WesterParse: A Transition-based Dependency Parser for Tonal Species Counterpoint

Robert Snarrenberg[a]
Department of Music, Washington University in St. Louis, One Brookings Drive, St. Louis, MO, U.S.A.

Keywords: Music Analysis, Melodic Parsing, Tonal Syntax, Algorithmic Analysis, Westergaard.

Abstract: This article describes the syntax parser that is a principal component of WesterParse, a software program designed to evaluate tonal species counterpoint in the version developed by Peter Westergaard (1975). The parser produces interpretations of the pitch-syntactic structure of simple tonal lines. The parser is written in Python and relies on the `music21` toolkit. Given a simple tonal line of the sort found in Westergaardian counterpoint, the parser can evaluate its structure and report whether the line is valid. To do so, the parser compiles a set of possible syntactic interpretations. 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 both simple and combined species for compliance with Westergaard's rules of voice leading.) After providing a synopsis of Westergaard's definition of simple tonal lines, the article describes the architecture of the software parser, the scanning process, and the central concept of dependency relations. The parsing operation is then illustrated using Fux's Dorian cantus firmus, and a closer look is taken at the process for parsing transitions.

1 INTRODUCTION

I have been teaching Peter Westergaard's species counterpoint in a class on tonal theory for 25 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 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 students are still in the flow of composing. So I decided to create a web-based application to supplement the in-class experience.

The application consists of a web page where student users can compose species counterpoint exercises and a server-based backend that can interpret data sent from the web page. Stephen Pentecost, from the Humanities Digital Workshop at Washington University, created the web interface. I wrote the backend in Python.

The user begins by deciding how many measures of counterpoint to write, how many parts, and what key signature to use. The webpage then presents the user with a blank set of staves. The user enters notes as desired, and a separate edit mode allows the user to go back and change notes and add or delete measures. When the exercise is complete, the user can ask the machine to evaluate the lines or evaluate the voice leading. Clicking "Evaluate Lines" sends a representation of the line in MusicXML to a parser, 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 user clicks "Evaluate Voice Leading."

This article describes the backend software component that evaluates individual lines, the so-called parser. Readers interested in testing the parser and counterpoint evaluator may visit talus.artsci.wustl.edu/westerparse/. The full code for WesterParse is freely available on github.com/snarrenberg/westerparse. Documentation of the software is available at westerparse.readthedocs.io.

[a] https://orcid.org/0000-0001-6705-188X

Snarrenberg, R.
WesterParse: A Transition-based Dependency Parser for Tonal Species Counterpoint.
DOI: 10.5220/0010482606690679
In *Proceedings of the 13th International Conference on Computer Supported Education (CSEDU 2021) - Volume 1*, pages 669-679
ISBN: 978-989-758-502-9

2 WESTERGAARDIAN LINES

In the early 1970s Westergaard developed an approach to teaching tonal music theory centered on the cognition of linear syntax and counterpoint rather than chords and harmony. The goal was to give the student "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).

One thing that 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" (Westergaard, 1975, 29). In Westergaard's formulation, the simple tonal lines of species counterpoint are constructed by applying rules that generate notes with certain order-dependent functions (see Appendix). These generative rules constitute a syntax for simple tonal lines.[1]

The forms of simple linear structure are based on those that 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 (rule A1); a tonic-triad pitch that lies in the register above the tonic pitch and acts as the initial structural element (A2); and pitches in the scale that fill the span between the initial and final elements with a complete stepwise motion (A3). Bass lines have a kernel structure consisting of an arpeggiation from the tonic degree (A2) to the dominant (A3) and on to another tonic degree (A1). (Westergaard adds a third, less constrained type of line that begins and ends on a tonic-triad pitch; I call this a generic line.)

The kernel structures may be elaborated by the addition of syntactically dependent elements: tonic-triad pitches may be repeated (B1), new tonic triad pitches may be inserted (B3), and stepwise transitions may be created between consecutions of identical pitches (neighboring motions, B2) or different pitches (passing motions, B4).

Figure 1 shows how a melody familiar to countless generations of species counterpoint students might be constructed as a bass line using Westergaard's rules, starting with the A-rules at the top and proceeding level by level until all the notes of the line have been generated.

[1]Simple lines are rhythmically uniform and relatively brief. The syntactic system for complex, rhythmically differentiated lines involves rhythmically and contrapuntally sensitive rules. Parsing the syntax of such lines lies outside the scope of this project.

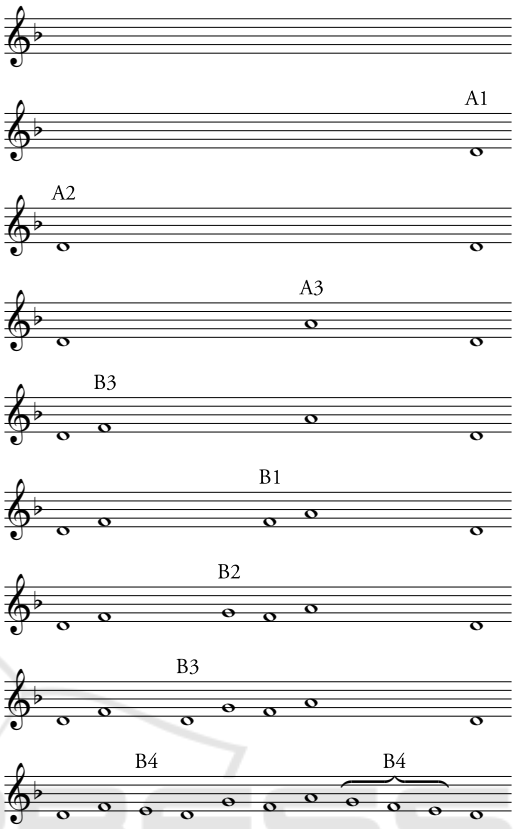

Figure 1: Constructing Fux's Dorian cantus firmus as a Westergaardian bass line.

3 THE PARSER

The interpretation of Fux's Dorian cantus firmus shown in Figure 1 builds the line, as it were, note by note, from the root node at the end of the line to the most deeply embedded, dependent notes in the middle. The top-down structure resembles a syntax tree. To arrive at such an interpretation, however, the software parser (or the human interpreter) has to begin with the given line of notes (the utterance) and then derive an interpretation, determining what rule is used to generate each note. And if there is more than one way to parse the syntax of the line, the parser ought to generate multiple interpretations.

The parser is designed to model the cognitive process of auditive interpretation, which occurs in time as the melody unfolds note by note. It is not a simple process. In fact, a listener cannot begin to attribute syntactic structure reliably to a melody without also determining what triad and scale to use as frames of reference. In the case of Fux's melody, it takes a few notes before a listener can grasp the mode and triad. The process might run something like this. A listener

assumes that the first note at the very least belongs to the tonic triad, unless evidence accumulates to the contrary. But which scale degree is it? $\hat{1}$, $\hat{3}$, or $\hat{5}$? After hearing the second note, the number of possibilities is reduced, for the only plausible interpretations of the two notes are $\hat{1}\,\hat{3}$ and $\hat{3}\,\hat{5}$. And after hearing the third note, there is no doubt: this is a line in the minor mode that starts with $\hat{1}\,\hat{3}\,\hat{2}$. A preliminary stage of parsing, then, involves the selection of such frames of reference as a plausible context for the syntactic interpretation.

The parser must also keep track of notes and all their contextually derived properties and syntactic functions. From a software standpoint, this requires an object-oriented programming language. I built the parser in Python in order to take advantage of the `music21` toolkit developed at MIT by Michael Cuthbert and his colleagues.[2] A significant advantage of `music21` is that its already robust collection of musical objects and relations is easily extensible. The parser accepts input in the form of a MusicXML source file, which is then converted by the `music21` toolkit. After conversion, the program can access the content in the source file in a variety of ways: parts, measures, notes, simultaneities, and so forth. Thanks to the design of `music21`, it is a relatively simple matter to extract the line of notes in each part of a contrapuntal exercise for parsing.

The basic architecture of the parser is borrowed from computational linguistics. It is modeled on software that processes a sentence word by word and outputs a syntactic model, a so-called "transition-based dependency parser." In simple terms, this kind of parser examines the transitions from word to word in a sentence and decides at each point whether two adjacent words are syntactically related as head and dependent (or, vice versa, as dependent and head), or whether to keep looking for such connections. Since dependency relations are rather different in music, I wrote all the interpretive routines from scratch[3].

Figure 2 shows the basic architecture of the parser. 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 transition parser examines the top element of the stack and the next element in the buffer and selects an action based on examination of the current interpretive state.[4]

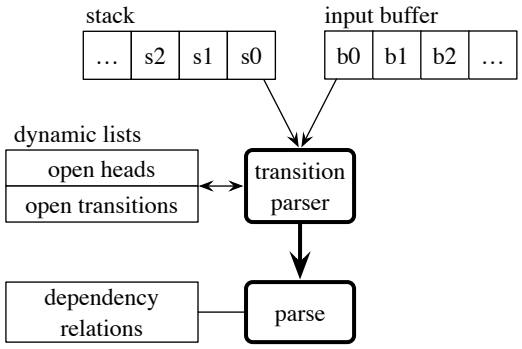

Figure 2: The architecture of WesterParse's line parser.

In addition to the stack and the input buffer, the parser maintains a set of dynamic lists (open heads, open transitions) and a 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 cantus firmus, D F E. At this point, the parser has placed D and F on the list of notes of available heads and has decided 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. As the line transitions from one note to the next, the parser is beginning to figure out dependency relations among the notes, hence the name of transition-based dependency parser.

The dependency relations make up the content of the interpretation, the so-called parse. So, what is meant by dependency relations? 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. 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 and the right. They have a lefthead and a righthead.[5] Take the succession [F E D]. One possible syntactic inter-

[2](Cuthbert and Ariza, 2010); see http://web.mit.edu /music21/doc/index.html.

[3]While many linguistics parsers now use machine learning methods and a set of training data, WesterParse takes an algorithmic approach, based on a set of rules for generating lines and a set of preferences for interpreting lines.

[4]By contrast, the linguistics parsers described in (Jurafsky and Martin, 2008, Chapter 13, "Syntactic Parsing") examine the top two elements of the stack. 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.

[5]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 mu-

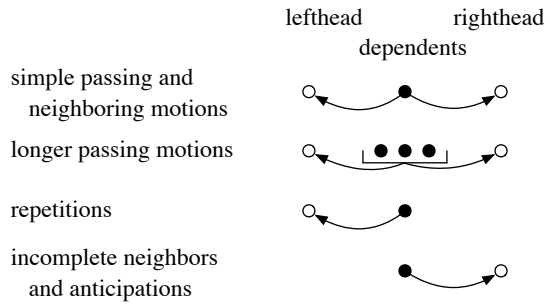

Figure 3: Four types of arc.

pretation 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 an arc. Arcs are of several types (see Fig. 3): 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 a custom Dependency object that is attached to each note in the parse. As completed arcs accumulate, they are stored in a Parse object.

The parser also needs to keep track of other variables and properties of notes (see Fig. 1). At the outset, it needs to set some global variables: the name of the keynote and the mode, from which the tonic triad and scale can be inferred. This is currently handled by a separate software component that infers the key of the source input using a set of custom algorithms. (The web application also allows the user to input a key and thus override the keyfinding algorithm.)

Keyfinding is a crucial first step, since the parser also needs to know the scale degree function of each pitch. This is stored in another custom object called Concrete Scale Degree (CSD). We are used to thinking of scale degrees as scale-degree classes, assigning all tonic degrees to the class "scale degree one," but it has proven helpful to have a more concrete scale degree function, one that distinguishes between, say, $\hat{1}$ and $\hat{8}$; this distinction allows the parser to hear $\hat{7}$ as adjacent to $\hat{8}$ but not to $\hat{1}$. The CSD stores a value representing the distance of a particular pitch from the core tonic degree. Scale degrees also have have directionality: in the minor mode, some degrees are bidirectional, some are ascending, and some descending. The parser will need to know, for example, that raised $\hat{6}$ is bidirectional but diatonic $\hat{6}$ is strictly descending.

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.

sic, 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, 185–87)).

Table 1: A partial ontology of WesterParse.

Object Classes	*Attributes*	*Values*
Context	parts	Part
	key	Key
Key	keynote	Pitch
	mode	major, minor
Part	notes	Note
	id	0, . . . , *n*
Note	pitch	A, . . . , G
	index	0, . . . , *n*
Concrete	value	. . . , –1, 0, 1, . . .
Scale	degree	. . . , $\hat{1}$, $\hat{2}$, . . .
Degree	direction	ascending, descending, bidirectional
Dependency	lefthead	None, or Note index
	righthead	None, or Note index
	dependents	None, or Note inidices
Rule	name	A1, . . . , B1, . . .
	level	0, . . . , *n*
Parse	linetype	primary, bass, generic
	arcs	lists of Note indices
	open heads	Note indices
	open transitions	Note indices

4 THE PARSING PROCEDURE

Let us look at how the parser works its way through Fux's Dorian cantus firmus. The first half of the process is illustrated in Figure 4. 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 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,

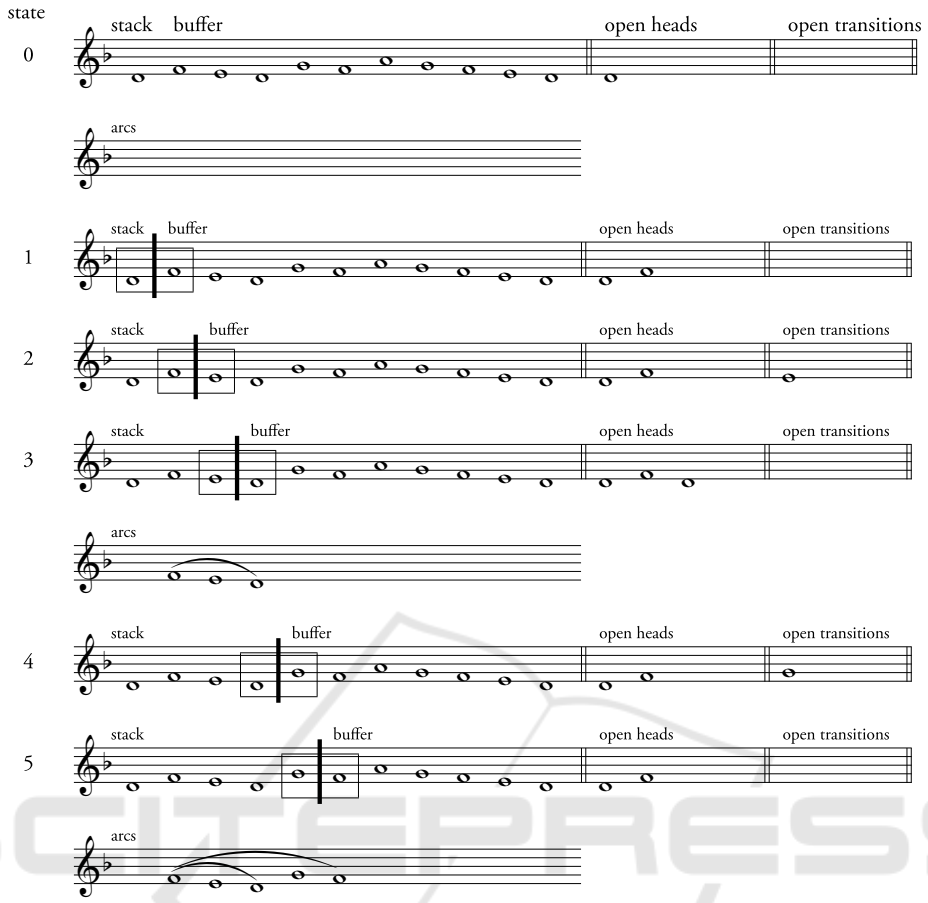

Figure 4: Parsing Fux's Dorian cantus firmus, states 0–5.

[F E D], which is added to the parse.[6] 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.

In subsequent stages, shown in Figure 5, 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 line segments (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 A1.

[6] In the musical representations of the parses, notes of the basic structure are identified by rule number, ties connect repetitions to their heads, slurs connect the notes of a stepwise motion, and parentheses enclose insertions.

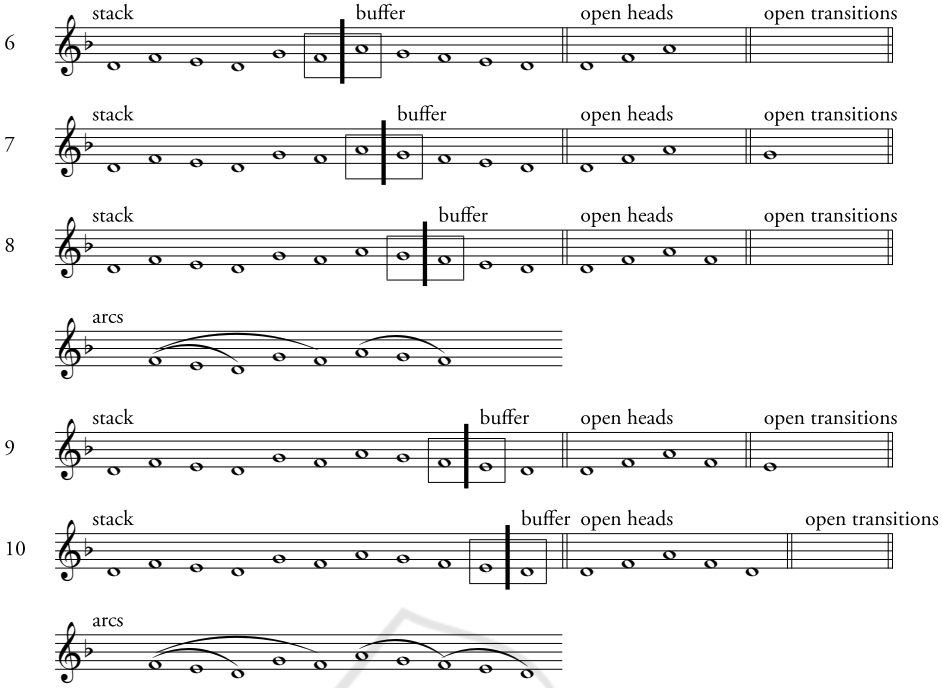

Figure 5: Parsing Fux's Dorian cantus firmus, states 6–10.

Ideally, the parser would look for this structure as it proceeds through the notes of the line. And 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 is somewhat intelligent. It knows that it only needs to 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 (see Fig. 6).

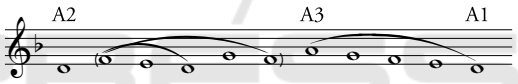

Figure 6: Parsing Fux's Dorian cantus firmus as a bass line.

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 other tonic triad pitches. 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 our 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, our 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. Figure 7 shows the parse that results when F is the candidate for A2.

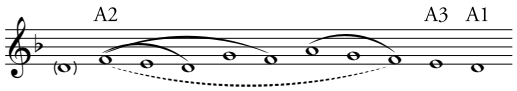

Figure 7: Parsing Fux's Dorian cantus firmus as a primary upper line from $\hat{3}$.

Our 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. Our 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 span into a single arc, as shown in Figure 8.

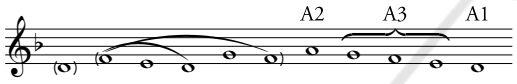

Figure 8: Parsing Fux's Dorian cantus firmus as a primary upper line from $\hat{5}$.

5 PARSING TRANSITIONS

The parser is principally concerned with evaluating transitions from one note to the next. So let us look a little closer at how the parser goes about this work. Let us call the notes I and J. The parser asks a series of questions having to do with I and J: their relation to the tonic triad, their intervallic relation, and the dynamic lists of open heads and transitions. Based on the answers, the parser assigns dependency relations, creates arcs where warranted, or returns error messages if the line is syntactically malformed.

The first questions asked of I and J are simple: Is I a tonic-triad pitch? Is J a tonic-triad pitch? If both are triad pitches, then the parser looks to see whether J can be the terminus of any open transitions; if I and J are identical in pitch, then J is interpreted as a repetition of I, else J is added to the list of open heads.

If either I or J is not a triad pitch, the parser looks to see whether the interval between I and J is a diatonic step or a consonant skip. (The parser is trained to think that simple tonal lines have no dissonant skips, so if it encounters a dissonant skip between I and J, it decides that the line is syntactically malformed and returns an error message to that effect.) The parser then considers whether there are any open transitions or open heads.

If neither I nor J is a triad pitch, the parser looks at their intervallic relation. If they form a skip of any kind or a repetition, the parser rejects the line as syntactically malformed, since the rules do not permit a skip or repetition between non-triad pitches.[7] If they form a step, the parser tries to interpret them as part of a single step motion.

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 Figure 9a, 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♮.

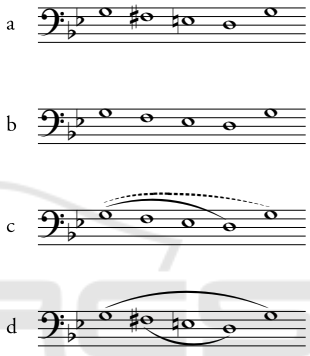

Figure 9: Some special cases in melodic minor.

Consider first how the parser would handle the line if the notes were F♮ and E♭ (Fig. 9b). 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 Figure 9c.

In the original case, 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 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

[7] In the version of third species that I teach, consonant skips between non-tonic-triad pitches are permitted within the measure, so the parser has been built to allow for this possibility.

Figure 9d.

A version of the line shown in Figure 10a was submitted by a student, and the parser initially rejected it, saying that the A was not generable in the key of C major (Fig. 10b). 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.

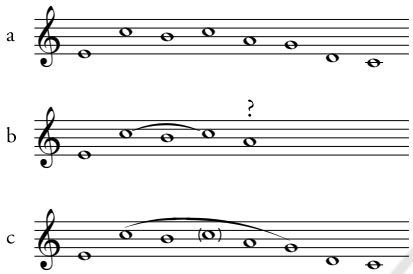

Figure 10: A case of retrospective reinterpretation.

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 listening 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 Figure 10c. 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 audition.

As already mentioned, the parser is biased toward simpler interpretations. So one of the things it will do is look to see whether there are two passing motions that share an inner node and direction. If so, it will merge them into a single arc and revise the dependency relations accordingly (Fig. 11a).
Likewise, if a neighbor motion is linked to a passing motion, the parser will embed the neighbor structure within the passing, making them both share the same lefthead (Fig. 11b).

Finally, consider the line fragment shown in Figure 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

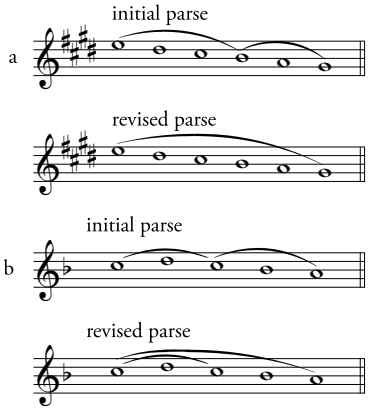

Figure 11: Two examples of bias toward simplicity.

progress, one of which attaches the B to an A, either as a left head or right head.

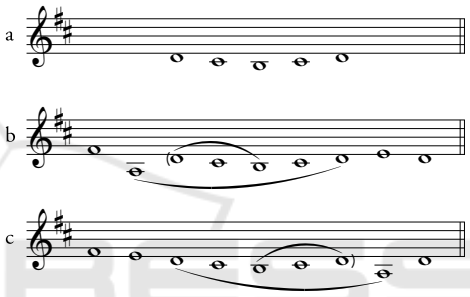

Figure 12: Handling a change of direction in mid-transition.

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 (Fig. 12b); failing that, it will need to wait and see whether there is an A later in the line that can serve as a right head, making the B a descending passing tone from the first D (Fig. 12c). In each case, the B also serves as the head of a subordinate transition to or from an inserted D.

6 IMPLEMENTATIONS OF WESTERPARSE

The main implementation of WesterParse is the web-based pedagogical application for composing and evaluating exercises in species counterpoint. This application provides the student user with reports on the syntactic validity of the individual lines and on conformity with the voice leading rules. If the parser finds that a line is syntactically invalid, it reports that fact and gives a few hints as to the nature of the syntactic problem, which the student must then solve on their own. If it finds a line is valid, it simply re-

ports that fact without providing any details on how the line can be constructed using Westergaard's rules; the student must then download the composition as a MusicXML file, open it in a music notation editor, and add annotations to indicate how the line is constructed; this ensures that the student has internalized the syntax rules. Voice leading reports detail all infractions, but it is left to the student to figure out how rectify matters. The student can also compose offline in music notation editor and then upload a MusicXML file to the WesterParse site for testing.

WesterParse can also be deployed offline in a Python environment. In this implementation, intended for music theorists, the user inputs a MusicXML file and has the option of sending the full set of legitimate parses to be displayed in a notation program. (I use the freeware program MuseScore, but just about any program that accepts MusicXML files could be used.) Having the option to display the full set of final-state interpretations allows the user to assess the reliability of the algorithms. The parser in this implementation also logs every step of the process, thus allowing the user to access all of the intermediate states, like those shown in Figures 4 and 5.

7 THE WESTERPARSE CORPUS

A corpus of examples drawn from Westergaard's text was compiled in order to test the validity of the WesterParse algorithms: 48 single lines and 29 complete examples of species counterpoint. WesterParse produces some minor deviations from Westergaard's analyses, attributable to minor differences in handling ambiguity. Westergaard, for example, allows rule A1 to attach to a nonfinal pitch, whereas WesterParse does not. On the whole, however, WesterParse reproduces Westergaard's analyses, where they are provided. The corpus also includes an additional collection of 27 lines and 41 counterpoint compositions, some of my own invention and others written by students. Further testing was performed by a group of 11 students in my Fall 2020 music theory class.

Three samples from the corpus are shown in Figure 13. The first of these is taken from Westergaard's text, where it is intended to illustrate 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 arcs; having previously decided that the E in bar 2 passed to the F in bar 6, it must then reject that arc 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 restoring the C 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.

8 EXTENDING WESTERPARSE

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 legitimate.

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. In order to handle longer lines, the contextualizer 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).

Developing WesterParse into a program that can parse more complex lines that unfold in a contrapuntal texture will require additional components. 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.[8] 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

[8]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 de Valk, 2015). Also see (Temperley, 2009).

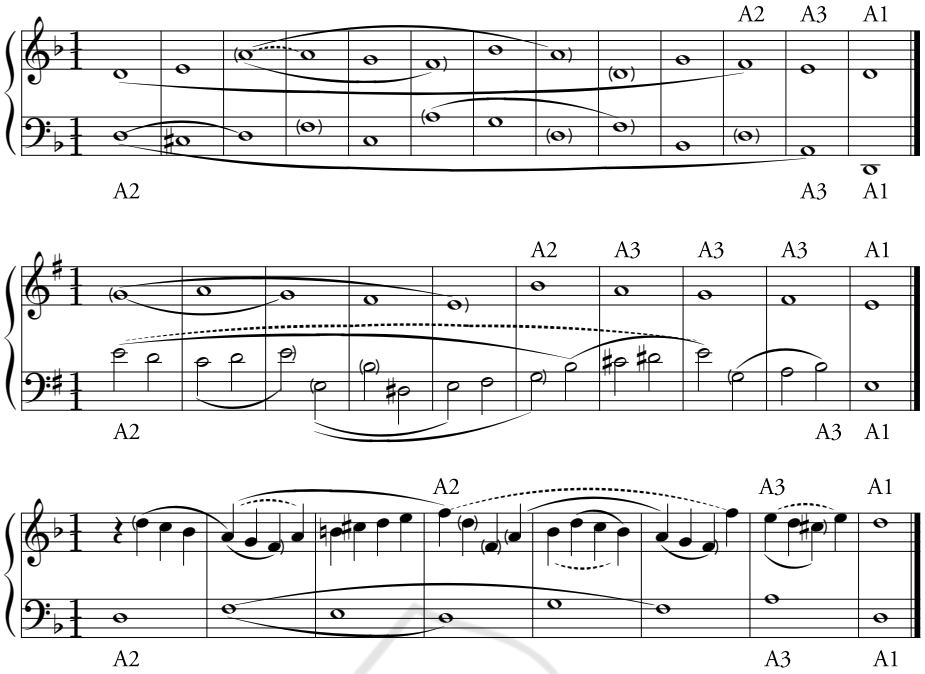

Figure 13: WesterParse's final-state interpretation of three sample exercises.

them to new notes with new timespans. The segregated streams are then sent on to individual parsers.

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 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 this in order to determine whether a pitch is to be generated as triad pitch (rules B1 and B3) or as 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.

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.[9]

REFERENCES

Cuthbert, M. S. and Ariza, C. (2010). `music21`: A toolkit for computer-aided musicology and symbolic music data. In Downie, J. S. and Veltkamp, R. C., editors, *11th International Society for Music Information Retrieval Conference (ISMIR 2010)*, pages 637–42.

Jurafsky, D. and Martin, J. H. (2008). *Speech and Language Processing*. Pearson Prentice Hall, 2nd ed. edition.

Lerdahl, F. and Jackendoff, R. (1983). *A Generative Theory of Tonal Music*. MIT Press.

[9]See (Temperley, 2007) for a review of literature on parsing meter and modulations as well as proposed solutions.

Marsden, A. (2010). Schenkerian analysis by computer: A proof of concept. *Journal of New Music Research*, 39(3):269–89.

Temperley, D. (2007). *Music and Probability*. MIT Press.

Temperley, D. (2009). A unified probabilistic model for polyphonic music analysis. *Journal of New Music Research*, 38(1):3–18.

Westergaard, P. (1975). *An Introduction to Tonal Theory*. Norton.

Weyde, T. and de Valk, R. (2015). Chord- and note-based approaches to voice separation. In Meredith, D., editor, *Computational Music Analysis*, pages 137–54. Springer.

APPENDIX

Westergaard's rules for the construction of simple, monotriadic lines. A-rules construct the basic structure. B-rules add secondary structures.

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.