A Survey of Phrase Projectivity in *Antigone*

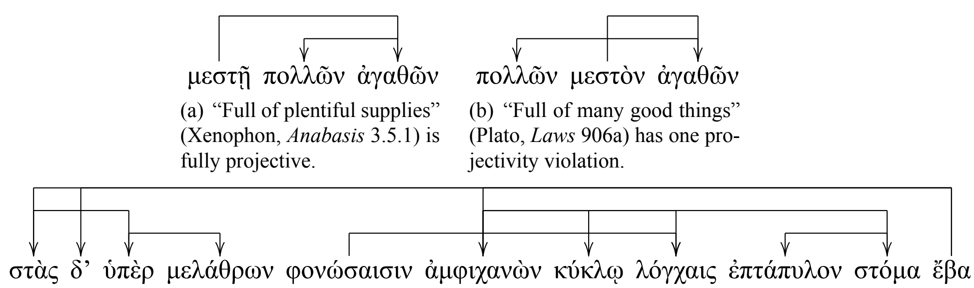
Jonathan Sterling

April 2013

In this paper, I will show how phrase projectivity (which corresponds to lacking hyperbaton) is linked to register and meter in Sophocles's *Antigone*, by developing a quantitative metric for projectivity and comparing it across lyrics, trimeters and anapaests.

1 Dependency Trees and Their Projectivity

A dependency tree encodes the head-dependent relation for a string of words, where arcs are drawn from heads to their dependents. We consider a phrase *projective* when these arcs do not cross each other, and *discontinuous* to the extent that any of the arcs intersect. Figure 1 illustrates the various kinds of projectivity violations that may occur.



(a) “Full of plentiful supplies” (Xenophon, *Anabasis* 3.5.1) is fully projective.

(b) “Full of many good things” (Plato, *Laws* 906a) has one projectivity violation.

(c) “And he stood over the rooftops, gaped in a circle with murderous spears around the seven-gated mouth, and left” (Sophocles, *Antigone* 117–120) has five projectivity violations (note that multiple arcs may intersect at a point).

Figure 1: A dependency path wrapping around itself is a projectivity violation, as in (b); interlacing adjacent phrases also violate projectivity, as in (c). Examples (a–b) drawn from Devine & Stephens.

In this paper, we shall use a concrete metric of projectivity, ω, given by the following ratio:

$$\omega = \frac{\text{number of violations}}{\text{number of arcs}}$$

The number of violations is simply the total number of intersections that occur in a tree. Appendix A deals with the development of a model and algorithm in the programming language Haskell to compute this quantity for a particular dependency tree.

2 The Perseus Treebank

The Perseus Ancient Greek Dependency Treebank is a massive trove of annotated texts that encode the all dependency relations in every sentence. The data is given in an XML (E**x**tensible **M**arkup **L**anguage) format resembling the following:

```xml
<sentence id="2900759">
  <word id="1" form="χρὴ" lemma="χρή" head="0" />
  <word id="2" form="δὲ" lemma="δέ" head="1" />
  ...
</sentence>
```

Every sentence is given a unique, sequential identifier; within each sentence, every word is indexed by its linear position and coreferenced with the linear position of its dominating head. In the case of the data for *Antigone*, the maximal head of each sentence has its own head given as `0`. Appendix B deals with parsing these XML representations into dependency trees for which we can compute ω.

3 Projectivity in Antigone

To observe the variation of projectivity within a text, then, one may make a selection of sentences that have something in common (such as meter), compute their trees and thence derive ω, and then average the results. Then that quantity may be compared with that of other selections.

To that end, I have selected passages from *Antigone* and organized them by type. Table 1 enumerates the lyric passages of the play, along with their computed mean ω values, and a final mean of means with the standard deviation of the set. Table 2 does the same for anapaests. Lastly, Table 3 gives the data for dialogue (which is in iambic trimeters), divided between medium-to-long speeches and stichomythia.

As can be seen from the data, lyrics have the highest degree of non-projectivity, followed by speeches, then anapaests, and then stichomythia. However, the standard deviation of the ω for anapaestic passages is so high that it may be difficult to say much of interest about them at all in respect to the questions that we are considering. So, let

Lines		ω
$100 \cdots 154$	*First choral ode*	0.64
$332 \cdots 375$	*Second choral ode*	0.40
$583 \cdots 625$	*Third choral ode*	0.44
$781 \cdots 800$	*Fourth choral ode*	0.43
$806 \cdots 816$	*Antigone's Kommos*	1.24
$823 \cdots 833$	*Antigone's Kommos (cntd.)*	0.61
$839 \cdots 882$	*Antigone's Kommos (cntd.)*	0.47
$944 \cdots 987$	*Fifth choral ode*	0.34
$1116 \cdots 1152$	*Sixth choral ode*	0.60
$1261 \cdots 1269$	*Kreon's Kommos*	0.28
$1283 \cdots 1292$	*Kreon's Kommos (cntd.)*	0.77
$1306 \cdots 1311$	*Kreon's Kommos (cntd.)*	0.29
$1317 \cdots 1325$	*Kreon's Kommos (cntd.)*	0.98
$1239 \cdots 1246$	*Kreon's Kommos (cntd.)*	0.44
		mean $\omega = 0.57$, sdev $= 0.27$

Table 1: Lyrics, including odes and kommoi.

Lines		ω
$155 \cdots 161$	*Kreon's Entrance*	0.17
$376 \cdots 383$	*Antigone's Entrance*	0.62
$526 \cdots 530$	*Ismene's Entrance*	0.05
$626 \cdots 630$	*Haimon's Entrance*	0.44
$801 \cdots 805$	*Antigone's Entrance*	1.08
$817 \cdots 822$	*Chorus to Antigone*	0.46
$834 \cdots 838$	*Chorus to Antigone*	0.03
$929 \cdots 943$	*Chorus, Kreon and Antigone*	0.25
$1257 \cdots 1260$	*Chorus before Kreon's Kommos*	0.00
$1347 \cdots 1353$	*Final anapaests of the Chorus*	0.36
		mean $\omega = 0.35$, sdev $= 0.33$

Table 2: Anapaests.

(a) Speeches and Dialogue

Lines		ω
$162 \cdots 210$	Kreon: *ἄνδρες, τὰ μὲν δὴ...*	0.25
$249 \cdots 277$	Guard: *οὐκ οἶδ'· ἐκεῖ γὰρ οὔτε...*	0.34
$280 \cdots 314$	Kreon: *παῦσαι, πρὶν ὀργῆς...*	0.39
$407 \cdots 440$	Guard: *τοιοῦτον ἦν τὸ πρᾶγμ'...*	0.40
$450 \cdots 470$	Antigone: *οὐ γάρ τί μοι Ζεὺς...*	0.39
$473 \cdots 495$	Kreon: *ἀλλ' ἴσθι τοι...*	0.49
$639 \cdots 680$	Kreon: *οὕτω γὰρ, ὦ παῖ...*	0.44
$683 \cdots 723$	Haimon: *πάτερ, θεοὶ φύουσιν...*	0.38
$891 \cdots 928$	Antigone: *ὦ τύμβος, ὦ νυμφεῖον...*	0.31
$998 \cdots 1032$	Teiresias: *γνώσῃ, τέχνης σημεῖα...*	0.35
$1033 \cdots 1047$	Kreon: *ὦ πρέσβυ, πάντες...*	0.18
$1064 \cdots 1090$	Teiresias: *ἀλλ' εὖ γέ τοι...*	0.64
$1155 \cdots 1172$	Messenger: *Κάδμου πάροικοι καὶ...*	0.35
$1192 \cdots 1243$	Messenger: *ἐγώ, φίλη δέσποινα...*	0.27
		mean $\omega = 0.37$, sdev = 0.11

(b) Stichomythia

Lines		ω
$536 \cdots 576$	*Ismene, Antigone and Kreon*	0.23
$728 \cdots 757$	*Haimon and Kreon*	0.27
$991 \cdots 997$	*Kreon and Teiresias*	0.53
$1047 \cdots 1063$	*Kreon and Teiresias*	0.09
$1172 \cdots 1179$	*Chorus and Messenger*	0.16
		mean $\omega = 0.26$, sdev = 0.17

Table 3: Dialogue (Trimeters)

us put the anapaests aside for the moment and deal exclusively with lyrics, speeches and stichomythias.

Whereas in prose, hyperbaton corresponds to *strong focus*, which "does not merely fill a gap in the addressee's knowledge but additionally evokes and excludes alternatives" (Devine & Stephens 303), hyperbaton in verse only entails weak focus, which emphasizes but does not exclude (ibid. 107).

As a result, hyperbaton in verse may be used to evoke a kind of elevated style without incidentally entailing more emphasis and other pragmatic effects than intended. And so it should not be surprising that lyric passages, which reside in the most poetic and elevated register present in tragic diction, should have proved in *Antigone* to have the highest proportion of projectivity violations.

Appendices

A Algorithm & Data Representation

Dependency trees are a recursive data structure with a head node, which may have any number of arcs drawn to further trees (this is called a *rose tree*). We represent them as a Haskell data-type as follows:

$$\textbf{data}\ \mathsf{Tree}\ \alpha = \alpha \curvearrowright [\mathsf{Tree}\ \alpha]$$

This can be read as "For all types α, a Tree of α is constructed from a *label* of type α and a *subforest* of Trees of α," where brackets are a notation for lists.

Given a tree, we can extract its root label or its subforest by pattern matching on its structure as follows:

$$\begin{aligned}
&\mathsf{getLabel} :: \mathsf{Tree}\ \alpha \to \alpha \\
&\mathsf{getLabel}\ (l \curvearrowright _) = l \\
&\mathsf{getForest} :: \mathsf{Tree}\ \alpha \to [\mathsf{Tree}\ \alpha] \\
&\mathsf{getForest}\ (_ \curvearrowright ts) = ts
\end{aligned}$$

A.1 From Edges to Trees

We shall consider each word index to be a *vertex*, and each pair of vertices to be an Edge, which we shall write as follows:

$$\textbf{data}\ \mathsf{Edge}\ \alpha = \alpha \leftrightarrow \alpha\ \textbf{deriving}\ \mathsf{Eq}$$

An $\mathsf{Edge}\ \alpha$ is given by two vertices of type α; the $\textbf{deriving}\ \mathsf{Eq}$ statement generates the code that is necessary to determine whether or not two Edges are equal using the $(\equiv)$ operator. In order to perform our analysis, we should wish to transform the raw list of edges into a tree structure. The basic procedure is as follows:

First, we try to find the root vertex of the tree. This will be a vertex that is given as the head of one of the words, but does not itself appear in the sentence:

$$\begin{aligned}
&\mathsf{rootVertex} :: \mathsf{Eq}\ \alpha \Rightarrow [\mathsf{Edge}\ \alpha] \to \mathsf{Maybe}\ \ \alpha \\
&\mathsf{rootVertex}\ es = \mathsf{find}\ (\notin \mathsf{deps})\ \mathsf{heads}\ \textbf{where} \\
&\quad \mathsf{heads} = [\![\ (\lambda(x \leftrightarrow y) \to x)\ es\ ]\!] \\
&\quad \mathsf{deps}\ \ = [\![\ (\lambda(x \leftrightarrow y) \to y)\ es\ ]\!]
\end{aligned}$$

If the data that we are working with are not well-formed, there is a chance that we will not find a root vertex; that is why the type is given as Maybe.

Then, given a root vertex, we look to find all the edges that it touches, and try to build the subtrees that are connected with those edges.

$$
\begin{aligned}
&\mathsf{onEdge} :: \mathsf{Eq}\ \alpha \Rightarrow \alpha \rightarrow \mathsf{Edge}\ \alpha \rightarrow \mathbb{B} \\
&\mathsf{onEdge}\ i\ (x \leftrightarrow y) = x \equiv i \vee y \equiv i
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{oppositeVertex} :: \mathsf{Eq}\ \alpha \Rightarrow \alpha \rightarrow \mathsf{Edge}\ \alpha \rightarrow \alpha \\
&\mathsf{oppositeVertex}\ i\ (x \leftrightarrow y) \\
&\quad \mid x \equiv i \qquad\quad = y \\
&\quad \mid otherwise = x
\end{aligned}
$$

This is done recursively until the list of edges is exhausted and we have a complete tree structure:

$$
\begin{aligned}
&\mathsf{treeFromEdges} :: \mathsf{Ord}\ \alpha \Rightarrow [\mathsf{Edge}\ \alpha] \rightarrow \mathsf{Maybe}\ (\mathsf{Tree}\ \alpha) \\
&\mathsf{treeFromEdges}\ es = [\![\ (\mathsf{buildWithRoot}\ es)\ (\mathsf{rootVertex}\ es)\ ]\!]\ \textbf{where} \\
&\quad \mathsf{buildWithRoot}\ es\ root = root \curvearrowright \mathsf{sortedChildren}\ \textbf{where} \\
&\qquad \mathsf{roots} \qquad\qquad\; = [\![\ (\mathsf{oppositeVertex}\ root)\ \mathsf{localVertices}\ ]\!] \\
&\qquad \mathsf{children} \qquad\quad\; = [\![\ (\mathsf{buildWithRoot}\ \mathsf{foreignVertices})\ \mathsf{roots}\ ]\!] \\
&\qquad \mathsf{localVertices} \quad\; = \mathsf{filter}\ (\mathsf{onEdge}\ root)\ es \\
&\qquad \mathsf{foreignVertices} = \mathsf{filter}\ (\neg \circ \mathsf{onEdge}\ root)\ es \\
&\qquad \mathsf{sortedChildren} \;\, = \mathsf{sortBy}\ (\mathsf{compare}\ \text{`on`}\ \mathsf{getLabel})\ \mathsf{children}
\end{aligned}
$$

A.2 Counting Violations: Computing ω

Violations are given as an integer tally:

$$\textbf{type}\ \mathsf{Violations} = \mathbb{Z}$$

The basic procedure for counting projectivity violations is as follows: flatten down the tree into a list of edges cross-referenced by their vertical position in the tree; then traverse the list and see how many times these edges intersect each other.

$$\textbf{type}\ \mathsf{Level} = \mathbb{Z}$$

The vertical position of a node in a tree is represented as its Level, counting backwards from the total depth of the tree. That is, the deepest node in the tree is at level 0, and the highest node in the tree is at level n, where n is the tree's depth.

$$
\begin{aligned}
&\mathsf{levels} :: \mathsf{Tree}\ \alpha \rightarrow [[\alpha]] \\
&\mathsf{levels}\ t = \mathsf{fmap}\ (\mathsf{fmap\ getLabel})\ \$ \\
&\quad \mathsf{takeWhile}\ (\neg \circ \mathsf{null})\ \$ \\
&\quad \mathsf{iterate}\ (\ggg\!= \mathsf{getForest})\ [t]
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{depth} :: \mathsf{Tree}\ \alpha \rightarrow \mathbb{Z} \\
&\mathsf{depth} = \mathsf{length} \circ \mathsf{levels}
\end{aligned}
$$

We can now annotate each node in a tree with what level it is at:

$$
\begin{aligned}
&\mathsf{annotateLevels} :: \mathsf{Tree}\ \alpha \rightarrow \mathsf{Tree}\ (\mathsf{Level}, \alpha) \\
&\mathsf{annotateLevels}\ tree = \mathsf{aux}\ (\mathsf{depth}\ tree)\ tree\ \textbf{where} \\
&\quad \mathsf{aux}\ l\ (x \curvearrowright ts) = (l, x) \curvearrowright [\![\ (\mathsf{aux}\ (l-1))\ ts\ ]\!]
\end{aligned}
$$

Then, we fold up the tree into a list of edges and levels:

$$
\begin{aligned}
&\mathsf{allEdges} :: \mathsf{Ord}\ \alpha \Rightarrow \mathsf{Tree}\ \alpha \rightarrow [(\mathsf{Level}, \mathsf{Edge}\ \alpha)] \\
&\mathsf{allEdges}\ tree = \mathsf{aux}\ (\mathsf{annotateLevels}\ tree)\ \textbf{where} \\
&\quad \mathsf{aux}\ ((\_, x) \curvearrowright ts) = ts \ggg\!= \mathsf{go}\ \textbf{where} \\
&\qquad \mathsf{go}\ t@((l, y) \curvearrowright \_) = (l, \mathsf{edgeWithRange}\ [x, y]) : \mathsf{aux}\ t
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{edgeWithRange} :: (\mathsf{Ord}\ \alpha) \Rightarrow [\alpha] \rightarrow \mathsf{Edge}\ \ \alpha \\
&\mathsf{edgeWithRange}\ xs = \mathsf{minimum}\ xs \leftrightarrow \mathsf{maximum}\ xs
\end{aligned}
$$

A handy way to think of edges annotated by levels is as a representation of the arc itself, where the vertices of the edge are the endpoints, and the level is the height of the arc.

If one end of an arc is between the ends of another, then there is a single intersection. If one arc is higher than another and the latter is in between the endpoints of the former, there is no violation; but if they are at the same level, or if the latter is higher than the former, there is a double intersection. Otherwise, there is no intersection.

$$
\begin{aligned}
&\mathsf{checkEdges} :: \mathsf{Ord}\ \alpha \Rightarrow (\mathsf{Level}, \mathsf{Edge}\ \alpha) \rightarrow (\mathsf{Level}, \mathsf{Edge}\ \alpha) \rightarrow \mathsf{Violations} \\
&\mathsf{checkEdges}\ (l, xy@(x \leftrightarrow y))\ (l', uv@(u \leftrightarrow v)) \\
&\quad \mid y \in_E uv \wedge u \in_E xy && = \mathsf{1} \\
&\quad \mid x \in_E uv \wedge v \in_E xy && = \mathsf{1} \\
&\quad \mid x \in_E uv \wedge y \in_E uv \wedge l \geqslant l' && = \mathsf{2} \\
&\quad \mid u \in_E xy \wedge v \in_E xy \wedge l \leqslant l' && = \mathsf{2} \\
&\quad \mid otherwise && = \mathsf{0}
\end{aligned}
$$

We determine whether a vertex is in the bounds of an edge using $\cdot \in_E \cdot$.

$$
\begin{aligned}
&\cdot \in_E \cdot :: \mathsf{Ord}\ \alpha \Rightarrow \alpha \to \mathsf{Edge}\ \alpha \to \mathbb{B} \\
&z \in_E x \leftrightarrow y = z > \mathsf{minimum}\ [x, y] \\
&\qquad\qquad\quad \wedge\ z < \mathsf{maximum}\ [x, y]
\end{aligned}
$$

We can now use what we've built to count the intersections that occur in a collection of edges. This is done by adding up the result of checkEdges of the combination of each edge with the subset of edges which are at or below its level:

$$
\begin{aligned}
&\mathsf{edgeViolations} :: \mathsf{Ord}\ \alpha \Rightarrow [(\mathsf{Level}, \mathsf{Edge}\ \alpha)] \to \mathsf{Violations} \\
&\mathsf{edgeViolations}\ xs = \mathsf{sum}\ [\![\ \mathsf{violationsWith}\ xs\ ]\!]\ \textbf{where} \\
&\quad \mathsf{rangesBelow}\ (l, \_) = \mathsf{filter}\ (\lambda(l', \_) \to l' \leqslant l)\ xs \\
&\quad \mathsf{violationsWith}\ x \quad = \mathsf{sum}\ [\![\ (\mathsf{checkEdges}\ x)\ (\mathsf{rangesBelow}\ x)\ ]\!]
\end{aligned}
$$

Finally, ω is computed for a tree as follows:

$$
\begin{aligned}
&\omega :: \mathsf{Ord}\ \alpha \Rightarrow \mathsf{Tree}\ \alpha \to \mathbb{Q} \\
&\omega\ tree = \frac{\mathsf{edgeViolations\ edges}}{\mathsf{length\ edges}}\ \textbf{where} \\
&\quad \mathsf{edges} = \mathsf{allEdges}\ tree
\end{aligned}
$$

B Parsing the Perseus Treebank

We can express the general shape of such a document as follows:

$$\textbf{type}\ \mathsf{Document} = [\mathsf{Sentence}]$$
$$\textbf{data}\ \mathsf{Sentence} = \mathsf{Sentence}\ \{\mathsf{sentenceId} :: \mathbb{Z}, \mathsf{sentenceEdges} :: [\mathsf{Edge}\ \mathbb{Z}]\}\ \textbf{deriving}\ \mathsf{Show}$$

To construct a Document from the contents of an XML file, it suffices to find all of the sentences.

$$\mathsf{documentFromXML} :: [\mathsf{Content}] \rightarrow \mathsf{Document}$$
$$\mathsf{documentFromXML}\ xml = \mathsf{catMaybes}\ [\![\ \mathsf{sentenceFromXML}\ \mathsf{elems}\ ]\!]\ \textbf{where}$$
$$\quad \mathsf{elems} = \mathsf{onlyElems}\ xml \ggg\!= \mathsf{findElements}\ (\mathsf{simpleName}\ \texttt{"sentence"})$$

Sentences are got by taking the contents of their `id` attribute, and extracting edges from their children.

$$\mathsf{sentenceFromXML} :: \mathsf{Element} \rightarrow \mathsf{Maybe}\ \mathsf{Sentence}$$
$$\mathsf{sentenceFromXML}\ e = [\![\ \mathsf{Sentence}\ (\mathsf{readAttr}\ \texttt{"id"}\ e)\ (\mathsf{pure}\ \mathsf{edges})\ ]\!]\ \textbf{where}$$
$$\quad \mathsf{edges} \quad = \mathsf{catMaybes}\ [\![\ \mathsf{edgeFromXML}\ \mathsf{children}\ ]\!]$$
$$\quad \mathsf{children} = \mathsf{findChildren}\ (\mathsf{simpleName}\ \texttt{"word"})\ e$$

An edge is got from an element by taking the contents of its `id` attribute with the contents of its `head` attribute.

$$\mathsf{edgeFromXML} :: \mathsf{Element} \rightarrow \mathsf{Maybe}\ (\mathsf{Edge}\ \mathbb{Z})$$
$$\mathsf{edgeFromXML}\ e =$$
$$\quad \textbf{case}\ \mathsf{findAttr}\ (\mathsf{simpleName}\ \texttt{"form"})\ e\ \textbf{of}$$
$$\qquad \mathsf{Just}\ x \mid x \in [\texttt{"."}, \texttt{","}, \texttt{";"}, \texttt{":"}] \rightarrow \mathsf{Nothing}$$
$$\qquad otherwise \rightarrow [\![\ (\mathsf{readAttr}\ \texttt{"head"}\ e)\ \leftrightarrow\ (\mathsf{readAttr}\ \texttt{"id"}\ e)\ ]\!]$$

Thence, turn a sentence into a tree by its edges using the machinery from Section A.1.

$$\mathsf{treeFromSentence} :: \mathsf{Sentence} \rightarrow \mathsf{Maybe}\ (\mathsf{Tree}\ \mathbb{Z})$$
$$\mathsf{treeFromSentence}\ (\mathsf{Sentence}\ \_\ ws) = \mathsf{treeFromEdges}\ ws$$

By applying treeFromSentence to every sentence within a document, we can generate all the trees in a document.

$$\mathsf{treesFromDocument} :: \mathsf{Document} \rightarrow [\mathsf{Tree}\ \mathbb{Z}]$$
$$\mathsf{treesFromDocument}\ ss = \mathsf{catMaybes}\ [\![\ \mathsf{treeFromSentence}\ ss\ ]\!]$$

By combining the above, we also may derive a document structure from a file on disk.

$$\begin{aligned}
&\mathsf{documentFromFile} :: \mathsf{FilePath} \to \mathsf{IO\ Document}\\
&\mathsf{documentFromFile}\ path = [\![\ (\mathsf{documentFromXML} \circ \mathsf{parseXML})\ (\mathsf{readFile}\ path)\ ]\!]
\end{aligned}$$

C Analysis of Data

We compute the mean ω of the trees contained in a document as follows:

$$\begin{aligned}
&\mathsf{analyzeDocument} :: \mathsf{Document} \to \mathbb{Q}\\
&\mathsf{analyzeDocument}\ doc = \mathsf{mean}\ [\![\ \omega\ (\mathsf{treesFromDocument}\ doc)\ ]\!]
\end{aligned}$$

We will wish to compare the ω for parts of *Antigone*. A section is given by a two sentence indices (a beginning and an end):

$$\textbf{data}\ \mathsf{Section} = \mathbb{Z} \cdots \mathbb{Z}$$

Then, the entire document can be cut down into smaller documents by section:

$$\begin{aligned}
&\mathsf{restrictDocument} :: \mathsf{Section} \to \mathsf{Document} \to \mathsf{Document}\\
&\mathsf{restrictDocument}\ (start \cdots finish) = \mathsf{filter\ withinSection}\ \textbf{where}\\
&\quad \mathsf{withinSection}\ (\mathsf{Sentence}\ i\ _) = i \geqslant start \wedge i \leqslant finish
\end{aligned}$$

Auxiliary Functions

$$\begin{aligned}
&\mathsf{simpleName} :: \mathsf{String} \to \mathsf{QName}\\
&\mathsf{simpleName}\ s = \mathsf{QName}\ s\ \mathsf{Nothing\ Nothing}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{readAttr} :: \mathsf{Read}\ \alpha \Rightarrow \mathsf{String} \to \mathsf{Element} \to \mathsf{Maybe}\ \ \alpha\\
&\mathsf{readAttr}\ n = \mathsf{fmap\ read} \circ \mathsf{findAttr}\ (\mathsf{simpleName}\ n)
\end{aligned}$$

$$\begin{aligned}
&\mathsf{mean} :: \mathsf{Fractional}\ n \Rightarrow [n] \to n\\
&\mathsf{mean} = [\![\ \mathsf{sum}\ \ /\ \ \mathsf{length}\ ]\!]
\end{aligned}$$

$$\begin{aligned}
&\mathsf{sdev} :: \mathsf{Floating}\ n \Rightarrow [n] \to n\\
&\mathsf{sdev}\ xs = \sqrt{\frac{\mathsf{sum}\ [\![\ (\lambda x \to x^2)\ [\![\ (-(\mathsf{mean}\ xs)+)\ xs\ ]\!]\ ]\!]}{\mathsf{length}\ xs - 1}}
\end{aligned}$$