the scale degree ($\hat{5}$, in the two cases just mentioned). A CSD also has a direction; in major scales, all degrees are bidirectional, while in minor the lower forms of degrees $\hat{6}$ and $\hat{7}$ are descending, the raised form of $\hat{6}$ is bidirectional, and the raised form of $\hat{7}$ is ascending.

species, the user will have provided measure numbers for the start of the predominant and dominant spans; the program then uses these values to calculate the spans of the initial tonic, predominant, dominant, and closing tonic spans and stores them in a dictionary.

### 3.2 The KeyFinder

The KeyFinder module examines a set of parts and either validates a key provided by the user or infers an appropriate key. The resulting key is passed to the Global Context.

To infer a key, the KeyFinder examines the context part by part, looking for several features that must be true of any valid Westergaardian line. The KeyFinder then sifts through the possibilities to see whether there is agreement among the parts. In a valid Westergaardian line, the first and last pitches must belong to the tonic triad, all pitches must belong to the scale, and at least one pitch in any leap must belong to the tonic triad.[10] The KeyFinder begins by finding all of the scales in which these three criteria are satisfied in each part. It is also the case that in a valid Westergaardian line, every non-tonic-triad pitch must eventually be displaced by step to a tonic-triad pitch, so the KeyFinder examines each part to find all of the keys in which only tonic-triad pitches are left hanging.[11] The lists resulting from these two steps are sifted to see which possibilities are common to all parts. If there are still multiple options for key of the context, the list is winnowed using two preference rules: (a) prefer most lines to end on the tonic degree, and (b) prefer major rather than minor if ambiguously mixed. If winnowed to one option, the appropriate mode and keynote are assigned to the context, otherwise an exception is raised and the failure to find a single key is reported to the user.

A user has the option of selecting a particular key. This is useful when the context is tonally ambiguous. Validation of a user-provided key tests for nominal validity ('Q-sharp diminished' is not a valid option) and then tests the selection against the criteria used in key inference. If the parts do not actually project the selected key, the program will report which criteria have not been met.

## 4 The EvaluateLines Function

### 4.1 Overview

WesterParse evaluates linear syntax for each selected part and then compiles a set of interpretations. Unless the user has selected a specific part for evaluation, WesterParse selects all parts for evaluation. If the user has selected one part for evaluation or there is just one part in the score, the user may also select a type of line.

WesterParse runs the Parser for each of the selected parts and collects errors, if any; it determines whether all of the selected parts are generable; and it creates a report to display to the user.[12] If all parts are generable, WesterParse assembles interpretations of the context

10. In the version of third species that I teach, the rules permit arpeggiations within local harmonies, so the last condition is relaxed for the third-species line.

11. Westergaard's line-writing rules ensure that nothing but tonic-triad pitches are left hanging by the time a line has concluded. David Lewin (1983) proposed a more radical rule, one that leaves no non-tonic (non-final) pitches hanging.

12. WesterParse also includes a method for extracting the data of each successful parse and printing it to a

from the parses of the individual lines. In the current implementation, WesterParse also winnows the sets of parses using some of Westergaard's counterpoint preferences for 2- and 3-part counterpoint. If there is only a single part, WesterParse returns all of the possible interpretations of the part as a primary, bass, or generic line. If there are two or three parts, WesterParse selects only bass line parses for the lowest line and looks for combinations with an upper line that can be interpreted as a primary line. If the user has chosen a specific part to interpret, it returns only the interpretations of that part; and if the user also specified a type of line, it further limits the output to just that type of line.

Finally, WesterParse outputs the interpretations of the context based on the value of the *show* variable. This could mean translating the content of the parse into musical notation (using a form of Schenkerian notation) or preparing a textual report. The program also includes a method for displaying the parse of a single line level by level, in the manner used by Westergaard; this method is not currently accessible. In the online pedagogical implementation, the value of *show* is set by default, so that the student user sees only a textual report on the parses, which tells the student whether each part is generable and, if not, what errors were encountered. Since the activity of interpretation is essential preparation for analyzing real music, the student is responsible for providing their own legitimate interpretations of the linear syntax of each part.

### 4.2 The Parser

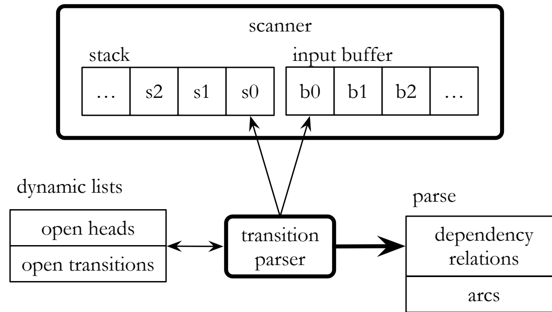
With approximately five thousand lines of code, the Parser is easily the most complex module in WesterParse. The Parser starts by preparing placeholders for parses and errors. If a line type was provided by the user, it limits the operation to that particular type; otherwise, it examines the line and tries to determine which of the three line types is plausible. Bass lines, for example, must begin and end with the tonic degree and also contain a dominant degree. Primary lines must end on the tonic degree and must contain a tonic-triad pitch that lies a third, fifth, or octave above the final tonic.

The Parser then conducts a preliminary parse, irrespective of line type. In effect, the preliminary parse determines whether the line can be generated using the four B rules (see Ex. 1). If preliminary parsing is unsuccessful, the operation of the Parser is interrupted and the errors are reported to the user. Otherwise, the Parser proceeds to decide on a set of possible structural interpretations (the A rules) using the inferred line types; it creates a Parse object to store each interpretation. During the preliminary parse, the Parser identifies notes that can serve as part of a basic structure. For example, there may be several dominant degrees in a bass line, any one of which could serve as a component of the bass arpeggiation; and in a primary line, there may be several different tonic-triad pitches that could serve as the initiation of the basic step motion (Urlinie). The Parser tests the plausibility of each structural candidate. The Parser also uses several different methods for constructing primary upper lines, so it may end up testing several dozen possible interpretations of a given part.

The successful parses are then gathered by line type; duplicate interpretations are removed, and the winnowed set is passed back to the Context object.

---

json file; the json files can then be subjected to data analysis.



Example 3: The architecture of WesterParse's line parser.

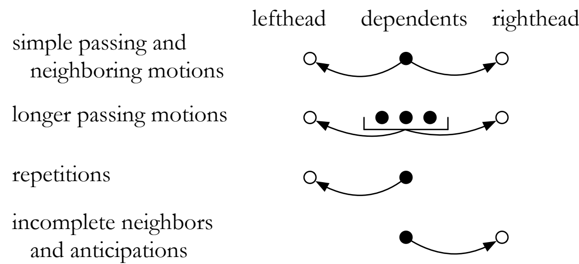
### 4.3 The Architecture of the Parser

WesterParse is structured as a transition-based dependency parser. See Example 3. The process begins with an input buffer, loaded with all of the notes in the line, and a stack, which is empty. A simple scanning function shifts notes from the buffer onto the stack, one at a time, until the buffer is exhausted. At each step, the Parser examines the top element of the stack and the next element in the buffer and selects an interpretive action based on examination of the current interpretive state.[13]

In addition to the stack and the input buffer, the Parser maintains a set of dynamic data structures (open heads and open transitions) and gradually compiles partial parse of the line. The dynamic lists keep track of open syntactic relations. They change in content during the course of interpreting the line. On the list of open heads are all the notes that can currently initiate a new step motion or get repeated. The list of open transitions contains notes that are in yet-to-be-completed step motions. Think of the opening notes of Fux's Dorian cantus firmus, D F E. When the Parser interprets the transition from F to E, it has already placed D and F on the list of available heads and decides that E is an open transition, on its way somewhere. The Parser has also decided that E is stepping away from F, which is to say, E depends on F; the dependency relationship is recorded in the Dependency objects attached to the two notes. So, as the line transitions from one note to the next, the parser is beginning to figure out dependency relations among the notes, hence the term transition-based dependency parser.

Dependency relations are central to the content of the interpretation, the so-called parse. So, what is a dependency relation? Take a consecutive pair of notes, X and Y. We will say that Y is syntactically dependent upon X if X is mentioned in the syntactic description of Y. Since X precedes Y, we will also say that X stands "to the left" of Y. If we find that Y repeats X, then Y is dependent on X. Repetitions are always dependent on a lefthand note, a so-called lefthead. Passing tones, by contrast, are dependent upon notes to the left

13. By contrast, the linguistics parsers described in Jurafsky and Martin 2008, Chapter 13, "Syntactic Parsing" instead examine the top two elements of the stack. And in the linguistics parsers, the syntactic category of each element has already been assigned prior to parsing, whereas WesterParse assigns and revises syntactic classifications during the parsing process itself.




Example 4: Four types of arc. Open circles indicate arc heads; filled circles, dependent members of arcs.

and the right. They have a lefthead and a righthead.[14] Take the succession [F E D]. One possible syntactic interpretation of the succession is this: "E passes between F and D." Under this interpretation, F and D are mentioned in the description of E. F is the lefthead of E. D is the righthead of E. And E is a dependent of both F and D.

A set of notes interconnected by dependency relations forms a syntactic unit called an arc. There are several types of arc (see Ex. 4): some arcs, like passing and neighboring motions, have heads to the left and the right with dependents in between. Others have only a lefthead (e.g., repetition), and some notes (e.g., insertions) do not have a head, per se. In more complex tonal lines, it is possible that an arc might have only a righthead (e.g., incomplete neighbor or anticipation).

As the Parser works its way through the line, then, it sets dependency attributes for each note, storing the information in the Dependency object that is attached to each note in the parse. As completed arcs accumulate, they are stored in a separate list.[15]

When the parse is finished, the Parser assigns a Rule object to each note in the line. This object stores the name of the note's syntactic function and its structural level.

### 4.4 The Parser Illustrated: Fux's Dorian Cantus Firmus

Before describing the actual mechanism of the transition parsing algorithm, let us look at how the Parser works its way through Fux's Dorian cantus firmus. I regard this as a hypothetical model of how a Westergaardian listener perceives linear prolongation.

The first half of the process is illustrated in Example 5. To initialize the parser, the first note D is moved onto the stack and is also added to the list of open heads; at this stage

14. This is a significant point of difference between language and music. In language, each word other than the root note has but a single head, and so linguistic dependency can be represented in strictly binary trees or graphs. In music, however, some notes are clearly transitional between an earlier and a later note. Restricting theory to binary relations leads to false dependency choices, as can be seen, for example, in Lerdahl and Jackendoff's account of neighboring motions (see Lerdahl and Jackendoff 1983, 113–14 and 185–87).

15. Once the parse is complete, the arcs are converted into Arc objects, which store information about their content and type. A list of Arcs forms part of the parse data that can be extracted for further analysis. In the course of writing this article, I discovered that the Dependency objects do not actually store a full account of a note's relationships to other notes in the line; once a note has been assigned to an arc, its dependency relations can be overwritten as the parser searches for new connections. If it proves somehow useful to track the complete set of a note's dependency relations, I may rethink how the parser stores this information, but for now it can be gleaned from the arcs.

Example 5: Parsing Fux's Dorian cantus firmus, states 0–5.

there are no open transitions and no completed arcs. From this initial state, the parser scans forward, listens to F (state 1), and adds F to the list of open heads. The parser scans forward again (state 2), listens to E, and adds it to the list of open transitions. Several things happen after the parser hears the fourth note, D (state 3). The parser connects E to this D as a righthead and then listens back through the list of open heads, searching for a lefthead. The parser is biased toward finding a lefthead in proximity, so it looks no further than F. The parser then creates an arc, [F E D], which is added to the parse.[16] Meanwhile, E is removed from the list of open transitions and the most recent D is added to the list of open heads.

The parser listens to G (state 4). Realizing that the only available precursor to G is the F, it removes the intervening D from the list of open heads. The list is pruned, we might say. The parser also adds G to the open transitions. When it listens to F (state 5), it makes an arc [F G F], adds this arc to the parse, and then prunes back the list of open transitions. In general, when the parser finds an arc, it prunes interior elements from the list of open transitions and prunes embedded heads from the list of open heads.

16. When parses are represented in musical notation, notes of the basic structure are identified by rule number, ties connect repetitions to their heads (ties are dotted if no neighbor intervenes), slurs connect the notes of a passing motion, and parentheses enclose insertions; a left or right parenthesis is removed if a step connection is made to or from the inserted note.

Example 6: Parsing Fux's Dorian cantus firmus, states 6–10.

In subsequent stages, shown in Example 6, the parser hears A and adds it to the list of open heads, then hears G and adds it to the list of open transitions. Upon hearing F (state 8), the parser uses it as the righthead of a new arc, [A G F], adding F to the open heads, and removing G from the list of open transitions. Upon hearing E (state 9), the parser adds it to the list of open transitions. When the parser hears the final D (state 10), it creates an arc, [F E D], adds the final D to the open heads, and removes E from the list of open transitions.

The parse, at this stage, is nevertheless incomplete. The parser has compiled a list of syntactic units (arcs), but at least one note (the first) does not belong to any arc, and the arcs are not yet integrated into an overarching structure. To integrate the arcs into a complete interpretation, the parser has to decide what type of line it wants to hear.

Suppose that the parser is told to see whether the line makes sense as a bass line. If so, the line will have to end and begin on a tonic pitch, and in between the beginning and the end it will have to touch on $\hat{5}$. The rules imply that these three notes are conceptually prior to all other notes in the line. Which is to say, they are not dependent upon any other notes. In the way that the rules are framed, A1 is something like a root node. A2 is partially dependent on A1 (at least in terms of order), and A3 is dependent upon both A1 and A2.

Ideally, perhaps, the parser would look for this structure as it proceeds through the notes of the line. A future version of the parser may incorporate simultaneous parsing of basic structure, but for now the procedure has been relegated to what we might think of as a retroauditive parse.

Once the buffer is empty, the parser scans the line again, looking for notes that could function in a basic structure, assuming that the line is of a certain type (bass, primary,

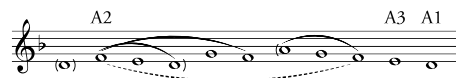
generic). The parser need only look at notes that are not dependents of others, so at this stage it uses just the list of open heads that remained in play at the end of the initial parse. The parser examines these open heads and assembles lists of candidates for each of the structural components (A1, A2, A3) and then tries to generate an interpretation for each list.

The parser now has something to say about the function of the first note in Fux's cantus firmus: it functions as "the initial pitch of the bass arpeggiation," rule A2. Looking for A1 in a bass line is only a matter of confirming that the last note is a tonic degree. The only remaining question is whether there are any candidates for A3. What the parser discovers in this particular case is that there is only one candidate: the A in the middle of the line. Hence the parser generates a single parse of the cantus firmus as a bass line (Ex. 7).

Example 7: Parsing Fux's Dorian cantus firmus as a bass line.

What if we ask the parser to see whether the cantus firmus makes sense as an upper line? Like bass lines, primary upper lines must end on the tonic degree. But while bass lines must start on the tonic degree, primary upper lines can begin on any tonic-triad pitch. And the initial note of the line need not be the note that functions as A2. The rule specifies that at some point, the upper line has to reach a tonic-triad pitch ($\hat{3}$, $\hat{5}$, or $\hat{8}$) that lies above A1. Rule A3 specifies that A2 has to then be connected to A1 via a continuous, descending step motion. In effect, there are three options, corresponding to the three forms of the Urlinie posited by Schenker. As with the bass line, the notes that function as A2, A3, and A1 must be conceptually prior to all other notes in the line.

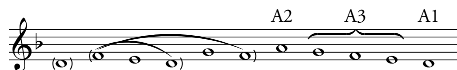
The task for the parser, then, is to figure out whether there are any candidates for A2 and then to find out which of these candidates, if any, can be connected to A1 via a step motion. It turns out that Fux's Dorian cantus firmus is structurally ambiguous when taken as a primary upper line. There are several candidates for A2: any of the Fs and also the A. Our parser considers it more plausible to take the first of the Fs as a candidate. Which is to say, the parser has a preference for interpreting later instances of a pitch as repetitions, operating on the principle that it is easier to interpret the future in terms of the past than vice versa. Example 8 shows the parse that results when the first F is the candidate for A2.




Example 8: Parsing Fux's Dorian cantus firmus as a primary upper line from $\hat{3}$.

The parser has other preferences built into it. The reader may have noticed in the initial run of the parser that the span after the high A was interpreted as an arc from A down to F followed by an arc from F down to D. The parser, however, considers it simpler to hear this span not as two arcs but as a single arc, a single step motion from A down to D, and will do so if it can. Of course, if F is functioning as A2, then it has priority, and

our parser hears the line as returning to F instead of passing through it. But if the parser tries out the A as a candidate for A2, it will fuse that final span into a single arc, as shown in Example 9.




Example 9: Parsing Fux's Dorian cantus firmus as a primary upper line from $\hat{5}$.

### 4.5 Parsing Transitions: The Algorithm

The ParseTransition function is the main engine of the Parser module. So let us look a little more closely at how the program goes about the work of evaluating transitions from one note to the next. Let us call these notes I and J. The parser asks a series of questions having to do with I and J: their relation to the governing triad, their intervallic relation, and whether J is related to notes on the dynamic lists of open heads and transitions (see Ex. 10). In first, second, and fourth species, the governing triad is simply the tonic triad. In third species, the tonic triad is the frame of reference for the overall line, but each bar also has a local triadic frame of reference. In harmonic species, the composition is segmented into three or four harmonic spans, each governed by its own triad (I, V, and II). Parsing third species and harmonic species requires maintaining not only global lists of open heads and transitions, but also local lists. Based on the answers, the parser assigns dependency relations, creates arcs where warranted, or returns error messages if the line is syntactically malformed.

The parsing algorithm considers eleven different cases, most of which have complex subtypes. Illustrations are provided for a few of the more interesting cases. In Case 1, both I and J belong to the governing triad. If I and J are also identical in pitch, J is interpreted as a repetition of I. If not identical, the parser looks to see whether J can be the terminus of an open transition, starting with the most recent; if so, it creates an arc and adds J to the list of open heads. Otherwise, it simply adds J to the list of open heads. In general, when searching the past for connections to the present, the parser is biased toward finding the shortest, most recent connection.

In cases 2, 3, and 4, either I or J does not belong to the governing triad, and the interval between them is a diatonic step. In Case 2, I is not a triad pitch, and J belongs to the triad of the next local span. This case involves a transition through end of one bar to the harmony of the next and arises only in third species and harmonic species. If I is an open local transition, the parser ends the transition at J and creates an arc. In Case 3, I is a triad pitch but J is not. If there are open transitions, the parser asks whether J continues a transition, starting with the most recent. If there are no open transitions but there are open heads, it tries to attach J to an open head, starting with the most recent. In Case 4, I is not a triad pitch but J is. Since I is an open transition, the parser looks to see whether J resolves a transition, starting with the most recent (I); the parser takes into account the directionality of the earlier pitch and the direction of its step relation to J. In third species, J is added to the local harmony if needed.

In Case 5, neither I nor J is in the triad. When such stepwise successions between

| *Case* | *I in triad* | *J in triad* | *Relation between I and J* |
|---|---|---|---|
| 1 | T | T | unison or consonant skip |
| 2 | F | T* | step |
| 3 | T | F | step |
| 4 | F | T† | step |
| 5 | F | F | step |
| 6 | F | T | consonant skip |
| 7 | T | F | consonant skip |
| 8 | F | F | consonant skip |
| 9 | F | F | unison |
| 10 | | | dissonant skip |
| 11 | | | compound interval |

*The governing triad is the local triad of the next timespan.
†The governing triad is the tonic triad.

Example 10: Transition cases analyzed by the parser.

a

b

c

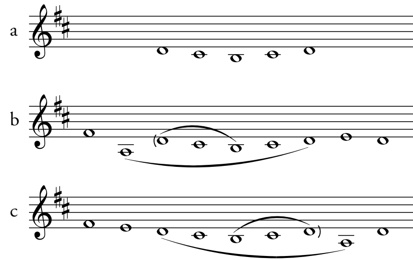
d

Example 11: Some special cases in melodic minor.

non-triad pitches occur in melodic minor, the parser has to pay special attention to the directionality of scale degrees. The parser is designed to hear raised $\hat{7}$ as either the lower neighbor to $\hat{8}$ or an ascending passing tone, so when it listens to the line in Example 11a, it resists thinking that the F♯ is part of a descending passing motion. The parser instead honors the upward directionality of F♯ by keeping it on the list of open transitions until G returns. But it must decide how to interpret the E♮.

Consider first how the parser would handle the line if the notes were F♮ and E♭ (Ex. 11b). In this case, the line steps down twice, and those descents match the directionality of the two scale degrees. F was already interpreted as a dependent of G and has G as its lefthead, so when the parser hears E♭, it connects it to F, because it has the same directionality, and adds all of F's dependencies to E♭; it also adds E♭ to F's list of dependents and vice versa. The parser then removes F from the list of open transitions. F has been displaced. When it hears D, it makes D the righthead of E♭ and, by extension, F, as shown in Example 11c.

In the original case (Ex. 11a), the descending step from F♯ to E♮ contradicts the directionality of F♯. So the parser makes E♮ dependent on F♯, but that is as far as it goes; F♯ is assigned as the lefthead of E but remains on the list of open transitions. In other




Example 12: Handling a change of direction in mid-transition.

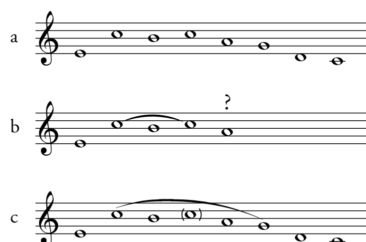
words, E♮ is not integrated into a step motion with F♯, because the line is not going in the right direction for that: F♯'s arc must go up by step. The resulting parse is shown in Example 11d.

Even major mode lines can be challenging. Consider the line fragment shown in Example 12a. If this sequence of notes is embedded in a line that is in the key of D major, the parser needs to know how to handle the change of direction after B, which implies that there are two transitions in progress, one of which attaches the B to an A, either as a lefthead or righthead. The parser therefore needs to consider whether there is an A already on the list of open heads, in which case it will interpret the B as a passing tone rising up to the second D (Ex. 12b); failing that, it will need to wait and see whether there is an A later in the line that can serve as a righthead, making the B a descending passing tone from the first D (Ex. 12c). In each case, the B also serves as the head of a subordinate transition to or from an inserted D.

So, Case 5 has five subtypes, depending on the directionality of I and J. (a) If the directionality of I and J match, or I is bidirectional and J is ascending, the parser asks whether I already has a lefthead. If not, it searches for one in the list of open heads, starting with the most recent. (b) If I is ascending and J is descending, and I steps up to J, J acts as the terminus of a transition through I and is assigned as the righthead of I. Also, J is added to the list of open transitions, while I is removed from the list. (c) If I is ascending and J is bidirectional, and I is raised $\hat{7}$ in minor and J is raised $\hat{6}$, then J is passing from I, so I is assigned as the lefthead of J, and J is appended to the list of open transitions; I also remains on the list of open transitions. (d) If I is bidirectional and J is descending (as is the case in minor from raised $\hat{6}$ to lowered $\hat{7}$), then I is a transition that terminates at J, so J is assigned as the righthead of I and added to the list of open transitions, while I is removed from the list of open transitions. (e) Otherwise, I and J are codependent; J inherits the lefthead of I, I is removed from the list of open transitions, and J is added to the list.

In cases 6 and 7, either I or J does not belong to the governing triad, and the interval between them is a consonant skip. In Case 6, I is not in the triad, but J is. If I has no lefthead, the parser reviews the list of open heads to find one. J is added to the list of open heads. In Case 7, I is in the triad, but J is not. If there are open transitions, the parser looks to see whether J continues a transition in progress; if that fails, it looks to see whether J connects to a head that precedes the open transitions; and if that fails, it returns an error. If there are open heads, the parser looks to see whether J can be made a dependent of one (starting with the most recent, as always); if that is not successful, the

parser searches the line for a possible step-related antecedent (as either head or transition) which was previously removed from consideration. This is one of the few cases where the algorithm attempts to revise the syntactic interpretation.




Example 13: A case of retrospective reinterpretation.

A version of the line shown in Example 13a was submitted by a student when I was initially developing WesterParse, and, despite being a valid Westergaardian line, the parser had rejected it, saying that the A was not generable in the key of C major (Ex. 13b). This is because the parser has a built-in bias for resolving transitions as soon as possible. So when the parser heard the second C, it decided that B was a lower neighbor and removed B from the list of open transitions. And then, when it subsequently heard A, it did not know what to make of it: there was no B on the list of open transitions that could link to A, and there was no G on the list of open heads that could link to A. The parsing algorithm thus had to be revised. Now, upon hearing A, the parser takes an extra moment to forget the partial parse it has constructed. First it clears the dependency relations. Then it selectively forgets the second C and starts over, loading all of the notes back into the buffer, with the exception of C, and listens again. Now it can hear the step connection between B and A. Later on it figures out that the intervening C was an independent insertion, an interjection, as it were, resulting in the parse shown in Example 13c. In this respect, the parser's activity mimics the phenomenology of acts of listening, in which interpretations are developed and then revised as new information becomes available.

In Case 8, neither I nor J belong to the governing harmony. In first, second, and fourth species, this produces an error. If the harmony derived from the notes on the downbeat in third species is ambiguous (e.g., forming an octave), I and J might be interpreted as disambiguating and defining the harmony. This is rare.

In the final group of cases, the interval between I and J is either a unison between non-triad pitches (Case 9), a dissonant skip (Case 10), or a skip larger than an octave (Case 11). These are always errors.

### 4.6 Parsing Local Contexts

In order to parse third species and harmonic species, the parser creates a separate buffer for the notes in local contexts as they arise and parses the transitions therein. Doing so requires that the parser also construct and maintain lists of heads, transitions, and arcs for each local context. In harmonic species, the triadic harmony of the local context is set in advance, but in third species, it must be inferred. The two consonant notes on the beat belong to the local triad, of course, but they do not necessarily define a triad, so the parser often

has to further define the local harmony by paying attention to which off-the-beat notes are approached or left by leap. In third species, the local parser also tries to extend local passing motions from or into bordering contexts. If the lefthead of a local passing motion is in the list of global transitions, it tries to stretch the arc back to the global lefthead of that note, and if the righthead of a local passing motion is the last note in the local context, it looks to see whether the arc can be extended to the end on the first note in the next context. In either case, the heads of the extended arc must be linearly consonant. When the parse of the local context is completed, the parser integrates the results with the ongoing global parse. Local arcs are added to the list of global arcs, and local heads are returned to the buffer so that the parser can try to connect them to earlier notes.

### 4.7 Building Parse Objects

The content of the preliminary parse lacks a kernel syntactic structure. The parser's next task, therefore, is to construct, where possible, a structure defined by Westergaard's A rules. Different sets of construction methods are used for generic, bass, and primary line types. In some cases, it is necessary to reinterpret or modify the results of the preliminary parse.

If the user has already specified a line type, the parser will only try to construct that type. Otherwise, it will try to construct parses for each of the three types, as follows: If the line type is *generic*, the function verifies that the line begins and ends on triad pitches, assigns rules A1 and A2 accordingly, and then looks for a possible step connection between these terminal pitches. If the line type is *bass*, the function verifies that the line begins and ends on a tonic degree (assigning rules A1 and A2), then assembles a list of notes that could complete the basic arpeggiation (rule A3) and builds a Parse object for each A3 candidate. If the line type is *primary*, the function verifies that the line ends on a tonic degree (assigning rule A1) and then assembles a list of notes that could initiate a basic step motion (rule A2). The function uses eight different methods to determine whether a valid basic step motion (rule A3) exists for each A2 candidate and attempts to build a Parse object using each method; not every method yields a result. The specific methods for each line type are discussed below.[17]

If a basic arc is created, the parser searches the list of open heads in the line to see whether any of them can be attached to the basic arc as repetitions, thereby improving the coherence of the interpretation. In bass lines, this means searching for repetitions of A2 and A3. In primary lines, it means searching for repetitions of A2 and any A3 that is also a tonic-triad pitch.

The parser creates a Parse object to store each possible interpretation. Each Parse object is assigned a unique label (e.g., "parse_BL04"). If the parse is successful, the parser adds interpretive information. It assign rule labels to notes in the secondary structures. In third species, it tests for the resolution of local harmonic insertions. It consolidates arcs into longer passing motions, if possible. And then it calculates the structural level of each note. Once all arcs for the parse have been settled, the parser creates an Arc object for each and adds these to the parse's arc dictionary. The parser makes note-by-note lists of rule labels and parentheses, to be used by the Context object when constructing representations of

17. Parts that are interpreted as primary lines are also checked for compliance with a rule that requires that the final consecution be a step; this rule is not in Westergaard.

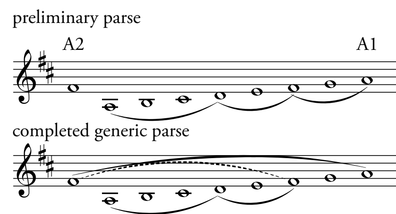
the parse in musical notation. The parser also calculates the hierarchical levels of the arcs. The results are collected and passed to the Global Context.

### 4.8 Bass Lines

As already mentioned, the preliminary parse has identified a set of notes that might function as the dominant of the bass arpeggiation (rule A3). The parser now creates a parse for each candidate. It tries to increase the coherence of the parse by looking to see whether any open heads can be attached as repetitions of A2 or A3. If the line is in third species, a particular choice of A3 might conflict with other arcs in the parse, so the parser tries to remove any lower-level arcs that cut across the A3 candidate.[18]

### 4.9 Generic Lines

If the preliminary parse was successful, the line has already passed the generic test, which is to say, there will be at least one successful parse as a generic line type. If the first and last notes are identical in pitch, then there can be no basic step motion and hence the basic arc consists of the two terminal components. If different, the parser searches for a basic step motion that connects the terminals. First it checks to see whether such an arc was created during the preliminary parse. If not, it looks for a step motion whose lefthead can be attached to the first terminal (A2). That is, it looks for an arc that concludes on the last note (A1), the lefthead of which has the same pitch as A2; it then creates a repetition between those pitches and reassigns the lefthead of the arc to A2. As can be seen in Example 14, the process of reinterpretation increases the coherence of the parse.




Example 14: Attaching an arc to A2 in a generic line.

If neither of these procedures has generated a basic arc, the parser looks for two arcs that can be embedded or merged into a basic step motion. (Similar methods are used for primary lines and will be discussed and illustrated below.) If all else fails, the parser creates a basic arc from the two terminal components.

### 4.10 Primary Lines

A set of candidates for A2 are deduced from the preliminary parse, each of which is tested for viability as the initiating tone of a complete step motion to A1. But to save some time with reinterpretation, the parser looks first to see whether the line ends with a neighboring arc to A1 and, if so, whether there is a descending passing arc that terminates at the

18. Removing arcs should necessitate reinterpretation of the affected spans, but the parser does not yet do this, so this routine is still not yet functioning optimally.

beginning of the neighbor. If so, the parser reinterprets the span, as illustrated by the two examples in Example 15.[19]

Example 15: Reinterpreting a final neighbor arc.

The parser then uses eight different methods for finding the basic arc in a primary line from each A2 candidate. Method 0 looks for an existing arc that passes from A2 to A1. If present, that arc becomes the basic arc. Method 1 looks for an existing arc that passes to A1 but starts from a later note that has the same pitch as the A2 candidate. If present, the lefthead of that arc is reinterpreted as a repetition of A2 and the arc's lefthead is reassigned to A2. See Example 16.

Example 16: Primary parse, method 1.

Methods 2 and 3 look, respectively for two or three passing arcs that can be merged into a basic step motion. The line shown in Example 17a, the end of which has already been reinterpreted after the preliminary parse, can be parsed as either a line from $\hat{5}$ or $\hat{8}$ using these methods, as shown in b and c, respectively.

Example 17: Primary parse, methods 2 and 3.

19. Westergaard allows the basic step motion of a primary line to conclude prior to the end of the line and thus would interpret the initial tone of the neighboring arcs as A1 in these examples.

Method 4 takes an existing $\hat{5}$-$\hat{4}$-$\hat{3}$ arc, using the longest if more than one is present, and tries to find a connection through $\hat{2}$ and $\hat{1}$ to complete a basic arc. See Example 18; a later method will be used to interpret the basic arc as beginning with the second note, a more plausible choice.

preliminary parse
A2 A1

final interpretation
A2 A1

Example 18: Primary parse, method 4.

Method 5 looks for a nonfinal arc from A2 whose terminus has the same pitch as A1 and then extends the arc to end on A1, if possible. The result is not always plausible, as seen in Example 19. Method 7 produces the same parse. The line is best interpreted as the generic line resulting from the preliminary parse.

preliminary parse
A2 A1

final interpretation
A2 A1

Example 19: Primary parse, method 5.

If A2 is $\hat{3}$, method 6 looks for a nonfinal lower neighbor arc that could be transformed into a passing to A1, and extends the arc to end there if possible. This situation is rare. An example is shown in Example 20; the same result is obtained from method 7.

preliminary parse
A2 A1

final interpretation
A2 A1

Example 20: Primary parse, method 6.

Method 7 takes the A2 candidate and then searches backwards from A1 to find a step connection to A2. It is the most radical method, and in its search for a basic arc, it often ends up jettisoning much of the preliminary parse. It works better for the simpler lines of first species and less well for third species. However, it can often produce credible results. It is currently the only method to take the line shown in Example 18 and find a basic arc starting with the second note; see Example 21.

preliminary parse
A2 A1

final interpretation
A2 A1

Example 21: Primary parse, method 7.

### 4.11 Harmonic Lines

In order to parse harmonic counterpoint, the user must provide information about the beginning of the predominant span (if any) and the beginning of the dominant span. The parser constructs four harmonic timespans from this information: initial tonic, predominant (which may be null), dominant, and closing tonic (always just the final bar); it also constructs the triadic frames of reference for these spans (using II for the predominant and V for the dominant). The spans are used during the preliminary parse as well as in the construction of the Parse objects. A short function in the context module verifies whether the harmonic segmentation conforms to the rules for harmonic species.

To construct a harmonic bass line, the parser confirms that the line begins and ends on the tonic degree. By definition, A1 lies in the closing tonic span and A2 lies in the initial tonic span. The parser then searches for instances of $\hat{5}$ that lie within the dominant span and uses these as candidates for A3. If there is a predominant span, the parser searches within it for candidates for A4, which is an additional rule that generates a predominant bass pitch. The parser then assembles valid pairings of A3 and A4 candidates and builds a Parser object for each.

Constructing a harmonic primary line is more complicated. The parser finds candidates for A2 within the initial tonic span and then uses a dozen different methods to construct a basic step motion across the harmonic spans, using arcs that occur within a harmonic span or connect from one span to the next. The methods are listed in Example 22 (methods 0–7 are used for monotriadic primary lines; see above). The rules for harmonic counterpoint permit a variety of different ways of distributing notes of the basic step motion across the three or four harmonic spans (see Schenker 1935, figs. 9–11). Several examples are shown in Example 23.

### 4.12 Determining Rules, Generative Levels, and Syntactic Units

The parser runs a function that assigns rules to notes in secondary structures. It finds any note that does not yet have a rule assigned to it and determines the appropriate rule based on the note's dependency relations. It looks first for neighboring and passing arcs, and then repetitions, and then notes that are independent. It is still possible at this stage that a note turns out not to be generable by a rule, in which case the parser records an error in the Parse object. Special rules are assigned for repetitions and insertions of local harmonic pitches in third species; again, this is a modification of the species that is not found in Westergaard.

Another function tests for the resolution of local insertions in third species. As already

| Method | Components from the Preliminary Parse | Comment |
|---|---|---|
| 8 | [$\hat{8}$–$\hat{7}$–$\hat{6}$] [$\hat{6}$–$\hat{5}$–$\hat{4}$] [$\hat{4}$–$\hat{3}$–$\hat{2}$] [$\hat{1}$] | |
| 9 | [$\hat{8}$–$\hat{7}$–$\hat{6}$–$\hat{5}$–$\hat{4}$] [$\hat{4}$–$\hat{3}$–$\hat{2}$] [$\hat{1}$] | $\hat{2}$ in the dominant |
| 10 | [$\hat{8}$–$\hat{7}$–$\hat{6}$–$\hat{5}$–$\hat{4}$] [$\hat{4}$–$\hat{3}$–$\hat{2}$] [$\hat{1}$] | $\hat{2}$ in the predominant |
| 11 | [$\hat{8}$–$\hat{7}$–$\hat{6}$–$\hat{5}$] [$\hat{5}$–$\hat{4}$–$\hat{3}$–$\hat{2}$] [$\hat{1}$] | |
| 12 | [$\hat{8}$–$\hat{7}$–$\hat{6}$–$\hat{5}$] [$\hat{4}$–$\hat{3}$–$\hat{2}$] [$\hat{1}$] | $\hat{2}$ in the dominant |
| 13 | [$\hat{8}$–$\hat{7}$–$\hat{6}$–$\hat{5}$] [$\hat{4}$–$\hat{3}$–$\hat{2}$] [$\hat{1}$] | $\hat{2}$ in the predominant |
| 14 | [$\hat{5}$–$\hat{4}$–$\hat{3}$–$\hat{2}$] [$\hat{1}$] | $\hat{2}$ in the dominant |
| 15 | [$\hat{5}$–$\hat{4}$–$\hat{3}$–$\hat{2}$] [$\hat{1}$] | $\hat{2}$ in the predominant |
| 16 | [$\hat{5}$] [$\hat{4}$–$\hat{3}$–$\hat{2}$] [$\hat{1}$] | $\hat{2}$ in the dominant |
| 17 | [$\hat{5}$] [$\hat{4}$–$\hat{3}$–$\hat{2}$] [$\hat{1}$] | $\hat{2}$ in the predominant |
| 18 | [$\hat{3}$] [$\hat{2}$] [$\hat{1}$] | $\hat{2}$ in the dominant |
| 19 | [$\hat{3}$] [$\hat{2}$] [$\hat{1}$] | $\hat{2}$ in the predominant |

Example 22: Methods for building the basic arc of a harmonic primary line.

Example 23: Parses of a harmonic primary line using methods 11, 13, 17, and 19.

mentioned, the parser is biased toward simpler interpretations. So one of the things it will do at the end of the parsing process is to see whether there are two passing motions that share an inner node and direction and sum to a consonant interval. If so, it will merge them into a single arc and revise the dependency relations and rules accordingly (Ex. 24). Likewise, if a neighbor motion is linked to a subsequent passing motion, the parser will embed the neighbor structure within the passing, making them both share the same lefthead; embeddings of this type are executed during the preliminary parse.

Example 24: Merging two passing arcs.

Having now finalized the interpretation of all arcs, the parser can record the details of the interpretation. The most difficult task is to calculate the structural level of each note. The parser assigns levels to notes in the basic arc, starting with 0 for the final note generated by rule A1, 1 for A2, and 2 for any notes generated by A3. If the line's first note is not in the basic arc, the parser sets its level to 3. The parser now compiles a list of fillable spans within the line. A span is defined by two notes (initial and final), and the length of a span is equal to the number of intervening notes; the length of the span between consecutive notes is 0. Only non-zero spans are fillable. The span list is initialized with a root span that extends from the first to the last note of a line. Then the parser adds fillable spans before and after the basic arc, if they exist.[20] Next it adds any fillable spans found in the basic arc. The parser then looks at every span in the list to see whether a secondary arc fits into it. This is the core of the function. Priority is given to a secondary arc that connects both edges of the span, then one that is tethered to the left edge, then the right edge, and then any that lie independently within the span. Preference is also give to the longer arcs. Once a choice is made for an arc that fits inside a span, the function assigns generative levels to the arc's dependent components (rule levels inside the span are determined by the rule levels of the left and right edges), revises the span list accordingly, and continues until the list of secondary arcs is exhausted. At this point, all of the arcs and their components have been accounted for, so all that remains is to calculate the level of independent inserted pitches. The parser examines every span that contains only inserted pitches, testing to ensure compliance with the restrictions of rule B3 (the function that does this is too complex to explain here).

The parser creates an Arc object for each arc and adds it to the parse's arc dictionary. It records the positional indexes of the notes in the arc, as well as the arc's category (basic, secondary), type (passing, neighboring, repetition, arpeggiation), and subtype (rising or falling for passing arcs; upper or lower for neighboring arcs). And it records the scale degree content of the arc. The parser also compiles two lists that will be used in preparing representations in musical notation. In the first list, each note in the line is represented by a tuple consisting of the note's index, rule name, and generative level. The second list records every note that will be enclosed in parentheses in the analytical notation. Another function derives the hierarchical level of each arc based on the generation level of the arc's most deeply embedded component and adds this information to the arc dictionary.

The final step is to collect parses and pass that information back to the Global Context. The parser gathers all the attempted parses of a line and discards any that have errors. It removes duplicate parses and also removes parses of primary lines if the same basic arc was produced by a more reliable method. If no parses were successful, it returns an error message. Otherwise, it adds the successful parses to a dictionary, according to line type, and records whether the line is generable as a primary, generic, or bass line.

### 4.13 Supporting Functions

A few dozen utility functions support the operation of the parser. Some check the harmonic status of a note, others evaluate the linear relation of two notes. Many different

20. In the current implementation, there will never be a span after the basic arc, but Westergaard allows it.

| *data source* | *part* | *parse label* | *line type* | *species* |
|---|---|---|---|---|
| WP001 | 0 | parse10_PL | primary | first |

| *ref* | *note index* | *offset* | *csd value* | *rule label* | *gen level* | *left paren* | *right paren* |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0.0 | 0 | B3 | 3 | True | True |
| 1 | 1 | 4.0 | 2 | B3 | 4 | True | False |
| 2 | 2 | 8.0 | 1 | B4 | 8 | False | False |
| 3 | 3 | 12.0 | 0 | B3 | 7 | False | True |
| 4 | 4 | 16.0 | 3 | B2 | 6 | False | False |
| 5 | 5 | 20.0 | 2 | B1 | 5 | False | True |
| 6 | 6 | 24.0 | 4 | A2 | 1 | False | False |
| 7 | 7 | 28.0 | 3 | A3 | 2 | False | False |
| 8 | 8 | 32.0 | 2 | A3 | 2 | False | False |
| 9 | 9 | 36.0 | 1 | A3 | 2 | False | False |
| 10 | 10 | 40.0 | 0 | A1 | 0 | False | False |

| *ref* | *arc* | *category* | *type* | *subtype* | *csd content* | *level* |
|---|---|---|---|---|---|---|
| 0 | [1, 2, 3] | secondary | passing | falling | [2, 1, 0] | 2 |
| 1 | [1, 4, 5] | secondary | neighbor | upper | [2, 3, 2] | 1 |
| 2 | [6, 7, 8, 9, 10] | basic | passing | falling | [4, 3, 2, 1, 0] | 0 |

Example 25: Parse data for Fux's Dorian cantus firmus, parse10_PL.

functions work with arcs: creating arcs using the Dependency objects attached to notes, checking or getting the properties of a given arc, and modifying the content of an arc and the dependency relations of its components.

## 4.14 Extracting Parse Data

Example 25 presents the data extracted from the parse of Fux's Dorian cantus firmus shown earlier in Example 9. The data has three parts: a set of metadata about the source, an array of data about the individual notes, and an array of data about the arcs. These tables illustrate WesterParse's native manner of representing syntactic information. Obviously, this format is difficult for humans to read, which is why WesterParse has been designed to translate interpretive data into musical notation.[21]

Reference numbers in the two arrays provide access to the rows. The note indexes refer to the position of the note in its line; often this is the same as the row number, but because `music21` counts tied-over notes as two separate notes, the note indexes in fourth species are nonsequential integers. Offset measures the distance of the note's attack from the beginning of the piece, counting in quarter-note units. Note indexes provide the link to the arcs array, where they appear as components of the lists in the second column.

21. In the analytical notation I use for WesterParse, left and right parentheses can be used separately, but MusicXML and `music21` treat parentheses as a unit, so the output of WesterParse will look slightly different when rendered by a notation program such as MuseScore.

### 4.15 Evaluating Parses and Contrapuntal Combinations

As is perhaps already clear to the reader, not every successful parse is equally plausible. A competent listener will likely have reasons for preferring one parse over another. Since evaluating among legitimate parses is a separate analytical activity, I did not build that directly into WesterParse. But the parse data extracted from WesterParse can be used to evaluate among the parses of a line. Take, for example, the line shown above in Example 17. WesterParse actually generates seven different parses of it as a primary line. These are shown in Example 26. A number of cognitive preferences might influence a listener's interpretive choice among these options. One preference already mentioned has to do with a bias toward interpreting later notes in terms of earlier notes; or, in other words, using what you already know to interpret whatever may come. In primary upper lines, this means selecting A2 as early as possible. Simplicity is another preference. One measure of simplicity is the number of generative levels. Parses b and c have the least number of levels. Those two parses, on the other hand, also have the most complex basic structures. Listeners may also prefer short spans of connection following non–tonic-triad pitches; such pitches connect to some note in the past and also to some note in the future, so they place the greatest demands on the listener's working memory. To minimize those demands, listeners will prefer to resolve such pitches as speedily as possible. Hence parse b is preferable to parse c, because the B♭ is resolved directly. Another likely preference is coherence. This might be measured in terms of the number of secondary arcs that are connected to the basic arc. Coherence is also inversely proportionate to the number of independent notes (enclosed in parentheses). Parses a–d are more coherent than parses e–g. And all things considered, parses a, b, and d seem preferable to the others. Each of the metrics mentioned here can easily be constructed from the parse data collected in WesterParse.

Contrapuntal combination with a bass line can influence a listener's interpretive choice. One form of contrapuntal coordination that reduces working memory is using the same type of structure simultaneously in two different parts (parallelism). When possible, listeners may also want to coordinate the basic structures of the lines: looking for the nearest timing of A2 in both the bass line and the primary line, and coordinating A3 in the bass with the last of the A3 pitches in the primary line. It is a fairly simple matter to gather the parses of a pair of lines and sort through the combinations to look for those that maximize these forms of coordination.[22]

## 5 The EvaluateCounterpoint Function

### 5.1 Preparing the Score for Analysis

The EvaluateCounterpoint function creates a Global Context object and then calls the VoiceLeadingChecker module to examine the local voice leading for conformity with Westergaard's rules.

The voice-leading checker prepares the score for evaluation by making pairwise extractions of the parts, called duets. It then slice the duets into vertical segments (simultaneities). Each simultaneity has access to a dictionary of its contents, arranged by part,

22. For a succinct summary of Westergaard's principles of disambiguation, see Peles 1997.

Example 26: Complete parse set of a primary line.

making it easy to access the simultaneous contents of one or more given parts.[23] Many voice-leading rules involve what happens between a pair of parts, so the checker makes extensive use of two bits of information derived from the verticalities: VoicePairs (VPs) and VoiceLeadingQuartets (VLQs). A VP is a pair of simultaneous notes:

$$\text{VP} = \begin{array}{r|l} \text{voice 1:} & \text{note} \\ \text{voice 2:} & \text{note} \end{array}$$

A VLQ consists of a pair of simultaneous two-note consecutions:

$$\text{VLQ} = \begin{array}{r|ll} \text{voice 1:} & \text{note 1} & \text{note 2} \\ \text{voice 2:} & \text{note 1} & \text{note 2} \end{array}$$

VPs are useful for checking the rules that control dissonance, and VLQs are useful for checking rules that forbid various forms of motion.[24] Although the `music21` voice-leading module contains many such tests, Westergaard's voice-leading rules are sufficiently different that I deemed it necessary to write custom voice-leading tests for WesterParse.

23. The latest version of `music21` (version 3.8) has a method for constructing so-called verticalities, but unfortunately its verticality object does not give access to the components by part, and the object's list of component pitches ignores unison duplication; both deficiencies make it unsuitable for use in evaluating counterpoint.

24. In `music21`, the ordering of parts in a VLQ is not fixed (i.e., voice 1 is not always the top or bottom voice). This inconsistency proved problematic for the analysis of strict counterpoint. I am grateful to Tony Li, who wrote a method for getting VLQs that eliminated this problem; the current iteration of WesterParse adapts some of the code that he wrote.

## 5.2 How WesterParse Evaluates Voice Leading

WesterParse checks each duet for forbidden forms of motion and control of dissonance, depending upon which simple species the duet represents (e.g., 1:1, 1:2, 1:3, or 1:4). The function is currently unable to evaluate combined species (e.g., 1:2:2).

Many rules about forbidden forms of motion apply across species. For example, the rules of first species, originally formulated for consecutive beats (on the beat to on the beat) apply in all species whenever both lines move to new notes on the beat. This happens regularly in first, second, and third species, and at the end of fourth species, where the syncopations are broken. Hence there is a single function to check these situations. It checks a duet for conformity with the rules that prohibit or restrict the following: similar motion to or from a unison; similar motion to an octave; similar motion to a fifth; parallel motion to unison, octave, or fifth; and voice crossing, voice overlap, and cross relations. Each counterpoint species has additional motion rules that are peculiar to it. Westergaard has rules, for example, that govern nonconsecutive parallel perfect intervals in first and second species. In second and third species, there are rules governing the motion from beat to beat. And in third species there are additional rules that govern motion from off the beat to next but not immediately following on the beat. Subroutines within the functions for each species check the duets for conformity with these rules.

Control of dissonance is checked by one of two functions: the one checks first, second, and third species, and the other checks fourth species. Checking control of dissonance in three or more parts requires access not only to notes in a given duet of lines but also the bass line, if different.

The voice-leading checker also evaluates the intervals between consecutive notes. In its current form, the voice-leading checker does not duplicate any of the consecution rules that are handled by the rules of linear syntax (e.g., the prohibition of dissonant skips). In second, third, and fourth species, a function checks the intervals between consecutive notes to ensure that there are no immediate repetitions (Westergaard permits direct repetitions in the whole note line); if the line is in fourth species, the function also confirms that the pitches of tied-over notes match. In Westergaardian species, there is also a set of consecution rules that have a contrapuntal component: the rules that address situations in which the bass leaps a perfect fourth; these rules in effect prohibit the implication of six-four chords, either within a bar or across the barline. To facilitate this evaluation, WesterParse has a function that compiles a list of fourth leaps in the bass.[25]

Infractions of the rules are automatically recorded and then reported to the user when the evaluation is completed.

# 6 Results and Testing

As part of the development process, the outputs of WesterParse were compared to examples provided by Westergaard. Of the 48 lines in the WesterParse corpus that are taken from Westergaard's text, Westergaard provides interpretations for only 25. Of those 25,

25. Another rule in this implementation (but not found in Westergaard) also has a contrapuntal aspect: the global rule that ensures a step connection from the penultimate measure to the final pitch of a primary line. This test is currently implemented in the main WesterParse module but will be relocated in a future release.

he provides two interpretation of 6, yielding a total of 31 interpretations. WesterParse generates 26 of those interpretations (some of the lines are ambiguous, so WesterParse also generates other interpretations).

Only 5 of Westergaard's interpretations are not reproduced by WesterParse. Three of those cases are due to the fact that Westergaard allows a nonfinal pitch to be generated by rule A1, whereas WesterParse does not. The other two cases are due to WesterParse's bias toward resolving non–tonic-triad pitches as soon as possible, and its concomitant reluctance to attribute a passing function to a tonic-triad pitch. There are no inexplicable discrepancies.

## 7 Plans for Development

Plans for further development of the WesterParse pedagogical environment include allowing the student user to add a syntax interpretation to a line and having the parser determine whether it is a legitimate interpretation. Unfortunately, while musical notation is a highly efficient data exchange format for humans, it is not so for computers.

I am also working on evaluating the interest of a line, mostly in terms of the degree to which the line avoids various forms of monotony and lacks complexity. This will be used to provide feedback to the student. Implementing Westergaard's preference rules for sonorities will provide another type of feedback for students.

A longer-range goal is to incorporate Westergaard's analysis of the rhythm of linear elaborations (chapters 3 and 7), and thus give the parser the ability to analyze rhythmically differentiated lines. But developing WesterParse into a program that can parse more complex lines that unfold in a contrapuntal texture will require many additional components and capabilities. One such component must be a stream segregator that is able to sort simultaneous notes into different lines. If the input source is a MusicXML file that is already divided into single-line parts, as it is in the web application, stream segregation is relatively simple. For more complex contexts, the segregator needs to have additional abilities. It needs to be able to split simultaneously sounding notes into separate streams.[26] It may also need to monitor the texture, deciding when an additional simultaneous note is supplemental and when it is the inauguration of an additional stream. The segregator also needs to be able to determine whether a stream is a compound line; if so, it will need to extract the pitches of the compound line and re-assign them to new notes with new timespans. The segregated streams are then sent on to individual parsers. In order to handle longer lines, the software will need to incorporate some form of grouping structure constraints. Some of the computational challenges of implementing this sort of analysis are addressed in Marsden 2010.

A contextualizer needs to gather information from the parsers, store relevant information about the context, and then share that information among the parsers. For example, the parsers need to provide information that can be used to determine the tonality of the passage. The contextualizer collects and analyzes information at the outset to determine a likely candidate for the tonic triad and the mode of the passage. This is information that

26. For a review of relevant literature on computational stream segregation and a discussion of a neural network model for automatic voice separation, see Weyde and Valk 2015. Also see Temperley 2009.

belongs to the global context. Each parser uses this information to determine the structure of its line.

If the input is a musical passage of harmonically rich counterpoint, the contextualizer also needs to maintain a list of local contexts. We might know that a particular span, for example, unfolds within a tonic triad, while the next span unfolds within a dominant triad, and so forth. The parsers need to know (or learn) this in order to determine whether a pitch is to be generated as a triad pitch (rules B1 and B3) or a transition (rules B2 and B4). The same pitch might be generated by one or the other category of rule, depending upon the context. The parsers will also have to maintain local lists. As long as a parser is interpreting a line solely in terms of the global tonic triad, it only needs to maintain one list of open heads and one list of open transitions. But if local contexts are engaged, it needs to maintain similar lists for each context, in addition to the global lists. It needs to be able to tell, for example, whether a note during the dominant span is part of a local transition or whether it belongs to a global transition. And the contextualizer has to be capable of handling many layers of embedded contexts.

The contextualizer should also handle negotiations among parsers in cases where one or more lines is structurally ambiguous (as is the case with the Fux's Dorian cantus firmus). The contextualizer would also be responsible for deciding when a passage modulates into a new key, gathering input from the individual parsers as they encounter interpretive anomalies in the current key and then negotiating a new state of agreement. And the contextualizer must be responsible for inferring the meter and any changes to the metrical system.[27]

## 8 Acknowledgements

I am indebted above all to Stephen Pentecost, Senior Digital Humanities Specialist in the Humanities Digital Workshop at Washington University. Steve has been a valuable resource from early on in the project. Steve created the web interface for the pedagogical application and generously answered my newbie questions about data structures and analysis, GitHub packaging, django, Python, and much more. I am grateful to Joe Loewenstein, Director of the Humanities Digital Workshop and the Interdisciplinary Project in the Humanities, for inviting me to give talks on WesterParse to HDW groups and for encouragement in developing this project. I am also grateful to Ben Duane for recommending early on that I have a look at `music21`, and to both Ben Duane and Paul Steinbeck for comments on an earlier draft of this article.

I am grateful to the students in my Theory III class who tested the web implementation. In particular, I received useful feedback from Cameron Perrin (who also wrote some lovely counterpoint), Rachel Kadlick, Maria Crusey, Cade Edney, and Jasen Vest. Nicole Shin made valuable contributions to testing early versions of the software and improving the online help pages.

Several graduate students at Washington University also contributed to the project. Lisa Mumme typeset examples for the corpus, Tony Li wrote a function for extracting

27. See Temperley 2007 for a review of literature on parsing meter and modulations as well as proposed solutions.

voice-leading quartets as well as some routines for analyzing sonority, and Varun Chandrasekhar composed a corpus of examples that I am using to test functions for evaluating the interest of lines.

I learned a great deal about Westergaard's approach from long and frequent discussions with my former colleague and friend, Marion Guck, whose counterpoint course I inherited when she moved to another institution. Finally, of course, I am grateful to Peter Westergaard …

## References

Alphonce, Bo H. 1980. "Music Analysis by Computer—a Field for Theory Formation." *Computer Music Journal* 4, no. 2 (Summer): 26–35. https://doi.org/http://dx.doi.org/10.2307/3680080.

Cuthbert, Michael Scott, and Christopher Ariza. 2010. "`music21`: A Toolkit for Computer-Aided Musicology and Symbolic Music Data." In *11th International Society for Music Information Retrieval Conference (ISMIR 2010),* edited by J. Stephen Downie and Remco C. Veltkamp, 637–42. Utrecht.

Jurafsky, Daniel, and James H. Martin. 2008. *Speech and Language Processing.* 2nd ed. Upper Saddle River, NJ: Pearson Prentice Hall.

Lerdahl, Fred, and Ray Jackendoff. 1983. *A Generative Theory of Tonal Music.* Cambridge: MIT Press.

Lewin, David. 1983. "An Interesting Global Rule for Species Counterpoint." *In Theory Only* 6 (8): 19–44.

Marsden, Alan. 2010. "Schenkerian Analysis by Computer: A Proof of Concept." *Journal of New Music Research* 39 (3): 269–89. https://doi.org/http://dx.doi.org/10.1080/09298215.2010.503898.

Maus, Fred Everett. 1992. "Teaching with Westergaard's Counterpoint Rules." *Music Theory Pedagogy Online* 6.

Peles, Stephen. 1997. "An Introduction to Westergaard's Tonal Theory." *In Theory Only* 13 (1-4): 73–94.

Schenker, Heinrich. 1935. *Der freie Satz.* Vienna: Universal. Vol. 3 of *Neue musikalische Theorien und Phantasien*.

Snarrenberg, Robert. 2021. "WesterParse: A Transition-based Dependency Parser for Tonal Species Counterpoint." In *Proceedings of the 13th International Conference on Computer Supported Education — Volume 1: CSME,* 669–79. INSTICC, SciTePress. https://doi.org/http://dx.doi.org/10.5220/0010482606690679.

Temperley, David. 2007. *Music and Probability.* Cambridge: MIT Press.

———. 2009. "A Unified Probabilistic Model for Polyphonic Music Analysis." *Journal of New Music Research* 38 (1): 3–18. https://doi.org/http://dx.doi.org/10.1080/09298210902928495.

Westergaard, Peter. 1975. *An Introduction to Tonal Theory.* New York: Norton.

Weyde, Tillman, and Reinier de Valk. 2015. "Chord- and Note-Based Approaches to Voice Separation." In *Computational Music Analysis,* edited by David Meredith, 137–54. Springer.