CCG Parsing Algorithm with Incremental Tree Rotation

Miloš Stanojević
School of Informatics
University of Edinburgh
m.stanojevic@ed.ac.uk

Mark Steedman
School of Informatics
University of Edinburgh
steedman@inf.ed.ac.uk

Abstract

The main obstacle to incremental sentence processing arises from right-branching constituent structures, which are present in the majority of English sentences, as well as from optional constituents that adjoin on the right, such as right adjuncts and right conjuncts. In CCG, many right-branching derivations can be replaced by semantically equivalent left-branching incremental derivations.

The problem of right-adjunction is more resistant to solution, and has been tackled in the past using *revealing*-based approaches that often rely either on the higher-order unification over lambda terms (Pareschi and Steedman, 1987) or heuristics over dependency representations that do not cover the whole CCGbank (Ambati et al., 2015).

We propose a new incremental parsing algorithm for CCG following the same *revealing* tradition of work but having a purely syntactic approach that does not depend on access to a distinct level of semantic representation. This algorithm can cover the whole CCGbank, with greater incrementality and accuracy than previous proposals.

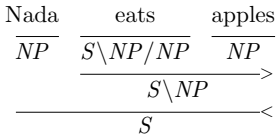
1 Introduction

Combinatory Categorial Grammar (CCG) (Ades and Steedman, 1982; Steedman, 2000) is a mildly context sensitive grammar formalism that is attractive both from a cognitive and an engineering perspective. Compared to other grammar formalisms, the aspect in which CCG excels is incremental sentence processing. CCG has a very flexible notion of constituent structure which allows (mostly) left-branching derivation trees that are easier to process incrementally. Take for instance the derivation tree in Figure 1a. If we use a non-incremental shift-reduce parser (as done in the majority of transition-based parsers for CCG (Zhang and Clark, 2011; Xu et al., 2014; Xu, 2016)) we will be able to establish the semantic connection between the subject "Nada" and the verb "eats" only when we reach the end of the sentence. This is undesirable for several reasons. First, human sentence processing is much more incremental, so that the meaning of the prefix "Nada eats" is available as soon as it is read (Marslen-Wilson, 1973). Second, if we want a predictive model—either for better parsing or language modelling—it is crucial to establish relations between the words in the prefix as early as possible.

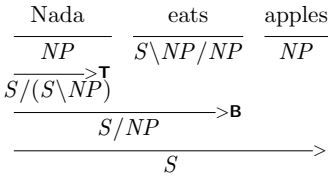
To address this problem, a syntactic theory needs to be able to represent partial constituents like "Nada eats" and have mechanisms to build them just by observing the prefix. In CCG solutions for these problems come out of the theory naturally. CCG categories can represent partial structures and these partial structures can combine into bigger (partial) structures using CCG combinators recursively. Figure 1b shows how CCG can incrementally process the example sentence via a different derivation tree that generates the same semantics more incrementally by being left-branching.

This way of doing incremental processing seems straightforward except for one obstacle: optional constituents that attach from the right, i.e. right adjuncts. Because they are optional, it is impossible to predict them with certainty. This forces an eager incremental processor to make an uninformed decision very early and, if later that decision turns out to be wrong, to backtrack to repair the mistake. This behaviour would imply that human processors have difficulty in processing right adjuncts, but that does not seem to be the case. For instance, let's say that after incrementally processing "Nada eats apples" we encounter right adjunct "regularly" as in Figure 2a. The parser will be stuck at this point because there is no way to at-

Proceedings of NAACL-HLT 2019, pages 228–239
Minneapolis, Minnesota, June 2 - June 7, 2019.

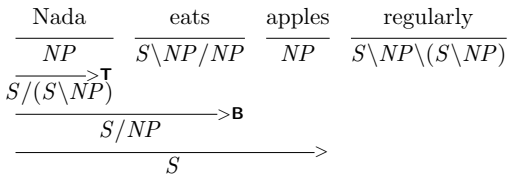


(a) Right-branching derivation.

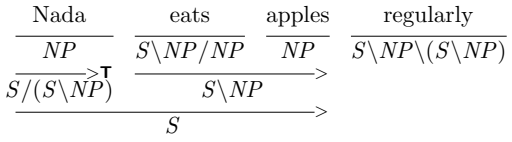


(b) Left-branching derivation.

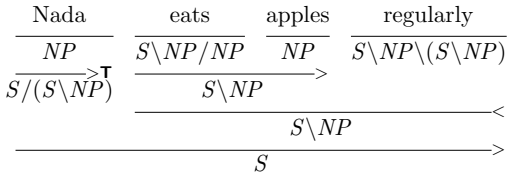
Figure 1: Semantically equivalent CCG derivations.



(a) Problem – *S\NP* that needs to be modified was never built.



(b) Incremental tree rotation *reveals S\NP*.



(c) Right adjunct is attached to the revealed node.

Figure 2: Right adjunction.

tach the right adjunct of a verb phrase to a sentence constituent. A simple solution would be some sort of limited back-tracking where we would look if we could extract the verb-phrase, attach its right adjunct, and then put the derivation back together. But how do we do the extraction of the verb-phrase "eats apples" when that constituent was never built during the incremental left-branching derivation?

Pareschi and Steedman (1987) proposed to *reveal* the constituent that is needed, the verb-phrase in our example, by having an elegant way of reanalysing the derivation. This reanalysis does not repeat parsing from scratch but instead runs a single CCG combinatory rule backwards. In the example at hand, first we recognise that right adjunction needs to take place because we have a category of shape *X\X* (concretely *(S\NP)\(S\NP)* but in the present CCG notation slashes "associate to the left", so we drop the first pair of brackets). Thanks to the type of the adjunct we know that the constituent that needs to be revealed is of type *X*, in our case *S\NP*. Now, we take the constituent on the left of the right adjunct, in our example constituent *S*, and look for CCG category *Y* and combinatory rule *C* that satisfies the following relation: *C(Y, S\NP) = S*. The solution to this type equation is *Y=NP* and *C=<*.

To confine revealing to delivering constituents that the parser *could have* built if it had been less greedy for incrementality, and exclude revelation of unsupported types, such as PP in Figure 2a, the process must be constrained by the actual derivation. Pareschi and Steedman proposed to do so by accessing the semantic representation in parallel, using higher-order unification, which is in general undecidable and may be unsound unless defined over a specific semantic representation.

Ambati et al. (2015) propose an alternative method for revealing where dependencies are used as a semantic representation (instead of first-order logic) and special heuristics are used for revealing (instead of higher order unification). This is computationally a much more efficient approach and appears sound, but requires distinct revealing rules for each constituent type and has specific difficulties with punctuation.

In this paper we propose a method of revealing that does not depend on any specific choice of semantic representation, can discover multiple possible revealing options if they are available, is sound and complete and computationally efficient, and gives state-of-the-art parsing results. The algorithm works by building left-branching derivations incrementally, but, following Niv (1993, 1994), as soon as a left branching derivation is built, its derivation tree is rebalanced to be right-branching. When all such constituents' derivation trees are right-branching, revealing becomes a trivial operation where we just traverse the right spine looking for the constituent(s) of the right type to be modified by the right adjunct.

We call this rebalancing operation *tree rota-*

tion since it is a technical term established in the field of data structures for similar operation of balanced binary search trees (Adelson-Velskii and Landis, 1962; Guibas and Sedgewick, 1978; Okasaki, 1999; Cormen et al., 2009). Figure 2b shows the right rotated derivation "Nada eats apples" next to the adjunct. Here we can just look up the required *S\NP* and attach the right adjunct to it as in Figure 2c.

2 Combinatory Categorial Grammar

CCG is a lexicalized grammar formalism where each lexical item in a derivation has a category assigned to it which expresses the ways in which the lexical item can be used in the derivation. These categories are put together using combinatory rules.

The binary combinatory rules we use are:

$$X/Y \quad Y \Rightarrow X \qquad (>)$$
$$Y \quad X\backslash Y \Rightarrow X \qquad (<)$$
$$X/Y \quad Y/Z \Rightarrow X/Z \qquad (>\mathbf{B})$$
$$Y\backslash Z \quad X\backslash Y \Rightarrow X\backslash Z \qquad (<\mathbf{B})$$
$$Y/Z \quad X\backslash Y \Rightarrow X/Z \qquad (<\mathbf{B}_{\times})$$
$$Y/Z|W \quad X\backslash Y \Rightarrow X/Z|W \qquad (<\mathbf{B}^2_{\times})$$
$$X/Y \quad Y/Z|W \Rightarrow X/Z|W \qquad (>\mathbf{B}^2)$$

Each binary combinatory rule has one primary and one secondary category as its inputs. The primary functor is the one that selects; while the secondary category is the one that is selected. In forward combinatory rules the primary functor is always the left argument, while in the backward combinatory rules it is always the right.

It is useful to look at the mentioned combinatory rules in a generalised way. For instance, if we look at forward combinatory rules we can see that they all follow the same pattern of combining X/Y with a category that starts with Y. The only difference among them is how many subcategories follow Y in the secondary category. In case of forward function application there will be nothing following Y so we can treat forward function application as a generalised forward composition combinator of the zeroth order $>\mathbf{B0}$. Standard forward function composition $>\mathbf{B}$ will be a generalised composition of first order $>\mathbf{B1}$ while $>\mathbf{B}^2$ will be $>\mathbf{B2}$. Same generalisation can be applied to backward combinators. There is a low bound on the order of combinatory rules, around 2 or 3.

Following Hockenmaier and Steedman (2007), the proclitic character of conjunctions is captured in a syncategorematic rule combining them with the right conjunct, with the result later combining with the left conjunct [1] :

$$conj \quad X \Rightarrow X[conj] \qquad (>\mathbf{\Phi})$$
$$X \quad X[conj] \Rightarrow X \qquad (<\mathbf{\Phi})$$

Some additional unary and binary type-changing rules are also needed to process the derivations in CCGbank (Hockenmaier and Steedman, 2007). We use the same type-changing rules as those described in (Clark and Curran, 2007).

Among the unary combinatory rules the most important one is type-raising. The first reason for that is that it allows CCG to handle constructions like argument cluster coordination in a straightforward way. Second, it allows CCG to be much more incremental as seen from the example in Figure 1b. Type-raising rules are expressed in the following way:

$$X \Rightarrow Y/(Y\backslash X) \qquad (>\mathbf{T})$$
$$X \Rightarrow Y\backslash(Y/X) \qquad (<\mathbf{T})$$

Type-raising, is strictly limited to applying to category types that are *arguments*, such as NP, PP, etc., making it analogous to grammatical *case* in languages like Latin and Japanese, in spite of the lack of morphological case in English.

3 Parsing

CCG derivations can be parsed with the same shift-reduce mechanism used for CFG parsing (Steedman, 2000). In the context of CFG parsing, the shift-reduce algorithm is not incremental, because CFG structures are mostly right-branching, but in CCG by changing the derivation via the combinatory rules we also change the level of incrementality of the algorithm.

As usual, the shift-reduce algorithm consists of a stack of the constituents built so far and a buffer with words that are yet to be processed. Parsing starts with the stack empty and the buffer containing the whole sentence. The end state is a stack with only one element and an empty buffer. Transitions between parser states are:

- *shift(X)* – moves the first word from the buffer to the stack and labels it with category *X*,
- *reduceUnary(C)* – applies a unary combinatory rule *C* to the topmost constituent on the stack,
- *reduceBinary(C)* – applies a binary combinatory rule *C* to the two topmost constituents on

[1] This notation differs unimportantly from Steedman (2000) who uses a ternary coordination rule, and more recent work in which conjunctions are $X\backslash X/X$.

the stack.

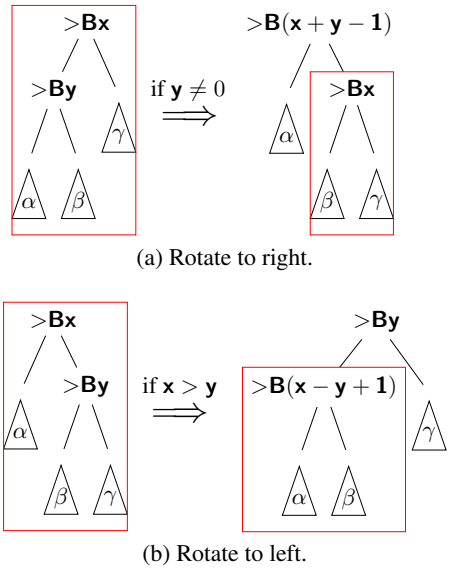
CCG shift-reduce parsers are often built over right-branching derivations that obey Eisner normal form (Eisner, 1996). Processing left-branching derivations is not any different except that it requires an opposite normal form.

Our revealing algorithm adds a couple of modifications to this default shift-reduce algorithm. First, it guarantees that all the trees stored on the stack are right-branching – this still allows left-branching parsing and only adds the requirement of adjusting newly reduced trees on the stack to be right leaning. Second, it adds *revealing* transitions that exploit the right-branching guarantee to apply right adjunction. Both tree rotation and revealing are performed efficiently as described in the following subsections.

3.1 Tree rotation

A naïve way of enforcing right-branching guarantee is to do a complete transformation of the subtree on the stack into a right-branching one. However, that would be unnecessarily expensive. Instead we do *incremental tree rotation* to right. If we assume that all the elements on the stack are respecting this right-branching form (our inductive case), this state can be disturbed only by *reduceBinary* transition (*shift* just adds a single word which is trivially right-branching and *reduceUnary* does not influence the direction of branching). The *reduceBinary* transition will take two topmost elements on the stack that are already right-branching and put them as children of some new binary node. We need to repair that potential "imperfection" on top of the tree. This is done by recursively rotating the nodes as in Figure 3a.[2]

This figure shows one of the sources of CCG's spurious ambiguity: parent-child relation of the combinatory rules with the same directionality. Here we concentrate on forward combinators because they are the most frequent in our data—most backward combinators disappear with the addition of forward type-raising and the addition of special right adjunct transitions—but the same method can be applied to backward combinatory rules as a mirror image. Having two combinatory rules of the same directionality is necessary



(a) Rotate to right.

(b) Rotate to left.

Figure 3: Tree rotation operations. The red square signifies recursion. Variables x and y represent the orders of the combinatory rules.

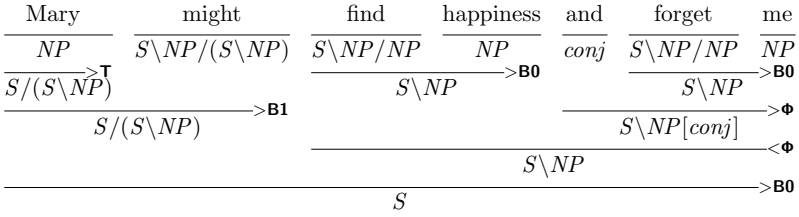
but not sufficient condition for spurious ambiguity. As visible on the Figure 3a side condition, the lower combinator must not be >**B0**. The tree rotation function assumes that both of the children are "perfect"—meaning right-branching[3]— and that the only imperfection is on the root node. The method repairs this imperfection on the root by applying the tree rotation transformation, but it also creates a new node as a right child and that node might be imperfect. That is why the method goes down the right node recursively until all the imperfections are removed and the whole tree becomes fully right-branching. In the worst case the method will reach the bottom of the tree, but often only 3 or 4 nodes need to be transformed to make the tree perfectly the right branching The worst case complexity of repairing the imperfection is $O(n)$ which makes the complexity of the whole parsing algorithm $O(n^2)$ for building a single derivation.

As a running example we will use a derivation tree in Figure 4a for which a transition sequence is given in Figure 4b. Here tree rotation is used in transitions 6 and 8 that introduce imperfections. In transition 6 a single tree rotation at the top was enough to correct the imperfection, while in transition 8 recursive tree rotation function went to depth two.

If the upper and lower combinators are both >**B2** the topmost combinator on the right will be-

[2] Although we do not discuss the operations on the semantic predicate-argument structure that correspond to tree-rotation, the combinatory semantics of the rules themselves guarantees that such operations can be done uniformly and in parallel.

[3] By right branching we mean as right branching as it is allowed by CCG formalism and predicate-argument structure.



(a) Derivation tree.

	transition		stack
1	shift	⟹	■ Mary
2	reduceUnary(>**T**)	⟹	■ >**T** Mary
3	shift	⟹	■ >**T** Mary ■ might
4	reduceBinary(>**B1**)	⟹	■ >**B1** (>**T** Mary) might
5	shift	⟹	■ >**B1** (>**T** Mary) might ■ find
6	reduceBinary(>**B1**)	⟹	■ >**B1** (>**B1** (>**T** Mary) might) find
	rotate to right	⟹	■ >**B1** (>**T** Mary) (>**B1** might find)
7	shift	⟹	■ >**B1** (>**T** Mary) (>**B1** might find) ■ happiness
8	reduceBinary(>**B1**)	⟹	■ >**B0** (>**B1** (>**T** Mary) (>**B1** might find)) happiness
	rotate to right	⟹	■ >**B0** (>**T** Mary) (>**B0** (>**B1** might find) happiness)
	rotate to right	⟹	■ >**B0** (>**T** Mary) (>**B0** might (>**B0** find happiness))
9	shift	⟹	■ >**B0** (>**T** Mary) (>**B0** might (>**B0** find happiness)) ■ and
10	shift	⟹	■ >**B0** (>**T** Mary) (>**B0** might (>**B0** find happiness)) ■ and ■ forget
11	shift	⟹	■ >**B0** (>**T** Mary) (>**B0** might (>**B0** find happiness)) ■ and ■ forget ■ me
12	reduceBinary(>**B0**)	⟹	■ >**B0** (>**T** Mary) (>**B0** might (>**B0** find happiness)) ■ and ■ >**B0** forget me
13	reduceBinary(>**Φ**)	⟹	■ >**B0** (>**T** Mary) (>**B0** might (>**B0** find happiness)) ■ >**Φ** and (>**B0** forget me)
14	reveal	⟹	■ two options for right adjunction and both with <**Φ** combinator: option 1: >**B0** might (>**B0** find happiness) option 2: >**B0** find happiness
15	pick option 2	⟹	■ >**B0** (>**T** Mary) (>**B0** might (<**Φ** (>**B0** find happiness) (>**Φ** and (>**B0** forget me))))

(b) Transition sequence for the derivation tree.

Figure 4: Example of the algorithm run over a sentence with tensed VP coordination.

come >**B3**, a combinatory rule that may be unnecessary for defining the competence grammar of human languages, but which is required if parsing performance is to be as incremental as possible. Fortunately, the configuration with two connected >**B2** combinatory rules appears very rarely in CCGbank.

Many papers have been published on using left-branching CCG derivations but, to the best of our knowledge, none of them explains how are they constructed from right-branching CCGbank trees. A very simple algorithm for that can be made using our tree rotation function. Here we use rotation in the opposite direction i.e. rotation to left (Figure 3b). We cannot apply this operation from the top node of the CCGbank tree because that tree does not satisfy the assumption of the algorithm: immediate children are not "perfect" (here perfect means being left-branching). That is why we start from the bottom of the tree with terminal nodes that are trivially "perfect" and apply tree transformation to each node in post-order traversal.

This incremental tree rotation algorithm is inspired by the AVL self-balancing binary search trees (Adelson-Velskii and Landis, 1962) and Red-Black trees (Guibas and Sedgewick, 1978; Okasaki, 1999). The main difference is that here we are trying to do the opposite of AVLs: instead of making the tree perfectly balanced we are trying to make it perfectly unbalanced, i.e. leaning to the right (or left). Also, our imperfections start at the top and are pushed to the bottom of the tree which is in contrast to AVLs trees where imperfections start at the bottom and get pushed to the top.

The last important point about tree rotation concerns punctuation rules. All punctuation is attached to the left of the highest possible node in case of left-branching derivations (Hockenmaier and Bisk, 2010), while in the right-branching derivations we lower the punctuation to the bottom left neighbouring node. Punctuation has no influence on the predicate-argument structure so it is safe to apply this transformation.

3.2 Revealing transitions

If the topmost element on the stack is of the form $X\backslash X$ and the second topmost element on the stack has on its right edge one or more constituents of a type $X|\$$ we allow *reveal* transition.[4] This is a more general way of revealing than approaches of Pareschi and Steedman (1987) and Ambati et al. (2015) who attempt to reveal only constituents of type X while we reveal any type that has X as its prime element (that is the meaning of $X|\$$ notation).

We also treat X[conj] as right adjuncts of the left conjunct. Similarly to the previous case, if the topmost element on the stack is X[conj] and the right edge of the second topmost element on the stack has constituent(s) of type X, they are revealed for possible combination via $<\mathbf{\Phi}$ combinator.

If *reveal* transition is selected, as in transition 14 in Figure 4b, the parser enters into a mode of choosing among different constituents labelled $X|\$$ that could be modified by the right adjunct $X\backslash X$. After particular $X|\$$ node is chosen $X\backslash X$ is combined with it and the rest of the tree above X node is rebuilt in the same way. This rebuild is fully deterministic and is done quickly even though in principle it could take $O(n)$ to compute. Even in the worst case scenario, it does not make the complexity of the algorithm go higher than $O(n^2)$.

The ability of our algorithm to choose among different possible revealing options is unique among all the proposals for revealing. For transition 15 in Figure 4b the parser can choose whether to adjoin (coordinate) to a verb phrase that already contains a left modifier or without. This is similar to *Selective Modifier Placement* strategy from older Augmented Transition Network (ATN) systems (Woods, 1973) which finds all the attachment options that are syntactically legal and then allows the parser to choose among those using some criteria. Woods (1973) suggests using lexical semantic information for this selection, but in his ATN system only handwritten semantic selection rules were used. Here we will also use selection based on the lexical content but it will be broad coverage and learned from the data. This ability to semantically select the modifier's attachment point is essential for good parsing results as will be shown.

[4]The "$ notation" is from (Steedman, 2000) where $ is used as a (potentially empty) placeholder variable ranging over multiple arguments.

4 Neural Model

The neural probabilistic model that chooses which transition should be taken next conditions on the whole state of the configuration in a similar way to RNNG parser (Dyer et al., 2016). The words in the sentence are first embedded using the concatenation of top layers of ELMo embeddings (Peters et al., 2018) that are normalised to L2 norm and then refined with two layers of bi-LSTM (Graves et al., 2005). The neural representation of the terminal is composed of concatenated ELMo embedding and supertag embedding.

The representation of a subtree combines:

- span representation – we subtract representation of the leftmost terminal from the representation of the rightmost terminal as done in LSTM-Minus architecture (Wang and Chang, 2016),
- combinator and category embeddings,
- head words encoding – because each constituent can have a set of heads, for instance arising from coordination, we model representation of heads with DeepSet architecture (Zaheer et al., 2017) over representations of head terminals.

We do not use recursive neural networks like Tree-LSTM (Tai et al., 2015) to encode subtrees because of the frequency of tree rotation. These operations are fast, but they would trigger frequent recomputation of the neural tree representation, so we opted for a mechanism that is invariant to rebranching.

The stack representation is encoded using Stack-LSTM (Dyer et al., 2015). The configuration representation is the concatenation of the stack representation and the representation of the rightmost terminal in the stack. The next *non-revealing* transition is chosen by a two-layer feed-forward network.

If the *reveal* transition is triggered, the system needs to choose which among the candidate nodes $X|\$$ to adjoin the right modifier $X\backslash X$ to. The number of these modifiers can vary so we cannot use a simple feed-forward network to choose among them. Instead, we use the mechanism of *Pointer networks* (Vinyals et al., 2015), which works in a similar way to attention (Bahdanau et al., 2014) except that attention weights are interpreted as probabilities of selecting any particular node. Attention is computed over representations of each candidate node. Because we expect that there

	Waiting time	Connect-edness
Right-branching	4.29	5.01
Left-branching	2.32	3.15
Ambati et al. (2015)*	0.69	2.15
Revealing our	**0.46**	**1.72**

Table 1: Train set measure of incrementality. *: taken from (Ambati et al., 2015)

	heads	SMP	LF	UF	Sup. Tag
Left	yes	—	89.2	95.1	95.0
Right	yes	—	89.1	95.0	**95.1**
Revealing	no	yes	89.3	95.2	94.9
Revealing	yes	no	88.8	94.9	94.9
Revealing	yes	yes	**89.5**	**95.4**	**95.1**

Table 2: Development set F1 results with greedy decoding for CCG dependencies.

could be some preference for attaching adjuncts high or low in the tree we add to the context representation of each node two position embeddings (Vaswani et al., 2017) that encode the candidate node's height and depth in the current tree.

We optimize for maximum log-likelihood on the training set, using only the most frequent supertags and the most important combinators. To avoid discarding sentences with rare supertags and type-changing rules we use all supertags and combinatory rules during training but do not add their probability to the loss function. The number of supertags used is 425, as in the Easy-CCG parser, and the combinatory rules that are used are the same as in C&C parser. The loss is minimised for 15 epochs on the training portion of CCGbank (Hockenmaier and Steedman, 2007) using Adam with learning rate 0.001. Dimensionality is set to 128 in all cases, except for ELMo set at 300. Dropout is applied only to the ELMo input with a rate of 0.2. The parser is implemented in Scala using the DyNet toolkit (Neubig et al., 2017) and is available at https://github.com/stanojevic/Rotating-CCG.

5 Experiments

5.1 How incremental is the Revealing algorithm?

To measure the incrementality of the proposed algorithm we use two evaluation metrics: *waiting time* and *connectedness*. Waiting time is the average number of nodes that need to be shifted before the dependency between two nodes is established. The minimal value for a fully incremental algorithm is 0 (the single shift that is always necessary is not counted). Connectedness is defined as the average stack size before a shift operation is performed (the initial two shifts are forced so they are not taken in the average). The minimal value for connectedness is 1. We have computed these measures on the training portion of the CCGbank for standard non-incremental right-branching derivations, the more incremental left-branching derivations and our revealing derivations. We also put in the results numbers for the previous proposal of revealing by Ambati et al. (2015) taken from their paper but these numbers should be taken with caution, because it is not clear from the paper whether the authors computed them in the same way and on the same portion of the dataset as we did. Table 1 results shows that our revealing derivations are significantly more incremental even in comparison to previous revealing proposals, and barely use more than the minimal amount of stack memory.

5.2 Which algorithm gives the best parsing results?

We have tested on the development set which of the parsing algorithms gives best parsing accuracy. All the algorithms use the same neural architecture and training method except for the revealing operations that require additional mechanisms to choose the node for revealing. This allows us to isolate machine learning factors and see which of the parsing strategies works the best.

There are two methods that are often used for evaluating CCG parsers. They are both based on "deep" dependencies extracted from the derivation trees. The first is from (Clark et al., 2002) and is closer to categorial grammar view of dependencies. The second is from (Clark and Curran, 2007) and is meant to be more formalism independent and closer to standard dependencies (Caroll et al., 1998). We opt for the first option for development as we find it more robust and reliable but we report both types on the test set.

Table 2 shows the results on development set. The *heads* column shows if the head words representation is used for computing the representation of the nodes in the tree. The *SMP* column shows if Selective Modifier Placement is used: whether we choose where to attach right adjunct based only

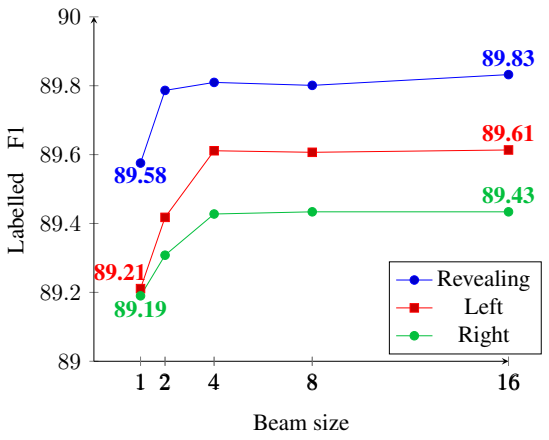


Figure 5: Influence of beam size on the dev results.

on the position embeddings or also on the node's lexical content. First we can see that Revealing approach that uses head representation and does selective modifier placement outperforms all the models both on labelled and unlabelled dependencies. Ablation experiments show that SMP was the crucial component: without it the Revealing model is much worse. This is a clear evidence that attachment heuristics are not enough and also that previous approaches that extract only single revealing option are sub-optimal.

A possible reason why Revealing model works better than Left and Right branching models is that Left and Right models need to commit early on whether there will be a right adjunct in the future or not. If they make a mistake during greedy decoding there will be no way to repair that mistake. This is not an issue for the Revealing model because it can attach right adjuncts at any point and does not need to forecast them. A natural question then is if these improvements of Revealing model will stay if we use a bigger beam. Figure 5 shows exactly that experiment. We see that the model that gains the most from the biggest beam is for the Left-branching condition, which is expected since that is the model that commits to its predictions the most — it commits with type-raising, unlike Right model, and it commits with predicting right adjunction, unlike Revealing model. With an increased beam Left model equals the Revealing greedy model. But if all the models use the same beam the Revealing model remains the best. An interesting result is that the small beam of size 4 is enough to get the maximal improvement. This probably reflects the low degree of lexical ambiguity that is unresolved at each point during parsing.

	Tag	UF	LF
Lewis and Steedman (2014)	93.0	88.6	81.3
Ambati et al. (2015)	91.2	89.0	81.4
Hockenmaier (2003)	92.2	92.0	84.4
Zhang and Clark (2011)	93.1	—	85.5
Clark and Curran (2007)	94.3	93.0	87.6
Revealing (beam=1)	**95.2**	**95.5**	**89.8**
Revealing (beam=4)	**95.4**	**95.8**	**90.2**

Table 3: Test set F1 results for prediction of supertags (Tag), unlabelled (UF) and labelled (LF) CCG dependencies extracted using scripts from Hockenmaier (2003) parser.

		Dev LF	Test LF
Clark and Curran (2007)		83.8	85.2
Lewis and Steedman (2014)		—	86.1
Yoshikawa et al. (2017)		86.8	87.7
Xu et al. (2016)		87.5	87.8
Lewis et al. (2016)	tri-train	87.5	88.1
Vaswani et al. (2016)		87.8	88.3
Lee et al. (2016)	tri-train	88.4	88.7
Yoshikawa et al. (2017)	tri-train	87.7	88.8
Revealing (beam=1)		**90.8**	**90.5**

Table 4: F1 results for labelled dependencies extracted with *generate* program of C&C parser (Clark and Curran, 2007).

5.3 Comparison to other published models

We compute test set results for our Revealing model and compare it to most of the previous results on CCGbank using both types of dependencies. Table 3 shows results with (Clark et al., 2002) style dependencies. Here we get state-of-the-art results by a large margin, probably mostly thanks to the machine learning component of our parser. An interesting comparison to be made is against EasyCCG parser of Lewis and Steedman (2014). This parser uses a neural supertagger of accuracy that is not too far from ours, but the dependencies extracted by our parser are much more accurate. This shows that a richer probabilistic model that we use contributes more to the good results than the exact A* search that EasyCCG does with a more simplistic model. Another comparison of relevance would be with the revealing model of Ambati et al. (2015) but the comparison of the algorithms is difficult since the machine learning component is very different: Ambati uses a structured perceptron while our model is a heavily parametrized neural network.

In Table 4 we show results with the second type of dependencies used for CCG evaluation. All the models, except Clark and Curran (2007), are neural and use external embeddings. From the presented models only Revealing and Xu et al. (2016) are transition based. All other models have a global search either via CKY or A* search. Our revealing-based parser that does only greedy search is outperforming all of them including those trained on large amounts of unlabelled data using semi-supervised techniques like tri-training (Lewis et al., 2016; Lee et al., 2016; Yoshikawa et al., 2017).

In some sense, all the neural models in Table 4 are implicitly trained in semi-supervised way because they use pretrained embeddings that are estimated on unlabelled data. The quality of ELMo embeddings is probably one of the reasons why our parser achieves such good results. However, another semi-supervised training method, namely tri-training, is particularly attractive because, unlike ELMo, it is trained on a CCG parsing objective which is more closely aligned to what we want to do. All tri-training models are trained on much larger dataset that in addition to CCGbank also includes 43 million word corpus automatically annotated with silver CCG derivations by Lewis et al. (2016). It is likely that incorporating tri-training into our training setup will further increase the improvement over other models.

6 Other relevant work

Recurrent Neural Network Grammar (RNNG) (Dyer et al., 2016) is a fully incremental top-down parsing model. Because it is top-down it has no issues with right branching structures, but right adjuncts would still make parsing more difficult for RNNG because they will have to be predicted even earlier than in Left- and Right- branching derivations in CCG.

Left-corner parsers (which can be seen as a more constrained version of CCG Left-branching parsing strategy) seem more psychologically realistic than top-down parsers (Abney and Johnson, 1991; Resnik, 1992; Stanojević and Stabler, 2018). Some proposals about handling right adjunction in left-corner parsing are based on extension to generalized left-corner parsers (Demers, 1977; Hale, 2014) that can force some grammar rules (in particular right-adjunction rules) to be less incremental. Our approach does not decrease incrementality of the parser in this way. On the contrary, having a special mechanism for right adjunction makes parser both more incremental and more accurate.

Revealing based on higher order unification by Pareschi and Steedman (1987) was also proposed by Steedman (1990) as the basis for CCG explanation of gapping. The present derivation-based mechanism for revealing does not extend to gapping, and is targeting to model only derivations that could be explained with a standard CCG grammar derived from CCGbank. While that guarantees that we stay in the safe zone of sound and complete "standard" CCG derivations, it would be good as a future work to extend support for gapping and other types of derivations not present in CCGbank.

Niv (1993, 1994) proposed an alternative to the unification-based account of Pareschi and Steedman similar to our proposal for online tree rotation. Niv's parser is mostly a formal treatment of left-to-right rotations evaluated against psycholinguistic garden paths, but lacks the wide coverage implementation and statistical parsing model as a basis for resolving attachment ambiguities.

7 Conclusion

We have presented a revealing-based incremental parsing algorithm that has special transitions for handling right-adjunction. The parser is neutral with regard to the particular semantic representation used. It is computationally efficient, and can reveal all possible constituents types. It is the most incremental CCG parser yet proposed, and has state-of-the-art results against all published parsers trained on the CCGbank under both dependency recovery measures that are in use for the purpose.

Acknowledgments

This work was supported by ERC H2020 Advanced Fellowship GA 742137 SEMANTAX grant.

References

Steven P. Abney and Mark Johnson. 1991. Memory requirements and local ambiguities of parsing strategies. *Journal of Psycholinguistic Research*, 20:233–249.

G M Adelson-Velskii and E M Landis. 1962. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3(2):263–266.

Anthony Ades and Mark Steedman. 1982. On the order of words. *Linguistics and Philosophy*, 4:517–558.

Bharat Ram Ambati, Tejaswini Deoskar, Mark Johnson, and Mark Steedman. 2015. An Incremental Algorithm for Transition-based CCG Parsing. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 53–63. Association for Computational Linguistics.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

John Caroll, Ted Briscoe, and Antonio Sanfilippo. 1998. Parser evaluation: a survey and a new proposal. In *First International Conference on language resources & evaluation: Granada, Spain, 28-30 May 1998*, pages 447–456. European Language Resources Association.

Stephen Clark and James R Curran. 2007. Wide-coverage efficient statistical parsing with CCG and log-linear models. *Computational Linguistics*, 33(4):493–552.

Stephen Clark, Julia Hockenmaier, and Mark Steedman. 2002. Building deep dependency structures with a wide-coverage CCG parser. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, pages 327–334. Association for Computational Linguistics.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition*, 3rd edition. The MIT Press.

Alan J. Demers. 1977. Generalized left corner parsing. In *4th Annual ACM Symposium on Principles of Programming Languages*, pages 170–181.

Chris Dyer, Miguel Ballesteros, Wang Ling, Austin Matthews, and Noah A. Smith. 2015. Transition-based dependency parsing with stack long short-term memory. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 334–343, Beijing, China. Association for Computational Linguistics.

Chris Dyer, Adhiguna Kuncoro, Miguel Ballesteros, and Noah A Smith. 2016. Recurrent neural network grammars. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 199–209.

Jason Eisner. 1996. Efficient normal-form parsing for Combinatory Categorial Grammar. In *Proceedings of the 34th annual meeting on Association for Computational Linguistics*, pages 79–86. Association for Computational Linguistics.

Alex Graves, Santiago Fernández, and Jürgen Schmidhuber. 2005. Bidirectional LSTM Networks for Improved Phoneme Classification and Recognition. In *Proceedings of the 15th International Conference on Artificial Neural Networks: Formal Models and Their Applications - Volume Part II*, ICANN'05, pages 799–804, Berlin, Heidelberg. Springer-Verlag.

Leo J. Guibas and Robert Sedgewick. 1978. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, SFCS '78, pages 8–21, Washington, DC, USA. IEEE Computer Society.

John T. Hale. 2014. *Automaton Theories of Human Sentence Comprehension*. CSLI, Stanford.

Julia Hockenmaier. 2003. *Data and models for statistical parsing with Combinatory Categorial Grammar*. Ph.D. thesis, University of Edinburgh. College of Science and Engineering. School of Informatics.

Julia Hockenmaier and Yonatan Bisk. 2010. Normal-form Parsing for Combinatory Categorial Grammars with Generalized Composition and Type-raising. In *Proceedings of the 23rd International Conference on Computational Linguistics*, COLING '10, pages 465–473, Stroudsburg, PA, USA. Association for Computational Linguistics.

Julia Hockenmaier and Mark Steedman. 2007. CCGbank: a corpus of CCG derivations and dependency structures extracted from the Penn Treebank. *Computational Linguistics*, 33(3):355–396.

Kenton Lee, Mike Lewis, and Luke Zettlemoyer. 2016. Global Neural CCG Parsing with Optimality Guarantees. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2366–2376. Association for Computational Linguistics.

Mike Lewis, Kenton Lee, and Luke Zettlemoyer. 2016. Lstm ccg parsing. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 221–231, San Diego, California. Association for Computational Linguistics.

Mike Lewis and Mark Steedman. 2014. A* CCG parsing with a supertag-factored model. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 990–1000.

William Marslen-Wilson. 1973. Linguistic structure and speech shadowing at very short latencies. *Nature*, 244:522–523.

Graham Neubig, Chris Dyer, Yoav Goldberg, Austin Matthews, Waleed Ammar, Antonios Anastasopoulos, Miguel Ballesteros, David Chiang, Daniel Clothiaux, Trevor Cohn, Kevin Duh, Manaal Faruqui, Cynthia Gan, Dan Garrette, Yangfeng Ji, Lingpeng Kong, Adhiguna Kuncoro, Gaurav Kumar, Chaitanya Malaviya, Paul Michel, Yusuke Oda, Matthew Richardson, Naomi Saphra, Swabha Swayamdipta, and Pengcheng Yin. 2017. Dynet: The dynamic neural network toolkit. *arXiv preprint arXiv:1701.03980*.

Michael Niv. 1993. *A Computational Model of Syntactic Processing: Ambiguity Resolution from Interpretation*. Ph.D. thesis, University of Pennsylvania. IRCS Report 93-27.

Michael Niv. 1994. A psycholinguistically motivated parser for CCG. In *Proceedings of the 32nd annual meeting on Association for Computational Linguistics*, pages 125–132. Association for Computational Linguistics.

Chris Okasaki. 1999. Red-black trees in a functional setting. *Journal of functional programming*, 9(4):471–477.

Remo Pareschi and Mark Steedman. 1987. A Lazy Way to Chart-parse with Categorial Grammars. In *Proceedings of the 25th Annual Meeting on Association for Computational Linguistics*, ACL '87, pages 81–88, Stroudsburg, PA, USA. Association for Computational Linguistics.

Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep contextualized word representations. In *Proc. of NAACL*.

Philip Resnik. 1992. Left-corner parsing and psychological plausibility. In *Proceedings of the 14th International Conference on Computational Linguistics, COLING 92*, pages 191–197.

Miloš Stanojević and Edward Stabler. 2018. A sound and complete left-corner parsing for minimalist grammars. In *Proceedings of the Eight Workshop on Cognitive Aspects of Computational Language Learning and Processing*, pages 65–74. Association for Computational Linguistics.

Mark Steedman. 1990. Gapping as constituent coordination. *Linguistics and philosophy*, 13(2):207–263.

Mark Steedman. 2000. *The Syntactic Process*. MIT Press, Cambridge, MA.

Kai Sheng Tai, Richard Socher, and Christopher D. Manning. 2015. Improved semantic representations from tree-structured long short-term memory networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1556–1566, Beijing, China. Association for Computational Linguistics.

Ashish Vaswani, Yonatan Bisk, Kenji Sagae, and Ryan Musa. 2016. Supertagging With LSTMs. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 232–237. Association for Computational Linguistics.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 5998–6008. Curran Associates, Inc.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer networks. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 2692–2700. Curran Associates, Inc.

Wenhui Wang and Baobao Chang. 2016. Graph-based Dependency Parsing with Bidirectional LSTM. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2306–2315. Association for Computational Linguistics.

William Woods. 1973. An experimental parsing system for Transition Network Grammars. In Randall Rustin, editor, *Natural Language Processing*, pages 111–154. Algorithmics Press, New York.

Wenduan Xu. 2016. LSTM shift-reduce CCG parsing. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 1754–1764.

Wenduan Xu, Michael Auli, and Stephen Christopher Clark. 2016. Expected f-measure training for shift-reduce parsing with recurrent neural networks. In *Proceedings of the 54nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics.

Wenduan Xu, Stephen Clark, and Yue Zhang. 2014. Shift-reduce CCG parsing with a dependency model. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 218–227.

Masashi Yoshikawa, Hiroshi Noji, and Yuji Matsumoto. 2017. A* ccg parsing with a supertag and dependency factored model. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 277–287. Association for Computational Linguistics.

M. Zaheer, S. Kottur, M. Ravanbakhsh, B. Poczos, R. Salakhutdinov, and A. Smola. 2017. Deep sets. In *NIPS*. (Accepted for oral presentation, 1.23

Yue Zhang and Stephen Clark. 2011. Shift-reduce CCG parsing. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies-Volume 1*, pages 683–692. Association for Computational Linguistics.