Guidelines for the CLEAR Style Constituent to Dependency Conversion

Jinho D. Choi
choijd@colorado.edu

Martha Palmer
mpalmer@colorado.edu

Center for Computational Language and EducAtion Research

University of Colorado Boulder

Institute of Cognitive Science
Technical Report 01-12

Contents

1 Introduction

1.1 Motivation

Most current state-of-the-art dependency parsers take various statistical learning approaches (Mcdonald and Pereira, 2006; Nivre, 2008; Huang and Sagae, 2010; Rush and Petrov, 2012). The biggest advantage of statistical parsing is found in the ability to adapt to new data without modifying the parsing algorithm. Statistical parsers can be trained on data from new domains, genres, or languages as long as they are provided with sufficiently large training data from the new sources. On the other hand, this is also the biggest drawback for statistical parsing because annotating such large training data is manually intensive work that is costly and time consuming.

Although a few manually annotated dependency Treebanks are available for English (Rambow et al., 2002; Čmejrek et al., 2004), constituent Treebanks are still more dominant (Marcus et al., 1993; Weischedel et al., 2011). It has been shown that the Penn Treebank style constituent trees can reliably be converted to dependency trees using head-finding rules and heuristics (Johansson and Nugues, 2007; de Marneffe and Manning, 2008a; Choi and Palmer, 2010). By automatically converting these constituent trees to dependency trees, statistical dependency parsers have access to a larger amount of training data. Few tools are available for constituent to dependency conversion. Two of the most popular ones are the LTH and the Stanford dependency converters.[1] The LTH converter had been used to provide English data for the CoNLL'07-09 shared tasks (Nivre et al., 2007; Surdeanu et al., 2008; Hajič et al., 2009). The LTH converter makes several improvements over its predecessor, Penn2Malt,[2] by adding syntactic and semantic dependencies retained from function tags (e.g., `PRD`, `TMP`) and producing long-distance dependencies caused by empty categories or gapping relations.[3] The Stanford converter was used for the SANCL'12 shared task (Petrov and McDonald, 2012), and is perhaps the most widely used dependency converter at the moment. The Stanford converter gives fine-grained dependency labels useful for many NLP tasks. Appendix B shows descriptions of the CoNLL and the Stanford dependency labels generated by these two tools.

Both converters perform well for most cases; however, they are somewhat customized to the Penn Treebank (mainly to the Wall Street Journal corpus; see Marcus et al. (1993)), so do not work as well when applied to different corpora. For example, the OntoNotes Treebank (Weischedel et al., 2011) contains additional constituent tags not used by the Penn Treebank (e.g., `EDITED`, `META`), and shows occasional departures from the Penn Treebank guidelines (e.g., inserting `NML` phrases, separating hyphenated words; see Figure 1). These new formats affect the ability of existing tools to find correct dependencies, motivating us to aim for a more resilient approach.

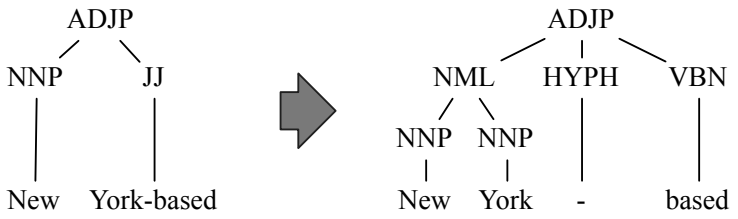

Figure 1: Structural differences in the Penn Treebank (left) and the OntoNotes Treebank (right). The hyphenated word is tokenized, `HYPH`, and the nominal phrase is grouped, `NML`, in the OntoNotes.

Producing more informative trees provides additional motivation. The Stanford converter generates dependency trees without using information such as function tags (Appendix A.3), empty categories (Section 2), or

[1] The LTH dependency converter: `http://nlp.cs.lth.se/software/treebank_converter/`
The Stanford dependency converter: `http://nlp.stanford.edu/software/stanford-dependencies.shtml`

[2] Penn2Malt: `http://stp.lingfil.uu.se/~nivre/research/Penn2Malt.html`

[3] The term "long-distance dependency" is used to indicate dependency relations between words that are not within the same domain of locality.

gapping relations (Section 5.1), which is provided in manually annotated but not in automatically generated constituent trees. This enables the Stanford converter to generate the same kind of dependencies given either manually or automatically generated constituent trees. However, it sometimes misses important details such as long-distance dependencies, which can be retrieved from empty categories, or produces unclassified dependencies that can be disambiguated by function tags. This becomes an issue when this converter is used for generating dependency trees for training because statistical parsers trained on these trees would not reflect these details.

The dependency conversion described here takes the Stanford dependency approach as the core structure and integrates the CoNLL dependency approach to add long-distance dependencies, to enrich important relations like object predicates, and to minimize unclassified dependencies. The Stanford dependency approach is taken for the core structure because it gives more fine-grained dependency labels and is currently used more widely than the CoNLL dependency approach. For our conversion, head-finding rules and heuristics are completely reanalyzed from the previous work to handle constituent tags and relations not introduced by the Penn Treebank. Our conversion has been evaluated with several different constituent Treebanks (Marcus et al., 1993; Nielsen et al., 2010; Weischedel et al., 2011; Verspoor et al., 2012) and showed robust results across these corpora.

1.2 Background

1.2.1 Dependency graph

A dependency structure can be represented as a directed graph. For a given sentence $s = w_1, \ldots, w_n$, where w_i is the i'th word token in the sentence, a dependency graph $G_s = (V_s, E_s)$ can be defined as follows:

$$\begin{aligned} V_s &= \{w_0 = \text{root}, w_1, \ldots, w_n\} \\ E_s &= \{(w_i \xrightarrow{r} w_j) : i \neq j, w_i \in V_s, w_j \in V_s - \{w_0\}, r \in R_s\} \\ R_s &= \text{A subset of all dependency relations in } s \end{aligned}$$

$w_i \xrightarrow{r} w_j$ is a directed edge from w_i to w_j with a label r, which implies that w_i is the head of w_j with a dependency relation r. A dependency graph is considered *well-formed* if it satisfied all of the following properties:

- **Root**: there must be a unique vertex, w_0, with no incoming edge.
  $\neg[\exists k.\, (w_0 \leftarrow w_k)]$
- **Single head**: each vertex $w_{i>0}$ must have a single incoming edge.
  $\forall i.\, [i > 0 \Rightarrow \forall j.\, [(w_i \leftarrow w_j) \Rightarrow \neg[\exists k.\, (k \neq j) \wedge (w_i \leftarrow w_k)]]]$
- **Connected**: there must be an undirected path between any two vertices.[4]
  $[\forall i, j.\, (w_i - w_j)]$, where $w_i - w_j$ indicates an undirected path between w_i and w_j.
- **Acyclic**: a directed path between any two vertices must not be cyclic.
  $\neg[\exists i, j.(w_i \ {}^{*}\!\leftarrow w_j) \wedge (w_i \rightarrow^* w_j)]$, where $w_i \rightarrow^* w_j$ indicates a directed path from w_i to w_j.

Sometimes, *projectivity* is also considered a property of a well-formed dependency graph. When projectivity is considered, no crossing edge is allowed when all vertices are lined up in linear-order and edges are drawn above the vertices (Figure 2). Preserving projectivity can be useful because it enables regeneration of the original sentence from its dependency graph without losing the word order. More importantly, it reduces parsing complexity to $O(n)$ (Nivre and Scholz, 2004). Although preserving projectivity has a few advantages, non-projective dependencies are often required, especially in flexible word order languages, to represent correct dependencies. Even in rigid word order languages such as English, non-projective dependencies are necessary to represent long-distance dependencies. In Figure 3, there is no way of describing the dependency relations for both "*bought* → *yesterday*" and "*car* → *is*" without having their edges cross. Because of such cases, projectivity is dropped from the properties of a well-formed dependency graph for this research.

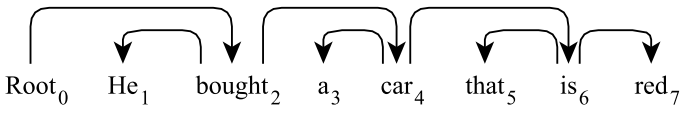

Figure 2: An example of a projective dependency graph.

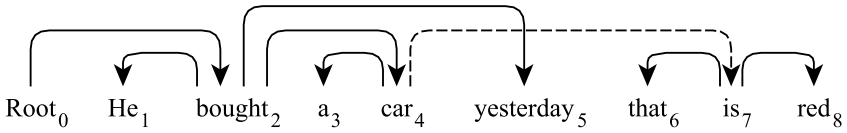

Figure 3: An example of a non-projective dependency graph. The dependency between *car* and *is* is non-projective because it crosses the dependency between *bought* and *yesterday*.

A well-formed dependency graph, with or without the projective property, satisfies all of the conditions for tree structures, so is called a 'dependency tree'.

1.2.2 Types of empty categories

Empty categories are syntactic units, usually nominal phrases, that appear in the surface form to signal the canonical locations of syntactic elements in its deep structure (Cowper, 1992; Chomsky, 1995). Table 1 shows a list of empty categories used in constituent Treebanks for English. Some of these empty categories have overloaded meanings. For instance, `*PRO*` indicates empty subjects caused by different pro-drop cases (e.g., control, imperative, nominalization). See Bies et al. (1995); Taylor (2006) for more details about these empty categories.

Type	Description
`*PRO*`	Empty subject of pro-drop (e.g., control, ECM, imperative, nominalization)
`*T*`	Trace of *wh*-movement and topicalization
`*`	Trace of subject raising and passive construction
`0`	Null complementizer
`*U*`	Unit (e.g., $)
`*ICH*`	Pseudo-attach: Interpret Constituent Here
`*?*`	Placeholder for ellipsed material
`*EXP*`	Pseudo-attach: EXPletives
`*RNR*`	Pseudo-attach: Right Node Raising
`*NOT*`	Anti-placeholder in template gapping
`*PPA*`	Pseudo-attach: Permanent Predictable Ambiguity

Table 1: A list of empty categories used in constituent Treebanks for English.

1.3 Overview

Figure 4 shows the overview of our constituent to dependency conversion. Given a constituent tree, empty categories are mapped to their antecedents first (step 2; see Section 2). This step relocates phrasal nodes regarding certain kinds of empty categories that may cause generation of non-projective dependencies.[5] Once empty categories are mapped, special cases such as apposition, coordination, or small clauses are handled

[4] An 'undirected path' implies a path between two vertices, regardless of their directionality.

[5] Although phrases in constituency trees are relocated, word order in dependency trees remains the same.

next (step 3; see Sections 3.2 to 3.4). Finally, general cases are handled using head-finding rules and heuristics (step 4; see Sections 3.1 and 4).

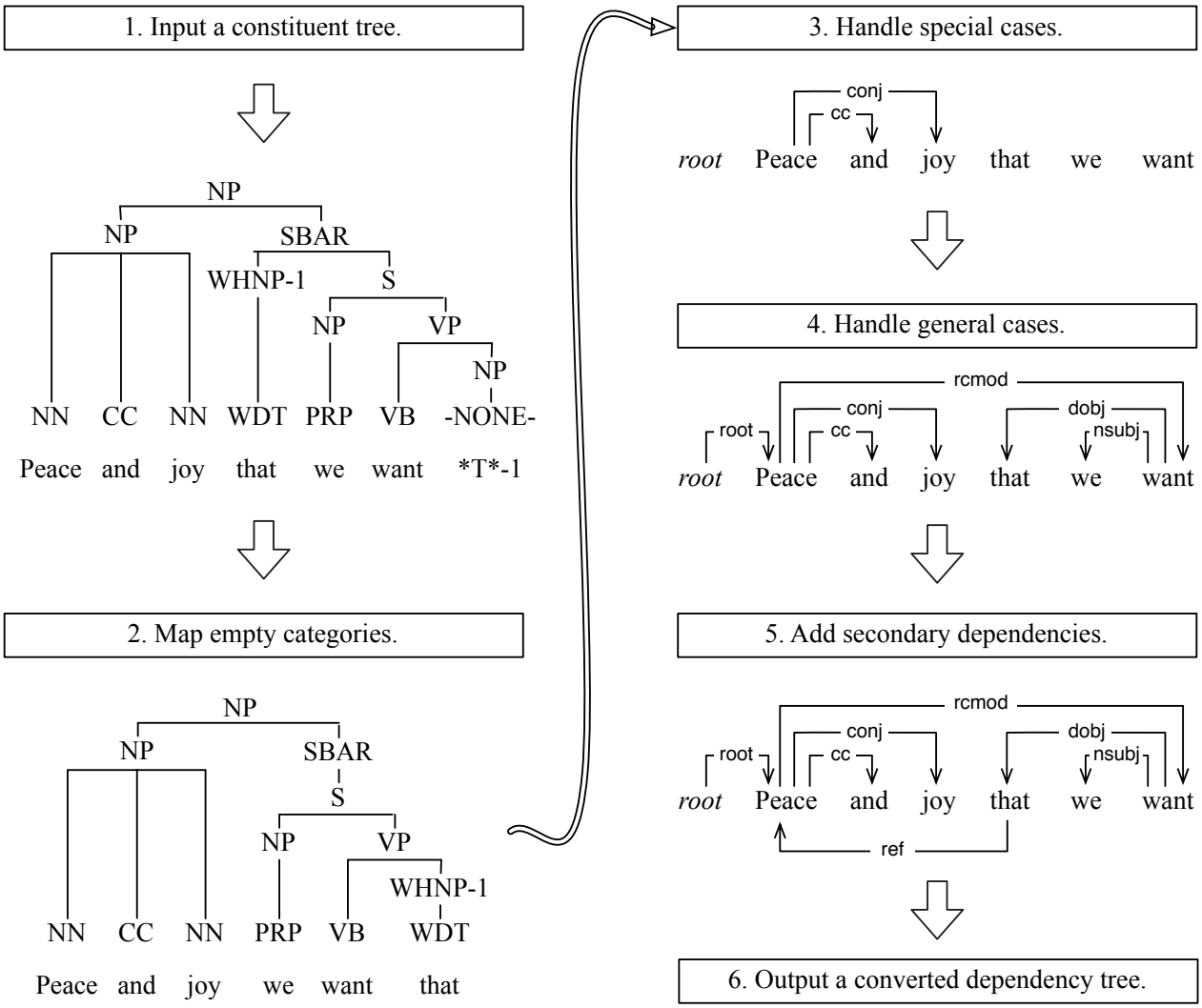

Figure 4: The overview of constituent to dependency conversion.

Secondary dependencies are added as a separate layer of this dependency tree (step 5; see Section 5). Additionally, syntactic and semantic function tags in the constituent tree are preserved as features of individual nodes in the dependency tree (not shown in Figure 4; see Appendix A.3).

2 Mapping empty categories

Most long-distance dependencies can be represented without using empty categories in dependency structure. In English, long-distance dependencies are caused by certain linguistic phenomena such as *wh*-movement, topicalization, discontinuous constituents, etc. It is difficult to find long-distance dependencies during automatic parsing because they often introduce dependents that are not within the same domain of locality, resulting in non-projective dependencies (McDonald and Satta, 2007; Koo et al., 2010; Kuhlmann and Nivre, 2010).

Four types of empty categories are used to represent long-distance dependencies during our conversion: `*T*`, `*RNR*`, `*ICH*`, and `*PPA*` (see Table 1). Note that the CoNLL dependency approach used `*EXP*` to represent extraposed elements in expletive constructions, which is not used in our approach because the annotation of `*EXP*` is somewhat inconsistent across different corpora.

2.1 Wh-movement

Wh-movement is represented by `*T*` in constituent trees. In Figure 5, `WHNP-1` is moved from the object position of the subordinate verb *liked* and leaves a trace, `*T*-1`, at its original position. Figure 6 shows a dependency tree converted from the constituent tree in Figure 5. The dependency of `WHNP-1` is derived from its original position so that it becomes a direct object of *liked* (`DOBJ`; Section 4.5.2).

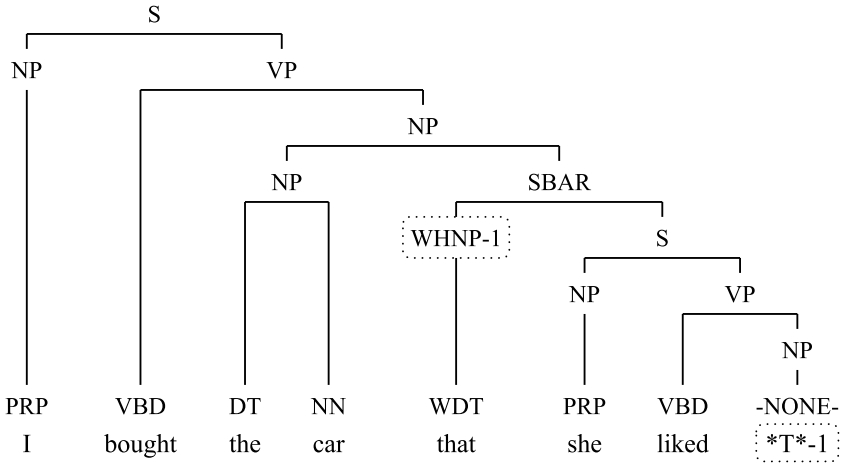

Figure 5: An example of *wh*-movement.

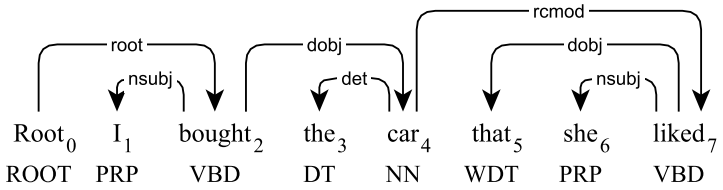

Figure 6: A dependency tree converted from the constituent tree in Figure 5

Wh-complementizers can be moved from several positions. In Figure 7, `WHNP-1` is moved from the prepositional phrase, `PP`, so in Figure 8, the complementizer *what* becomes an object of the preposition *in* (`POBJ`; Section 4.12.2). Notice that the `POBJ` dependency is non-projective; it crosses the dependency between *knew* and *was*. This is a typical case of a non-projective dependency caused by *wh*-movement.

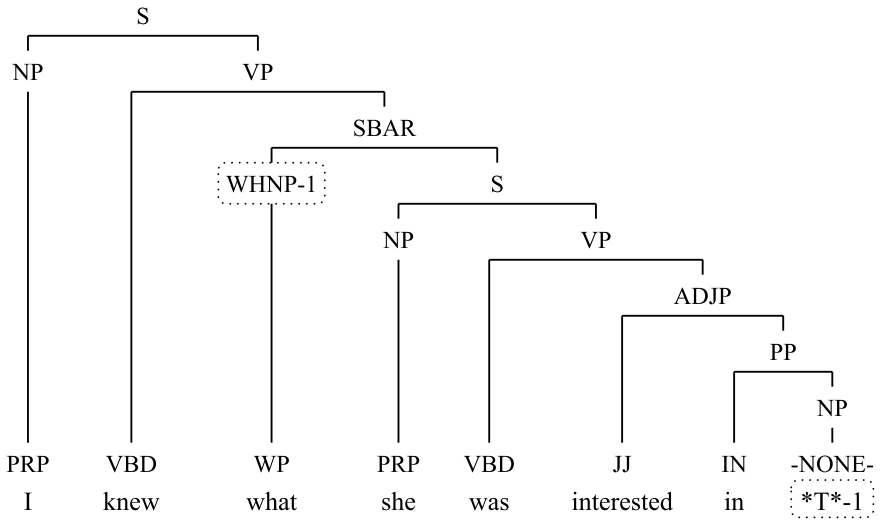

Figure 7: Another example of *wh*-movement.

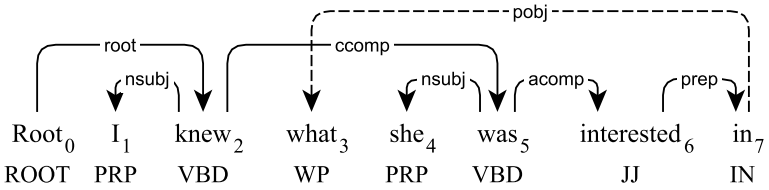

Figure 8: A dependency tree converted from the constituent tree in Figure 7. The dependency derived from the *wh*-movement, `POBJ`, is indicated by a dotted line.

2.2 Topicalization

Topicalization is also represented by `*T*`. In Figure 9, `S`-1 is moved from the subordinate clause, `SBAR`, and leaves a trace behind. In Figure 10, the head of `S`-1, *liked*, becomes a dependent of the matrix verb *seemed* (`ADVCL`; Section 4.9.1), and the preposition *like* becomes a dependent of the subordinate verb *liked* (`MARK`; Section 4.9.3). The `MARK` dependency is non-projective such that it crosses the dependency between *Root* and *seemed*.

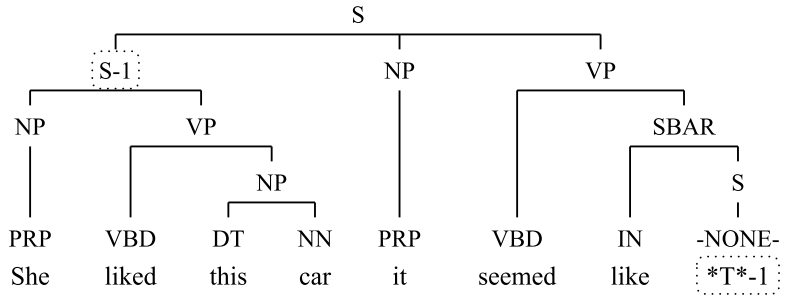

Figure 9: An example of topicalization.

There are a few cases where `*T*` mapping causes cyclic dependency relations. In Figure 11, `*T*`-1 is mapped to `S`-1 that is an ancestor of itself. Thus, the head of `S`-1, *bought*, becomes a dependent of the subordinate

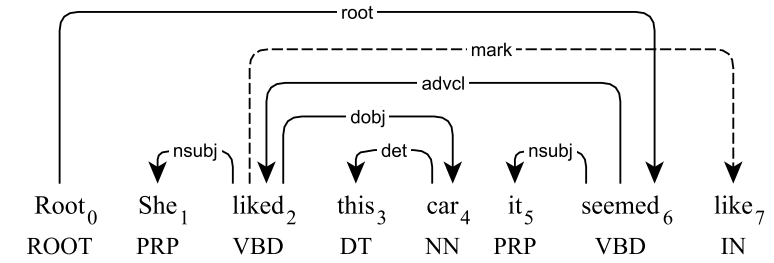

Figure 10: A dependency tree converted from the constituent tree in Figure 9. The dependency derived from the topicalization, `MARK`, is indicated by a dotted line.

verb *said* while the head of the subordinate clause, *said*, becomes a dependent of the matrix verb *bought*. Since this creates a cyclic relation in the dependency tree, such traces are ignored during our conversion (Figure 12).

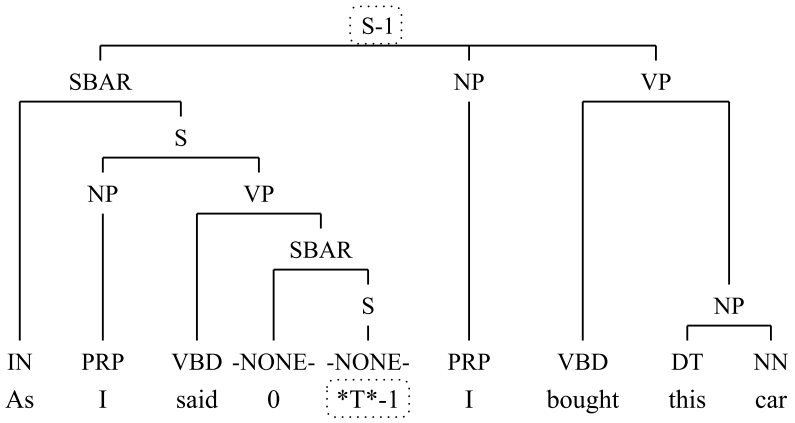

Figure 11: An example of topicalization, where a topic movement creates a cyclic relation.

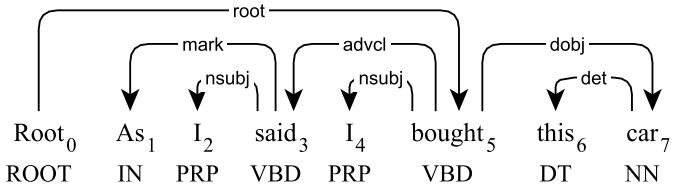

Figure 12: A dependency tree converted from the constituent tree in Figure 11.

2.3 Right node raising

Right node raising occurs in coordination where a constituent is governed by multiple parents that are not on the same level (Levine, 1985). Right node raising is represented by `*RNR*` in constituent trees. In Figure 13, `NP-1` should be governed by both `PP-1` and `PP-2`, where `*RNR*-1`'s are located. Making `NP-1` dependents of both `PP-1` and `PP-2` breaks the **single head** property (Section 1.2.1); instead, the dependency of `NP-1` is derived from its closest `*RNR*-1` in our conversion. In Figure 14, *her* becomes a dependent of the head of `PP-2`, *in*. The dependency between *her* and the head of `PP-1`, *for*, is preserved as a secondary dependency, `REF`

(referent; see Section 5). Thus, *her* is a dependent of only PP-2 in our dependency tree while the dependency to PP-2 can still be retrieved through the secondary dependency.[6]

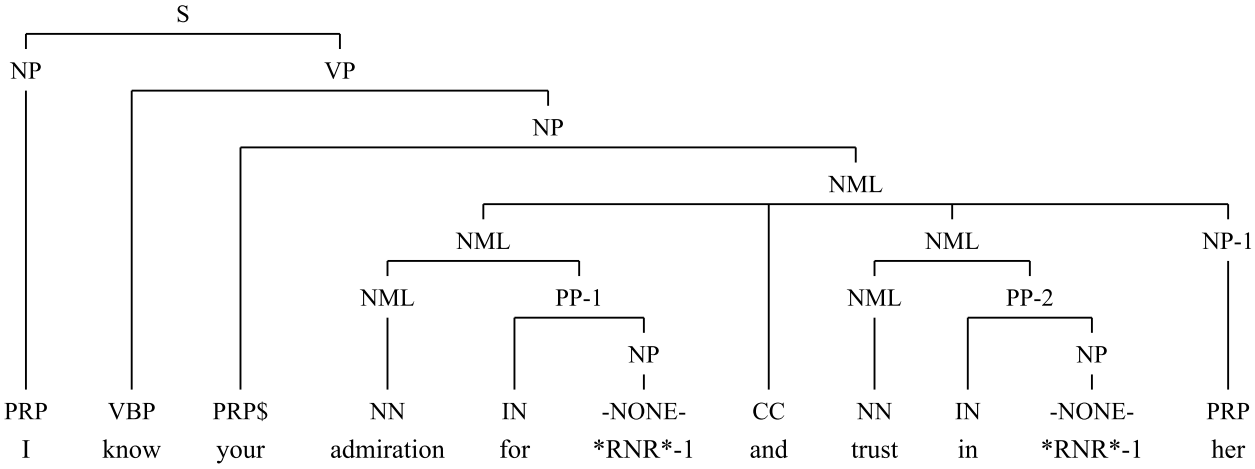

Figure 13: An example of right node raising.

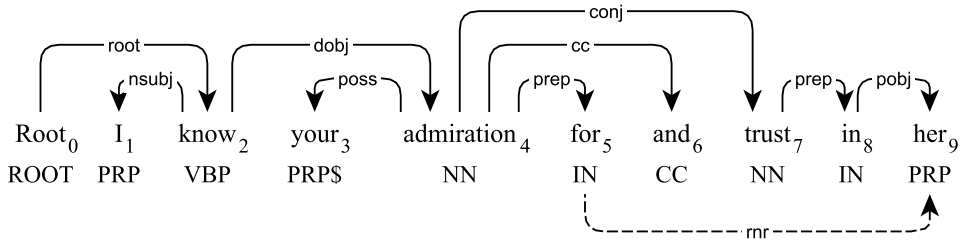

Figure 14: A dependency tree converted from the constituent tree in Figure 13. The secondary dependency, RNR, is added to a separate layer to preserve tree properties.

Note that the CoNLL dependency approach makes *her* a dependent of the head of PP-1, which creates a non-projective dependency (the dependency between *for* and *her* in Figure 15). This non-projective dependency is avoided in our approach without losing any referential information.

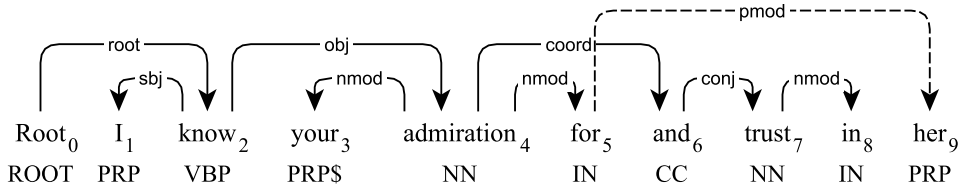

Figure 15: A CoNLL style dependency tree converted from the constituent tree in Figure 13. The dependency caused by right node raising, PMOD, is indicated by a dotted line.

[6]Secondary dependencies are not commonly used in dependency structures. These are dependencies derived from gapping relations, referent relations, right node raising, and open clausal subjects, which may break tree properties (Section 5). During our conversion, secondary dependencies are preserved in a separate layer so they can be learned either jointly or separately from other dependencies.

2.4 Discontinuous constituent

A discontinuous constituent is a constituent that is separated from its original position by some intervening material. The original position of a discontinuous constituent is indicated by `*ICH*` in constituent trees. In Figure 16, `PP-1` is separated from its original position, `*ICH*-1`, by the adverb phrase, `ADVP`. Thus, in Figure 17, the head of the prepositional phrase, *than*, becomes a prepositional modifier (`PREP`; Section 4.12.3) of the head of the adjective phrase (`ADJP-2`), *expensive*. The `PREP` dependency is non-projective; it crosses the dependency between *is* and *now*.

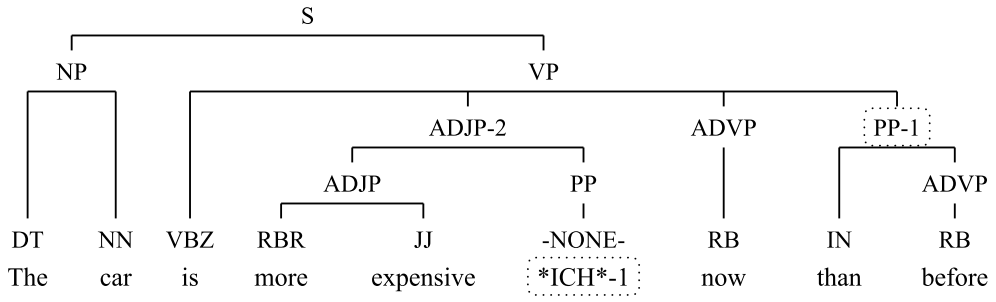

Figure 16: An example of discontinuous constituents.

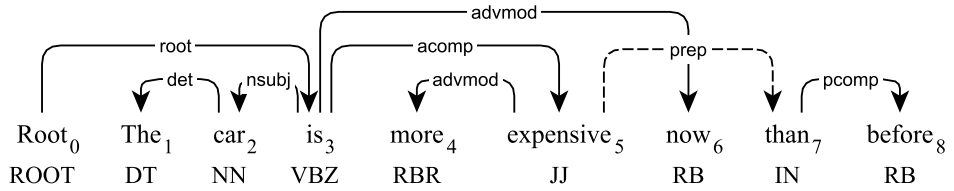

Figure 17: A dependency tree converted from the constituent tree in Figure 16. The dependency derived from the `*ICH*` movement, `PREP`, is indicated by a dotted line.

3 Finding dependency heads

3.1 Head-finding rules

Table 2 shows head-finding rules (henceforth, headrules) derived from various constituent Treebanks. For each phrase (or clause) in a constituent tree, the head of the phrase is found by using its headrules, and all other nodes in the phrase become dependents of the head. This procedure goes on recursively until every constituent in the tree becomes a dependent of one other constituent, except for the top constituent, which becomes the root of the dependency tree. A dependency tree generated by this procedure is guaranteed to be well-formed (Section 1.2.1), and may or may not be non-projective, depending on how empty categories are mapped (Section 2).

ADJP	r	JJ*|VB*|NN*;ADJP;IN;RB|ADVP;CD|QP;FW|NP;*
ADVP	r	VB*;RP;RB*|JJ*;ADJP;ADVP;QP;IN;NN;CD;NP;*
CONJP	l	CC;VB*;NN*;TO|IN;*
EDITED	r	VP;VB*;NN*|PRP|NP;IN|PP;S*;*
EMBED	r	S*;FRAG|NP;*
FRAG	r	VP;VB*;-PRD;S|SQ|SINV|SBARQ;NN*|NP;PP;SBAR;JJ*|ADJP;RB|ADVP;INTJ;*
INTJ	l	VB*;NN*;UH;INTJ;*
LST	l	LS|CD;NN;*
META	l	NP;VP|S;*
NAC	r	NN*;NP;S|SINV;*
NML	r	NN*|NML;CD|NP|QP|JJ*|VB*;*
NP	r	NN*|NML;NX;PRP;FW;CD;NP;-NOM;QP|JJ*|VB*;ADJP;S;SBAR;*
NX	r	NN*;NX;NP;*
PP	l	RP;TO;IN;VB*;PP;NN*;JJ;RB;*
PRN	r	VP;NP;S|SBARQ|SINV|SQ;SBAR;*
PRT	l	RP;PRT;*
QP	r	CD;NN*;JJ;DT|PDT;RB;NP|QP;*
RRC	l	VP;VB*;-PRD;NP|NN*;ADJP;PP;*
S	r	VP;VB*;-PRD;S|SQ|SINV|SBARQ;SBAR;NP;PP;*
SBAR	r	VP;S|SQ|SINV;SBAR*;FRAG|NP;*
SBARQ	r	VP;SQ|SBARQ;S|SINV;FRAG|NP;*
SINV	r	VP;VB*;MD;S|SINV;NP;*
SQ	r	VP;VB*;SQ;S;MD;NP;*
UCP	l	*
VP	l	VP;VB*;MD|TO;JJ*|NN*|IN;-PRD;NP;ADJP|QP;S;*
WHADJP	r	JJ*|VBN;WHADJP|ADJP;*
WHADVP	r	RB*|WRB;WHADVP;*
WHNP	r	NN*;WP|WHNP;NP|NML|CD;JJ*|VBG;WHADJP|ADJP;DT;*
WHPP	l	IN|TO;*
X	r	*

Table 2: Head-finding rules. `l`/`r` implies the search direction for the leftmost/rightmost constituent. `*`/`+` implies 0/1 or more characters and `-TAG` implies any POS tag with the specific function tag. `|` implies a logical OR and `;` is a delimiter between POS tags. Each rule gives higher precedence to the left (e.g., `VP` takes the highest precedence in `S`).

Notice that the headrules in Table 2 give information about which constituents can be the heads, but do not show which constituents cannot be the heads. Some constituents are more likely to be dependents than heads. In Figure 18, both *Three times* and *a week* are noun phrases under another noun phrase. According to our headrules, the rightmost noun phrase, `NP-TMP`, is chosen to be the head of this phrase. However,

NP-TMP is actually an adverbial modifier of NP-H (NPADVMOD; Section 4.9.5); thus, NP-H should be the head of this phrase instead. This indicates that extra information is required to retrieve correct heads for this kind of phrases.

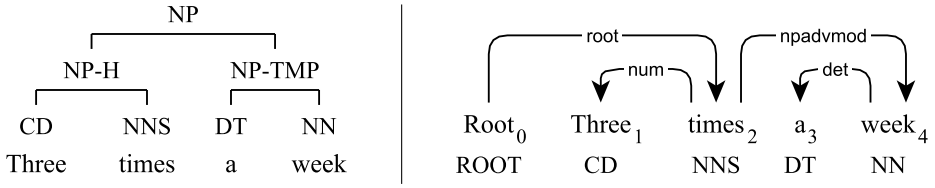

Figure 18: An example of a noun phrase modifying another noun phrase.

The *getHead*(N, R) method in Algorithm 3.1 finds the head of a phrase (lines 2-7) and makes all other constituents in the phrase dependents of the head (lines 8-11). The input to the method is the ordered list of children N and the corresponding headrules R of the phrase. The *getHeadFlag*(C) method in Algorithm 3.2 returns the head-flag of a constituent C, which indicates the dependency precedence of C: the lower the flag is, the sooner C can be chosen as the head. For example, NP-TMP in Figure 18 is skipped during the first iteration (line 2 in Algorithm 3.1) because it has the adverbial function tag TMP, so gets a flag of 1 (line 1 in Algorithm 3.2). Alternatively, NP-H is not skipped because it gets a flag of 0. Thus, NP-H becomes the head of this phrase.

Algorithm 3.1 : *getHead*(N, R)

Input: An ordered list N of constituent nodes that are siblings,
The headrules R of the parent of nodes in N.
Output: The head constituent of N with respect to R.
All other nodes in N become dependents of the head.

```
1:  if the search direction of R is r then N.reverse()        # the 2nd column in Table 2
2:  for flag in {0...3} do
3:    for tags in R do                                        # e.g,. tags ← NN*|NML
4:      for node in N do
5:        if (flag = getHeadFlag(node)) and (node is tags) then
6:          head ← node
7:          break the highest for-loop
8:  for node in N do
9:    if node ≠ head then
10:     node.head ← head
11:     node.label ← getDependencyLabel(node, node.parent, head)      # Section 4.3
12: return head
```

Algorithm 3.2 : *getHeadFlag*(C)

Input: A constituent C.
Output: The head-flag of C, that is either 0, 1, 2, or 3.

```
1:  if hasAdverbialTag(C) return 1                                  # Section 4.9
2:  if isMetaModifier(C) return 2                                   # Section 4.14.4
3:  if (C is an empty category) or isPunctuation(C) return 3        # Section 4.14.8
4:  return 0
```

The following sections describe heuristics to resolve special cases such as apposition, coordination, and small clauses, where correct heads cannot always be retrieved by headrules alone.

3.2 Apposition

Apposition is a grammatical construction where multiple noun phrases are placed side-by-side and later noun phrases give additional information about the first noun phrase. For example, in a phrase "*John, my brother*", both *John* and *my brother* are noun phrases such that *my brother* gives additional information about its preceding noun phrase, *John*. The *findApposition*(C) method in Algorithm 3.3 makes each appositional modifier a dependent of the first noun phrase in a phrase (lines 8-9). An appositional modifier is either a noun phrase without an adverbial function tag (line 5), any phrase with the function tag `HLN|TTL` (headlines or titles; line 6), or a reduced relative clause containing a noun phrase with the function tag `PRD` (non-`VP` predicate; line 7).

Algorithm 3.3 : *findApposition*(C)

Input: A constituent C.
Output: `True` if C contains apposition; otherwise, `False`.

```
1:  if (C is not NP|NML) or (C contains NN*) or (C contains no NP) return False
2:  let f be the first NP|NML in C that contains no POS   # skip possession modifier
3:  b ← False
4:  for s in all children of C preceded by f do
5:    if ((s is NML|NP) and (not hasAdverbialTag(s)))      # Section 4.9
6:       or (s has HLN|TTL)
7:       or ((s is RRC) and (s contains NP-PRD)) then
8:      s.head ← f
9:      s.label ← APPOS
10:     b ← True
11: return b
```

3.3 Coordination

Several approaches have been proposed for coordination representation in dependency structure. The Stanford dependency approach makes the leftmost conjunct the head of all other conjuncts and conjunctions. The Prague dependency approach makes the rightmost conjunction the head of all conjuncts and conjunctions (Čmejrek et al., 2004). The CoNLL dependency approach makes each preceding conjunct or conjunction the head of its following conjunct or conjunction.

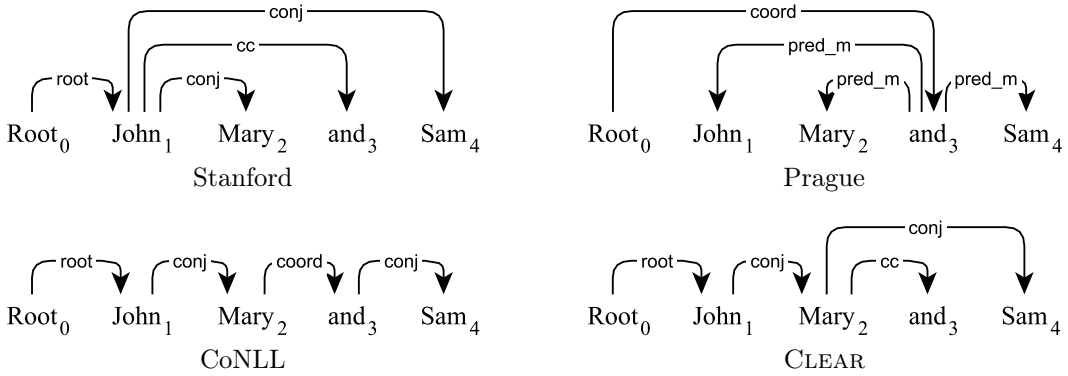

Figure 19: Different ways of representing coordination in dependency structure.

Our conversion takes an approach similar to the CoNLL dependency approach, which had been shown to work better for transition-based dependency parsing (Nilsson et al., 2006). There is one small change in our

approach such that conjunctions do not become the heads of conjuncts (CLEAR in Figure 19). This way, conjuncts are always dependents of their preceding conjuncts whether or not conjunctions exist in between.

The *getCoordinationHead*(C) method in Algorithm 3.4 finds dependencies between conjuncts and returns the head of the leftmost conjunct in C. The algorithm begins by checking if C is coordinated (line 1). For each constituent in C, the algorithm checks if it matches the conjunct head pattern of C (line 21), which varies by C's phrase type. For instance, only a non-auxiliary verb or a verb phrase can be a conjunct head in a verb phrase (see *getConjunctHeadPattern*(C) in Algorithm 3.6). When a coordinator (a conjunction, comma, or colon) is encountered, a sub-span is formed (line 9). If the span includes at least one constituent matching the conjunct head pattern, it is considered a new conjunct and the head of the conjunct is retrieved by the headrule of C (line 10). The head of the current conjunct becomes a dependent of the head of its preceding conjunct if it exists (see *getConjunctHead*$(S, R, pHead)$ in Algorithm 3.8). If there is no constituent matching the pattern, all constituents within the span become dependents of the head of the previous conjunct if it exists (lines 16-19). This procedure goes on iteratively until all constituents in C are encountered. Note that the *getCoordinationHead*(C, R) method is called before the *findApposition*(C) method in Algorithm 3.3; thus, a constituent can be a conjunct or an appositional modifier, but not both.

Algorithm 3.4 : *getCoordinationHead*(C, R)

Input: A constituent C and the headrule R of C.
Output: The head of the leftmost conjunct in C if exists; otherwise, `null`.

```
1:  if not containsCoordination(C) return null
2:  p ← getConjunctHeadPattern(C)
3:  pHead ← null                                          # previous conjunct head
4:  isPatternFound ← False
5:  let f be the first child of C
6:  for c in all children of C do
7:    if isCoordinatingConjunction(c) or (c is ,|:) then  # Section 4.10.2
8:      if isPatternFound then
9:        let S be a sub-span of C from f to c (exclusive)
10:       pHead ← getConjunctHead(S, R, pHead)
11:       c.head ← pHead
12:       c.label ← getDependencyLabel(c, C, pHead)       # Section 4.3
13:       isPatternfound ← False
14:       let f be the next sibling of c in C
15:     elif pHead ≠ null then
16:       let S be a sub-span of C from f to c (inclusive)
17:       for s in S do
18:         s.head ← pHead
19:         s.label ← getDependencyLabel(s, C, pHead)     # Section 4.3
20:       let f be the next sibling of c in C
21:   elif isConjunctHead(c, C, p) then isPatternFound ← True   # a conjunct is found
22: if pHead = null return null                           # no conjunct is found
23: let S be a sub-span of C from f to c (inclusive)
24: if S is not empty then getConjunctHead(S, R, pHead)
25: return the head of the leftmost conjunct
```

The *containsCoordination*(C) method in Algorithm 3.5 decides whether a constituent C is coordinated. C is coordinated if it is an unlike coordinated phrase (line 1), a noun phrase containing a constituent with the function tag `ETC` as the rightmost child (line 2-4), or contains a conjunction followed by a conjunct (lines 5-9). The *getConjunctHeadPattern*(C) method in Algorithm 3.6 returns a pattern that matches potential conjunct heads of C. In theory, a verb phrase should contain at least one non-auxiliary verb or a verb phrase that

matches the pattern (VP|VB$^\flat$ in line 9); however, this is not always true in practice (e.g., VP-ellipsis, randomly omitted verbs in web-texts). Moreover, phrases such as unlike coordinated phrases, quantifier phrases, or fragments do not always show clear conjunct head patterns. The default pattern of * is used for these cases, indicating that any constituent can be the potential head of a conjunct in these phrases.

Algorithm 3.5 : *containsCoordination*(C)

Input: Constituent C.
Output: `True` if C contains coordination; otherwise, `False`.

1: **if** C is `UCP` **return** `True` # unlike coordinated phrase
2: **if** (C is `NML|NP`) and (C contains `-ETC`) **then** # *et cetera* (*etc.*)
3: **let** e **be** a child of N with `-ETC`
4: **if** e is the rightmost element besides punctuation **return** `True`
5: **for** f **in** all children of C **do** # skip pre-conjunctions
6: **if** not (*isCoordinatingConjunction*(f) or *isPunctuation*(f)) **then** # App. 4.10.2, 4.14.8
7: **break**
8: **let** N **be** all children of C preceded by f
9: **return** N contains `CC|CONJP`

Algorithm 3.6 : *getConjunctHeadPattern*(C)

Input: A constituent C.
Output: The conjunct head pattern of C if exists; otherwise, the default pattern, *.
If C contains no child satisfying the pattern, returns the default pattern, *.
VB$^\flat$ implies a non-auxiliary verb (Section 4.6).
S$^\flat$ implies a clause without an adverbial function tag (Section 4.9).

1: **if** C is `ADJP` **then** $p \leftarrow$ `ADJP|JJ*|VBN|VBG`
2: **elif** C is `ADVP` **then** $p \leftarrow$ `ADVP|RB*`
3: **elif** C is `INTJ` **then** $p \leftarrow$ `INTJ|UH`
4: **elif** C is `PP` **then** $p \leftarrow$ `PP|IN|VBG`
5: **elif** C is `PRT` **then** $p \leftarrow$ `PRT|RP`
6: **elif** C is `NML|NP` **then** $p \leftarrow$ `NP|NML|NN*|PRP|-NOM`
7: **elif** C is `NAC` **then** $p \leftarrow$ `NP`
8: **elif** C is `NX` **then** $p \leftarrow$ `NX`
9: **elif** C is `VP` **then** $p \leftarrow$ `VP|VB`$^\flat$
10: **elif** C is `S` **then** $p \leftarrow$ `S`$^\flat$`|SINV|SQ|SBARQ`
11: **elif** C is `SQ` **then** $p \leftarrow$ `S`$^\flat$`|SQ|SBARQ`
12: **elif** C is `SINV` **then** $p \leftarrow$ `S`$^\flat$`|SINV`
13: **elif** C is `SBAR*` **then** $p \leftarrow$ `SBAR*`
14: **elif** C is `WHNP` **then** $p \leftarrow$ `NN*|WP`
15: **elif** C is `WHADJP` **then** $p \leftarrow$ `JJ*|VBN|VBG`
16: **elif** C is `WHADVP` **then** $p \leftarrow$ `RB*|WRB|IN`
17: **if** (p is not found) or (C contains no p) **return** *
18: **return** p

A pattern p retrieved by the *getConjunctHeadPattern*(C) method in Algorithm 3.6 is used in the *isConjunctHead*(C, P, p) method in Algorithm 3.7 to decide whether a constituent C is a potential conjunct head of its parent P. No subordinating conjunction is considered a conjunct head in a subordinate clause (line 1); this rule is added to prevent a complementizer such as *whether* from being the head of a clause starting with expressions like *whether or not*. When the default pattern is used, the method accepts any constituent except for a few special cases (lines 3-7). The method returns `True` if C matches p (line 9).

Algorithm 3.7 : *isConjunctHead*(C, P, p)

Input: Constituents C and P, where P is the parent of C, and the conjunct head pattern p of P.
Output: `True` if C matches the conjunct head pattern; otherwise, `False`.

```
1:  if (P is SBAR) and (C is ID|DT) return False   # Section 4.9.3
2:  if p is * then                                 # the default pattern
3:    if isPunctuation(C) return False             # Section 4.14.8
4:    if isInterjection(C) return False            # Section 4.14.3
5:    if isMetaModifier(C) return False            # Section 4.14.4
6:    if isParentheticalModifier(C) return False   # Section 4.14.5
7:    if isAdverbialModifier(C) return False       # Section 4.9.2
8:    return True
9:  if C is p return True
10: return False
```

Finally, the *getConjunctHead*$(S, R, pHead)$ method in Algorithm 3.8 finds the head of a conjunct S and makes this head a dependent of its preceding conjunct head, $pHead$. The head of S is found by the *getHead*(N, R) method in Algorithm 3.1 where R is the headrule of S's parent. The dependency label `CONJ` is assigned to this head except for the special cases of interjections and punctuation (lines 4-6).

Algorithm 3.8 The *getConjunctHead*$(S, R,$ *pHead*$)$ method.

Input: A constituent C, a sub-span S of C, the headrule R of C, and the previous conjunct head $pHead$ in C.
Output: The head of S. All other nodes in S become dependents of the head.

```
1: cHead ← getHead(S, R)                                  # Section 3.1
2: if pHead ≠ null then
3:   cHead.head ← pHead
4:   if   isInterjection(C) then cHead.label ← INTJ      # Section 4.14.3
5:   elif isPunctuation(C) then cHead.label ← PUNCT      # Section 4.14.8
6:   else cHead.label ← CONJ                             # Section 4.10.1
7: return cHead
```

3.4 Small clauses

Small clauses are represented as declarative clauses without verb phrases in constituent trees. Small clauses may not contain internal subjects. In Figure 20, both `S`-1 and `S`-2 are small clauses but `S`-1 contains an internal subject, *me*, whereas the subject of `S`-2 is controlled externally. This distinction is made because `S`-1 can be rewritten as a subordinate clause such as "*I am her friend*" whereas such a transformation is not possible for `S`-2. In other words, *me her friend* as a whole is an argument of *considers* whereas *me* and *her friend* are separate arguments of *calls*.

Figure 21 shows dependency trees converted from the trees in Figure 20. A small clause with an internal subject is considered a clausal complement (`CCOMP`; the left tree in Figure 20) whereas one without an internal subject is considered an object predicate (`OPRD`; the right tree in Figure 20), implying that it is a non-`VP` predicate of the object. This way, although *me* has no direct dependency to *friend*, their relation can be inferred through this label. Note that the CoNLL dependency approach uses the object predicate for both kinds of small clauses such that *me* and *her friend* become separate dependents of *considers*, as they are for *calls*. This analysis is not taken in our approach because we want our dependency trees to be consistent

with the original constituent trees. Preserving the original structure makes it easier to integrate additional information to the converted dependency trees that has been already annotated on top of these constituent trees (e.g., semantic roles in PropBank).

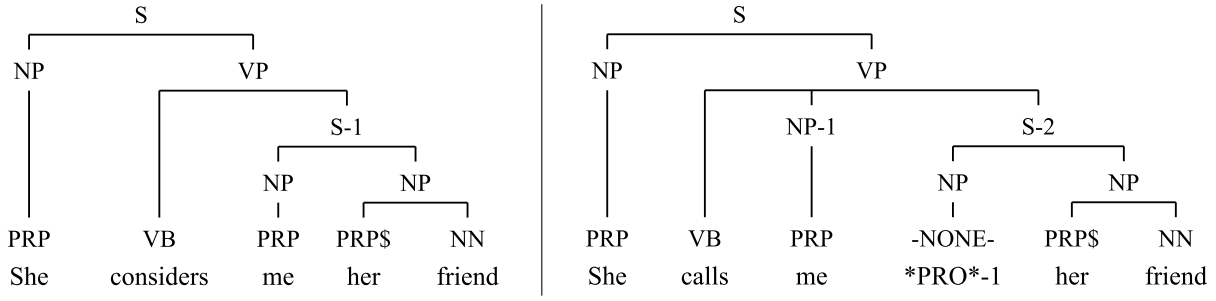

Figure 20: Examples of small clauses with internal (left) and external (right) subjects.

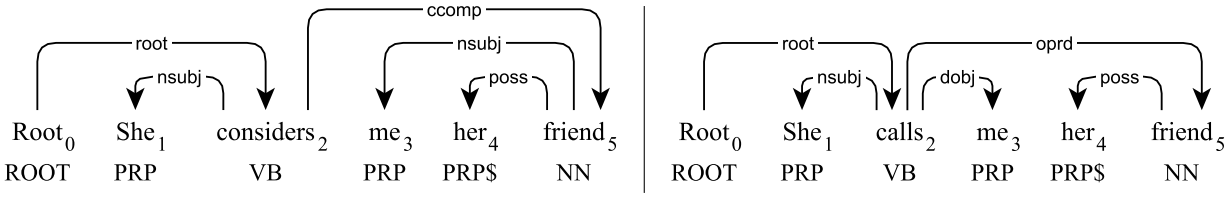

Figure 21: Dependency trees converted from the constituent trees in Figure 20.

For passive constructions, `OPRD` is applied to both kinds of small clauses because a dependency between the object and the non-`VP` predicate is lost by the `NP` movement. In Figure 22, *I* is moved from the object position to the subject position of *considered* (`NSUBJPASS`; Section 4.4.6); thus, it is no longer a dependent of *friend*. The dependency between *I* and *friend* can be inferred through `OPRD` without adding more structural complexity to the tree.

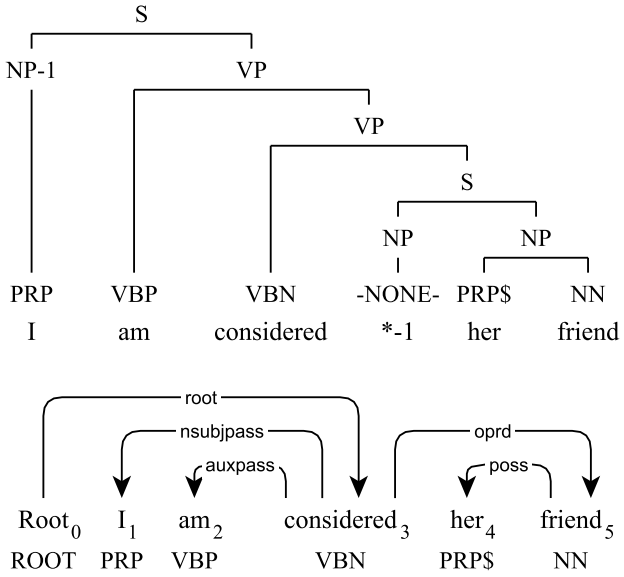

Figure 22: An example of a small clause in a passive construction.

3.5 Hyphenation

Recent Treebanks tokenize certain hyphenated words.[7] In Figure 23, a noun phrase "*The Zhuhai-Hong Kong-Macao bridge*" is tokenized to "*The Zhuhai - Hong Kong - Macao bridge*". In our dependency approach, these hyphenated words are assigned special dependency labels, `HMOD` (modifier in hyphenation) and `HYPH` (hyphen), which are borrowed from the CoNLL dependency approach. In Figure 24, $-_3$ and $-_6$ become dependents of *Kong* and *Macao* respectively with the dependency label, `HYPH`. Similarly, *Zhuhai* and *Kong* become dependents of *Kong* and *Macao* respectively with the dependency label, `HMOD`.

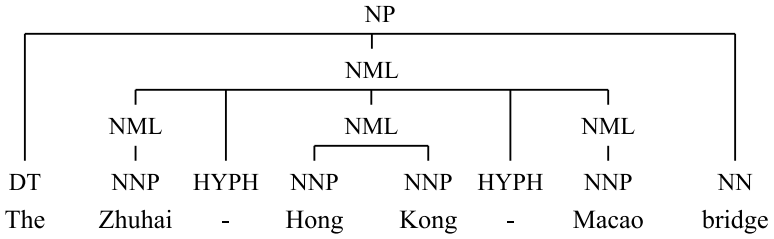

Figure 23: Examples of hyphenated words.

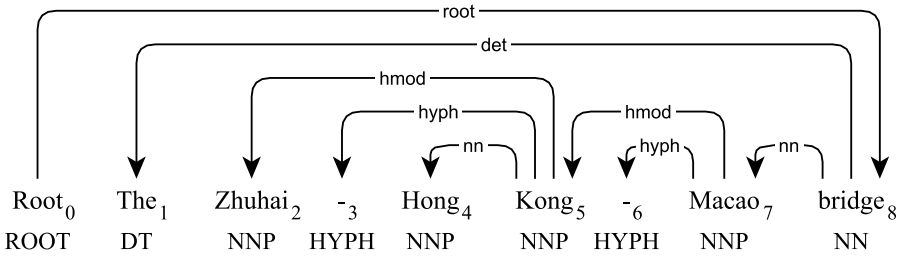

Figure 24: A dependency tree converted from the constituent tree in Figure 23.

The *findHyphenation*(C) method in Algorithm 3.9 finds dependencies in hyphenations and returns `True` if such dependencies are found; otherwise, returns `False`.

Algorithm 3.9 : *findHyphenation*(C)

Input: A constituent C whose POS tag is `VP`.
Output: `True` if C contains hyphens; otherwise, `False`.

1: $b \leftarrow$ `False`
2: $i \leftarrow 0$
3: **while** $i + 2 <$ the total number of C's children **do**
4: $c_i \leftarrow i$'th child of C
5: $c_{i+1} \leftarrow (i+1)$'th child of C
6: $c_{i+2} \leftarrow (i+2)$'th child of C
7: **if** c_{i+1} is `HYPH` **then**
8: c_i.head $\leftarrow c_{i+2}$; c_i.label $\leftarrow$ `HMOD`
9: c_{i+1}.head $\leftarrow c_{i+2}$; c_{i+1}.label $\leftarrow$ `HYPH`
10: $b \leftarrow$ `True`
11: $i \leftarrow i + 1$
12: **return** b

[7] See Section 1.1 for more details about the format changes in recent Treebanks.

4 Assigning dependency labels

4.1 CLEAR dependency labels

Table 3 shows a list of dependency labels, called the CLEAR dependency labels, generated by our dependency conversion. These labels are mostly inspired by the Stanford dependency approach, partially borrowed from the CoNLL dependency approach, and newly introduced by the CLEAR dependency approach to minimize unclassified dependencies. The following subsections show detailed descriptions of the CLEAR dependency labels. Section 4.2 shows a comparison between the CLEAR and the Stanford dependencies.

Label	Description	Label	Description
ACOMP	Adjectival complement	META**	Meta modifier
ADVCL	Adverbial clause modifier	NEG	Negation modifier
ADVMOD	Adverbial modifier	NMOD*	Modifier of nominal
AGENT	Agent	NN	Noun compound modifier
AMOD	Adjectival modifier	NPADVMOD	Noun phrase as ADVMOD
APPOS	Appositional modifier	NSUBJ	Nominal subject
ATTR	Attribute	NSUBJPASS	Nominal subject (passive)
AUX	Auxiliary	NUM	Numeric modifier
AUXPASS	Auxiliary (passive)	NUMBER	Number compound modifier
CC	Coordinating conjunction	OPRD*	Object predicate
CCOMP	Clausal complement	PARATAXIS	Parataxis
COMPLM	Complementizer	PARTMOD	Participial modifier
CONJ	Conjunct	PCOMP	Complement of a preposition
CSUBJ	Clausal subject	POBJ	Object of a preposition
CSUBJPASS	Clausal subject (passive)	POSS	Possession modifier
DEP	Unclassified dependent	POSSESSIVE	Possessive modifier
DET	Determiner	PRECONJ	Pre-correlative conjunction
DOBJ	Direct object	PREDET	Predeterminer
EXPL	Expletive	PREP	Prepositional modifier
HMOD*	Modifier in hyphenation	PRT	Particle
HYPH*	Hyphen	PUNCT	Punctuation
INFMOD	Infinitival modifier	QUANTMOD	Quantifier phrase modifier
INTJ**	Interjection	RCMOD	Relative clause modifier
IOBJ	Indirect object	ROOT	Root
MARK	Marker	XCOMP	Open clausal complement

Table 3: A list of the CLEAR dependency labels. Labels followed by * are borrowed from the CoNLL dependency approach. Labels followed by ** are newly introduced by the CLEAR dependency approach. HMOD and HYPH labels are added later.

4.2 Comparison to the Stanford dependency approach

Treating dependency trees generated by the Stanford dependency approach as gold-standard, the CLEAR dependency approach shows a labeled attachment score of 95.39%, an unlabeled attachment score of 90.39%, and a label accuracy of 93.01%. For comparison, the OntoNotes Treebank is used, which consists of various corpora in multiple genres. Out of 138K dependency trees generated by our conversion, 3.69% of them contain at least one non-projective dependency. Out of 2.6M dependencies, 3.62% are unclassified by the Stanford converter whereas 0.23% are unclassified by our approach, that is a 93.65% proportional reduction in error. A dependency is considered unclassified if it is assigned with the label, DEP (Section 4.14.2). Table 5 shows a list of the top 40 dependency labels generated by our approach that are unclassified by the Stanford dependency approach.[8]

[8]The following options are used for the Stanford dependency conversion, which is the same setup that was used for the SANCL'12 shared task (Petrov and McDonald, 2012): `-basic -conllx -keepPunct -makeCopulaHead`.

CLEAR	Count	Stanford
ACOMP	20,325	ACOMP(98.19)
ADVCL	33,768	ADVCL(53.43), XCOMP(19.79), DEP(11.33), CCOMP(6.67), PARTMOD(6.04)
ADVMOD	101,134	ADVMOD(96.38)
AGENT	4,756	PREP(99.62)
AMOD	131,971	AMOD(97.93)
APPOS	17,869	APPOS(54.80), DEP(40.56)
ATTR	22,597	ATTR(81.87), NSUBJ(15.41)
AUX	106,428	AUX(99.98)
AUXPASS	19,289	AUXPASS(99.99)
CC	68,522	CC(99.26)
CCOMP	42,354	CCOMP(78.50), DEP(12.16), XCOMP(6.73)
COMPLM	13,130	COMPLM(94.94)
CONJ	61,270	CONJ(97.42)
CSUBJ	1,766	CSUBJ(92.19), DEP(5.32)
CSUBJPASS	72	CSUBJPASS(91.67), DEP(6.94)
DEP	4,046	DEP(90.06), NSUBJ(5.98)
DET	214,488	DET(99.82)
DOBJ	112,856	DOBJ(98.90)
EXPL	4,373	EXPL(99.20)
INFMOD	5,697	INFMOD(98.05)
INTJ	10,947	DEP(99.44)
IOBJ	2,615	IOBJ(86.16), DOBJ(10.48)
MARK	21,235	MARK(82.07), DEP(12.18), COMPLM(5.66)
META	5,620	DEP(99.00)
NEG	18,585	NEG(95.71)
NMOD	923	DEP(68.47), AMOD(30.23)
NN	149,201	NN(99.51)
NPADVMOD	21,267	TMOD(41.11), DEP(24.77), NPADVMOD(14.70), DOBJ(8.08), NSUBJ(5.01)
NSUBJ	208,934	NSUBJ(99.52)
NSUBJPASS	16,994	NSUBJPASS(99.82)
NUM	30,412	NUM(99.91)
NUMBER	3,456	NUMBER(98.96)
OPRD	2,855	DEP(49.91), ACOMP(26.90), XCOMP(22.42)
PARATAXIS	3,662	PARATAXIS(77.01), DEP(22.23)
PARTMOD	9,945	PARTMOD(94.17), DEP(5.39)
PCOMP	12,702	PCOMP(88.99), POBJ(7.98)
POBJ	222,115	POBJ(99.89)
POSS	45,156	POSS(99.91)
POSSESSIVE	16,608	POSSESSIVE(99.99)
PRECONJ	574	PRECONJ(76.83), DEP(20.56)
PREDET	2,409	PREDET(94.65), DEP(4.61)
PREP	231,742	PREP(97.52)
PRT	10,149	PRT(96.21), DEP(3.79)
PUNCT	280,452	PUNCT(93.39), DEP(6.56)
QUANTMOD	3,467	QUANTMOD(83.50), DEP(14.94)
RCMOD	22,781	RCMOD(96.28), DEP(3.09)
ROOT	132,225	ROOT(99.98)
XCOMP	25,909	XCOMP(89.61), CCOMP(7.13), DEP(3.17)

Table 4: Mappings between the CLEAR and the Stanford dependency labels. The CLEAR column show the CLEAR dependency labels. The Count column shows the count of each label. The Stanford column shows labels generated by the Stanford converter in place of the CLEAR dependency label with probabilities (in %); labels with less than 3% occurrences are discarded.

PUNCT	23.98	MARK	3.37	AMOD	0.83	PCOMP	0.49	PRECONJ	0.15
INTJ	14.18	PREP	1.97	NMOD	0.82	COMPLM	0.47	PREDET	0.14
APPOS	9.44	OPRD	1.86	NSUBJ	0.72	ACOMP	0.43	CSUBJ	0.12
META	7.25	ADVMOD	1.56	PARTMOD	0.70	NEG	0.39	INFMOD	0.12
NPADVMOD	6.86	XCOMP	1.07	NN	0.69	POBJ	0.29	IOBJ	0.09
CCOMP	6.71	PARATAXIS	1.06	QUANTMOD	0.67	DET	0.22	POSS	0.02
ADVCL	4.98	CONJ	1.06	CC	0.64	DOBJ	0.22	NUM	0.01
DEP	4.75	RCMOD	0.92	PRT	0.50	ATTR	0.18	AGENT	0.01

Table 5: A list of the CLEAR dependency labels that are unclassified by the Stanford dependency approach. The first column shows the unclassified CLEAR dependency labels and the second column shows their proportions to unclassified dependencies in the Stanford dependency approach (in %).

Table 4 shows mappings between the CLEAR and the Stanford dependency labels. Some labels in the Stanford dependency approach are not used in our conversion. For instance, multi-word expressions (`MWE`) are not used in our approach because it is not clear how to identify multi-word expressions systematically. Furthermore, purpose clause modifiers (`PURPCL`) and temporal modifiers (`TMOD`) are not included as dependencies but added as separate features of individual nodes in our dependency trees (see Appendix A.3 for more details about these additional features).

4.3 Dependency label heuristics

The *getDependencyLabel*(C, P, p) in Algorithm 4.2 assigns a dependency label to a constituent C by using function tags and inferring constituent relations between C, P, and p, where P is the parent of C and p is the head constituent of P. Heuristics described in this algorithm are derived from careful analysis of several constituent Treebanks (Marcus et al., 1993; Nielsen et al., 2010; Weischedel et al., 2011; Verspoor et al., 2012) and manually evaluated case-by-case. All supplementary methods are described in the following subsections. Algorithms followed by * (e.g., *setPassiveSubject*(D, H)* in Algorithm 4.4) are called after the *getDependencyLabel*(C, P, p) method and applied to all dependency nodes.

The *getSimpleLabel*(C) method in Algorithm 4.1 returns the dependency label of a constituent C if it can be inferred from the POS tag of C; otherwise, `null`.

Algorithm 4.1 : *getSimpleLabel*(C)

Input: A constituent C.
Output: The dependency label of C if it can be inferred from the POS tag of C; otherwise, `null`.

1: **let** d **be** the head dependent of C
2: **if** C is `HYPH` **return** `HYPH` # Section 4.7.2
3: **if** C is `ADJP|WHADJP|JJ*` **return** `AMOD` # Section 4.14.1
4: **if** C is `PP|WHPP` **return** `PREP` # Section 4.12.3
5: **if** C is `PRT|RP` **return** `PRT` # Section 4.14.7
6: **if** *isPreCorrelativeConjunction*(C) **return** `PRECONJ` # Section 4.10.3
7: **if** *isCoordinatingConjunction*(C) **return** `CC` # Section 4.10.2
8: **if** *isParentheticalModifier*(C) **return** `PARATAXIS` # Section 4.14.5
9: **if** *isPunctuation*$(C|d)$ **return** `PUNCT` # Section 4.14.8
10: **if** *isInterjection*$(C|d)$ **return** `INTJ` # Section 4.14.3
11: **if** *isMetaModifier*(C) **return** `META` # Section 4.14.4
12: **if** *isAdverbialModifier*(C) **return** `ADVMOD` # Section 4.9.2
13: **return** `null`

Algorithm 4.2 : *getDependencyLabel*(C, P, p)

Input: Constituents C, P, and p.
P is the parent of C, and p is the head constituent of P.
Output: The dependency label of C with respect to p in P.

let c **be** the head constituent of C
let d **be** the head dependent of C
if *hasAdverbialTag*(C) **then** # Section 4.9
  if C is `S|SBAR|SINV` **return** `ADVCL`
  if C is `NML|NP|QP` **return** `NPADVMOD`
if (label $\leftarrow$ *getSubjectLabel*(C)) $\neq$ `null` **return** label # Section 4.4
if C is `UCP` **then**
  c.add(all function tags of C)
  return *getDependencyLabel*(c, P, p)
if P is `VP|SINV|SQ` **then**
  if C is `ADJP` **return** `ACOMP`
  if (label $\leftarrow$ *getObjectLabel*(C)) $\neq$ `null` **return** label # Section 4.5
  if *isObjectPredicate*(C) **return** `OPRD` # Section 4.5.4
  if *isOpenClausalComplement*(C) **return** `XCOMP` # Section 4.8.3
  if *isClausalComplement*(C) **return** `CCOMP` # Section 4.8.2
  if (label $\leftarrow$ *getAuxiliaryLabel*(C)) $\neq$ `null` **return** label # Section 4.6
if P is `ADJP|ADVP` **then**
  if *isOpenClausalComplement*(C) **return** `XCOMP` # Section 4.8.3
  if *isClausalComplement*(C) **return** `CCOMP` # Section 4.8.2
if P is `NML|NP|WHNP` **then**
  if (label $\leftarrow$ *getNonFiniteModifierLabel*(C)) $\neq$ `null` **return** label # Section 4.11
  if *isRelativeClauseModifier*(C) **return** `RCMOD` # Section 4.11.10
  if *isClausalComplement*(C) **return** `CCOMP` # Section 4.8.2
if *isPossessionModifier*(C, P) **return** `POSS` # Section 4.14.6
if (label $\leftarrow$ *getSimpleLabel*(C)) $\neq$ `null` **return** label # Section 4.3
if P is `PP|WHPP` **return** *getPrepositionModifierLabel*(C, d) # Section 4.12
if (C is `SBAR`) or *isOpenClausalComplement*(C) **return** `ADVCL` # Section 4.8.3
if (P is `PP`) and (C is `S*`) **return** `ADVCL`
if C is `S|SBARQ|SINV|SQ` **return** `CCOMP`
if P is `QP` **return** (C is `CD`) ? `NUMBER` : `QUANTMOD`
if (P is `NML|NP|NX|WHNP`) or (p is `NN*|PRP|WP`) **then**
  return *getNounModifierLabel*(C) # Section 4.11
if (label $\leftarrow$ *getSimpleLabel*(c)) $\neq$ `null` **return** label # Section 4.3
if d is `IN` **return** `PREP`
if d is `RB*` **return** `ADVMOD`
if (P is `ADJP|ADVP|PP`) or (p is `JJ*|RB*`) **then**
  if C is `NML|NP|QP|NN*|PRP|WP` **return** `NPADVMOD`
  return `ADVMOD`
return `DEP`

4.4 Arguments: subject related

Subject-related labels consist of agents (AGENT), clausal subjects (CSUBJ), clausal passive subjects (CSUBJPASS), expletives (EXPL), nominal subjects (NSUBJ), and nominal passive subjects (NSUBJPASS).

Algorithm 4.3 : *getSubjectLabel*(C, d)

Input: Constituents C and d, where d is the head dependent of C.
Output: CSUBJ, NSUBJ, EXPL, or AGENT if C is a subject-related argument; otherwise, null.

```
1: if C has SBJ then
2:   if C is S* return CSUBJ      # Section 4.4.2
3:   if d is EX return EXPL       # Section 4.4.4
4:   return NSUBJ                 # Section 4.4.5
5: if C has LGS return AGENT      # Section 4.4.1
6: return null
```

Algorithm 4.4 : *setPassiveSubject*(D, H)*

Input: Dependents D and H, where H is the head of D.
Output: If D is a passive subject, append PASS to its label.

```
1: if H contains AUXPASS then
2:   if   D is CSUBJ then D.label ← CSUBJPASS   # Section 4.4.3
3:   elif D is NSUBJ then D.label ← NSUBJPASS   # Section 4.4.6
```

4.4.1 AGENT: agent

An agent is the complement of a passive verb that is the surface subject of its active form. In our approach, the preposition *by* is included as a part of AGENT.

(1) The car was bought [by John] AGENT(bought, by), POBJ(by, John)
(2) The car bought [by John] is red AGENT(bought, by), POBJ(by, John)

4.4.2 CSUBJ: clausal subject

A clausal subject is a clause in the subject position of an active verb. A clause with a SBJ function tag is considered a CSUBJ.

(1) [Whether she liked me] doesn't matter CSUBJ(matter, liked)
(2) [What I said] was true CSUBJ(was, said)
(3) [Who I liked] was you CCOMP(was, liked), NSUBJ(was, you)

In (3), *Who I liked* is topicalized such that it is considered a clausal complement (CCOMP) of *was*; *you* is considered a nominal subject (NSUBJ) of *was*.

4.4.3 CSUBJPASS: clausal passive subject

A clausal passive subject is a clause in the subject position of a passive verb. A clause with the SBJ function tag that depends on a passive verb is considered a CSUBJPASS.

(1) [Whoever misbehaves] will be dismissed CSUBJPASS(dismissed, misbehaves)

4.4.4 EXPL: expletive

An expletive is an existential *there* in the subject position.

(1) There was an explosion EXPL(was, There)

4.4.5 NSUBJ: nominal subject

A nominal subject is a non-clausal constituent in the subject position of an active verb. A non-clausal constituent with the SBJ function tag is considered a NSUBJ.

(1) [She and I] came home together NSUBJ(came, She)
(2) [Earlier] was better NSUBJ(was, Earlier)

4.4.6 NSUBJPASS: nominal passive subject

A nominal passive subject is a non-clausal constituent in the subject position of a passive verb. A non-clausal constituent with the SBJ function tag that depends on a passive verb is considered a NSUBJPASS.

(1) I [am] drawn to her NSUBJPASS(drawn, I)
(2) We will [get] married NSUBJPASS(married, We)
(3) She will [become] nationalized NSUBJPASS(nationalized, She)

4.5 Arguments: object related

Object-related labels consist of attributes (ATTR), direct objects (DOBJ), indirect objects (IOBJ), and object predicates (OPRD).

Algorithm 4.5 : *getObjectLabel*(C)

Input: A constituent C whose parent is VP|SINV|SQ.
Output: DOBJ or ATTR if C is in an object or an attribute; otherwise, null.

```
1: if C is NP|NML then
2:   if C has PRD return ATTR   # Section 4.5.1
3:   return DOBJ                # Section 4.5.2
4: return null
```

4.5.1 ATTR: attribute

An attribute is a noun phrase that is a non-VP predicate usually following a copula verb.

(1) This product is [a global brand] ATTR(is, brand)
(2) This area became [a prohibited zone] ATTR(became, zone)

4.5.2 DOBJ: direct object

A direct object is a noun phrase that is the accusative object of a (di)transitive verb.

(1) She bought me [these books] DOBJ(bought, books)
(2) She bought [these books] for me DOBJ(bought, books)

4.5.3 IOBJ: indirect object

An indirect object is a noun phrase that is the dative object of a ditransitive verb.

(1)	She bought [me] these books	IOBJ(bought, me)
(2)	She bought these books [for me]	PREP(bought, for)
(3)	[What] she bought [me] were these books	DOBJ(bought, What), IOBJ(bought, me)
(4)	I read [them] [one by one]	DOBJ(read, them), NPADVMOD(read, one)

In (2), *for me* is considered a prepositional modifier although it is the dative object in an unshifted form. This information is preserved with a function tag DTV as additional information in our representation (Section 6.2). In (3), *What* and *me* are considered direct and indirect objects of *bought*, respectively. In (4), the noun phrase *one by one* is not considered an IOBJ, but an adverbial noun phrase modifier (NPADVMOD) because it carries an adverbial function tag, MNR. This kind of information is also preserved with semantic function tags in our representation (Section 6.1).

Algorithm 4.6 : *setIndirectObject*$(C)^*$

Input: A dependent D.
Output: If D is an indirect object, set its label to IOBJ.

1: **if** (D is DOBJ) and (D is followed by another DOBJ) **then** D.label $\leftarrow$ IOBJ

4.5.4 OPRD: object predicate

An object predicate is a non-VP predicate in a small clause that functions like the predicate of an object. Section 3.4 describes how object predicates are derived.

(1)	She calls [me] [her friend]	DOBJ(calls, me), OPRD(calls, friend)
(2)	She considers [[me] her friend]	CCOMP(considers, friend), NSUBJ(me, friend)
(3)	I am considered [her friend]	OPRD(considered, friend)
(4)	I persuaded [her] [to come]	DOBJ(persuaded, her), XCOMP(persuaded, come)

In (2), the small clause *me her friend* is considered a clausal complement (CCOMP) because we treat *me* as the subject of the non-VP predicate, *her friend*. In (4), the open clause *to come* does indeed predicate over *her* but is not labeled as an OPRD but rather an open clausal complement (XCOMP). This is because the dependency between *her* and *come* is already shown in our representation as an open clausal subject (XSUBJ) whereas such information is not available for the non-VP predicates in (1) and (3); thus, without labeling them as object predicates, it can be difficult to infer the relation between the objects and object predicates.

Algorithm 4.7 : *isObjectPredicate*(C)

Input: A constituent C.
Output: True if C is an object predicate; otherwise, False.

1: **if** (C is S) and (C contains no VP) and (C contains both SBJ and PRD) **then**
2: **if** the subject of C is an empty category **return** True
3: **return** False

4.6 Auxiliaries

Auxiliary labels consist of auxiliaries (`AUX`) and passive auxiliaries (`AUXPASS`). The *getAuxiliaryLabel*(C) method in Algorithm 4.8 shows how these auxiliary labels are distinguished. Note that a passive auxiliary is supposed to modify only a past participle (`VBN`), which is sometimes annotated as a past tense verb (`VBD`). The condition in lines 5 and 8 resolves such an erroneous case. Lines 6-7 are added to handle the case of coordination where vp_1 is just an umbrella constituent that groups `VP` conjuncts together.

Algorithm 4.8 : *getAuxiliaryLabel*(C)

Input: A constituent C whose parent is `VP|SINV|SQ`.
Output: `AUX` or `AUXPASS` if C is an auxiliary or a passive auxiliary; otherwise, `null`.

```
1:  if C is MD|TO return AUX                                  # Section 4.6.1
2:  if (C is VB*) and (C contains VP) then
3:    if C is be|become|get then
4:      let vp1 be the first VP in C
5:      if vp1 contains VBN|VBD return AUXPASS                  # Section 4.6.2
6:      if (vp1 contains no VB*) and (vp1 contains VP) then     # for coordination
7:        let vp2 be the first VP in vp1
8:        if vp2 contains VBN|VBD return AUXPASS
9:    return AUX
10: return null
```

4.6.1 `AUX`: auxiliary

An auxiliary is an auxiliary or modal verb that gives further information about the main verb (e.g., tense, aspect). The preposition *to*, used for infinitive, is also considered an `AUX`. Auxiliary verbs for passive verbs are assigned with a separate dependency label `AUXPASS` (Section 4.6.2).

(1)	I [have] [been] seeing her	`AUX`(seeing, have), `AUX`(seeing, been)
(2)	I [will] meet her tomorrow	`AUX`(meet, will)
(3)	I [am] [going] [to] meet her tomorrow	`AUX`(meet, am), `AUX`(meet, going), `AUX`(meet, to)

4.6.2 `AUXPASS`: passive auxiliary

A passive auxiliary is an auxiliary verb, *be*, *become*, or *get*, that modifies a passive verb.

(1)	I [am] drawn to her	`AUXPASS`(drawn, am)
(2)	We will [get] married	`AUXPASS`(married, get)
(3)	She will [become] nationalized	`AUXPASS`(nationalized, become)

4.7 Hyphenation

4.7.1 `HMOD`: modifier in hyphenation

A modifier in hyphenation is a constituent preceding a hyphen, which modifies a constituent following the hyphen (see the example in Section 4.7.2).

4.7.2 `HYPH`: hyphen

A hyphen modifies a constituent following the hyphen.

(1)	New - York Times	`HMOD`(York, New), `HYPH`(York, -)

4.8 Complements

Complement labels consists of adjectival complements (ACOMP), clausal complements (CCOMP), and open clausal complements (XCOMP). Additionally, complementizers (COMPLM) are included to indicate the beginnings of clausal complements.

4.8.1 ACOMP: adjectival complement

An adjectival complement is an adjective phrase that modifies the head of a VP|SINV|SQ, that is usually a verb.

(1)	She looks [so beautiful]	ACOMP(looks, beautiful)
(2)	Please make [sure to invite her]	ACOMP(make, sure)
(3)	Are you [worried]	ACOMP(Are, worried)
(4)	[Most important] is your heart	ACOMP(is, important), NSUBJ(is, heart)

In (4), *Most important* is topicalized such that it is considered an ACOMP of *is* although it is in the subject position; *your heart* is considered a nominal subject (NSUBJ) of *is*.

4.8.2 CCOMP: clausal complement

A clausal complement is a clause with an internal subject that modifies the head of an ADJP|ADVP|NML|NP|WHNP|VP|SINV|SQ. For NML|NP|WHNP, a clause is considered a CCOMP if it is neither a infinitival modifier (Section 4.11.4), a participial modifier (Section 4.11.7), nor a relative clause modifier (Section 4.11.10).

(1)	She said [(that) she wanted to go]	CCOMP(said, wanted)
(2)	I am not sure [what she wanted]	CCOMP(sure, wanted)
(3)	She left no matter [how I felt]	CCOMP(matter, felt)
(4)	I don't know [where she is]	CCOMP(know, is)
(5)	She asked [should we meet again]	CCOMP(asked, meet)
(6)	I asked [why did you leave]	CCOMP(asked, leave)
(7)	I said [may God bless you]	CCOMP(said, bless)
(8)	The fact [(that) she came back] made me happy	CCOMP(fact, came)

In (4), *where she is* is considered a CCOMP although it carries arbitrary locative information. Clauses such as polar questions (5), *wh*-questions (6), or inverted declarative sentences (7) are also considered CCOMP. A clause with an adverbial function tag is not considered a CCOMP, but an adverbial clause modifier (Section 4.9.1).

Algorithm 4.9 : *isClausalComplement*(C)

Input: A constituent C whose parent is ADJP|ADVP|NML|NP|WHNP|VP|SINV|SQ.
Output: True if C is a clausal complement; otherwise, False.

```
1: if C is S|SQ|SINV|SBARQ return True
2: if C is SBAR then
3:   if C contains a wh-complementizer return True
4:   if C contains a null complementizer, 0 return True
5:   if C contains a complementizer, if, that, or whether then
6:     set the dependency label of the complementizer to COMPLM   # Section 4.8.4
7:     return True
8: return False
```

4.8.3 XCOMP: open clausal complement

An open clausal complement is a clause without an internal subject that modifies the head of an ADJP|ADVP|VP|SINV|SQ.

(1)	I want [to go]	XCOMP(want, go)
(2)	I am ready [to go]	XCOMP(ready, go)
(3)	It is too soon [to go]	XCOMP(soon, go)
(4)	He knows [how to go]	XCOMP(knows, go)
(5)	What do you think [happend]	XCOMP(think, happened)
(6)	He forced [me] [to go]	DOBJ(forced, me), XCOMP(forced, go)
(7)	He expected [[me] to go]	CCOMP(expected, go), NSUBJ(me, go)

In (7), *me to go* is not considered an XCOMP but a clausal complement (CCOMP) because *me* is considered a nominal subject (NSUBJ) of *go* (see Section 5.4 for more examples of open clauses).

Algorithm 4.10 : *isOpenClausalComplement*(C)

Input: A constituent C whose parent is ADJP|ADVP|VP.
Output: True if C is an open clausal complement; otherwise, False.

1: **if** C is S **then**
2: **return** (C contains VP) and (the subject of C is an empty category)
3: **if** (C is SBAR) and (C contains a null complementizer) **then**
4: **let** c **be** S in C
5: **return** *isOpenClausalComplement*(c)
6: **return** False

4.8.4 COMPLM: complementizer

A complementizer is a subordinating conjunction, *if*, *that*, or *whether*, that introduces a clausal complement (Section 4.8.2). A COMPLM is assigned when a clausal complement is found (see the line 6 of *isClausalComplement*(C) in Section 4.8.2).

(1)	She said [that] she wanted to go	COMPLM(wanted, that)
(2)	I wasn't sure [if] she liked me	COMPLM(liked, if)
(3)	I wasn't sure [whether] she liked me	COMPLM(liked, whether)

4.9 Modifiers: adverbial related

Adverbial related modifiers consist of adverbial clause modifiers (ADVCL), adverbial modifiers (ADVMOD), markers (MARK), negation modifiers (NEG), and noun phrases as adverbial modifiers (NPADVMOD).

Algorithm 4.11 : *hasAdverbialTag*(C)

Input: A constituent C.
Output: True if C has an adverbial function tag; otherwise, False.

1: **if** C has ADV|BNF|DIR|EXT|LOC|MNR|PRP|TMP|VOC **return** True
2: **return** False

4.9.1 ADVCL: adverbial clause modifier

An adverbial clause modifier is a clause that acts like an adverbial modifier. A clause with an adverbial function tag (see *hasAdverbialTag*(C)) is considered an ADVCL. Additionally, a subordinate clause or an open clause is considered an ADVCL if it does not satisfy any other dependency relation (see Appendices 4.8.2 and 4.8.3 for more details about clausal complements).

(1)	She came [as she promised]	ADVCL(came, promised)
(2)	She came [to see me]	ADVCL(came, see)
(3)	[Now that she is here] everything seems fine	ADVCL(seems, is)
(4)	She would have come [if she liked me]	ADVCL(come, liked)
(5)	I wasn't sure [if she liked me]	CCOMP(sure, liked)

In (2), *to see me* is an ADVCL (with a semantic role, purpose) although it may appear to be an open clausal complement of *came* (Section 4.8.3). In (4) and (5), *if she liked me* is considered an ADVCL and a clausal complement (CCOMP), respectively. This is because *if* in (3) creates a causal relation between the matrix and subordinate clauses whereas it does not serve any purpose other than introducing the subordinate clause in (4), just like a complementizer *that* or *whether*.

4.9.2 ADVMOD: adverbial modifier

An adverbial modifier is an adverb or an adverb phrase that modifies the meaning of another word. Other grammatical categories can also be ADVMOD if they modify adjectives.

(1)	I did [not] know her	ADVMOD(know, not)
(2)	I invited her [[as] well]	ADVMOD(invited, well), ADVMOD(well, as)
(3)	She is [already] [here]	ADVMOD(is, already), ADVMOD(is, here)
(4)	She is [so] beautiful	ADVMOD(beautiful, so)
(5)	I'm not sure [any] more	ADVMOD(more, any)

In (5), *any* is a determiner but considered an ADVMOD because it modifies the adjective, *more*.

Algorithm 4.12 : *isAdverbialModifier*(C)

Input: A constituent C.
Output: `True` if C is an adverbial function tag; otherwise, `False`.

```
1: if C is ADVP|RB*|WRB then
2:   let P be the parent of C
3:   if (P is PP) and (C's previous sibling is IN|TO) and (C is the last child of P) return False
4: return True
```

4.9.3 MARK: maker

A marker is a subordinating conjunction (e.g., *although*, *because*, *while*) that introduces an adverbial clause modifier (Section 4.9.1).

(1)	She came [as she promised]	MARK(promised, as)
(2)	She came [because she liked me]	MARK(liked, because)

The *setMarker*(C, P) method is called after P is identified as an adverbial modifier.

Algorithm 4.13 : *setMarker*(C, P)

Input: Constituents C and P, where P is the parent of C.
Output: If C is a marker, set its label to `MARK`.

1: **if** (P is `SBAR`) and (P is `ADVCL`) and (C is `IN|DT|TO`) **then** C.label $\leftarrow$ `MARK`

4.9.4 `NEG`: negation modifier

A negation modifier is an adverb that gives negative meaning to its head.

(1)	She decided not to come	`NEG`(come, not)
(2)	She didn't come	`NEG`(come, n't)
(3)	She never came	`NEG`(came, never)
(4)	This cookie is no good	`NEG`(is, no)

Algorithm 4.14 : *setNegationModifier*$(D)^*$

Input: A dependent D.
Output: If D is a negation modifier, set its label to `NEG`.

1: **if** (D is `NEG`) and (D is *never*|*not*|*n't*|*'nt*|*no*) **then** D.label $\leftarrow$ `NEG`

4.9.5 `NPADVMOD`: noun phrase as adverbial modifier

An adverbial noun phrase modifier is a noun phrase that acts like an adverbial modifier. A noun phrase with an adverbial function tag (see *hasAdverbialTag*(C)) is considered an `NPADVMOD`. Moreover, a noun phrase modifying either an adjective or an adverb is also considered an `NPADVMOD`.

(1)	Three times [a week]	`NPADVMOD`(times, week)
(2)	It is [a bit] surprising	`NPADVMOD`(surprising, bit)
(3)	[Two days] ago	`NPADVMOD`(ago, days)
(4)	It [all] feels right	`NPADVMOD`(feels, all)
(5)	I wrote the letter [myself]	`NPADVMOD`(wrote, myself)
(6)	I met her [last week]	`NPADVMOD`(met, week)
(7)	She lives [next door]	`NPADVMOD`(lives, door)

In (6) and (7), both *last week* and *next door* are considered `NPADVMOD` although they have different semantic roles, temporal and locative, respectively. These semantic roles can be retrieved from function tags and preserved as additional information (Section 6.1).

4.10 Modifiers: coordination related

Coordination related modifiers consist of conjuncts (`CONJ`), coordinating conjunctions (`CC`), and pre-correlative conjunctions (`PRECONJ`).

4.10.1 `CONJ`: conjunct

A conjunct is a dependent of the leftmost conjunct in coordination. The leftmost conjunct becomes the head of a coordinated phrase. Section 3.3 describes how conjuncts are derived.

(1) John, [Mary], and [Sam] CONJ(John, Mary), CONJ(John, Sam)
(2) John, [Mary], and [so on] CONJ(John, Mary), CONJ(John, on)
(3) John, [Mary], [Sam], [etc.] CONJ(John, Mary), CONJ(John, Sam), CONJ(John, etc.)

Although there is no coordinating conjunction in (3), the phrase is considered coordinated because of the presence of *etc.*

4.10.2 CC: coordinating conjunction

A coordinating conjunction is a dependent of the leftmost conjunct in coordination.

(1) John, Mary, [and] Sam CC(John, and)
(2) I know John [[as] [well] as] Mary CC(John, as), ADVMOD(as, as), ADVMOD(as, well)
(3) [And], I know you CC(know, And)

In (1), *and* becomes a CC of *John*, which is the leftmost conjunct. In (2), *as well as* is a multi-word expression so the dependencies between *as* and the others are not so meaningful but there to keep the tree connected. In (3), *And* is supposed to join the following clause with its preceding clause; however, since we do not derive dependencies across sentences, it becomes a dependent of the head of this clause, *know*.

Algorithm 4.15 : *isCoordinatingConjunction*(C)

Input: A constituent C.
Output: `True` if C is a coordinating conjunction; otherwise, `False`.

1: **return** C is `CC|CONJP`

4.10.3 PRECONJ: pre-correlative conjunction

A pre-correlative conjunction is the first part of a correlative conjunction that becomes a dependent of the first conjunct in coordination.

(1) [Either] John [or] Mary PRECONJ(John, Either), CC(John, or), CONJ(John, Mary)
(2) [Not only] John [but also] Mary PRECONJ(John, Not), CC(John, but), CONJ(John, Mary)

Algorithm 4.16 : *isPreCorrelativeConjunction*(C)

Input: A constituent C.
Output: `True` if C is a pre-correlative conjunction; otherwise, `False`.

1: **if** (C is `CC`) and (C is *both* | *either* | *neither* | *whether*) **return** `True`
2: **if** (C is `CONJP`) and (C is *not only*) **return** `True`
3: **return** `False`

4.11 Modifiers: noun phrase related

Noun phrase related modifiers consist of appositional modifiers (APPOS), determiners (DET), infinitival modifiers (INFMOD), modifiers of nominals (NMOD), noun compound modifiers (NN), numeric modifiers (NUM), participial modifiers (PARTMOD), possessive modifiers (POSSESSIVE), predeterminers (PREDET), and relative clause modifiers (RCMOD).

Algorithm 4.17 : *getNonFiniteModifierLabel*(C)

Input: A constituent C whose parent is NML|NP|WHNP.
Output: INFMOD or PARTMOD.

1: **if** *isOpenClausalComplement*(C) or (C is VP) **then** # Section 4.8.3
2: **if** *isInfinitivalModifier*(C) **return** INFMOD # Section 4.11.4
3: **return** PARTMOD # Section 4.11.7

Algorithm 4.18 : *getNounModifierLabel*(C)

Input: A constituent C whose parent is NML|NP|NX|WHNP.
Output: AMOD, DET, NN, NUM, POSSESSIVE, PREDET, or NMOD.

1: **if** C is VBG|VBN **return** AMOD # Section 4.14.1
2: **if** C is DT|WDT|WP **return** DET # Section 4.11.3
3: **if** C is PDT **return** PREDET # Section 4.11.9
4: **if** C is NML|NP|FW|NN* **return** NN # Section 4.11.5
5: **if** C is CD|QP **return** NUM # Section 4.11.6
6: **if** C is POS **return** POSSESSIVE # Section 4.11.8
7: **return** NMOD # Section 4.11.1

4.11.1 NMOD: modifier of nominal

A modifier of nominal is any unclassified dependent that modifies the head of a noun phrase.

4.11.2 APPOS: appositional modifier

An appositional modifier of an NML|NP is a noun phrase immediately preceded by another noun phrase, which gives additional information to its preceding noun phrase. A noun phrase with an adverbial function tag (Section 4.9.1) is not considered an APPOS. Section 3.2 describes how appositional modifiers are derived.

(1)	John, [my brother]	APPOS(John, bother)
(2)	The year [2012]	APPOS(year, 2012)
(3)	He [himself] bought the car	APPOS(He, himself)
(4)	Computational Linguistics [(CL)]	APPOS(Linguistics, CL)
(5)	The book, Between You and Me	APPOS(book, Between)
(6)	MacGraw-Hill Inc., New York	NPADVMOD(Inc., York)

4.11.3 DET: determiner

A determiner is a word token whose POS tag is DT|WDT|WP that modifies the head of a noun phrase.

(1)	[The] US military	DET(military, The)
(2)	[What] kind of movie is this	DET(movie, What)

4.11.4 INFMOD: infinitival modifier

An infinitival modifier is an infinitive clause or phrase that modifies the head of a noun phrase.

(1)	I have too much homework [to do]	INFMOD(homework, do)
(2)	I made an effort [to come]	INFMOD(effort, come)

Algorithm 4.19 : *isInfinitivalModifier*(C)

Input: A constituent C whose parent is `NML|NP|WHNP`.
Output: `True` if C is an infinitival modifier; otherwise, `False`.

1: **if** C is `VP` **then** $vp \leftarrow C$
2: **else**
3: **let** t **be** the first descendant of C that is `VP`
4: $vp \leftarrow (t \text{ exists}) \ ? \ t : \texttt{null}$
5: **if** $vp \neq \texttt{null}$ **then**
6: **let** t **be** the first child of vp that is `VP`
7: **while** t exists **do**
8: $vp \leftarrow t$
9: **if** vp's previous sibling is `TO` **return** `True`
10: **let** t **be** the first child of vp that is `VP`
11: **if** vp contains `TO` **return** `True`
12: **return** `False`

4.11.5 `NN`: noun compound modifier

A noun compound modifier is any noun that modifies the head of a noun phrase.

(1) The [US] military `PREDET`(military, US)
(2) The [video] camera `PREDET`(camera, video)

4.11.6 `NUM`: numeric modifier

A numeric modifier is any number or quantifier phrase that modifies the head of a noun phrase.

(1) [14] degrees `NUM`(degrees, 14)
(2) [One] nation, [fifty] states `NUM`(nation, One), `NUM`(states, fifty)

4.11.7 `PARTMOD`: participial modifier

A participial modifier is a clause or phrase whose head is a verb in a participial form (e.g., gerund, past participle) that modifies the head of a noun phrase.

(1) I went to the party [hosted by her] `PARTMOD`(party, hosted)
(2) I met people [coming to this party] `PARTMOD`(people, coming)

4.11.8 `POSSESSIVE`: possessive modifier

A possessive modifier is a word token whose POS tag is `POS` that modifies the head of a noun phrase.

(1) John['s] car `NMOD`(John, 's)

4.11.9 `PREDET`: predeterminer

A predeterminer is a word token whose POS tag is `PDT` that modifies the head of a noun phrase.

(1) [Such] a beautiful woman `PREDET`(woman, Such)
(2) [All] the books we read `PREDET`(books, All)

4.11.10 RCMOD: relative clause modifier

A relative clause modifier is a either relative clause or a reduced relative clause that modifies the head of an NML|NP|WHNP.

(1)	I bought the car [(that) I wanted]	RCMOD(car, wanted)
(2)	I was the first person [to buy this car]	INFMOD(person, buy)
(3)	This is the car [for which I've waited]	RCMOD(car, waited)
(4)	It is a car [(that is) worth buying]	RCMOD(car, worth)

In (2), *to buy this car* is considered an infinitival modifier (INFMOD) although it contains an empty *wh*-complementizer in the constituent tree. (4) shows an example of a reduced relative clause.

Algorithm 4.20 : *isRelativeClauseModifier*(C)

Input: A constituent C whose parent is NML|NP|WHNP.
Output: True if C is a relative clause modifier; otherwise, False.

1: **if** C is RRC **return** True
2: **if** (C is SBAR) and (C contains a *wh*-complementizer) **return** True
3: **return** False

4.12 Modifiers: prepositional phrase related

Prepositional phrase related modifiers consist of complements of prepositions, objects of prepositions, and prepositional modifiers.

Algorithm 4.21 : *getPrepositionModifierLabel*(C, d)

Input: A constituent C whose parent is NP|WHPP, and the head dependent d of C.
Output: POBJor PCOMP.

1: **if** (C is NP|NML) or (d is W*) **return** POBJ # Section 4.12.2
2: **return** PCOMP # Section 4.12.1

4.12.1 PCOMP: complement of a preposition

A complement of a preposition is any dependent that is not a POBJ but modifies the head of a prepositional phrase.

(1)	I agree with [what you said]	PCOMP(with, said)

4.12.2 POBJ: object of a preposition

An object of a preposition is a noun phrase that modifies the head of a prepositional phrase, which is usually a preposition but can be a verb in a participial form such as VBG.

(1)	On [the table]	POBJ(On, table)
(2)	Including us	POBJ(Including, us)
(3)	Given us	POBJ(Given, us)

4.12.3 PREP: prepositional modifier

A prepositional modifier is any prepositional phrase that modifies the meaning of its head.

(1)	Thank you [for coming [to my house]]	PREP(Thank, for), PREP(coming, to)
(2)	Please put your coat [on the table]	PREP(put, on)
(3)	Or just give it [to me]	PREP(give, to)

In (1), *to my house* is a PREP carrying a semantic role, direction. These semantic roles are preserved as additional information in our representation (Section 6.1). In (2), *on the table* is a PREP, which is considered the locative complement of *put* in some linguistic theories. Furthermore, in (3), *to me* is the dative object of *give* in the unshifted form, which is also considered a PREP in our analysis. This kind of information is also preserved with syntactic function tags in our representation (Section 6.2).

4.13 Modifiers: quantifier phrase related

Quantifier phrase related modifiers consist of number compound modifiers (NUMBER) and quantifier phrase modifiers (QUANTMOD).

4.13.1 NUMBER: number compound modifier

A number compound modifier is a cardinal number that modifies the head of a quantifier phrase.

(1)	[Seven] million dollars	NUMBER(million, Seven), NUM(dollars, million)
(2)	[Two] to [three] hundred	NUMBER(hundred, Two), NUMBER(hundred, three)

4.13.2 QUANTMOD: quantifier phrase modifier

A quantifier phrase modifier is a dependent of the head of a quantifier phrase.

(1)	[More] [than] five	AMOD(five, More), QUANTMOD(five, than)
(2)	[Five] [to] six	QUANTMOD(six, Five), QUANTMOD(six, to)

Quantifier phrases often form a very flat hierarchy, which makes it hard to derive correct dependencies for them. In (1), *More than* is a multi-word expression that should be grouped into a separate constituent (e.g., [More than] one); however, this kind of analysis is not used in our constituent trees. Thus, *More* and *than* become an AMOD and a QUANTMOD of *five*, respectively. In (2), *to* is more like a conjunction connecting *Five* to *six*, which is not explicitly represented. Thus, *Five* and *to* become QUANTMODs of *six* individually. More analysis needs to be done to derive correct dependencies for quantifier phrases, which will be explored in future work.

4.14 Modifiers: miscellaneous

Miscellaneous modifiers consists of adjectival modifiers (AMOD), unclassified dependents (DEP), interjections (INTJ), meta modifiers (META), parenthetical modifiers (PARATAXIS), possession modifiers (POSS), particles (PRT), punctuation (PUNCT), and roots (ROOT).

4.14.1 AMOD: adjectival modifier

An adjectival modifier is an adjective or an adjective phrase that modifies the meaning of another word, usually a noun.

(1)	A [beautiful] girl	AMOD(girl, beautiful)
(2)	A [five year old] girl	AMOD(girl, old)
(3)	[How many] people came	AMOD(people, many)

4.14.2 DEP: unclassified dependent

An unclassified dependent is a dependent that does not satisfy conditions for any other dependency.

4.14.3 INTJ: interjection

An interjection is an expression made by the speaker of an utterance.

(1) [Well], it is my birthday INTJ(is, Well)
(2) I [um] will throw a party INTJ(throw, um)

Algorithm 4.22 : *isInterjection*(C)

Input: A constituent C.
Output: `True` if C is an interjection; otherwise, `False`.

1: **return** C is `INTJ|UH`

4.14.4 META: meta modifier

A meta modifier is code (1), embedded (2), or meta (3) information that is randomly inserted in a phrase or clause.

(1) [choijd] My first visit META(visit, choijd)
(2) I visited Boulder and {others} [other cities] META(Boulder, others), CONJ(Boulder, cities)
(3) [Applause] Thank you META(Thank, Applause)

Algorithm 4.23 : *isMetaModifier*(C)

Input: A constituent C.
Output: `True` if C is a meta modifier; otherwise, `False`.

1: **return** C is `CODE|EDITED|EMBED|LST|META`

4.14.5 PARATAXIS: parenthetical modifier

A parenthetical modifier is an embedded chunk, often but not necessarily surrounded by parenthetical notations (e.g,. brackets, quotes, commas, etc.), which gives side information to its head.

(1) She[, I mean,] Mary was here PARATAXIS(was, mean)
(2) [That is to say,] John was also here PARATAXIS(was, is)

Algorithm 4.24 : *isParentheticalModifier*(C)

Input: A constituent C.
Output: `True` if C is a parenthetical modifier; otherwise, `False`.

1: **return** C is `PRN`

4.14.6 POSS: possession modifier

A possession modifier is either a possessive determiner (PRP$) or a NML|NP|WHNP containing a possessive ending that modifies the head of a ADJP|NML|NP|QP|WHNP.

(1)	I bought [his] car	POSS(car, his)
(2)	I bought [John's] car	POSS(car, John)
(3)	This building is [Asia's] largest	POSS(largest, Asia)

Note that *Asia's* in (3) is a POSS of *largest*, which is an adjective followed by an elided *building*. Such an expression does not occur often but we anticipate it to appear more when dealing with informal texts (e.g., text-messages, conversations, web-texts).

Algorithm 4.25 : *isPossessionModifier*(C)

Input: Constituents C and P, where P is the parent of C.
Output: True if C is a possession modifier; otherwise, False.

1: **if** C is PRP$ **return** True
2: **if** P is ADJP|NML|NP|QP|WHNP **then**
3: **return** C contains POS
4: **return** False

4.14.7 PRT: particle

A particle is a preposition in a phrasal verb that forms a verb-particle construction.

(1)	Shut [down] the machine	PRT(Shut, down)
(2)	Shut the machine [down]	PRT(Shut, down)

4.14.8 PUNCT: punctuation

Any punctuation is assigned the dependency label PUNCT.

Algorithm 4.26 : *isPunctuation*(C)

Input: A constituent C.
Output: True if C is punctuation; otherwise, False.

1: **return** (C is :|,|.|``|''|-LRB-|-RRB-|HYPH|NFP|SYM|PUNC)

4.14.9 ROOT: root

A root is the root of a tree that does not depend on any node in the tree but the artificial root node whose ID is 0. A tree can have multiple roots only if the top constituent contains more than one child in the original constituent tree (this does not happen with the OntoNotes Treebank but happens quite often with medical corpora).

5 Adding secondary dependencies

Secondary dependencies are additional dependency relations derived from gapping relations (Section 5.1), relative clauses (Section 5.2), right node raising (Section 5.3), and open clausal complements (Section 5.4). These are separated from the other types of dependencies (Section 4) because they can break tree properties (e.g., single head, acyclic) when combined with the others. Preserving tree structure is important because most dependency parsing algorithms assume their input to be trees. Secondary dependencies give deeper representations that allow extraction of more complete information from the dependency structure.

5.1 GAP: gapping

Gapping is represented by co-indexes (with the = symbol) in constituent trees. Gapping usually happens in forms of coordination where some parts included in the first conjunct do not appear in later conjuncts (Jackendoff, 1971). In Figure 25, the first conjunct, VP-3, contains the verb *used*, which does not appear in the second conjunct, VP-4, but is implied for both NP=1 and PP=2. The CoNLL dependency approach makes the conjunction, *and*, the heads of both NP=1 and PP=2, and adds an extra label, GAP, to their existing labels (ADV-GAP and GAP-OBJ in Figure 27). Although this represents the gapping relations in one unified format, statistical dependency parsers perform poorly on these labels because they do not occur frequently enough and are often confused with regular coordination.

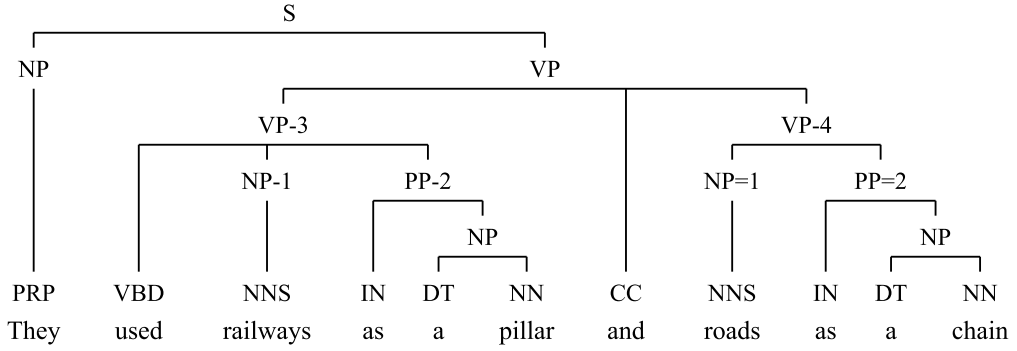

Figure 25: An example of a gapping relation.

In our approach, gapping is represented as secondary dependencies; this way, it can be trained separately from the other types of dependencies. The GAP dependencies in Figure 26 show how gapping is represented in our structure: the head of each constituent involving a gap (*road*, as_9) becomes a dependent of the head of the leftmost constituent not involving a gap (*railways*, as_4).

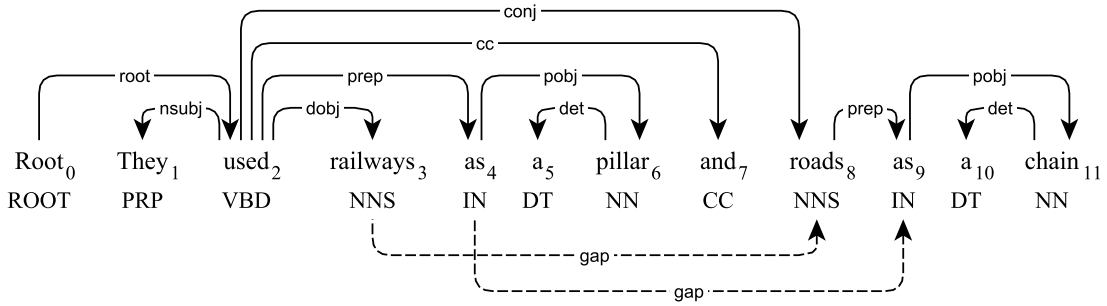

Figure 26: The CLEAR dependency tree converted from the constituent tree in Figure 25. The gapping relations are represented by the secondary dependencies, GAP.

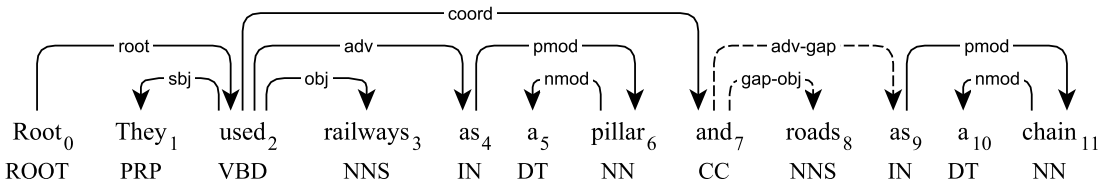

Figure 27: The CoNLL dependency tree converted from the constituent tree in Figure 25. The dependencies derived from the gapping relations, `ADV-GAP`, `GAP-OBJ`, are indicated by dotted lines.

5.2 `REF`: referent

A referent is the relation between a *wh*-complementizer in a relative clause and its referential head. In Figure 28, the relation between the complementizer *which* and its referent *Crimes* is represented by the `REF` dependency. Referent relations are represented as secondary dependencies because integrating them with other dependencies breaks the single-head tree property (e.g., *which* would have multiple heads in Figure 28).

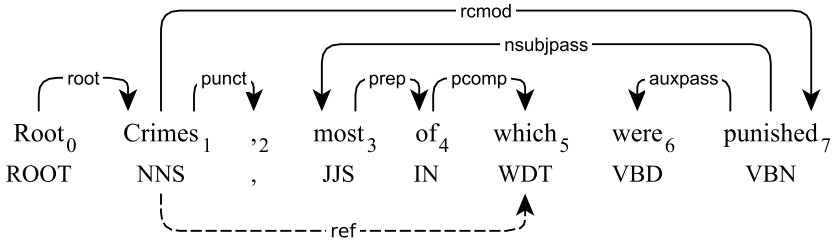

Figure 28: An example of a referent relation. The referent relation is represented by the secondary dependency, `REF`.

Algorithm 5.1 : *linkReferent*(C)

Input: A constituent C.

if C is `WHADVP|WHNP|WHPP` **then**
  let c **be** the *wh*-complementizer of C
  let s **be** the topmost `SBAR` of C
  if the parent of s is `UCP` **then** $s \leftarrow s$.parent
  if *isRelativizer*(c) and (s has no `NOM`) **then**
    let p **be** the parent of s
    ref $\leftarrow$ `null`
    if p is `NP|ADVP` **then**
      let *ref* **be** the previous sibling of s that is `NP|ADVP`, respectively
    elif p is `VP` **then**
      let t **be** the previous sibling of s that has `PRD`
      if s has `CLF` **then** *ref* $\leftarrow t$
      if (C is `WHNP`) and (t is `NP`) **then** *ref* $\leftarrow t$
      if (C is `WHPP`) and (t is `PP`) **then** *ref* $\leftarrow t$
      if (C is `WHADVP`) and (t is `ADVP`) **then** *ref* $\leftarrow t$
    if *ref* $\neq$ `null` **then**
      while *ref* has an antecedent **do** *ref* $\leftarrow$ *ref*.antecedent
      c.rHead $\leftarrow$ *ref*
      c.rLabel $\leftarrow$ `REF`

The *linkReferent*(C) method in Algorithm 5.1 finds a *wh*-complementizer and makes it a dependent of its referent. Note that referent relations are not provided in constituent trees; however, they are manually annotated in the PropBank as `LINK-SLC` (Bonial et al., 2010, Chap. 1.8). This algorithm was tested against the PropBank annotation using gold-standard constituent trees and showed an F1-score of approximately 97%.

Algorithm 5.2 : *isRelativizer*(C)

Input: A constituent C.
Output: `True` if C is a relativizer linked to some referent; otherwise, `False`.

1: **return** C is `0` | *that* | *when* | *where* | *whereby* | *wherein* | *whereupon* | *which* | *who* | *whom* | *whose*

5.3 `RNR`: right node raising

As mentioned in Section 2.3, missing dependencies caused by right node raising are preserved as secondary dependencies. In Figure 14 (page 12), *her* should be a dependent of both *for* and *in*; however, it is a dependent of only *for* in our structure because making it a dependent of both nodes breaks a tree property (e.g., *her* would have multiple heads). Instead, the dependency between *her* and *for* is preserved with the `RNR` dependency. Figure 30 shows another example of right node raising where the raised constituent, `VP`-2, is the head of the constituents that it is raised from, `VP`-4 and `VP`-5. In this case, *done* becomes the head of can_2 with the dependency label, `RNR`.

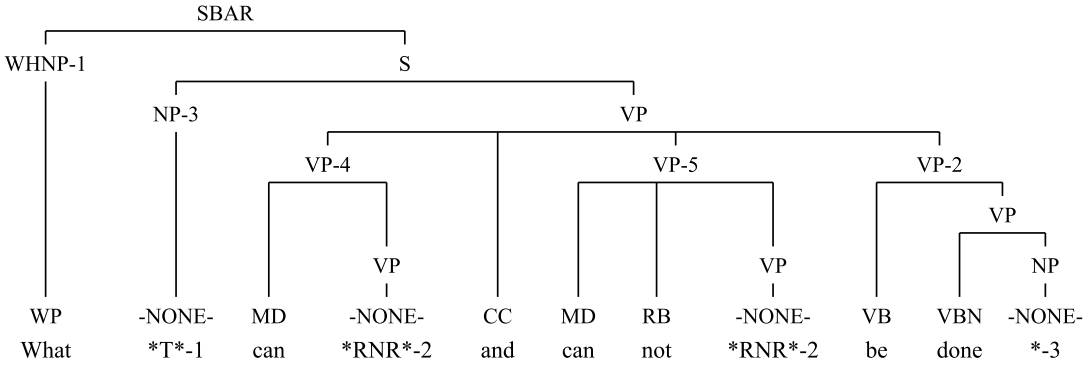

Figure 29: An example of right node raising where the raised constituent is the head.

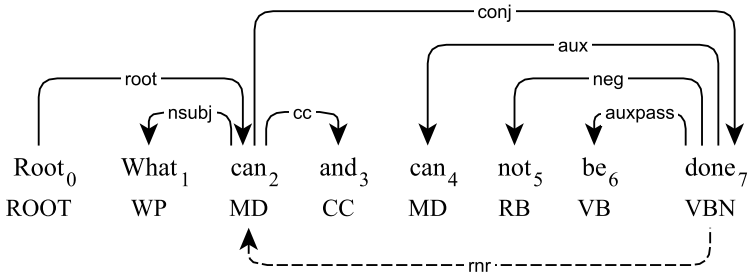

Figure 30: The dependency tree converted from the constituent tree in Figure 29. Right node raising is represented by the secondary dependency, `RNR`.

5.4 XSUBJ: open clausal subject

An open clausal subject is the subject of an open clausal complement (usually non-finite) that is governed externally. Open clausal subjects are often caused by raising and control verbs (Chomsky, 1981). In Figure 31, the subject of *like* is moved to the subject position of the raising verb *seemed* (subject raising) so that *She* becomes the syntactic subject of *seemed* as well as the open clausal subject of *like* (see Figure 32).

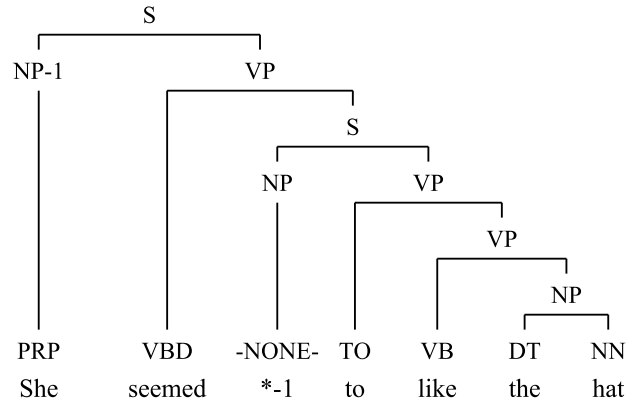

Figure 31: An example of an open clausal subject caused by a subject raising.

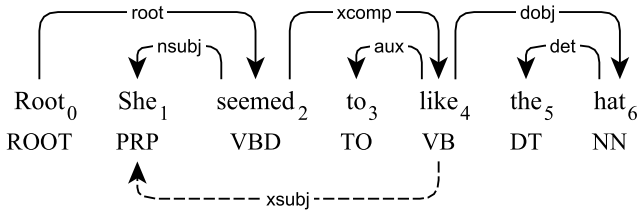

Figure 32: The dependency tree converted from the constituent tree in Figure 31. The open clausal subject is represented by the secondary dependency, XSUBJ.

In Figure 33, the subject of *wear* is shared with the object of the control verb *forced* (object control) so that *me* becomes the direct object of *forced* as well as the open clausal subject of *wear* (Figure 34). Alternatively, *me* in Figure 35 is not considered the direct object of *expected* but the subject of *wear*; this is a special case called "exceptional case marking (ECM)", which appears to be very similar to the object control case but is handled differently in constituent trees (see Taylor (2006) for more details about ECM verbs).

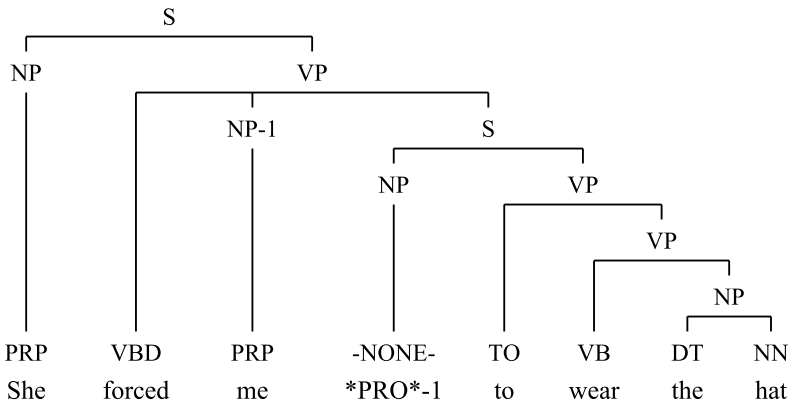

Figure 33: An example of an open clausal subject caused by an object raising.

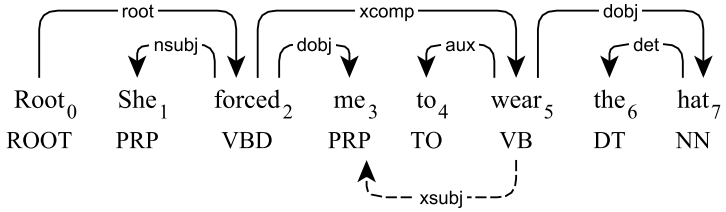

Figure 34: A dependency tree converted from the constituent tree in Figure 33. The open clausal subject is represented by the secondary dependency, `XSUBJ`.

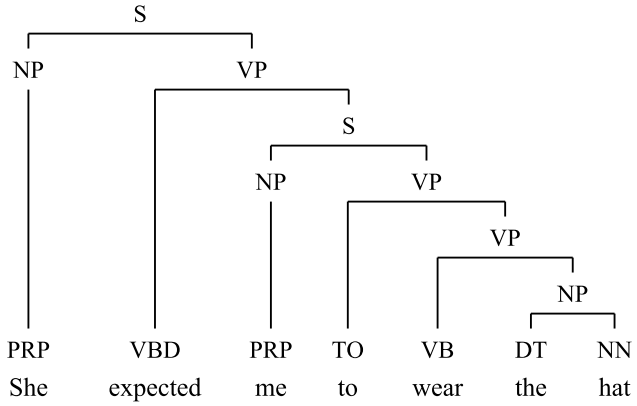

Figure 35: An example of exceptional case marking.

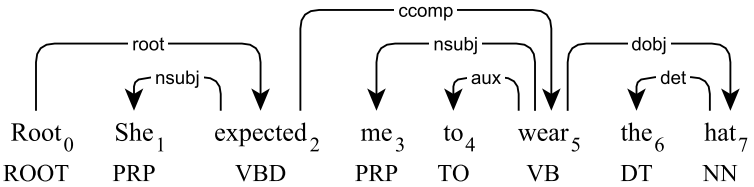

Figure 36: A dependency tree converted from the constituent tree in Figure 35.

6 Adding function tags

6.1 SEM: semantic function tags

When a constituent is annotated with a semantic function tag (BNF, DIR, EXT, LOC, MNR, PRP, TMP, and VOC; see Appendix A.3), the tag is preserved with the head of the constituent as an additional feature. In Figure 37, the subordinate clause SBAR is annotated with the function tag PRP, so the head of the subordinate clause, *is*, is annotated with the semantic tag in our representation (Figure 38). Note that the CoNLL dependency approach uses these semantic tags in place of dependency labels (e.g., the dependency label between *is* and *let* would be PRP instead of ADVCL). These tags are kept separate from the other kinds of dependency labels in our approach so they can be processed either during or after parsing. The semantic function tags can be integrated easily into our dependency structure by replacing dependency labels with semantic tags (Figure 39).

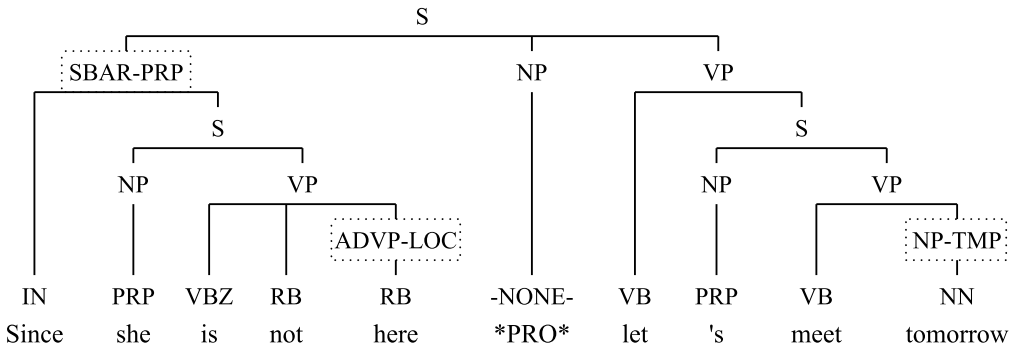

Figure 37: A constituent tree with semantic function tags. The phrases with the semantic function tags are indicated by dotted boxes.

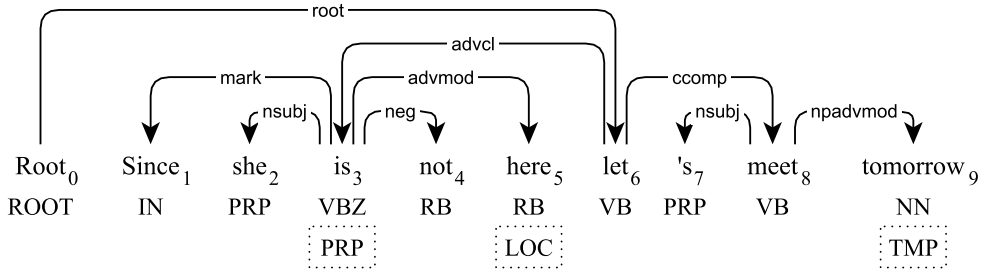

Figure 38: A dependency tree converted from the constituent tree in Figure 37. The function tags PRP, LOC, and TMP are preserved as additional features of *is*, *here*, and *tomorrow*, respectively.

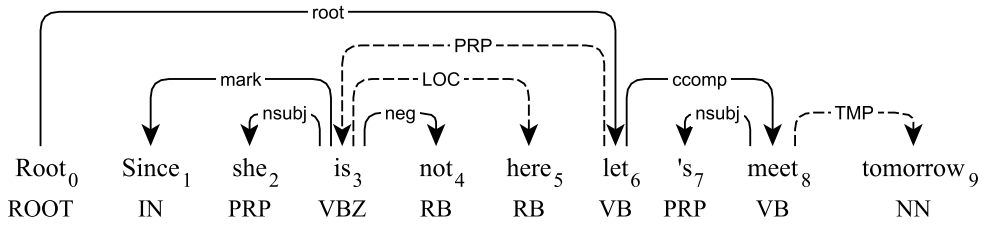

Figure 39: Another dependency tree converted from the constituent tree in Figure 37. The function tags, PRP, LOC, and TMP, replace the original dependency labels, ADVCL, ADVMOD, and NPADVMOD.

6.2 SYN: syntactic function tags

When a constituent is annotated with one or more syntactic function tags (ADV, CLF, CLR, DTV, NOM, PUT, PRD, RED, and TPC; see Appendix A.3), all tags are preserved with the head of the constituent as additional features. In Figure 40, the noun phrase NP-1 is annotated with the function tag PRD and TPC so the head of the noun phrase, *slap*, is annotated with both tags in our representation (Figure 41). Similarly to the semantic function tags (Section 6.1), syntactic function tags can also be integrated into our dependency structure by replacing dependency labels with syntactic tags.

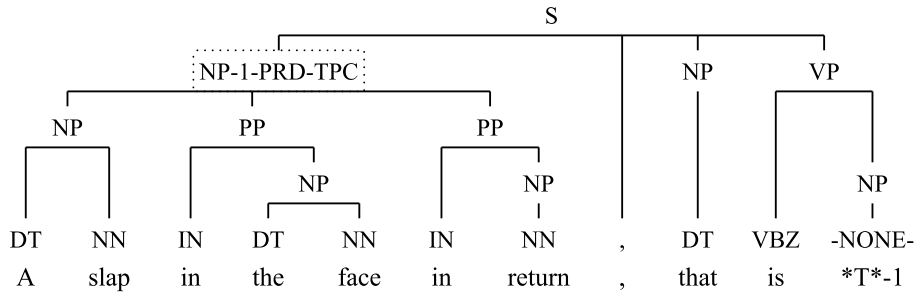

Figure 40: A constituent tree with syntactic function tags. The phrase with the syntactic function tags is indicated by a dotted box.

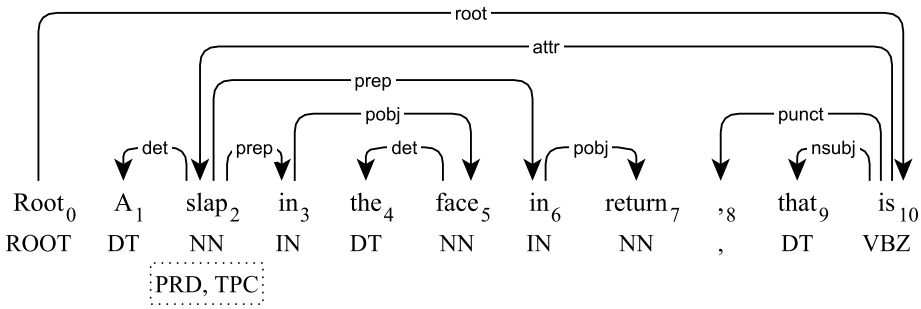

Figure 41: A dependency tree converted from the constituent tree in Figure 40. The function tags, PRD and TPC, are preserved as additional features of *slap*.

References

Ann Bies, Mark Ferguson, Karen Katz, and Robert MacIntyre. Bracketing Guidelines for Treebank II Style Penn Treebank Penn Treebank Project. Technical report, University of Pennsylvania, 1995.

Claire Bonial, Olga Babko-Malaya, Jinho D. Choi, Jena Hwang, and Martha Palmer. PropBank Annotation Guidelines. Technical report, University of Colorado at Boulder, 2010.

Jinho D. Choi and Martha Palmer. Robust Constituent-to-Dependency Conversion for Multiple Corpora in English. In *Proceedings of the 9th International Workshop on Treebanks and Linguistic Theories*, TLT'9, 2010.

Noam Chomsky. *Lectures in Government and Binding*. Dordrecht, Foris, 1981.

Noam Chomsky. *The Minimalist Program*. The MIT Press, 1995.

Elizabeth A. Cowper. *A Concise Introduction to Syntactic Theory*. The University of Chicago Press, 1992.

Marie-Catherine de Marneffe and Christopher D. Manning. The Stanford typed dependencies representation. In *Proceedings of the COLING workshop on Cross-Framework and Cross-Domain Parser Evaluation*, 2008a.

Marie-Catherine de Marneffe and Christopher D. Manning. Stanford Dependencies manual. `http://nlp.stanford.edu/software/dependencies_manual.pdf`, 2008b.

Jan Hajič, Massimiliano Ciaramita, Richard Johansson, Daisuke Kawahara, Maria Antònia Martí, Lluís Màrquez, Adam Meyers, Joakim Nivre, Sebastian Padó, Jan Štěpánek, Pavel Straňák, Mihai Surdeanu, Nianwen Xue, and Yi Zhang. The CoNLL-2009 Shared Task: Syntactic and Semantic Dependencies in Multiple Languages. In *Proceedings of the 13th Conference on Computational Natural Language Learning: Shared Task*, CoNLL'09, pages 1–18, 2009.

Liang Huang and Kenji Sagae. Dynamic Programming for Linear-Time Incremental Parsing. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, ACL'10, 2010.

Ray Jackendoff. Gapping and Related Rules. *Linguistic Inquiry*, 2:21–35, 1971.

Richard Johansson. *Dependency-based Semantic Analysis of Natural-language Text*. PhD thesis, Lund University, 2008.

Richard Johansson and Pierre Nugues. Extended Constituent-to-dependency Conversion for English. In *Proceedings of the 16th Nordic Conference of Computational Linguistics*, NODALIDA'07, 2007.

Terry Koo, Alexander M. Rush, Michael Collins, Tommi Jaakkola, and David Sontag. Dual Decomposition for Parsing with Non-Projective Head Automata. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, EMNLP'10, pages 1288–1298, 2010.

Marco Kuhlmann and Joakim Nivre. Transition-Based Techniques for Non-Projective Dependency Parsing. *Northern European Journal of Language Technology*, 2(1):1–19, 2010.

Robert D. Levine. Right Node (non)-Raising. *Linguistic Inquiry*, 16(3):492–497, 1985.

Mitchell P. Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. Building a Large Annotated Corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330, 1993.

Ryan Mcdonald and Fernando Pereira. Online Learning of Approximate Dependency Parsing Algorithms. In *Proceedings of the Annual Meeting of the European American Chapter of the Association for Computational Linguistics*, EACL'06, pages 81–88, 2006.

Ryan McDonald and Giorgio Satta. On the Complexity of Non-Projective Data-Driven Dependency Parsing. In *Proceedings of the 10th International Conference on Parsing Technologies*, IWPT'07, pages 121–132, 2007.

Rodney D. Nielsen, James Masanz, Philip Ogren, Wayne Ward, James H. Martin, Guergana Savova, and Martha Palmer. An architecture for complex clinical question answering. In *Proceedings of the 1st ACM International Health Informatics Symposium*, IHI'10, pages 395–399, 2010.

Jens Nilsson, Joakim Nivre, and Johan Hall. Graph Transformations in Data-Driven Dependency Parsing. In *Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics*, COLING:ACL'06, pages 257–264, 2006.

Joakim Nivre. Algorithms for deterministic incremental dependency parsing. *Computational Linguistics*, 34 (4):513–553, 2008.

Joakim Nivre and Mario Scholz. Deterministic Dependency Parsing of English Text. In *Proceedings of the 20th International Conference on Computational Linguistics*, COLING'04, pages 64–70, 2004.

Joakim Nivre, Johan Hall, Sandra Kübler, Ryan McDonald, Jens Nilsson, Sebastian Riedel, and Deniz Yuret. The CoNLL 2007 Shared Task on Dependency Parsing. In *Proceedings of the CoNLL Shared Task Session of EMNLP-CoNLL 2007*, pages 915–932, 2007.

Slav Petrov and Ryan McDonald. Overview of the 2012 Shared Task on Parsing the Web. In *Proceedings of the 1st Workshop on Syntactic Analysis of Non-Canonical Language: shared task*, SANCL'12, 2012.

Owen Rambow, Cassandre Creswell, Rachel Szekely, Harriet Taber, and Marilyn Walker. A Dependency Treebank for English. In *Proceedings of the 3rd International Conference on Language Resources and Evaluation (LREC'02)*, 2002.

Alexander M. Rush and Slav Petrov. Vine Pruning for Efficient Multi-Pass Dependency Parsing. In *Proceedings of the 12th Annual Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, NAACL:HLT'12, 2012.

Mihai Surdeanu, Richard Johansson, Adam Meyers, Lluís Màrquez, and Joakim Nivre. The CoNLL-2008 Shared Task on Joint Parsing of Syntactic and Semantic Dependencies. In *Proceedings of the 12th Conference on Computational Natural Language Learning: Shared Task*, CoNLL'08, pages 59–177, 2008.

Ann Taylor. Treebank 2a Guidelines. `http://www-users.york.ac.uk/~lang22/TB2a_Guidelines.htm`, 2006.

M. Čmejrek, J. Cuřín, and J. Havelka. Prague Czech-English Dependency Treebank: Any Hopes for a Common Annotation Scheme? In *HLT-NAACL'04 workshop on Frontiers in Corpus Annotation*, pages 47–54, 2004.

Karin Verspoor, Kevin B. Cohen, Arrick Lanfranchi, Colin Warner, Helen L. Johnson, Christophe Roeder, Jinho D. Choi, Christopher Funk, Yuriy Malenkiy, Miriam Eckert, Nianwen Xue, William A. Baumgartner Jr., Mike Bada, Martha Palmer, and Hunter Larry E. A corpus of full-text journal articles is a robust evaluation tool for revealing differences in performance of biomedical natural language processing tools. *BMC Bioinformatics*, 2012. in press.

Ralph Weischedel, Eduard Hovy, Martha Palmer, Mitch Marcus, Robert Belvin, Sameer Pradhan, Lance Ramshaw, and Nianwen Xue. OntoNotes: A Large Training Corpus for Enhanced Processing. In Joseph Olive, Caitlin Christianson, and John McCary, editors, *Handbook of Natural Language Processing and Machine Translation*. Springer, 2011.

A Constituent Treebank Tags

This appendix shows tags used in various constituent Treebanks for English (Marcus et al., 1993; Nielsen et al., 2010; Weischedel et al., 2011; Verspoor et al., 2012). Tags followed by * are not the typical Penn Treebank tags but used in some other Treebanks.

A.1 Part-of-speech tags

Word level tags			
ADD	Email	POS	Possessive ending
AFX	Affix	PRP	Personal pronoun
CC	Coordinating conjunction	PRP$	Possessive pronoun
CD	Cardinal number	RB	Adverb
CODE	Code ID	RBR	Adverb, comparative
DT	Determiner	RBS	Adverb, superlative
EX	Existential there	RP	Particle
FW	Foreign word	TO	To
GW	Go with	UH	Interjection
IN	Preposition or subordinating conjunction	VB	Verb, base form
JJ	Adjective	VBD	Verb, past tense
JJR	Adjective, comparative	VBG	Verb, gerund or present participle
JJS	Adjective, superlative	VBN	Verb, past participle
LS	List item marker	VBP	Verb, non-3rd person singular present
MD	Modal	VBZ	Verb, 3rd person singular present
NN	Noun, singular or mass	WDT	*Wh*-determiner
NNS	Noun, plural	WP	*Wh*-pronoun
NNP	Proper noun, singular	WP$	Possessive *wh*-pronoun
NNPS	Proper noun, plural	WRB	*Wh*-adverb
PDT	Predeterminer	XX	Unknown
Punctuation like tags			
$	Dollar	-LRB-	Left bracket
:	Colon	-RRB-	Right bracket
,	Comma	HYPH	Hyphen
.	Period	NFP	Superfluous punctuation
‘‘	Left quote	SYM	Symbol
’’	Right quote	PUNC	General punctuation

Table 6: A list of part-of-speech tags for English.

A.2 Clause and phrase level tags

Clause level tags			
S	Simple declarative clause		
SBAR	Clause introduced by a subordinating conjunction		
SBARQ	Direct question introduced by a *wh*-word or a *wh*-phrase		
SINV	Inverted declarative sentence		
SQ	Inverted yes/no question, or main clause of a *wh*-question		
Phrase level tags			
ADJP	Adjective phrase	NX	N-bar level phrase
ADVP	Adverb phrase	PP	Prepositional phrase
CAPTION*	Caption	PRN	Parenthetical phrase
CIT*	Citation	PRT	Particle
CONJP	Conjunction phrase	QP	Quantifier Phrase
EDITED	Edited phrase	RRC	Reduced relative clause
EMBED	Embedded phrase	TITLE*	Title
FRAG	Fragment	TYPO	Typo
HEADING*	Heading	UCP	Unlike coordinated phrase
INTJ	Interjection	VP	Verb phrase
LST	List marker	WHADJP	*Wh*-adjective phrase
META	Meta data	WHADVP	*Wh*-adverb phrase
NAC	Not a constituent	WHNP	*Wh*-noun phrase
NML	Nominal phrase	WHPP	*Wh*-prepositional phrase
NP	Noun phrase	X	Unknown

Table 7: A list of clause and phrase level tags for English.

A.3 Function tags

Syntactic roles			
ADV	Adverbial	PUT	Locative complement of *put*
CLF	*It*-cleft	PRD	Non-VP predicate
CLR	Closely related constituent	RED*	Reduced auxiliary
DTV	Dative	SBJ	Surface subject
LGS	Logical subject in passive	TPC	Topicalization
NOM	Nominalization		
Semantic roles			
BNF	Benefactive	MNR	Manner
DIR	Direction	PRP	Purpose or reason
EXT	Extent	TMP	Temporal
LOC	Locative	VOC	Vocative
Text and speech categories			
ETC	Et cetera	SEZ	Direct speech
FRM*	Formula	TTL	Title
HLN	Headline	UNF	Unfinished constituent
IMP	Imperative		

Table 8: A list of function tags for English.

B Dependency Labels

B.1 CoNLL dependency labels

This appendix shows a list of the CoNLL dependency labels. See Johansson (2008, Chap. 4) for more details about the CoNLL dependency labels.

Labels retained from function tags			
ADV	Unclassified adverbial	MNR	Manner
BNF	Benefactor	PRD	Predicative complement
DIR	Direction	PRP	Purpose or reason
DTV	Dative	PUT	Locative complement of *put*
EXT	Extent	SBJ	Subject
LGS	Logical subject	TMP	Temporal
LOC	Locative	VOC	Vocative
Labels inferred from constituent relations			
AMOD	Modifier of adjective or adverb	OPRD	Object predicate
CONJ	Conjunct	P	Punctuation
COORD	Coordination	PMOD	Modifier of preposition
DEP	Unclassified dependency	PRN	Parenthetical
EXTR	Extraposed element	PRT	Particle
GAP	Gapping	QMOD	Modifier of quantifier
IM	Infinitive marker	ROOT	Root
NMOD	Modifier of nominal	SUB	Subordinating conjunction
OBJ	Object or clausal complement	VC	Verb chain

Table 9: A list of the CoNLL dependency labels.

B.2 Stanford dependency labels

This appendix shows a list of the Stanford dependency labels. See de Marneffe and Manning (2008b) for more details about Stanford dependency labels.

Label	Description	Label	Description
ABBREV	Abbreviation modifier	NPADVMOD	Noun phrase as ADVMOD
ACOMP	Adjectival complement	NSUBJ	Nominal subject
ADVCL	Adverbial clause modifier	NSUBJPASS	Nominal subject (passive)
ADVMOD	Adverbial modifier	NUM	Numeric modifier
AGENT	Agent	NUMBER	Element of compound number
AMOD	Adjectival modifier	PARATAXIS	Parataxis
APPOS	Appositional modifier	PARTMOD	Participial modifier
ATTR	Attribute	PCOMP	Prepositional complement
AUX	Auxiliary	POBJ	Object of a preposition
AUXPASS	Auxiliary (passive)	POSS	Possession modifier
CC	Coordination	POSSESSIVE	Possessive modifier
CCOMP	Clausal complement	PRECONJ	Preconjunct
COMPLM	Complementizer	PREDET	Predeterminer
CONJ	Conjunct	PREP	Prepositional modifier
COP	Copula	PREPC	Prepositional clausal modifier
CSUBJ	Clausal subject	PRT	Phrasal verb particle
CSUBJPASS	Clausal subject (passive)	PUNCT	Punctuation
DEP	Dependent	PURPCL	Purpose clause modifier
DET	Determiner	QUANTMOD	Quantifier phrase modifier
DOBJ	Direct object	RCMOD	Relative clause modifier
EXPL	Expletive	REF	Referent
INFMOD	Infinitival modifier	REL	Relative
IOBJ	Indirect object	ROOT	Root
MARK	Marker	TMOD	Temporal modifier
MWE	Multi-word expression	XCOMP	Open clausal complement
NEG	Negation modifier	XSUBJ	Controlling subject
NN	Noun compound modifier		

Table 10: A list of the Stanford dependency labels.