On Parsing as Tagging

Afra Amini Ryan Cotterell

{afra.amini, ryan.cotterell}@inf.ethz.ch

Abstract

There have been many proposals to reduce constituency parsing to tagging in the literature. To better understand what these approaches have in common, we cast several existing proposals into a unifying pipeline consisting of three steps: linearization, learning, and decoding. In particular, we show how to reduce tetratagging, a state-of-the-art constituency tagger, to shift-reduce parsing by performing a right-corner transformation on the grammar and making a specific independence assumption. Furthermore, we empirically evaluate our taxonomy of tagging pipelines with different choices of linearizers, learners, and decoders. Based on the results in English and a set of 8 typologically diverse languages, we conclude that the linearization of the derivation tree and its alignment with the input sequence is the most critical factor in achieving accurate taggers.

1 Introduction

The automatic syntactic analysis of natural language text is an important problem in natural language processing (NLP). Due to the ambiguity present in natural language, many parsers for natural language are statistical in nature, i.e., they provide a probability distribution over syntactic analyses for a given input sequence. A common design pattern for natural language parsers is to re-purpose tools from formal language theory that were often designed for compiler construction (Aho and Ullman, 1972), to construct probability distributions over derivation trees (syntactic analyses). Indeed, one of the most commonly deployed parsers in NLP is a statistical version of the classic shift-reduce parser, which has been widely applied in constituency parsing (Sagae and Lavie, 2005; Zhang and Clark, 2009; Zhu et al., 2013), dependency parsing (Fraser, 1989; Yamada and Matsumoto, 2003; Nivre, 2004), and even for the parsing of more exotic formalisms, e.g., CCG parsing (Zhang and Clark, 2011; Xu et al., 2014).

Another relatively recent trend in statistical parsing is the idea of reducing parsing to tagging (Gómez-Rodríguez and Vilares, 2018; Strzyz et al., 2019; Vilares et al., 2019; Vacareanu et al., 2020; Gómez-Rodríguez et al., 2020; Kitaev and Klein, 2020). There are a number of motivations behind this reduction. For instance, Kitaev and Klein (2020) motivate tetratagging, their novel constituency parsing-as-tagging scheme, by its ability to parallelize the computation during training and to minimize task-specific modeling. The proposed taggers in the literature are argued to exhibit the right balance in the trade-off between...
accuracy and efficiency. Nevertheless, there are several ways to cast parsing as a tagging problem, and the relationship of each of these ways to transition-based parsing remains underdeveloped.

To better understand the relationship between parsing as tagging and transition-based parsing, we identify a common pipeline that many parsing-as-tagging approaches seem to follow. This pipeline, as shown in Fig. 1, consists of three steps: linearization, learning, and decoding. Furthermore, we demonstrate that Kitaev and Klein’s (2020) tetratagging may be derived under two additional assumptions on a classic shift–reduce parser: (i) a right-corner transform in the linearization step and (ii) factorized scoring of the rules in the learning step. We find that (i) leads to better alignment with the input sequence, while (ii) leads to parallel execution.

In light of these findings, we propose a taxonomy of parsing-as-tagging schemes based on different choices of linearizations, learners, and decoders. Furthermore, we empirically evaluate the effect that the different choice points have on the performance of the resulting model. Based on experimental results, one of the most important factors in achieving the best-performing tagger is the order in which the linearizer enumerates the tree and the alignment between the tags and the input sequence. We show that taggers with in-order linearization are the most accurate parsers, followed by their pre-order and post-order counterparts, respectively. This result is in agreement with Liu and Zhang (2017), whose results also hint at the superiority of in-order shift–reduce parsers.

Finally, we argue that the effect of the linearization step on parsing performance can be explained by the deviation between the resulting tag sequence and the input sequence, which should be minimized. We theoretically show that in-order linearizers always generate tag sequences with zero deviation. On the other hand, the deviation of pre- and post-order linearizations in the worst case grows in order of sentence length. Empirically, experiments on a set of 9 typologically diverse languages show that the deviation varies across languages and negatively correlates with parsing performance. Indeed, we find that the deviation appears to be the most important factor in predicting parser performance.

2 Preliminaries

Before diving into the details of parsing as tagging, we introduce the notation that is used throughout the paper. We assume that we have a weighted grammar in Chomsky normal form \( \mathcal{G} = (\mathcal{N}, S, \Sigma, \mathcal{R}) \) where \( \mathcal{N} \) is a finite set of nonterminal symbols, \( S \in \mathcal{N} \) is a distinguished start symbol, \( \Sigma \) is an alphabet of terminal symbols, and \( \mathcal{R} \) is a set of productions. Each production can take on two forms, either \( X \to YZ \), where \( X, Y, Z \in \mathcal{N} \), or \( X \to x \), where \( X \in \mathcal{N} \) and \( x \in \Sigma \).

\[ \text{Let } w = w_1 w_2 \cdots w_N \text{ be the input sequence of } N \text{ words, where } w_n \in \Sigma. \text{ Next, we give a definition of a derivation tree } d_w \text{ that yields } w. \]

**Definition 1 (Derivation Tree).** A derivation tree \( d_w \) is a labeled and ordered tree. We denote the nodes in \( d_w \) as \( \xi \). The helper function \( \rho(\cdot) \) returns the label of a node. If the node is an interior node, then \( \rho(\xi) = X \text{ where } X \in \mathcal{N} \). If the node is a leaf, then \( \rho(\xi) = w \text{ where } w \in \Sigma. \)

Note that under Def. 1 a derivation tree \( d_w \) must have \( |w| = N \) leaves, which, in left-to-right order, are labeled as \( w_1, w_2, \ldots, w_N \). We additionally define the helper function \( \pi(\cdot) \) that returns the production associated with an interior node. Specifically, if \( \xi \) is an interior node of \( d_w \) in a CNF grammar, then \( \pi(\xi) \) returns either \( \rho(\xi) \) or \( \rho(\xi') \) if \( \xi \) has two children \( \xi' \) and \( \xi'' \) or \( \rho(\xi) \) if \( \xi \) has one child \( \xi'. \) We further denote the non-negative weight of each production in the grammar with score \( \geq 0. \)

We now define the following distribution over derivation trees:

\[ p(d_w \mid w) \propto \prod_{\xi \in d_w} \text{score}(\pi(\xi)) \]  

(1)

where we abuse notation to use set inclusion \( \xi \in d_w \) to denote an interior node in the derivation tree \( d_w \). In words, Eq. (1) says that the probability of a derivation tree is proportional to the product of the scores of its productions.

3 Linearization

In this and the following two sections, we go through the parsing-as-tagging pipeline and discuss the choice points at each step. We start with the first step, which is to design a linearizer. The goal of this step is to efficiently encode all we need to know about the parse tree in a sequence of tags.

---

1 We assume that the empty string is not in the yield of \( \mathcal{G} \)—otherwise, we would require a rule of the form \( S \to \varepsilon \).

2 Note that, for every \( w \), we must have at least one derivation tree with a non-zero score in order to be able to normalize the distribution.
Table 1: Examples of different linearizations for the derivation tree of the sentence *She enjoys reading papers*. Note that $a_w^{pre}$ is the action sequence of a shift–reduce parser, $\Phi_w^{pre}(d_w)$ is the pre-order linearization, $\Phi_w^{post}(d_w)$ is the post-order linearization, and $\Phi_w^{lin}(d_w)$ is the in-order linearization (tetragger).

### Definition 2 (Gómez-Rodríguez and Vilares (2018)). A linearizer is a function $\Phi_w : D_w \rightarrow \mathcal{T}^M$ that maps a derivation tree $d_w$ to a sequence of tags of length $M$, where $D_w$ is the set of derivation trees with yield $w$, $\mathcal{T}$ is the set of tags, $M$ is $O(N)$, and $N$ is the length of $w$. We further require that $\Phi_w$ is a total and injective function. This means that each derivation tree $d_w$ in $D_w$ is mapped to a unique tagging sequence $t$, but that some tagging sequences do not map back to derivation trees.

We wish to contextualize Def. 2 by contrasting parsing-as-tagging schemes, as formalized in Def. 2, with classical transition-based parsers. As written, Def. 2 subsumes many transition-based parsers, e.g., Nivre’s (2008) arc-standard and arc-eager parsers. However, in the case of most transition-based parsers, the linearizer $\Phi_w$ is a bijection between the set of derivation trees $D_w$ and the set of valid tagging sequences, which we denote as $\mathcal{T}_w \subset \mathcal{T}^M$. The reason for this is that most transition-based parsers require a global constraint to ensure that the resulting tag sequence corresponds to a valid parse.

In contrast, in Def. 2, we require a looser condition—namely, that $\Phi_w : D_w \rightarrow \mathcal{T}^M$ is an injection. Therefore, tagging-based parsers allow the prediction of invalid linearizations, i.e., sequences of tags that cannot be mapped back to a valid derivation tree. More formally, the model can place non-zero score on elements in $\mathcal{T}^M \setminus \mathcal{T}_w$. However, the hope is that the model learns to place zero score on elements in $\mathcal{T}^M \setminus \mathcal{T}_w$, and, thus, enforces a hard constraint. Therefore, parsing-as-tagging schemes come with the advantage that they make use of a simpler structure prediction framework, i.e., tagging. Nevertheless, in practice, they still learn a useful parser that only predicts valid sequences of tags due to the expressive power of the neural network backbone.

### 3.1 Pre-Order and Post-Order Linearization

In this section, we discuss how to define linearizers based on the actions that shift–reduce parsers take. A shift–reduce parser gives a particular linearization of a derivation tree via a sequence of shift and reduce actions. A top-down shift–reduce parser enumerates a derivation tree $d_w$ in a pre-order fashion, taking a $\text{shift}(X \rightarrow x)$ action when visiting a node $\xi$ with $\pi(\xi) = X \rightarrow x$, and a $\text{reduce}(X \rightarrow YZ)$ action when $\pi(\xi) = X \rightarrow Y Z$. The parser halts when every node is visited. As an example, consider the sentence: *She enjoys reading papers*, with the parse tree depicted in Table 1. The sequence of actions used by a top-down shift–reduce parser is shown under the column $a_w^{pre}$. Given the assumption that the grammar is in Chomsky normal form, the parser takes $N$ shift actions and $N - 1$ reduce actions, resulting in a sequence of length $M = 2N - 1$. From this sequence of actions, we construct the tags output by the pre-order linearization $\Phi_w^{pre}$ as follows:

1. We show reduce actions with double arrows $\Rightarrow$, $\Rightarrow$, and shift actions with $\ni$, $\ni$.

2. For shift actions, we encode whether the node being shifted is a left or a right child of

---

3Nivre (2008) discusses dependency parsers, but both arc-standard and arc-eager can be easily modified to work for constituency parsing, the subject of this work.
its parent.\footnote{We assume that part-of-speech tags are given. Otherwise, the identity of the non-terminal being shifted should be encoded in \( \kappa \) and \( \varphi \) tags as well.} We show left terminals with \( \varphi \) and right terminals with \( \kappa \).

3. For \texttt{REDUCE} actions, we encode the identity of the non-terminal being reduced as well as whether it is a left or a right child. We denote reduce actions that create non-terminals that are left (right) children as \( \varphi \) (\( \kappa \)).

The output of such a linearization \( \Phi^\text{pre}_w \) is shown in Table 1. Similarly, a bottom-up shift–reduce parser can be viewed as post-order traversal of the derivation tree; see \( \Phi^\text{post}_w \) in Table 1 for an example.

### 3.2 In-Order Linearization

On the other hand, the linearization function used in tetratagger does an in-order traversal \cite{LiuZhang2017} of the derivation tree. Similarly to the pre- and post-order linearization, when visiting each node, it encodes the direction of the node relative to its parent in the tag sequence, i.e., whether this node is a left child or a right child. For a given sentence \( w \), the tetratagging linearization \( t \) also has a length of \( M = 2N - 1 \) actions. An example can be found under the column \( \Phi^\text{w}_w \) in Table 1. The novelty of tetratagging lies in how it builds a fixed-length tagging schema using an in-order traversal of the derivation tree. While this approach might seem intuitive, the connection between this particular linearization and shift–reduce actions is not straightforward. The next derivation states that the tetratag sequence can be derived by shift–reduce parsing on a transformed derivation tree.

#### Derivation 1 (Informal). There exists a sequence transformation function that transforms bottom-up shift–reduce actions \( \alpha^\text{post}_w \) on the right-corner transformed derivation tree to tetratags.

A similar statement holds for transforming top-down shift–reduce actions on the left-corner derivation trees to tetratags; a more formal derivation can be found in App. B.

### 3.3 Deviation

The output of the linearization function is a sequence of tags \( t \). Next, we need to align this sequence with the input sequence \( w \). Inspired by \cite{GomezRodriguezetal2020}, we define an alignment function as follows.

#### Definition 3. The alignment function \( \Theta^K_w \) maps each word in an input sequence of length \( N \) to at most \( K \) tags in a tag sequence of length \( M \): \( \Theta^K_w(w, t) = (t_{11}, t_{12}, \ldots, t_{1K}, \ldots, t_{NK}) \). We say \( w_n \) is mapped to \( t_{n1}, \ldots, t_{nK} \).

Because all of the aforementioned linearizers generate tag sequences of length \( 2N - 1 \), a natural alignment schema, which we call \texttt{paired alignment} \( \Theta^1 \), is to assign two tags to each word, except for the last word, which is only assigned to one tag. Fig. 1 depicts how such an alignment is applied to a sequence of pre-order linearizations with an example input sequence.

#### Definition 4. We define deviation of \( \Theta^K_w(w, t) \) to be the distance between a word and the tag corresponding to shifting this word into the stack. Formally, for the \( n \)th word in the sequence, if \( t_{nk} = \text{SHIFT}(w_n) \), then the deviation would be \( |n - l| \). We further call a sequence of shift–reduce tags \texttt{shift-aligned} with the input sequence if and only if it has zero deviation for any given word and sentence.

Note that if the paired alignment is used with an in-order shift–reduce linearizer, as done in tetratagger, it will be shift-aligned. However, such schema will not necessarily be shift aligned when used with pre- or post-order linearizations and leads to non-zero deviations. More formally, the following three propositions hold.

#### Proposition 1. For any input sequence \( w \), the paired alignment of this sequence with the in-order linearization of its derivation tree, denoted \( \Theta^1(w, \Phi^\text{in}_w) \), has a maximum deviation of zero, i.e., it is shift-aligned.

\textit{Proof.} In a binary derivation tree, in-order traversal results in a sequence of tags where each shift is followed by a reduce. Since words are shifted from left to right, each word will be mapped to its corresponding shift tag by \( \Theta^1 \), thus creating a shift-aligned sequence of tags.

#### Proposition 2. For any input sequence \( w \) of length \( N \), the maximum deviation of the paired alignment applied to a pre-order linearization \( \Theta^1(w, \Phi^\text{pre}_w) \) in the worst case is \( O(N) \).

\textit{Proof.} The deviation is bounded above by \( O(N) \). Now consider a left-branching tree. In this case, \( \Phi^\text{pre}_w \) starts with at most \( N - 1 \) reduce tags before shifting the first word. Therefore, the maximum
deviation, which happens for the first word, is $\lfloor \frac{N}{2} \rfloor$, which is $O(N)$, achieving the upper bound.

**Proposition 3.** For any input sequence $w$ with length $N$, the maximum deviation of the paired alignment applied to a post-order linearization is $O(N)$.

**Proof.** The proof is similar to the pre-order case above, but with a right-branching tree instead of a left-branching one.

Intuitively, if the deviation is small, it means that the structural information about each word is encoded close to the position of that word in the tag sequence. Thus, it should be easier for the learner to model this information due to the locality. Therefore, we expect a better performance from taggers with shift-aligned or low deviation linearizations. Later, in our experiments, we empirically test this hypothesis.

## 4 Learning

In this section, we focus on the second step in a parsing-as-tagging pipeline, which is to score a tag sequence. The parsers-as-taggers proposed in the literature often simplify the probability distribution over derivation trees, as defined in Eq. (1). We first discuss various approximations we can make to the score function $\text{score}(X \rightarrow Y Z)$. The three approximations we identify each make the model less expressive, but allow for more efficient decoding algorithms (see §5). Then, we exhibit how the crudest approximation allows for a fully parallelized tagger.

### 4.1 Approximating the Score Function

We identify three natural levels of approximation for $\text{score}(X \rightarrow Y Z)$, which we explicate below.

**Independent Approximation.** The first approximation is $\text{score}(X \rightarrow Y Z) = \text{score}(\bullet \rightarrow Y \bullet) \times \text{score}(\bullet \rightarrow \bullet Z)$, which we call the independent approximation. We can enact the independent approximation under all three linearization schemes $\Phi^\text{pre}_w$, $\Phi^\text{post}_w$, and $\Phi^\text{in}_w$. In each scheme, the tag encodes the nonterminal and its position with respect to its parent, i.e., whether it is a left child or a right child. The independent approximation entails that the score of each rule is the product of the scores of the left child and the right child.

**Left-Dependent Approximation.** Second, we consider the approximation $\text{score}(X \rightarrow Y Z) = \text{score}(X \rightarrow Y \bullet)$, which we call the left-dependent approximation. The left-dependent approximation is only valid under the pre-order linearization scheme. The idea is that, after generating a tag for a node $\rho(\xi) = X$ as a part of the production $\pi(\xi) = X \rightarrow Y Z$, we immediately generate a tag for its left child $Y$ independent of $Z$. Therefore, if we assume that the score of each tag only depends on the last generated tag, we simplify $\text{score}(X \rightarrow Y Z)$ as $\text{score}(X \rightarrow \bullet Z)$.

**Right-Dependent Approximation.** The third approximation is the right-dependent approximation, which takes the form $\text{score}(X \rightarrow Y Z) = \text{score}(X \rightarrow \bullet Z)$. The right-dependent approximation is only valid under the post-order linearization scheme. It works as follows: Immediately after generating the tag for the right child, the tag for its parent is generated. Again, if we assume that the score of each tag only depends on the last generated tag, we simplify $\text{score}(X \rightarrow Y Z)$ here as $\text{score}(X \rightarrow \bullet Z)$.

### 4.2 Increased Parallelism

One of the main benefits of modeling parsing as tagging is that one is then able to fine-tune pre-trained models, such as BERT (Devlin et al., 2019), to predict tags efficiently in parallel without the need to design a task-specific model architecture, e.g., as one does when building a standard transition-based parser. In short, the learner tries to predict the correct tag sequence from pre-trained BERT word representations and learns the constraints to enforce that the tagging encodes a derivation tree.

Given these representations, in the independent approximation, a common practice is to take $K$ independent projection matrices and apply them to the last subword unit of each word, followed by a softmax to predict the distribution over the $k$th tag. Given a derivation tree $d_w$ for a sentence $w$, our goal is to maximize the probability of the tag sequence that is the result of linearizing the derivation tree, i.e., $\Phi^\text{in}_w(d_w) = t$. The training objective is to maximize the probability

$$p(t \mid w) = \prod_{1 \leq n \leq N, 1 \leq k \leq K} p(T_{nk} = t_{nk} \mid w)$$

$$= \prod_{1 \leq n \leq N, 1 \leq k \leq K} \text{softmax}(W_k \cdot \text{bert}(w_n \mid w))t_{nk}$$
where \( p(T_{nk} = t_{nk} \mid w) \) is the probability of predicting the tag \( t \) for the \( k \)th tag assigned to \( w_n \), and \( \text{bert}(w_n \mid w) \) is the output of the last layer of bert processing \( w_n \) in the context of \( w \), which is a vector in \( \mathbb{R}^d \). Note that Eq. (2) follows from the independence approximation where the probability of each tag does not depend on the other tags in the sequence. Due to this approximation, there exist distributions that can be expressed by Eq. (1), but which can not be expressed by Eq. (2).

Making a connection between Eq. (1) and Eq. (2) helps us understand the extent to which we have restricted the expressive power of the model through independent assumptions to speed up training. For instance, we can properly model tagging sequences under a left- and right-dependent approximation using a conditional random field (CRF; Lafferty et al., 2001) with a first-order Markov assumption. Alternatively, we can create a dependency between the tags using an LSTM (Hochreiter and Schmidhuber, 1997) model. In our experiments, we compare these approximations by measuring the effect of the added expressive power on the overall parsing performance.

5 Decoding

In this section, we focus on the last step in a parsing-as-tagging pipeline, which is to decode a tag sequence into a tree. The goal of the decoding step is to find a valid sequence of tags \( t^* \) that are assigned the highest probability under the model for a given sentence, i.e.,

\[
t^* \overset{\text{def}}{=} \arg\max_{t \in T_w} p(t \mid w)
\]

where \( T_w \) is the set of tag sequences with a yield of \( w \). Again, we emphasize that not all sequences of tags are valid, i.e., not all tagging sequences can be mapped to a derivation tree. Therefore, in order to ensure we always return a valid tree, the invalid sequences must be detected and weeded out during the decoding process. This will generally require a more complicated algorithm.

5.1 Dynamic Programming

We extend the dynamic program (DP) suggested by Kitaev and Klein (2020) for decoding an in-order linearization to other variations of linearizers. Kitaev and Klein’s (2020) dynamic program relies on the fact that the validity of a tag sequence does not depend on individual elements in the stack. Instead, it depends only on the size of the stack at each derivation point. We show that the same observation holds for both \( \Phi_{\text{post}}^w \) and \( \Phi_{\text{pre}}^w \), which helps us to develop an efficient decoding algorithm for these linearizations.

Decoding \( \Phi_{\text{in}}^w \). We start by introducing Kitaev and Klein’s (2020) dynamic program for their \( \Phi_{\text{in}}^w \) model. Like many dynamic programs, theirs can be visualized as finding the highest-scoring path in a weighted directed acyclic graph (DAG). Each node in the DAG represents the number of generated tags \( i \) and the current stack size \( j \). Each edge represents a transition weighted by the score of the associated tag predicted by the learner. Only sequences of exactly \( 2N - 1 \) tags will be valid. The odd tags must either be \( \text{⇒} \) or \( \text{⇒} \), and the even tags must be either \( \text{⇒} \) or \( \text{⇒} \). The only accepted edges when

\[6\text{With the exception of the first tag that can only be \( \text{⇒} \).} \]
generating the $i$th tag ($1 < i \leq 2N - 1$) in the
sequence are shown in Fig. 2 left. The starting
node is $\langle 1, 1 \rangle$, with $\rhd$ in the stack, and the goal is
to reach the node: $\langle 2N - 1, 1 \rangle$.

**Decoding $\Phi^\text{pre}_w$ and $\Phi^\text{post}_w$.** To adapt the algorithm
introduced above to work with pre-order linearization,
we first investigate what makes a pre-order
tagging sequence valid:

- When a REDUCE tag ($\rhd$ or $\triangleright$) is generated,
  there must be at least one node in the stack,
  and after performing the REDUCE, the size of
  the stack is increased by one.

- Immediately after shifting a node, we either
  process (perform a SHIFT or REDUCE on) that
  node’s right sibling or shift the right sibling.
  Therefore, the only valid tags to generate sub-
  sequently are $\triangleright$ or $\triangleright$.

- Immediately after reducing a node, we process
  (perform a SHIFT or REDUCE on) its left child.
  Therefore, it is only valid to generate either
  $\triangleright$ or $\triangleright$.

The above constraints suggest that if we know
the previous tag type (SHIFT or REDUCE) and the
stack size at each step of generation, we can form
the valid transitions from each node $\langle i, j \rangle$. Such
transitions are shown in Fig. 2 center. Similarly, the
valid transitions for decoding post-order linearization
are shown in the right part of Fig. 2.

**Time Complexity.** The complexity of the dy-
namic program to find the highest-scoring path
depends on the number of unique tags $O(|\mathcal{N}|)$, the
length of the tag sequence $O(|\mathcal{N}|)$, and the stack
size $d$. Linearization schemes derived from the
right-corner transform tend to have smaller stack
sizes (Abney and Johnson, 1991; Schuler et al.,
2010). As a result, one can often derive a faster,$^7$ exact decoding algorithm for such schemes. Please
refer to App. C for a more formal treatment of the
dynamic program for finding the highest-scoring
tag sequence using these transitions (Algorithm 1).

**An $O(dN|\mathcal{N}|^2)$ Algorithm.** As discussed in §4,
one can break the independence assumption in
Eq. (2) by making each tag dependent on the previously
generated tag. Consequently, when finding the
highest-scoring path in the decoding step, we
should remember the last generated tag. There-
fore, in addition to $i$ and $j$, the generated tag must
be stored. This adds $O(|\mathcal{N}|)$ extra memory per
node, and increases the runtime complexity to
$O(dN|\mathcal{N}|^2)$. Please refer to App. C for a more
formal treatment of the dynamic program for finding
the highest-scoring tag sequence (Algorithm 2).

---

$^7$For purely left- or right-branching trees, the left-corner
or right-corner transformations, respectively, provably reduce
the stack size to one (Johnson, 1998). These transformations
could increase the stack size for centered-embedded trees.
However, such trees are rare in natural language as they are
difficult for humans to comprehend (Gibson, 1998).
5.2 Beam Search Decoding

We can speed up the dynamic programming algorithm above if we forego the desire for an exact algorithm. In such a case, applying beam search presents a reasonable alternative where we only keep track of \( h \) tag sequences with the highest score at each step of the tag generation. Although no longer an exact algorithm, beam search reduces the time complexity to \( \mathcal{O}(h \log \log N) \).\(^8\) This can result in a substantial speed-up over the dynamic programs presented above when \( d \), the maximum required stack size, is much larger than the beam size \( h \). We empirically measure the trade-off between accuracy and speed in our experiments.

6 Experiments

In our experimental section we aim to answer two questions: (i) What is the effect of each design decision on the efficiency and accuracy of a tagger, and which design decisions have the greatest effect? (ii) Are the best design choices consistent across languages?

**Data.** We use two data sources: the Penn Treebank (Marcus et al., 1993) for English constituency parsing and Statistical Parsing of Morphologically Rich Languages (SPMRL) 2013/2014 shared tasks (Seddah et al., 2013, 2014) for 8 languages: Basque, French, German, Hebrew, Hungarian, Korean, Polish, and Swedish. We provide the dataset statistics in Table 5. We perform similar preprocessing as Kitaev and Klein (2020), which we explain in App. F. For evaluation, the EVALB Perl script\(^9\) is used to calculate FMeasure of the parse tree. We use only one GPU node for the reported inference times. Further experimental details can be found in App. G.

6.1 What Makes a Good Tagger?

**Linearization.** First, we measure the effect of linearization and alignment on the performance of the parser. We train BERT\(^10\) on the English dataset. While keeping the learning schema fixed, we experiment with three linearizations: \( \Phi^{\text{in}}_w \), \( \Phi^{\text{post}}_w \), and \( \Phi^{\text{pre}}_w \). We observe that the in-order shift–reduce linearizer leads to the best FMeasure (95.15), followed by the top-down linearizer (94.56). There is a considerable gap between the performance of a tagger with the post-order linearizer (89.23) and other taggers, as shown in Table 3. The only difference between these taggers is the linearization function, and more specifically, the deviation between tags and the input sequence as defined in Def. 4. This result suggests that deviation is an important factor in designing accurate taggers.

**Learning Schema.** Second, we measure the effect of the independence assumption in scoring tagging sequences. We slightly change the scoring mechanism by training a CRF\(^11\) and an LSTM layer\(^12\) to make scoring over tags left- or right-dependent. We expect to see improvements with \( \Phi^{\text{pre}}_w \) and \( \Phi^{\text{post}}_w \), since in these two cases, creating such dependencies gives us more expressive power in terms of the distributions over trees that can be modeled. However, we only observe marginal improvements when we add left and right dependencies (see Table 3). Thus, we hypothesize that BERT already captures the dependencies between the words.

**Decoding Schema.** Third, we assess the trade-off between accuracy and speed caused by using beam search versus exact decoding. We take \( h = 10 \) as the beam size. As shown in Table 3, taggers with post-order linearization take longer to decode using the exact dynamic program. This is largely due to the fact that they need deeper stacks in the decoding process (see App. E for a comparison). In such scenarios, we see that beam search leads to a significant speed-up in decoding time, at the expense of a drop (between 3 to 6 points) in FMeasure. However, the drop in accuracy observed with the pre-order linearization is less than 1 point. Therefore, in some cases, beam search may be able to offer a sweet spot between speed and accuracy. To put these results into context, we see taggers with in-order and pre-order linearizations achieve comparable results to state-of-the-art parsers with custom architectures, where the state-of-the-art parser achieves 95.84 FMeasure (Zhou and Zhao, 2019).

6.2 Multilingual Parsing

As shown in the previous experiment, linearization plays an important role in constructing an accurate tagger. Moreover, we observe that in-order linearizers yield the best-performing taggers, followed by

---

\(^8\)This can be improved to \( \mathcal{O}(h\log \log N) \) with quick select.

\(^9\)http://nlp.cs.nyu.edu/evalb/

\(^10\)We use BERT-LARGE-UNCASED from the Huggingface library (Wolf et al., 2020).

\(^11\)We use our custom implementation of CRF with pytorch.

\(^12\)We use a two-layered biLSTM network, with the hidden size equal to the tags vocabulary size \(|T|\) (approximately 150 cells). For further details please refer to App. G.
Table 2: Comparison of FMeasure of different tagging schemata on the SPMRL test set

<table>
<thead>
<tr>
<th>Tagging Schemata</th>
<th>FMeasure</th>
<th>Sents/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>in-order-[INDEP.]</td>
<td>89.86</td>
<td>84.54</td>
</tr>
<tr>
<td>pre-order-[INDEP.]</td>
<td>87.98</td>
<td>84.76</td>
</tr>
<tr>
<td>post-order-[INDEP.]</td>
<td>81.05</td>
<td>56.97</td>
</tr>
</tbody>
</table>

Table 3: Comparison of parsing metrics on the WSJ test set.

<table>
<thead>
<tr>
<th>Parsing Schemata</th>
<th>FMeasure</th>
<th>Sents/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>in-order*[INDEP.]</td>
<td>91.59</td>
<td>156</td>
</tr>
<tr>
<td>pre-order*[INDEP.]</td>
<td>93.57</td>
<td>114</td>
</tr>
<tr>
<td>post-order*[INDEP.]</td>
<td>85.38</td>
<td>108</td>
</tr>
</tbody>
</table>

13We use BERT-BASE-MULTILINGUAL-CASED.
the algorithms that we experiment with because our work is primarily a theoretical analysis of existing work. Therefore, we foresee no ethical concerns in this work.

Limitations
This work only focuses on constituency parsing. Therefore, the results might not fully generalize to other parsing tasks, e.g., dependency parsing. Furthermore, in our experiments, we only focus on comparing design decisions of parsing-as-tagging schemes, and not on achieving state-of-the-art results. We believe that by using larger pre-trained models, one might be able to obtain better parsing performance with the tagging pipelines discussed in this paper. In addition, in our multilingual analysis, the only left-branching language that we had access to was Korean, therefore, more analysis needs to be done on left-branching languages.

Acknowledgments
We would like to thank Nikita Kitaev, Tim Vieira, Carlos Gómez-Rodríguez, and Jason Eisner for fruitful discussions on a preliminary draft of the paper. We would also like to thank anonymous reviewers for their insightful feedback during the review process. We also thank Zeerak Talat and Clément Guerner for their feedback on this manuscript. Afra Amini is supported by ETH AI Center doctoral fellowship.

References


John D. Lafferty, Andrew McCallum, and Fernando C. N. Pereira. 2001. Conditional random fields:


A Related Work

We provide a comparative review of parsing as tagging works on both dependency and constituency parsing, focusing on their design decisions at different steps of the pipeline.

A.1 Dependency Parsing as Tagging

Previous work on dependency parsing as tagging linearizes the dependency tree by iterating through the dependency arcs and encoding the relative position of the child with respect to its parent in the tag sequence (Li et al., 2018; Kiperwasser and Ballesteros, 2018). Each word is then aligned with the tag indicating its relative position with respect to its head. At the decoding step, to prevent the model from generating dependency arcs that do not form valid dependency trees, a tree constraint is often applied (Li et al., 2018). Similarly, Vacareanu et al. (2020) employ contextualized embeddings of the input with the same tagging schema. Strzyz et al. (2019) provide a framework and train and compare various dependency parsers as taggers. See also Strzyz (2021) for further comparison.

Closest to our work, Gómez-Rodríguez et al. (2020) suggest a linearization and alignment scheme that works with any transition-based parser to create a sequence of tags from a sequence of actions. Therefore, their method applies to both constituency and dependency parsing. For instance, turning a shift–reduce parser into a tag sequence requires creating a tag for all the REDUCE actions up to each SHIFT action. While this construction results in exactly \( N \) tags, there are in general an exponentially large number of tags. To compare this approach with the linearizations discussed in this paper, we must note that while such a schema is shift-aligned, it might not evenly align the tags with the input sequence. Moreover, the in-order linearization scheme discussed here has a small number of tags.

A.2 Constituency Parsing as Tagging

Gómez-Rodríguez and Vilares (2018) provide an in-order linearization of the derivation tree, where they encode nonterminals and their depth as a tagging sequence. Since the depth is encoded in the tags themselves, the tag set size is unbounded and is dependent on the input. Kitaev and Klein (2020) improve upon Gómez-Rodríguez and Vilares’s (2018) encoding by discovering a similar tagging system that has only four tags. In Kitaev and Klein’s (2020) linearization step, they encode both terminals and nonterminals in the tag sequence, as well as the direction of a node with respect to its parent. Using a pretrained BERT model to score tags, they show that their approach reaches state-of-the-art performance despite being architecturally simpler. Furthermore, the parser is significantly faster than competing approaches.

B Derivations

The following derivation provides the exact map and merge functions for transforming shift–reduce action sequences to tetragrams.

**Derivation 2.** Let \( d_{w}^{rc} \) be the derivation tree after the right-corner transformation for the sentence \( w \), and let \( a_{w}^{post-rc} \) be the sequence of SHIFT or REDUCE actions taken by a bottom-up shift–reduce parser processing \( d_{w}^{rc} \). Then, we have that

\[
\Phi_{w}^{in}(d_{w}) = \text{merge} \left( \text{map} \left( a_{w}^{post-rc} \right) \right) \tag{4}
\]

where map applies to each element in the action sequence and maps each action to:

\[
\begin{align*}
\text{map(REDUCE}(N \to N/N_{1} N_{2})) &= \kappa_

\text{map(REDUCE}(N/N_{1} \to N/N_{2} N_{3})) &= \kappa_{\dagger}(N)

\text{map(REDUCE}(N/N_{1} \to \varepsilon N_{2})) &= \dagger(N)

\text{map(SHIFT}(\bullet)) &= \odot
\end{align*}
\]

The merge function reads the sequence of tags from left to right and whenever it encounters a \( \odot \) tag followed immediately by a \( \kappa \), it merges them to a \( \kappa \) tag.
And ways to position These configurations are shown in the first row of Table 4. Now we apply the right-corner transformation proof followed immediately by a shift sequence of derivation 3.

Let derivation 3. map bottom-up shift–reduce linearizations for the transformed trees and split them at tetratagger, this split starts with either the last group consists of only one tag. Let’s focus on a specific split. According to the definition of proof.

\[ \Phi(w) \]

\[ (N_1/.) \]

\[ (N'/N_1) \]

\[ w_i \]

\[ d_{w} \]

\[ \Phi_{w}^{in} \]

\[ (N_1/.) \rightarrow \varepsilon X_i \]

\[ (N'/N_1) \rightarrow X_i \]

\[ w_i \]

\[ d_{w}' \]

\[ \Phi_{w}^{post} \]

\[ (N_1/.) \rightarrow \varepsilon X_i \]

\[ (N'/N_1) \rightarrow X_i \]

\[ w_i \]

\[ \text{merge} \]

\[ (N_1/.) \rightarrow \varepsilon X_i \]

\[ (N'/N_1) \rightarrow X_i \]

\[ w_i \]

Table 4: Equivalence of tetratags and bottom-up shift–reduce tags on the right-cornered derivation tree

**Proof.** We split tetratags \( \Phi_w(d_w) \) into \( n \) groups, where the first \( n - 1 \) groups consist of two tags and the last group consists of only one tag. Let’s focus on a specific split. According to the definition of tetratagger, this split starts with either \( \triangleright \) or \( \triangleleft \), each of which corresponds to a terminal node, i.e. a word in the input sequence: \( w_i \) with part-of-speech \( X_i \). Depending on the topology of \( d_w \), there are five distinct ways to position \( w_i \) in the derivation tree, each of which corresponds to unique tetratags for the split. These configurations are shown in the first row of Table 4. Now we apply the right-corner transformation to the derivation trees and we obtain the trees shown in the second row of Table 4. Next, we generate bottom-up shift–reduce linearizations for the transformed trees and split them at shift actions. We see that after applying map and merge on each of the splits, we obtain the exact same tetratags for all of the five possible configurations, as shown in the last row of Table 4.

**Derivation 3.** Let \( d_{w}' \) be the left-corner transformed derivation tree for the sentence \( w \), and \( a_{w}^{pre-lc} \) be the sequence of shift or reduce actions took by a top-down shift–reduce parser processing \( d_{w}' \). Then:

\[ \Phi_{w}^{in}(d_w) = \text{merge} \left( \text{map} \left( a_{w}^{pre-lc} \right) \right) \]  \( (5) \)

where map applies to each element in the action sequence and maps each action to:

\[ \text{map}(\text{reduce}(N \rightarrow N_2 N/N_1)) = \triangleright \]

\[ \text{map}(\text{reduce}(N/N_1 \rightarrow N_3 N/N_2)) = \triangleright (N_2) \]

\[ \text{map}(\text{reduce}(N/N_1 \rightarrow N_2 \varepsilon)) = \triangleleft (N) \]

\[ \text{map}(\text{shift}(\bullet)) = \triangleleft \]

And merge function reads the sequence of tags from left to right and whenever it encounters a \( \triangleright \) tag followed immediately by a \( \triangleleft \), it merges them to a \( \triangleright \) tag.

**Proof.** The proof is similar to the proof of right-corner and bottom-up shift–reduce.
C Decoding Algorithms

**Proposition 4.** Given a scoring function with independent approximation, \( \text{score}(X \rightarrow Y Z) \overset{\text{def}}{=} \text{score}(\cdot \rightarrow Y \cdot) \times \text{score}(\cdot \rightarrow Z) \), over derivation trees, we can find the highest-probability derivation in \( \mathcal{O}(dN|N|) \) using Algorithm 1.

**Proof sketch.** We give an outline of the construction. First, consider a grammar in Chomsky normal form \( \mathcal{G} \). Next, apply the standard shift–reduce transformation to \( \mathcal{G} \) in order to construct a pushdown automaton (PDA). Finally, note that under the independence assumption \( \text{score}(X \rightarrow Y Z) \overset{\text{def}}{=} \text{score}(\cdot \rightarrow Y \cdot) \times \text{score}(\cdot \rightarrow Z) \) we can reduce the runtime of the dynamic program that sums over all runs in the PDA from \( \mathcal{O}(N^3|N|^3) \) (Lang, 1974; Butoi et al., 2022) to \( \mathcal{O}(dN|N|) \) because we only need to keep track of the height of the stack \( d \) and not which elements are in it. Note that \( d = N \) in the worst case. This can be viewed as using Algorithm 1 to find the valid tagging sequence with the highest score in a directed acyclic graph, as shown in Fig. 2. ■

**Proposition 5.** Suppose we are given a scoring function with left- or right-dependent approximation, \( \text{score}(X \rightarrow Y Z) \overset{\text{def}}{=} \text{score}(X \rightarrow Y \cdot) \) or \( \text{score}(X \rightarrow Y Z) \overset{\text{def}}{=} \text{score}(X \rightarrow \cdot Z) \), over derivation trees. We can find the derivation tree with the highest score in \( \mathcal{O}(dN|N|^2) \) using Algorithm 2.

**Proof sketch.** We give an outline of the construction. First, consider a grammar in Chomsky normal form \( \mathcal{G} \). Next, apply the standard shift–reduce transformation to \( \mathcal{G} \) in order to construct a pushdown automaton (PDA). Finally, note that under the left- or right-dependent approximation, \( \text{score}(X \rightarrow Y Z) \overset{\text{def}}{=} \text{score}(X \rightarrow Y \cdot) \) or \( \text{score}(X \rightarrow Y Z) \overset{\text{def}}{=} \text{score}(X \rightarrow \cdot Z) \), we can reduce the runtime of the dynamic program that sums over all runs in the PDA from \( \mathcal{O}(N^3|N|^3) \) to \( \mathcal{O}(dN|N|^2) \) because we only need to keep track of the height of the stack \( d \), and the top element on the stack, but not the remaining elements in the stack. This can be viewed as using Algorithm 2 to find the valid tagging sequence with the highest score in a directed acyclic graph, as shown in Fig. 2. ■

**Algorithm 1** Dynamic program for decoding with independent scores

```plaintext
1: procedure DP(score) 
2: \( W[1, 1] = \log \text{score}(\cdot) \) \( \triangleright \) initialize the DP chart to zero
3: \( W[1, 2] = \log \text{score}(\cdot|S) \) \( \triangleright \) pre-order: \( W[1, 2] = \log \text{score}(\cdot|S) \)
4: for \( i = 2, \ldots, 2N - 1 \) \( \triangleright \) \( \text{pre-order: } W[2N - 1, 1] \)
5: \( d = \text{maximum size of the stack} \)
6: \( \text{shift}_{\text{score}} = \max_{\langle i-1,j+1 \rangle \rightarrow \langle i,j \rangle} \{W[i - 1, j + 1] + \log \text{score}(t)\} \)
7: \( \text{reduce}_{\text{score}} = \max_{\langle i-1,j \rangle \rightarrow \langle i,j \rangle} \{W[i - 1, j - 1] + \log \text{score}(t)\} \)
8: \( W[i, j] = \max(\text{shift}_{\text{score}}, \text{reduce}_{\text{score}}) \)
9: return \( W[2N - 1, 1] \) \( \triangleright \) \( \text{pre-order: } W[2N - 1, 0] \)
```

**Algorithm 2** Dynamic program for decoding with left- or right-dependent scores

```plaintext
1: procedure DP(score) 
2: \( W[1, 1] = \log \text{score}(\cdot) \) \( \triangleright \) initialize the DP chart to zero
3: \( W[1, 2] = \log \text{score}(\cdot|S) \) \( \triangleright \) \( \text{pre-order: } W[1, 2, \cdot|S] \)
4: for \( i = 2, \ldots, 2N - 1 \) \( \triangleright \) \( \text{pre-order: } W[2N - 1, 1, \cdot|S] \)
5: \( d = \text{maximum size of the stack} \)
6: for \( t = 1, \ldots, |T| \) : 
7: \( \text{shift}_{\text{score}} = \max_{\langle i-1,j+1,t' \rangle \rightarrow \langle i,j,t \rangle} \{W[i - 1, j + 1, t'] + \log \text{score}(t', t)\} \)
8: \( \text{reduce}_{\text{score}} = \max_{\langle i-1,j-1,t' \rangle \rightarrow \langle i,j,t \rangle} \{W[i - 1, j - 1, t'] + \log \text{score}(t', t)\} \)
9: \( W[i, j, t] = \max(\text{shift}_{\text{score}}, \text{reduce}_{\text{score}}) \)
10: return \( W[2N - 1, 1, \cdot|S] \) \( \triangleright \) \( \text{pre-order: } W[2N - 1, 0, \cdot|S] \)
```
### D Dataset Statistics

We use Penn Treebank for English and SPMRL 2013/2014 shared tasks for experiments on other languages. We further use available train/dev/test splits, the number of sentences in each split can be found in Table 5.

### E Stack Size

We empirically compare the stack size needed to parse English sentences using different linearizations in Fig. 4.

![Figure 4: With $\Phi_{\text{in}}^w$ and stack size of 6, all the trees in WSJ test set can be linearized, thus the coverage is 1. On the other extreme, in order to cover all the trees with $\Phi_{\text{post}}^w$, we need a stack of size 29.](image)

### F Preprocessing

Since our linearization works for binary trees, we do the following preprocessing: we collapse the unary rules by concatenating the nonterminal labels. We then binarize the tree using the nltk library. For in-order linearization, we first perform right-corner transformation of the tree and then run bottom-up shift reduce parsing on the transformed tree followed by the map and merge function introduced in App. B.\(^\text{14}\)

### G Experimental Setup

We experiment on 3 NVIDIA GeForce GTX 1080 Ti nodes. The batch size is 32 and sentences in each epoch are sampled without replacement. We use gradient clipping at 1.0 and learning rate $3e^{-5}$ with a warmup over the course of 160 training steps. We calculate FMeasure of the development set 4 times per epoch and cut the learning rate in half, whenever the FMeasure fails to improve. We set the initial epochs to 20, however, after 3 consecutive decays in the learning rate, training is terminated. The checkpoint with

\(^{14}\)We empirically test these map and merge functions, and verify that the sequence of transformed shift–reduce actions perfectly matches the original tetratags.
Recall Precision FMeasure

<table>
<thead>
<tr>
<th>Model</th>
<th>Recall</th>
<th>Precision</th>
<th>FMeasure</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Phi_w^{pre} + 2$-layered BiLSTM</td>
<td>93.76</td>
<td>94.5</td>
<td>93.9</td>
</tr>
<tr>
<td>$\Phi_w^{pre} + 3$-layered BiLSTM</td>
<td>93.93</td>
<td>94.45</td>
<td>94.19</td>
</tr>
</tbody>
</table>

Table 6: The effect of increasing BiLSTM layers on parsers’ performance.

the best development FMeasure is used for reporting the test scores. The stack depth for the decoding step is set to the maximum depth of the stack in the training set. In experiments with the LSTM model, we use a two-layered BiLSTM network. We observe in Table 6 that adding more layers has a minimal effect on parsers’ performance.