A Survey of Phrase Projectivity in the *Antigone*

Jonathan Sterling

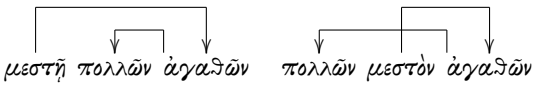
April 2013

THIS IS A DRAFT. It is currently lacking a bibliography, and the analysis is a bit just-so, I'm afraid. I also intend to add a section about how specific choices made about data representation by Perseus annotators can affect such an analysis as I have given here.

In this paper, I will show how, and to what degree, phrase projectivity corresponds with register and meter in Sophocles's *Antigone*, by developing a quantitative metric for projectivity and comparing it across lyrics, trimeters and anapaests using the data provided by the Perseus Ancient Greek Dependency Treebank. In the appendices, the formal algorithm for the computations done herein is developed in the programming language Haskell.

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 is a minimal pair that demonstrates how hyperbaton introduces a projectivity violation; in this case, a path of dependency "wraps around itself".



(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.

Figure 1: Examples drawn from Devine & Stephens.

In addition to the above, adjacent phrases (that is, phrases at the same level in the tree) may interlace, causing projectivity violations. This is commonly introduced by

Wackernagel's Law, as in Figure 2, where the placement of μὲν δὴ interlaces with the τὰ...πόλεος phrase.

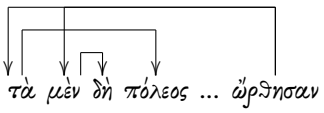


Figure 2: "[The gods] righted the matters of the city..." (*Antigone* 162–163) has one projectivity violation, due to the μὲν δὴ falling in Wackernagel's Position.

We consider a violation to have occured for each and every intersection of lines on such a drawing; thus, the hyperbaton of one word may introduce multiple violations. Consider, for instance, Figure 3, in which five violations are brought about by the displacement of φονώσαισιν. In this way, the number of intersections is a good heuristic for judging the severity of hyperbata.

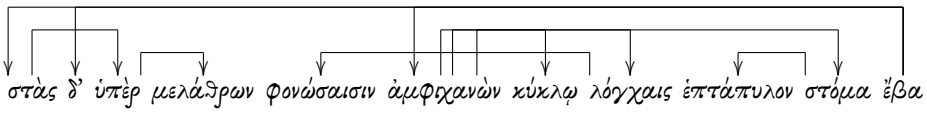


Figure 3: "And he stood over the rooftops, gaped in a circle with murderous spears around the seven-gated mouth, and left" (*Antigone* 117–120) has six projectivity violations, five of which are induced by the hyperbaton of φονώσαισιν, and one from the usual placement of δ' in Wackernagel's Position.

1.1 Counting Violations

Drawing trees and counting intersections is time-consuming and error-prone, especially since the number of intersections may vary if one is not consistent with the relative height of arcs and placement of endpoints. It is clear, then, that a computer ought to be able to do the job faster and more accurately than a human, given at least the head-dependent relations for a corpus.

The formal algorithm for counting the number of intersections is given in Appendix A.2, but I shall reproduce an informal and mostly nontechnical version of it here. First, we index each word in the sentence by its linear position, and cross-reference it with the linear position of its head:

στὰς δ'	ὑπὲρ	μελάθρων	φονώσαισιν	ἀμφικανὼν	κύκλῳ	λόγχαις	ἑπτάπυλον
1:11 2:11	3:1	4:3	5:8	6:11	7:6	8:6	9:10

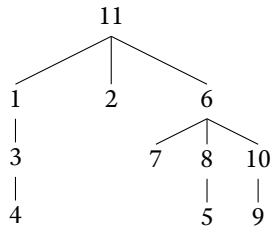


Figure 4: The dependency relations arranged in a non-linear tree.

level	*edges*
1	$3 \leftrightarrow 4, 5 \leftrightarrow 8, 9 \leftrightarrow 10$
2	$1 \leftrightarrow 3, 6 \leftrightarrow 7, 6 \leftrightarrow 8, 6 \leftrightarrow 10$
3	$1 \leftrightarrow 11, 2 \leftrightarrow 11, 6 \leftrightarrow 11$

Table 1: Edges of the tree in Figure 4 arranged by level.

στόμα ἔβα
10:6 11:_

Next, arrange the dependencies into a tree as in Figure 4. Then, counting upwards from the lowest edges (i.e. the lines) in the tree up to the topmost ones, make a list of edges indexed by vertical level as in Table 1.

Then, each edge in our table must be checked for violations against all the other edges in the table except those which are in a level higher than it. The level of the edge corresponds with the height at which we drew the arcs; this condition arises out of the fact that an arc cannot cross an arc that is above it.

Next, we must figure out all the possible ways for an arc to intersect another at given levels. These are enumerated in detail in the function checkEdges in Appendix A.2, but suffice it to say that they fall into a few main categories:

1. Both vertices of the higher edge are within the bounds of the lower edge. This is a double violation, as both sides of an arc will extrude through another.

2. One vertex of the higher edge is within the bounds of the lower edge, and the other vertex is not; this vertex is allowed to be equal to the second vertex of the lower edge. In either case, this is a single violation, as just one intersection occurs.

3. The edges are at the same level, and one vertex of the higher edge is neither within bounds of the other, nor equal to any of the vertices of the other. This is a single violation.

$2 \leftrightarrow 11$	$1 \leftrightarrow 3$	1
$6 \leftrightarrow 11$	$5 \leftrightarrow 8$	1
$6 \leftrightarrow 7$	$5 \leftrightarrow 8$	2
$6 \leftrightarrow 8$	$5 \leftrightarrow 8$	1
$6 \leftrightarrow 10$	$5 \leftrightarrow 8$	1
		$\mathsf{total} = 6$

Table 2: Projectivity violations which arise from the edges in Table 1.

Using this procedure, we shall have found the edge violations which are listed in Table 2, which are 6 in total.

1.2 $\wp$: a metric of projectivity

In order for our view of a text's overall projectivity to not be skewed by its length, we must have a ratio. For the purposes of this paper, we shall call this metric $\lambda\wp$, as given by the following ratio:

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

Now, this metric applies just as much to a single sentence as it does to a larger body of text. So, averages of $\wp$ should not be taken; rather, total numbers of violations and total numbers of arcs should be accumulated until $\wp$ may be computed for the entire body of text being examined.

2 The Perseus Treebank

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

```
<sentence id="2900759">
  <word id="1" form="λέγὴ" lemma="λέγή" 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 the *Antigone*, the maximal head of each

sentence has its own head given as `0`. Appendix B.1 deals with parsing these XML representations into dependency trees for which we can compute $\wp$.

The Perseus data also includes punctuation in the dependency trees, which we must of course filter out; a comma, for instance, may induce a technical hyperbaton, simply by virtue of what the Perseus editors have chosen to mark as its "head", to the extent that it means anything at all for a punctuation mark to have a head.

3 Projectivity in the *Antigone*

To observe the variation of projectivity within a text, then, one may make a selection of sentences that have something in common, compute their trees and thence derive a cumulative $\wp$ for the entire selection. Then that figure may be compared with that of other selections.

I have chosen to compare projectivity in lyrics, anapaests and trimeters. Lyrics I have divided into two categories: choral odes and laments, whereas I divide trimeters into medium-to-long speeches and stichomythia.

To that end, I have selected passages from the *Antigone* and organized them by type. Table 3 enumerates the lyric passages of the play, along with their computed $\wp$ values, and a cumulative $\wp$ value for the entire set. Table 4 does the same for anapaests. Lastly, Table 5 gives selections of 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. To try and understand why this is the case, it will be useful to discuss Greek hyperbaton in more general terms.

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 the *Antigone* to have the highest proportion of projectivity violations.

Within the lyric passages, the laments appear to have consistently higher $\wp$ than the choral odes, which may stem from their being much more emotive and personal

(a) Choral Odes

Lines		$\wp$
$100 \cdots 154$	*First choral ode*	0.83
$332 \cdots 375$	*Second choral ode*	0.58
$583 \cdots 625$	*Third choral ode*	0.71
$781 \cdots 800$	*Fourth choral ode*	0.67
$944 \cdots 987$	*Fifth choral ode*	0.47
$1116 \cdots 1152$	*Sixth choral ode*	0.64
	cumulative $\wp = 0.66$	

(b) Laments

Lines		$\wp$
$806 \cdots 816$	*Antigone's Lament*	1.37
$823 \cdots 833$	*Antigone's Lament (cntd.)*	0.78
$839 \cdots 882$	*Antigone's Lament (cntd.)*	0.63
$1261 \cdots 1269$	*Kreon's Lament*	0.38
$1283 \cdots 1292$	*Kreon's Lament (cntd.)*	1.34
$1306 \cdots 1311$	*Kreon's Lament (cntd.)*	0.37
$1317 \cdots 1325$	*Kreon's Lament (cntd.)*	1.13
$1239 \cdots 1246$	*Kreon's Lament (cntd.)*	0.60
	cumulative $\wp = 0.78$	

Table 3: Lyrics

Lines		$\wp$
$155 \cdots 161$	*Kreon's Entrance*	0.33
$376 \cdots 383$	*Antigone's Entrance*	0.91
$526 \cdots 530$	*Ismene's Entrance*	0.05
$626 \cdots 630$	*Haimon's Entrance*	0.91
$801 \cdots 805$	*Antigone's Entrance*	1.16
$817 \cdots 822$	*Chorus to Antigone*	0.57
$834 \cdots 838$	*Chorus to Antigone*	0.05
$929 \cdots 943$	*Chorus, Kreon and Antigone*	0.25
$1257 \cdots 1260$	*Chorus before Kreon's Kommos*	0.00
$1347 \cdots 1353$	*Final anapaests of the Chorus*	0.31
	cumulative $\wp = 0.47$	

Table 4: Anapaests.

(a) Speeches and Dialogue

Lines		$\wp$
162 · · · 210	Kreon: ἄνδρες, τὰ μὲν δὴ...	0.40
249 · · · 277	Guard: οὐκ οἶδ'· ἐκεῖ γὰρ οὔτε...	0.57
280 · · · 314	Kreon: παῦσαι, πρὶν ὀργῆς...	0.45
407 · · · 440	Guard: τοιοῦτον ἦν τὸ πρᾶγμ'...	0.56
450 · · · 470	Antigone: οὐ γάρ τί μοι Ζεὺς...	0.56
473 · · · 495	Kreon: ἀλλ' ἴσθι τοι...	0.45
639 · · · 680	Kreon: οὕτω γὰρ, ὦ παῖ...	0.55
683 · · · 723	Haimon: πάτερ, θεοὶ φύουσιν...	0.45
891 · · · 928	Antigone: ὦ τύμβος, ὦ νυμφεῖον...	0.48
998 · · · 1032	Teiresias: γνώσῃ, τέχνης σημεῖα...	0.57
1033 · · · 1047	Kreon: ὦ πρέσβυ, πάντες...	0.24
1064 · · · 1090	Teiresias: ἀλλ' εὖ γέ τοι...	0.82
1155 · · · 1172	Messenger: Κάδμου πάροικοι καὶ...	0.51
1192 · · · 1243	Messenger: ἐγώ, φίλη δέσποινα...	0.40
	cumulative $\wp$ =	0.50

(b) Stichomythia

Lines		$\wp$
536 · · · 576	*Ismene, Antigone and Kreon*	0.29
728 · · · 757	*Haimon and Kreon*	0.38
991 · · · 997	*Kreon and Teiresias*	0.70
1047 · · · 1063	*Kreon and Teiresias*	0.12
1172 · · · 1179	*Chorus and Messenger*	0.29
	cumulative $\wp$ =	0.32

Table 5: Dialogue (Trimeters)

in nature. It should be noted that, whilst the individual odes conform tightly to the cumulative $\wp$ of their category, there is a fair degree of variation among the laments. Likewise, the anapaests vary so wildly in their $\wp$ that it may be difficult to say very much about them that is relevant to the questions we are considering.

As for dialog, longer-form speeches are largely conformant in their $\wp$, with stichomythias varying a bit more. Speeches are a somewhat less projective than the stichomythias, being typically more eloquent and long-winded than their argumentative, choppy counterparts.

So far, the most surprising thing about the data is the degree to which certain verse-types vary in $\wp$ (or, if you like, the degree to which other types *don't*). The data draw us, then, to the following conclusions:

1. Non-projectivity varies within a single metrical type (lyrics, iambic trimeters, anapaests).

2. Certain registers seem to be more conventionalized with respect to $\wp$ than others; that is, choral odes and speeches do not vary greatly amongst themselves, but laments and anapaests do.

Lyric passages are in general less projective than anything else, but some laments reach a degree of non-projectivity that exceeds the most elliptical odes in the *Antigone*. Further, within the trimeters, speeches are less projective than stichomythias. From these things, then, we can say that that meter itself would not seem to be a primary factor for predicting incidence and severity of hyperbaton, but rather a secondary one at best.

That is to say, we know for a fact that passages in lyric meters have greater $\wp$ than passages in other meters. Yet, the variation of $\wp$ within that very meter indicates that there is some other factor involved, which very likely has to do with register along two different dimensions, which is to say, relative "dignity of style" and emotive force.

With regard to the very low $\wp$ found in the stichomythias, I suggest that it is the necessary shortness of each utterance which is at fault here. That is, the maximum "damage" that a hyperbaton can do is greatly lessened, when the ultimate depth of the phrase structure is limited by its length (whence, for instance, it is unlikely for a single hyperbaton to cause more than a few projectivity violations).

Appendices

The functions used in parsing and computing the data for this paper are developed here in the programming language Haskell. Haskell is a typed lambda calculus with inductive data types and type classes; the listings below use standard Haskell syntax with the exception of some infix operators to improve readability, and the addition of so-called "idiom brackets", which allow a more syntactically clean presentation of function application within a context.

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 Edge α is given by two vertices of type α; the **deriving** 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:

```
rootVertex :: Eq α ⇒ [Edge α] → Maybe α
rootVertex es = find (∉ deps) heads where
  heads = ⟦ (λ(x ↔ y) → x) es ⟧
  deps  = ⟦ (λ(x ↔ y) → y) es ⟧
```

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.

```
onEdge :: Eq α ⇒ α → Edge α → 𝔹
onEdge i (x ↔ y) = x ≡ i ∨ y ≡ i

oppositeVertex :: Eq α ⇒ α → Edge α → α
oppositeVertex i (x ↔ y)
  | x ≡ i       = y
  | otherwise = x
```

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

```
treeFromEdges :: Ord α ⇒ [Edge α] → Maybe (Tree α)
treeFromEdges es = ⟦ (buildWithRoot es) (rootVertex es) ⟧ where
  buildWithRoot es root = root ↷ sortedChildren where
    roots           = ⟦ (oppositeVertex root) localVertices ⟧
    children        = ⟦ (buildWithRoot foreignVertices) roots ⟧
    localVertices   = filter (onEdge root) es
    foreignVertices = filter (¬ ∘ onEdge root) es
    sortedChildren  = sortBy (compare `on` getLabel) children
```

A.2 Counting Violations

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.

```
type Level = 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.

```
levels :: Tree α → [[α]]
levels t = fmap (fmap getLabel) $
  takeWhile (¬ ∘ null) $
  iterate (>>=getForest) [t]


depth :: Tree α → ℤ
depth = length ∘ levels
```

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

```
annotateLevels :: Tree α → Tree (Level, α)
annotateLevels = ⟦ aux depth id ⟧ where
  aux l (x ↷ ts) = (l, x) ↷ ⟦ (aux (l − 1)) ts ⟧
```

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

```
allEdges :: Ord α ⇒ Tree α → [(Level, Edge α)]
allEdges = aux ∘ annotateLevels where
  aux ((_, x) ↷ ts) = ts >>= go where
    go t@((l, y) ↷ _) = (l, edgeWithRange [x, y]) : aux t


edgeWithRange :: Ord α ⇒ [α] → Edge α
edgeWithRange = ⟦ minimum ↔ maximum ⟧
```

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. Now, we can count the violations that occur between two arcs.

```
checkEdges :: Ord α ⇒ (Level, Edge α) → (Level, Edge α) → ℤ
checkEdges (l, xy@(x ↔ y)) (l', uv@(u ↔ v))
  | x ∈E uv ∧ ((y ⩾ v ∧ l > l') ∨ y > v) = 1
  | y ∈E uv ∧ ((x ⩽ u ∧ l > l') ∨ u < u) = 1
  | u ∈E xy ∧ ((v ⩾ y ∧ l < l') ∨ v > y) = 1
  | v ∈E xy ∧ ((u ⩽ x ∧ l < l') ∨ u < x) = 1
  | x ∈E uv ∧ y ∈E uv ∧ l ⩾ l'          = 2
```

$$
\begin{aligned}
&\mid u \in_E xy \wedge v \in_E xy \wedge l \leqslant l' && = 2 \\
&\mid otherwise && = 0
\end{aligned}
$$

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

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

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:

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

A.3 Computing $\wp$

We introduce a data type $\mathfrak{P}$ of integer-to-integer ratios which may be computed into a rational.

$$
\begin{aligned}
&\textbf{data}\ \mathfrak{P} = \mathfrak{P}\ \{\mathsf{violationCount} :: \mathbb{Z}, \mathsf{edgeCount} :: \mathbb{Z}\} \\
&\mathsf{compute}_\omega :: \mathfrak{P} \to \mathbb{Q} \\
&\mathsf{compute}_\omega = \left[\!\!\left[\ \frac{\mathsf{violationCount}}{\mathsf{edgeCount}}\ \right]\!\!\right]
\end{aligned}
$$

Furthermore, $\mathfrak{P}$s generate a monoid, which is an algebraic structure that abstracts out the notion of an identity and an associative binary operation that respects that identity. In this way, we can combine $\mathfrak{P}$ values:

$$
\begin{aligned}
&\textbf{instance}\ \mathsf{Monoid}\ \mathfrak{P}\ \textbf{where} \\
&\quad \mathcal{E} = \mathfrak{P}\ 0\ 0 \\
&\quad (\mathfrak{P}\ x\ y) \oplus (\mathfrak{P}\ u\ v) = \mathfrak{P}\ (x + u)\ (y + v)
\end{aligned}
$$

Finally, $\wp$ may be computed for trees.

$$
\begin{aligned}
&\wp :: \mathsf{Ord}\ \alpha \Rightarrow \mathsf{Tree}\ \alpha \to \mathfrak{P} \\
&\wp = [\![\ \mathfrak{P}\ \mathsf{edgeViolations}\ \mathsf{length}\ ]\!] \circ \mathsf{allEdges}
\end{aligned}
$$

B Working with the Perseus Treebank

B.1 Parsing the XML

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

```
type Document = [Sentence]
data Sentence = Sentence {sentenceId :: ℤ, sentenceEdges :: [Edge ℤ]}
```

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

```
documentFromXML :: [Content] → Document
documentFromXML xml = catMaybes ⟦ sentenceFromXML elems ⟧ where
  elems = onlyElems xml ≫= findElements (simpleName "sentence")
```

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

```
sentenceFromXML :: Element → Maybe Sentence
sentenceFromXML e = ⟦ Sentence (readAttr "id" e) (pure edges) ⟧ where
  edges    = catMaybes ⟦ edgeFromXML children ⟧
  children = findChildren (simpleName "word") e
```

An edge is got from an element by taking the contents of its `id` attribute with the contents of its `head` attribute. We make sure to filter out punctuation which would skew our data.

```
edgeFromXML :: Element → Maybe (Edge ℤ)
edgeFromXML e =
  case findAttr (simpleName "form") e of
    Just x | x ∈ [".",",",";",":"] → Nothing
    otherwise → ⟦ (readAttr "head" e) ↔ (readAttr "id" e) ⟧
```

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

```
treeFromSentence :: Sentence → Maybe (Tree ℤ)
treeFromSentence (Sentence _ ws) = treeFromEdges ws
```

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

```
treesFromDocument :: Document → [Tree ℤ]
treesFromDocument ss = catMaybes ⟦ treeFromSentence ss ⟧
```

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

```
documentFromFile :: FilePath → IO Document
documentFromFile path = ⟦ (documentFromXML ∘ parseXML) (readFile path) ⟧
```

B.2 Analysis of Data

We compute the cumulative $\wp$ of the trees contained in a document as follows:

```
analyzeDocument :: Document → 𝔓
analyzeDocument doc = mconcat ⟦ ℘ (treesFromDocument doc) ⟧
```

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

```
data Section = ℤ ··· ℤ
```

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

```
restrictDocument :: Section → Document → Document
restrictDocument (start ··· finish) = filter withinSection where
  withinSection (Sentence i _) = i ⩾ start ∧ i ⩽ finish
```