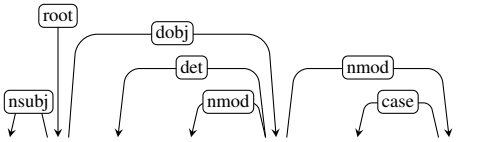
CHAPTER

14 Dependency Parsing

The focus of the two previous chapters has been on context-free grammars and their use in automatically generating constituent-based representations. Here we present another family of grammar formalisms called **dependency grammars** that are quite important in contemporary speech and language processing systems. In these formalisms, phrasal constituents and phrase-structure rules do not play a direct role. Instead, the syntactic structure of a sentence is described solely in terms of the words (or lemmas) in a sentence and an associated set of directed binary grammatical relations that hold among the words.

dependency grammars

The following diagram illustrates a dependency-style analysis using the standard graphical method favored in the dependency-parsing community.



(14.1)

Relations among the words are illustrated above the sentence with directed, labeled arcs from heads to dependents. We call this a **typed dependency structure** because the labels are drawn from a fixed inventory of grammatical relations. It also includes a *root* node that explicitly marks the root of the tree, the head of the entire structure.

typed dependency

Figure 14.1 shows the same dependency analysis as a tree alongside its corresponding phrase-structure analysis of the kind given in Chapter 12. Note the absence of nodes corresponding to phrasal constituents or lexical categories in the dependency parse; the internal structure of the dependency parse consists solely of directed relations between lexical items in the sentence. These relationships directly encode important information that is often buried in the more complex phrase-structure parses. For example, the arguments to the verb *prefer* are directly linked to it in the dependency structure, while their connection to the main verb is more distant in the phrase-structure tree. Similarly, *morning* and *Denver*, modifiers of *flight*, are linked to it directly in the dependency structure.

A major advantage of dependency grammars is their ability to deal with languages that are morphologically rich and have a relatively **free word order**. For example, word order in Czech can be much more flexible than in English; a grammatical *object* might occur before or after a *location adverbial*. A phrase-structure grammar would need a separate rule for each possible place in the parse tree where such an adverbial phrase could occur. A dependency-based approach would just have one link type representing this particular adverbial relation. Thus, a dependency grammar approach abstracts away from word order information, representing only the information that is necessary for the parse.

free word order

An additional practical motivation for a dependency-based approach is that the head-dependent relations provide an approximation to the semantic relationship be-

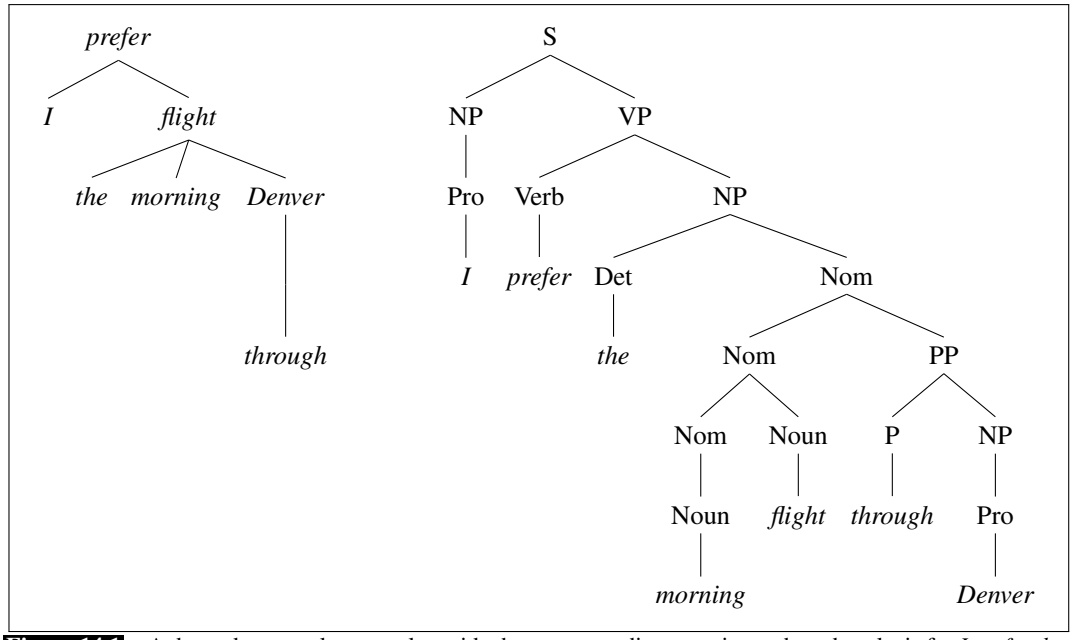


Figure 14.1 A dependency-style parse alongside the corresponding constituent-based analysis for *I prefer the morning flight through Denver.*

tween predicates and their arguments that makes them directly useful for many applications such as coreference resolution, question answering and information extraction. Constituent-based approaches to parsing provide similar information, but it often has to be distilled from the trees via techniques such as the head-finding rules discussed in Chapter 12.

In the following sections, we'll discuss in more detail the inventory of relations used in dependency parsing, as well as the formal basis for these dependency structures. We'll then move on to discuss the dominant families of algorithms that are used to automatically produce these structures. Finally, we'll discuss how to evaluate dependency parsers and point to some of the ways they are used in language processing applications.

14.1 Dependency Relations

grammatical relation

The traditional linguistic notion of **grammatical relation** provides the basis for the binary relations that comprise these dependency structures. The arguments to these relations consist of a **head** and a **dependent**. We've already discussed the notion of heads in Chapter 12 and Appendix C in the context of constituent structures. There, the head word of a constituent was the central organizing word of a larger constituent (e.g, the primary noun in a noun phrase, or verb in a verb phrase). The remaining words in the constituent are either direct, or indirect, dependents of their head. In dependency-based approaches, the head-dependent relationship is made explicit by directly linking heads to the words that are immediately dependent on them, bypassing the need for constituent structures.

head

dependent

grammatical function

In addition to specifying the head-dependent pairs, dependency grammars allow us to further classify the kinds of grammatical relations, or **grammatical function**,

Clausal Argument Relations	Description
NSUBJ	Nominal subject
DOBJ	Direct object
IOBJ	Indirect object
CCOMP	Clausal complement
XCOMP	Open clausal complement
Nominal Modifier Relations	**Description**
NMOD	Nominal modifier
AMOD	Adjectival modifier
NUMMOD	Numeric modifier
APPOS	Appositional modifier
DET	Determiner
CASE	Prepositions, postpositions and other case markers
Other Notable Relations	**Description**
CONJ	Conjunct
CC	Coordinating conjunction

Figure 14.2 Selected dependency relations from the Universal Dependency set. (de Marneffe et al., 2014)

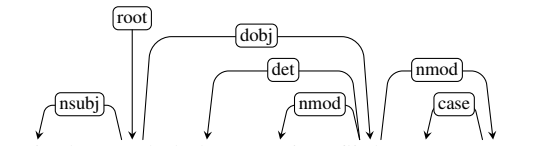
in terms of the role that the dependent plays with respect to its head. Familiar notions such as *subject*, *direct object* and *indirect object* are among the kind of relations we have in mind. In English these notions strongly correlate with, but by no means determine, both position in a sentence and constituent type and are therefore somewhat redundant with the kind of information found in phrase-structure trees. However, in more flexible languages the information encoded directly in these grammatical relations is critical since phrase-based constituent syntax provides little help.

Universal Dependencies

Not surprisingly, linguists have developed taxonomies of relations that go well beyond the familiar notions of subject and object. While there is considerable variation from theory to theory, there is enough commonality that efforts to develop a computationally useful standard are now possible. The **Universal Dependencies** project (Nivre et al., 2016) provides an inventory of dependency relations that are linguistically motivated, computationally useful, and cross-linguistically applicable. Fig. 14.2 shows a subset of the relations from this effort. Fig. 14.3 provides some example sentences illustrating selected relations.

The motivation for all of the relations in the Universal Dependency scheme is beyond the scope of this chapter, but the core set of frequently used relations can be broken into two sets: clausal relations that describe syntactic roles with respect to a predicate (often a verb), and modifier relations that categorize the ways that words that can modify their heads.

Consider the following example sentence:



(14.2)

The clausal relations NSUBJ and DOBJ identify the subject and direct object of the predicate *cancel*, while the NMOD, DET, and CASE relations denote modifiers of the nouns *flights* and *Houston*.

Relation	Examples with *head* and **dependent**
NSUBJ	**United** *canceled* the flight.
DOBJ	United *diverted* the **flight** to Reno.
	We *booked* her the first **flight** to Miami.
IOBJ	We *booked* **her** the flight to Miami.
NMOD	We took the **morning** *flight*.
AMOD	Book the **cheapest** *flight*.
NUMMOD	Before the storm JetBlue canceled **1000** *flights*.
APPOS	*United*, a **unit** of UAL, matched the fares.
DET	**The** *flight* was canceled.
	Which *flight* was delayed?
CONJ	We *flew* to Denver and **drove** to Steamboat.
CC	We flew to Denver **and** *drove* to Steamboat.
CASE	Book the flight **through** *Houston*.

Figure 14.3 Examples of core Universal Dependency relations.

14.2 Dependency Formalisms

In their most general form, the dependency structures we're discussing are simply directed graphs. That is, structures $G = (V,A)$ consisting of a set of vertices V, and a set of ordered pairs of vertices A, which we'll refer to as arcs.

For the most part we will assume that the set of vertices, V, corresponds exactly to the set of words in a given sentence. However, they might also correspond to punctuation, or when dealing with morphologically complex languages the set of vertices might consist of stems and affixes. The set of arcs, A, captures the head-dependent and grammatical function relationships between the elements in V.

Further constraints on these dependency structures are specific to the underlying grammatical theory or formalism. Among the more frequent restrictions are that the structures must be connected, have a designated root node, and be acyclic or planar. Of most relevance to the parsing approaches discussed in this chapter is the common, computationally-motivated, restriction to rooted trees. That is, a **dependency tree** is a directed graph that satisfies the following constraints:

dependency tree

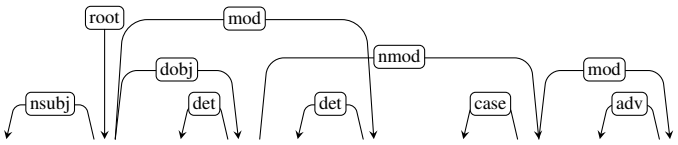
1. There is a single designated root node that has no incoming arcs.
2. With the exception of the root node, each vertex has exactly one incoming arc.
3. There is a unique path from the root node to each vertex in V.

Taken together, these constraints ensure that each word has a single head, that the dependency structure is connected, and that there is a single root node from which one can follow a unique directed path to each of the words in the sentence.

14.2.1 Projectivity

The notion of projectivity imposes an additional constraint that is derived from the order of the words in the input. An arc from a head to a dependent is said to be projective if there is a path from the head to every word that lies between the head and the dependent in the sentence. A dependency tree is then said to be projective if all the arcs that make it up are projective. All the dependency trees we've seen thus far have been projective. There are, however, many perfectly valid constructions which lead to non-projective trees, particularly in languages with a relatively flexible word order.

Consider the following example.



(14.3)

In this example, the arc from *flight* to its modifier *was* is non-projective since there is no path from *flight* to the intervening words *this* and *morning*. As we can see from this diagram, projectivity (and non-projectivity) can be detected in the way we've been drawing our trees. A dependency tree is projective if it can be drawn with no crossing edges. Here there is no way to link *flight* to its dependent *was* without crossing the arc that links *morning* to its head.

Our concern with projectivity arises from two related issues. First, the most widely used English dependency treebanks were automatically derived from phrase-structure treebanks through the use of head-finding rules (Chapter 12). The trees generated in such a fashion are guaranteed to be projective since they're generated from context-free grammars.

Second, there are computational limitations to the most widely used families of parsing algorithms. The transition-based approaches discussed in Section 14.4 can only produce projective trees, hence any sentences with non-projective structures will necessarily contain some errors. This limitation is one of the motivations for the more flexible graph-based parsing approach described in Section 14.5.

14.3 Dependency Treebanks

As with constituent-based methods, treebanks play a critical role in the development and evaluation of dependency parsers. Dependency treebanks have been created using similar approaches to those discussed in Chapter 12 — having human annotators directly generate dependency structures for a given corpus, or using automatic parsers to provide an initial parse and then having annotators hand correct those parsers. We can also use a deterministic process to translate existing constituent-based treebanks into dependency trees through the use of head rules.

For the most part, directly annotated dependency treebanks have been created for morphologically rich languages such as Czech, Hindi and Finnish that lend themselves to dependency grammar approaches, with the Prague Dependency Treebank (Bejček et al., 2013) for Czech being the most well-known effort. The major English dependency treebanks have largely been extracted from existing resources such as the Wall Street Journal sections of the Penn Treebank (Marcus et al., 1993). The more recent OntoNotes project (Hovy et al. 2006, Weischedel et al. 2011) extends this approach going beyond traditional news text to include conversational telephone speech, weblogs, usenet newsgroups, broadcasts, and talk shows in English, Chinese and Arabic.

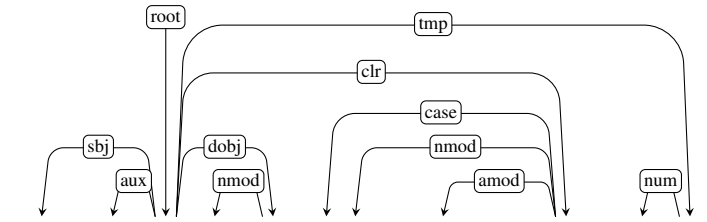
The translation process from constituent to dependency structures has two subtasks: identifying all the head-dependent relations in the structure and identifying the correct dependency relations for these relations. The first task relies heavily on the use of head rules discussed in Chapter 12 first developed for use in lexicalized probabilistic parsers (Magerman 1994, Collins 1999, Collins 2003). Here's a simple

and effective algorithm from Xia and Palmer (2001).

1. Mark the head child of each node in a phrase structure, using the appropriate head rules.
2. In the dependency structure, make the head of each non-head child depend on the head of the head-child.

When a phrase-structure parse contains additional information in the form of grammatical relations and function tags, as in the case of the Penn Treebank, these tags can be used to label the edges in the resulting tree. When applied to the parse tree in Fig. 14.4, this algorithm would produce the dependency structure in example 14.4.

(14.4)



The primary shortcoming of these extraction methods is that they are limited by the information present in the original constituent trees. Among the most important issues are the failure to integrate morphological information with the phrase-structure trees, the inability to easily represent non-projective structures, and the lack of internal structure to most noun-phrases, as reflected in the generally flat rules used in most treebank grammars. For these reasons, outside of English, most dependency treebanks are developed directly using human annotators.

14.4 Transition-Based Dependency Parsing

shift-reduce parsing

Our first approach to dependency parsing is motivated by a stack-based approach called **shift-reduce parsing** originally developed for analyzing programming languages (Aho and Ullman, 1972). This classic approach is simple and elegant, employing a context-free grammar, a stack, and a list of tokens to be parsed. Input tokens are successively shifted onto the stack and the top two elements of the stack are matched against the right-hand side of the rules in the grammar; when a match is found the matched elements are replaced on the stack (reduced) by the non-terminal from the left-hand side of the rule being matched. In adapting this approach for dependency parsing, we forgo the explicit use of a grammar and alter the reduce operation so that instead of adding a non-terminal to a parse tree, it introduces a dependency relation between a word and its head. More specifically, the reduce action is replaced with two possible actions: assert a head-dependent relation between the word at the top of the stack and the word below it, or vice versa. Figure 14.5 illustrates the basic operation of such a parser.

configuration

A key element in transition-based parsing is the notion of a **configuration** which consists of a stack, an input buffer of words, or tokens, and a set of relations representing a dependency tree. Given this framework, the parsing process consists of a sequence of transitions through the space of possible configurations. The goal of

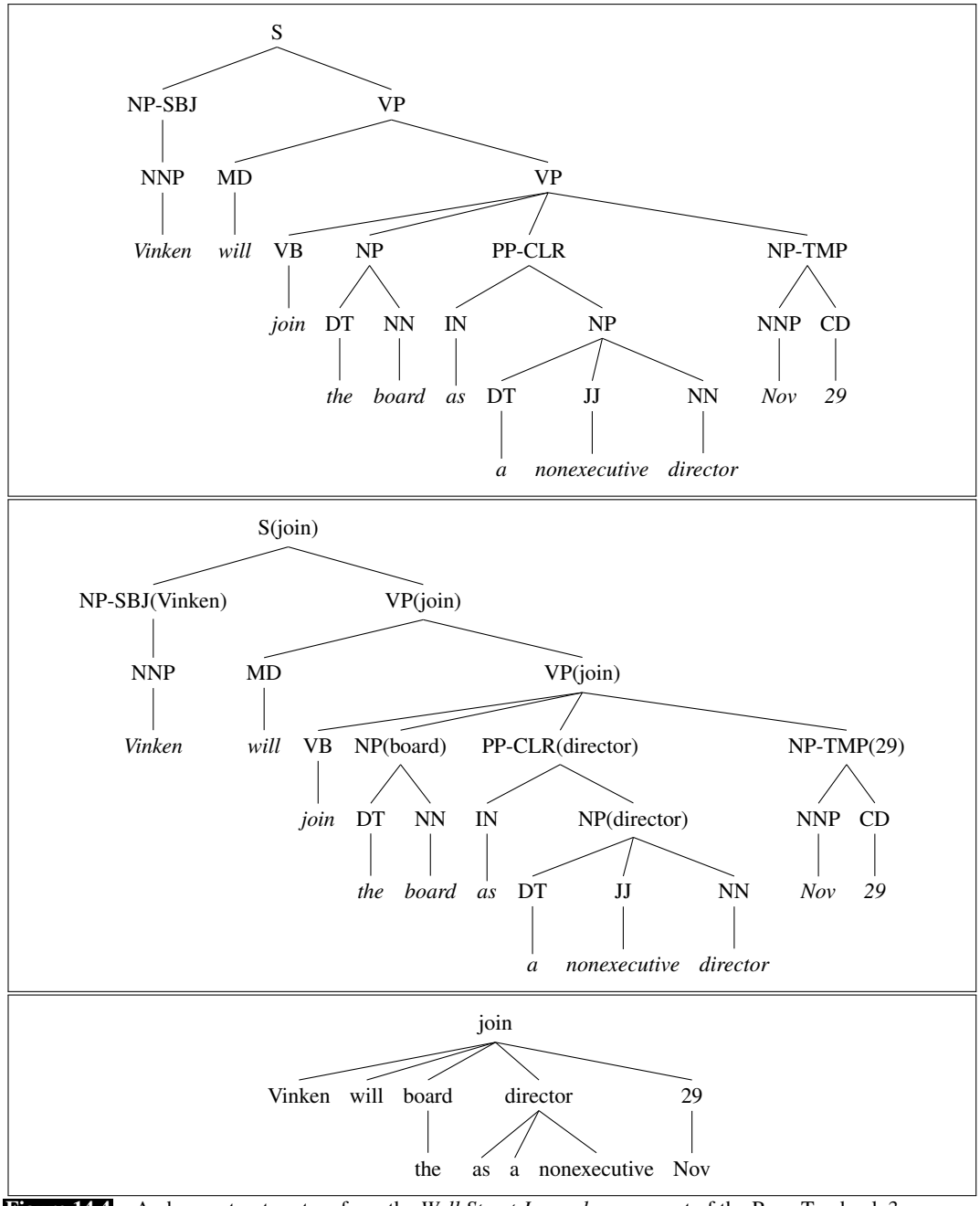


Figure 14.4 A phrase-structure tree from the *Wall Street Journal* component of the Penn Treebank 3.

this process is to find a final configuration where all the words have been accounted for and an appropriate dependency tree has been synthesized.

To implement such a search, we'll define a set of transition operators, which when applied to a configuration produce new configurations. Given this setup, we can view the operation of a parser as a search through a space of configurations for a sequence of transitions that leads from a start state to a desired goal state. At the start of this process we create an initial configuration in which the stack contains the ROOT node, the word list is initialized with the set of the words or lemmatized tokens

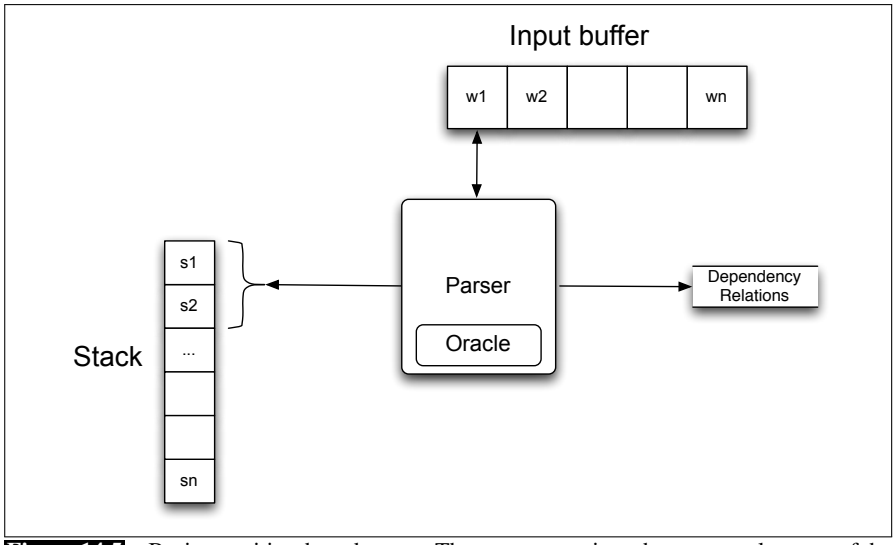


Figure 14.5 Basic transition-based parser. The parser examines the top two elements of the stack and selects an action based on consulting an oracle that examines the current configuration.

in the sentence, and an empty set of relations is created to represent the parse. In the final goal state, the stack and the word list should be empty, and the set of relations will represent the final parse.

In the standard approach to transition-based parsing, the operators used to produce new configurations are surprisingly simple and correspond to the intuitive actions one might take in creating a dependency tree by examining the words in a single pass over the input from left to right (Covington, 2001):

- Assign the current word as the head of some previously seen word,
- Assign some previously seen word as the head of the current word,
- Or postpone doing anything with the current word, adding it to a store for later processing.

To make these actions more precise, we'll create three transition operators that will operate on the top two elements of the stack:

- LEFTARC: Assert a head-dependent relation between the word at the top of the stack and the word directly beneath it; remove the lower word from the stack.
- RIGHTARC: Assert a head-dependent relation between the second word on the stack and the word at the top; remove the word at the top of the stack;
- SHIFT: Remove the word from the front of the input buffer and push it onto the stack.

arc standard

This particular set of operators implements what is known as the **arc standard** approach to transition-based parsing (Covington 2001, Nivre 2003). There are two notable characteristics to this approach: the transition operators only assert relations between elements at the top of the stack, and once an element has been assigned its head it is removed from the stack and is not available for further processing. As we'll see, there are alternative transition systems which demonstrate different parsing behaviors, but the arc standard approach is quite effective and is simple to implement.

To assure that these operators are used properly we'll need to add some preconditions to their use. First, since, by definition, the ROOT node cannot have any incoming arcs, we'll add the restriction that the LEFTARC operator cannot be applied when ROOT is the second element of the stack. Second, both reduce operators require two elements to be on the stack to be applied. Given these transition operators and preconditions, the specification of a transition-based parser is quite simple. Fig. 14.6 gives the basic algorithm.

```
function DEPENDENCYPARSE(words) returns dependency tree

  state ← {[root], [words], [] }  ; initial configuration
  while state not final
    t ← ORACLE(state)         ; choose a transition operator to apply
    state ← APPLY(t, state)   ; apply it, creating a new state
  return state
```

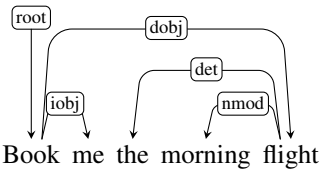
Figure 14.6 A generic transition-based dependency parser

At each step, the parser consults an oracle (we'll come back to this shortly) that provides the correct transition operator to use given the current configuration. It then applies that operator to the current configuration, producing a new configuration. The process ends when all the words in the sentence have been consumed and the ROOT node is the only element remaining on the stack.

The efficiency of transition-based parsers should be apparent from the algorithm. The complexity is linear in the length of the sentence since it is based on a single left to right pass through the words in the sentence. More specifically, each word must first be shifted onto the stack and then later reduced.

Note that unlike the dynamic programming and search-based approaches discussed in Chapters 12 and 13, this approach is a straightforward greedy algorithm — the oracle provides a single choice at each step and the parser proceeds with that choice, no other options are explored, no backtracking is employed, and a single parse is returned in the end.

Figure 14.7 illustrates the operation of the parser with the sequence of transitions leading to a parse for the following example.



(14.5)

Let's consider the state of the configuration at Step 2, after the word *me* has been pushed onto the stack.

Stack	Word List	Relations
[root, book, me]	[the, morning, flight]	

The correct operator to apply here is RIGHTARC which assigns *book* as the head of *me* and pops *me* from the stack resulting in the following configuration.

Stack	Word List	Relations
[root, book]	[the, morning, flight]	(book → me)

Step	Stack	Word List	Action	Relation Added
0	[root]	[book, me, the, morning, flight]	SHIFT	
1	[root, book]	[me, the, morning, flight]	SHIFT	
2	[root, book, me]	[the, morning, flight]	RIGHTARC	(book → me)
3	[root, book]	[the, morning, flight]	SHIFT	
4	[root, book, the]	[morning, flight]	SHIFT	
5	[root, book, the, morning]	[flight]	SHIFT	
6	[root, book, the, morning, flight]	[]	LEFTARC	(morning ← flight)
7	[root, book, the, flight]	[]	LEFTARC	(the ← flight)
8	[root, book, flight]	[]	RIGHTARC	(book → flight)
9	[root, book]	[]	RIGHTARC	(root → book)
10	[root]	[]	Done	

Figure 14.7 Trace of a transition-based parse.

After several subsequent applications of the SHIFT and LEFTARC operators, the configuration in Step 6 looks like the following:

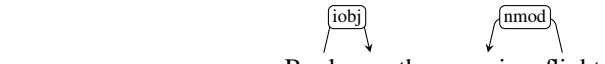
Stack	Word List	Relations
[root, book, the, morning, flight]	[]	(book → me)

Here, all the remaining words have been passed onto the stack and all that is left to do is to apply the appropriate reduce operators. In the current configuration, we employ the LEFTARC operator resulting in the following state.

Stack	Word List	Relations
[root, book, the, flight]	[]	(book → me) (morning ← flight)

At this point, the parse for this sentence consists of the following structure.

(14.6)



There are several important things to note when examining sequences such as the one in Figure 14.7. First, the sequence given is not the only one that might lead to a reasonable parse. In general, there may be more than one path that leads to the same result, and due to ambiguity, there may be other transition sequences that lead to different equally valid parses.

Second, we are assuming that the oracle always provides the correct operator at each point in the parse — an assumption that is unlikely to be true in practice. As a result, given the greedy nature of this algorithm, incorrect choices will lead to incorrect parses since the parser has no opportunity to go back and pursue alternative choices. Section 14.4.2 will introduce several techniques that allow transition-based approaches to explore the search space more fully.

Finally, for simplicity, we have illustrated this example without the labels on the dependency relations. To produce labeled trees, we can parameterize the LEFTARC and RIGHTARC operators with dependency labels, as in LEFTARC(NSUBJ) or RIGHTARC(DOBJ). This is equivalent to expanding the set of transition operators from our original set of three to a set that includes LEFTARC and RIGHTARC operators for each relation in the set of dependency relations being used, plus an additional one for the SHIFT operator. This, of course, makes the job of the oracle more difficult since it now has a much larger set of operators from which to choose.

14.4.1 Creating an Oracle

State-of-the-art transition-based systems use supervised machine learning methods to train classifiers that play the role of the oracle. Given appropriate training data, these methods learn a function that maps from configurations to transition operators.

As with all supervised machine learning methods, we will need access to appropriate training data and we will need to extract features useful for characterizing the decisions to be made. The source for this training data will be representative treebanks containing dependency trees. The features will consist of many of the same features we encountered in Chapter 8 for part-of-speech tagging, as well as those used in Appendix C for statistical parsing models.

Generating Training Data

Let's revisit the oracle from the algorithm in Fig. 14.6 to fully understand the learning problem. The oracle takes as input a configuration and returns as output a transition operator. Therefore, to train a classifier, we will need configurations paired with transition operators (i.e., LEFTARC, RIGHTARC, or SHIFT). Unfortunately, treebanks pair entire sentences with their corresponding trees, and therefore they don't directly provide what we need.

To generate the required training data, we will employ the oracle-based parsing algorithm in a clever way. We will supply our oracle with the training sentences to be parsed *along with* their corresponding reference parses from the treebank. To produce training instances, we will then *simulate* the operation of the parser by running the algorithm and relying on a new **training oracle** to give us correct transition operators for each successive configuration.

training oracle

To see how this works, let's first review the operation of our parser. It begins with a default initial configuration where the stack contains the ROOT, the input list is just the list of words, and the set of relations is empty. The LEFTARC and RIGHTARC operators each add relations between the words at the top of the stack to the set of relations being accumulated for a given sentence. Since we have a gold-standard reference parse for each training sentence, we know which dependency relations are valid for a given sentence. Therefore, we can use the reference parse to guide the selection of operators as the parser steps through a sequence of configurations.

To be more precise, given a reference parse and a configuration, the training oracle proceeds as follows:

- Choose LEFTARC if it produces a correct head-dependent relation given the reference parse and the current configuration,
- Otherwise, choose RIGHTARC if (1) it produces a correct head-dependent relation given the reference parse and (2) all of the dependents of the word at the top of the stack have already been assigned,
- Otherwise, choose SHIFT.

The restriction on selecting the RIGHTARC operator is needed to ensure that a word is not popped from the stack, and thus lost to further processing, before all its dependents have been assigned to it.

More formally, during training the oracle has access to the following information:

- A current configuration with a stack S and a set of dependency relations R_c
- A reference parse consisting of a set of vertices V and a set of dependency relations R_p

Step	Stack	Word List	Predicted Action
0	[root]	[book, the, flight, through, houston]	SHIFT
1	[root, book]	[the, flight, through, houston]	SHIFT
2	[root, book, the]	[flight, through, houston]	SHIFT
3	[root, book, the, flight]	[through, houston]	LEFTARC
4	[root, book, flight]	[through, houston]	SHIFT
5	[root, book, flight, through]	[houston]	SHIFT
6	[root, book, flight, through, houston]	[]	LEFTARC
7	[root, book, flight, houston]	[]	RIGHTARC
8	[root, book, flight]	[]	RIGHTARC
9	[root, book]	[]	RIGHTARC
10	[root]	[]	Done

Figure 14.8 Generating training items consisting of configuration/predicted action pairs by simulating a parse with a given reference parse.

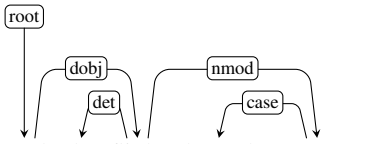
Given this information, the oracle chooses transitions as follows:

LEFTARC(r): **if** $(S_1\ r\ S_2) \in R_p$

RIGHTARC(r): **if** $(S_2\ r\ S_1) \in R_p$ **and** $\forall r', w\ s.t. (S_1\ r'\ w) \in R_p$ **then** $(S_1\ r'\ w) \in R_c$

SHIFT: **otherwise**

Let's walk through the steps of this process with the following example as shown in Fig. 14.8.



(14.7)

At Step 1, LEFTARC is not applicable in the initial configuration since it asserts a relation, (root ← book), not in the reference answer; RIGHTARC does assert a relation contained in the final answer (root → book), however *book* has not been attached to any of its dependents yet, so we have to defer, leaving SHIFT as the only possible action. The same conditions hold in the next two steps. In step 3, LEFTARC is selected to link *the* to its head.

Now consider the situation in Step 4.

Stack	Word buffer	Relations
[root, book, flight]	[through, Houston]	(the ← flight)

Here, we might be tempted to add a dependency relation between *book* and *flight*, which is present in the reference parse. But doing so now would prevent the later attachment of *Houston* since *flight* would have been removed from the stack. Fortunately, the precondition on choosing RIGHTARC prevents this choice and we're again left with SHIFT as the only viable option. The remaining choices complete the set of operators needed for this example.

To recap, we derive appropriate training instances consisting of configuration-transition pairs from a treebank by simulating the operation of a parser in the context of a reference dependency tree. We can deterministically record correct parser actions at each step as we progress through each training example, thereby creating the training set we require.

Features

Having generated appropriate training instances (configuration-transition pairs), we need to extract useful features from the configurations so we can train classifiers. The features that are used to train transition-based systems vary by language, genre, and the kind of classifier being employed. For example, morphosyntactic features such as case marking on subjects or direct objects may be more or less important depending on the language being processed. That said, the basic features that we have already seen with part-of-speech tagging and partial parsing have proven to be useful in training dependency parsers across a wide range of languages. Word forms, lemmas and parts of speech are all powerful features, as are the head, and dependency relation to the head.

In the transition-based parsing framework, such features need to be extracted from the configurations that make up the training data. Recall that configurations consist of three elements: the stack, the buffer and the current set of relations. In principle, any property of any or all of these elements can be represented as features in the usual way for training. However, to avoid sparsity and encourage generalization, it is best to focus the learning algorithm on the most useful aspects of decision making at each point in the parsing process. The focus of feature extraction for transition-based parsing is, therefore, on the top levels of the stack, the words near the front of the buffer, and the dependency relations already associated with any of those elements.

feature template

By combining simple features, such as word forms or parts of speech, with specific locations in a configuration, we can employ the notion of a **feature template** that we've already encountered with sentiment analysis and part-of-speech tagging. Feature templates allow us to automatically generate large numbers of specific features from a training set. As an example, consider the following feature templates that are based on single positions in a configuration.

$$\begin{aligned} &\langle s_1.w, op\rangle, \langle s_2.w, op\rangle \langle s_1.t, op\rangle, \langle s_2.t, op\rangle \\ &\langle b_1.w, op\rangle, \langle b_1.t, op\rangle \langle s_1.wt, op\rangle \end{aligned} \tag{14.8}$$

In these examples, individual features are denoted as $location.property$, where s denotes the stack, b the word buffer, and r the set of relations. Individual properties of locations include w for word forms, l for lemmas, and t for part-of-speech. For example, the feature corresponding to the word form at the top of the stack would be denoted as $s_1.w$, and the part of speech tag at the front of the buffer $b_1.t$. We can also combine individual features via concatenation into more specific features that may prove useful. For example, the feature designated by $s_1.wt$ represents the word form concatenated with the part of speech of the word at the top of the stack. Finally, op stands for the transition operator for the training example in question (i.e., the label for the training instance).

Let's consider the simple set of single-element feature templates given above in the context of the following intermediate configuration derived from a training oracle for Example 14.2.

Stack	Word buffer	Relations
[root, canceled, flights]	[to Houston]	(canceled → United)
		(flights → morning)
		(flights → the)

The correct transition here is SHIFT (you should convince yourself of this before

proceeding). The application of our set of feature templates to this configuration would result in the following set of instantiated features.

$$\begin{aligned}
\langle s_1.w = flights, op = shift \rangle \\
\langle s_2.w = canceled, op = shift \rangle \\
\langle s_1.t = NNS, op = shift \rangle \\
\langle s_2.t = VBD, op = shift \rangle \\
\langle b_1.w = to, op = shift \rangle \\
\langle b_1.t = TO, op = shift \rangle \\
\langle s_1.wt = flightsNNS, op = shift \rangle
\end{aligned} \tag{14.9}$$

Given that the left and right arc transitions operate on the top two elements of the stack, features that *combine* properties from these positions are even more useful. For example, a feature like $s_1.t \circ s_2.t$ concatenates the part of speech tag of the word at the top of the stack with the tag of the word beneath it.

$$\langle s_1.t \circ s_2.t = NNSVBD, op = shift \rangle \tag{14.10}$$

Not surprisingly, if two properties are useful then three or more should be even better. Figure 14.9 gives a baseline set of feature templates that have been employed (Zhang and Clark 2008, Huang and Sagae 2010, Zhang and Nivre 2011).

Note that some of these features make use of *dynamic* features — features such as head words and dependency relations that have been predicted at earlier steps in the parsing process, as opposed to features that are derived from static properties of the input.

Source	Feature templates		
One word	$s_1.w$	$s_1.t$	$s_1.wt$
	$s_2.w$	$s_2.t$	$s_2.wt$
	$b_1.w$	$b_1.w$	$b_0.wt$
Two word	$s_1.w \circ s_2.w$	$s_1.t \circ s_2.t$	$s_1.t \circ b_1.w$
	$s_1.t \circ s_2.wt$	$s_1.w \circ s_2.w \circ s_2.t$	$s_1.w \circ s_1.t \circ s_2.t$
	$s_1.w \circ s_1.t \circ s_2.t$	$s_1.w \circ s_1.t$	

Figure 14.9 Standard feature templates for training transition-based dependency parsers. In the template specifications s_n refers to a location on the stack, b_n refers to a location in the word buffer, w refers to the wordform of the input, and t refers to the part of speech of the input.

Learning

Over the years, the dominant approaches to training transition-based dependency parsers have been multinomial logistic regression and support vector machines, both of which can make effective use of large numbers of sparse features of the kind described in the last section. More recently, neural network, or deep learning, approaches of the kind described in Chapter 8 have been applied successfully to transition-based parsing (Chen and Manning, 2014). These approaches eliminate the need for complex, hand-crafted features and have been particularly effective at overcoming the data sparsity issues normally associated with training transition-based parsers.

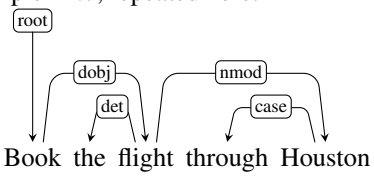
14.4.2 Advanced Methods in Transition-Based Parsing

The basic transition-based approach can be elaborated in a number of ways to improve performance by addressing some of the most obvious flaws in the approach.

Alternative Transition Systems

arc eager

The arc-standard transition system described above is only one of many possible systems. A frequently used alternative is the **arc eager** transition system. The arc eager approach gets its name from its ability to assert rightward relations much sooner than in the arc standard approach. To see this, let's revisit the arc standard trace of Example 14.7, repeated here.



Consider the dependency relation between *book* and *flight* in this analysis. As is shown in Fig. 14.8, an arc-standard approach would assert this relation at Step 8, despite the fact that *book* and *flight* first come together on the stack much earlier at Step 4. The reason this relation can't be captured at this point is due to the presence of the postnominal modifier *through Houston*. In an arc-standard approach, dependents are removed from the stack as soon as they are assigned their heads. If *flight* had been assigned *book* as its head in Step 4, it would no longer be available to serve as the head of *Houston*.

While this delay doesn't cause any issues in this example, in general the longer a word has to wait to get assigned its head the more opportunities there are for something to go awry. The arc-eager system addresses this issue by allowing words to be attached to their heads as early as possible, before all the subsequent words dependent on them have been seen. This is accomplished through minor changes to the LEFTARC and RIGHTARC operators and the addition of a new REDUCE operator.

- LEFTARC: Assert a head-dependent relation between the word at the front of the input buffer and the word at the top of the stack; pop the stack.
- RIGHTARC: Assert a head-dependent relation between the word on the top of the stack and the word at front of the input buffer; shift the word at the front of the input buffer to the stack.
- SHIFT: Remove the word from the front of the input buffer and push it onto the stack.
- REDUCE: Pop the stack.

The LEFTARC and RIGHTARC operators are applied to the top of the stack and the front of the input buffer, instead of the top two elements of the stack as in the arc-standard approach. The RIGHTARC operator now moves the dependent to the stack from the buffer rather than removing it, thus making it available to serve as the head of following words. The new REDUCE operator removes the top element from the stack. Together these changes permit a word to be eagerly assigned its head and still allow it to serve as the head for later dependents. The trace shown in Fig. 14.10 illustrates the new decision sequence for this example.

In addition to demonstrating the arc-eager transition system, this example demonstrates the power and flexibility of the overall transition-based approach. We were able to swap in a new transition system without having to make any changes to the

Step	Stack	Word List	Action	Relation Added
0	[root]	[book, the, flight, through, houston]	RIGHTARC	(root → book)
1	[root, book]	[the, flight, through, houston]	SHIFT	
2	[root, book, the]	[flight, through, houston]	LEFTARC	(the ← flight)
3	[root, book]	[flight, through, houston]	RIGHTARC	(book → flight)
4	[root, book, flight]	[through, houston]	SHIFT	
5	[root, book, flight, through]	[houston]	LEFTARC	(through ← houston)
6	[root, book, flight]	[houston]	RIGHTARC	(flight → houston)
7	[root, book, flight, houston]	[]	REDUCE	
8	[root, book, flight]	[]	REDUCE	
9	[root, book]	[]	REDUCE	
10	[root]	[]	Done	

Figure 14.10 A processing trace of *Book the flight through Houston* using the arc-eager transition operators.

underlying parsing algorithm. This flexibility has led to the development of a diverse set of transition systems that address different aspects of syntax and semantics including: assigning part of speech tags (Choi and Palmer, 2011a), allowing the generation of non-projective dependency structures (Nivre, 2009), assigning semantic roles (Choi and Palmer, 2011b), and parsing texts containing multiple languages (Bhat et al., 2017).

Beam Search

The computational efficiency of the transition-based approach discussed earlier derives from the fact that it makes a single pass through the sentence, greedily making decisions without considering alternatives. Of course, this is also the source of its greatest weakness – once a decision has been made it can not be undone, even in the face of overwhelming evidence arriving later in a sentence. Another approach is to systematically explore alternative decision sequences, selecting the best among those alternatives. The key problem for such a search is to manage the large number of potential sequences. **Beam search** accomplishes this by combining a breadth-first search strategy with a heuristic filter that prunes the search frontier to stay within a fixed-size **beam width**.

beam search

beam width

In applying beam search to transition-based parsing, we'll elaborate on the algorithm given in Fig. 14.6. Instead of choosing the single best transition operator at each iteration, we'll apply all applicable operators to each state on an agenda and then score the resulting configurations. We then add each of these new configurations to the frontier, subject to the constraint that there has to be room within the beam. As long as the size of the agenda is within the specified beam width, we can add new configurations to the agenda. Once the agenda reaches the limit, we only add new configurations that are better than the worst configuration on the agenda (removing the worst element so that we stay within the limit). Finally, to insure that we retrieve the best possible state on the agenda, the while loop continues as long as there are non-final states on the agenda.

The beam search approach requires a more elaborate notion of scoring than we used with the greedy algorithm. There, we assumed that a classifier trained using supervised machine learning would serve as an oracle, selecting the best transition operator based on features extracted from the current configuration. Regardless of the specific learning approach, this choice can be viewed as assigning a score to all the possible transitions and picking the best one.

$$\hat{T}(c) = argmaxScore(t,c)$$

With a beam search we are now searching through the space of decision sequences, so it makes sense to base the score for a configuration on its entire history. More specifically, we can define the score for a new configuration as the score of its predecessor plus the score of the operator used to produce it.

$$\begin{aligned} \mathit{ConfigScore}(c_0) &= 0.0 \\ \mathit{ConfigScore}(c_i) &= \mathit{ConfigScore}(c_{i-1}) + \mathit{Score}(t_i, c_{i-1}) \end{aligned}$$

This score is used both in filtering the agenda and in selecting the final answer. The new beam search version of transition-based parsing is given in Fig. 14.11.

```
function DEPENDENCYBEAMPARSE(words, width) returns dependency tree

  state ← {[root], [words], [], 0.0}     ;initial configuration
  agenda ← ⟨state⟩                       ;initial agenda

  while agenda contains non-final states
    newagenda ← ⟨⟩
    for each state ∈ agenda do
      for all {t | t ∈ VALIDOPERATORS(state)} do
        child ← APPLY(t, state)
        newagenda ← ADDTOBEAM(child, newagenda, width)
    agenda ← newagenda
  return BESTOF(agenda)

function ADDTOBEAM(state, agenda, width) returns updated agenda

  if LENGTH(agenda) < width then
    agenda ← INSERT(state, agenda)
  else if SCORE(state) > SCORE(WORSTOF(agenda))
    agenda ← REMOVE(WORSTOF(agenda))
    agenda ← INSERT(state, agenda)
  return agenda
```

Figure 14.11 Beam search applied to transition-based dependency parsing.

14.5 Graph-Based Dependency Parsing

Graph-based approaches to dependency parsing search through the space of possible trees for a given sentence for a tree (or trees) that maximize some score. These methods encode the search space as directed graphs and employ methods drawn from graph theory to search the space for optimal solutions. More formally, given a sentence S we're looking for the best dependency tree in $\mathscr{G}_s$, the space of all possible trees for that sentence, that maximizes some score.

$$\hat{T}(S) = \underset{t \in \mathscr{G}_S}{\operatorname{argmax}}\, score(t, S)$$

As with the probabilistic approaches to context-free parsing discussed in Appendix C, the overall score for a tree can be viewed as a function of the scores of the parts of the tree. The focus of this section is on **edge-factored** approaches where the

edge-factored

score for a tree is based on the scores of the edges that comprise the tree.

$$score(t,S) = \sum_{e \in t} score(e)$$

There are several motivations for the use of graph-based methods. First, unlike transition-based approaches, these methods are capable of producing non-projective trees. Although projectivity is not a significant issue for English, it is definitely a problem for many of the world's languages. A second motivation concerns parsing accuracy, particularly with respect to longer dependencies. Empirically, transition-based methods have high accuracy on shorter dependency relations but accuracy declines significantly as the distance between the head and dependent increases (McDonald and Nivre, 2011). Graph-based methods avoid this difficulty by scoring entire trees, rather than relying on greedy local decisions.

maximum spanning tree

The following section examines a widely-studied approach based on the use of a **maximum spanning tree** (MST) algorithm for weighted, directed graphs. We then discuss features that are typically used to score trees, as well as the methods used to train the scoring models.

14.5.1 Parsing

The approach described here uses an efficient greedy algorithm to search for optimal spanning trees in directed graphs. Given an input sentence, it begins by constructing a fully-connected, weighted, directed graph where the vertices are the input words and the directed edges represent *all possible* head-dependent assignments. An additional ROOT node is included with outgoing edges directed at all of the other vertices. The weights in the graph reflect the score for each possible head-dependent relation as provided by a model generated from training data. Given these weights, a maximum spanning tree of this graph emanating from the ROOT represents the preferred dependency parse for the sentence. A directed graph for the example *Book that flight* is shown in Fig. 14.12, with the maximum spanning tree corresponding to the desired parse shown in blue. For ease of exposition, we'll focus here on unlabeled dependency parsing. Graph-based approaches to labeled parsing are discussed in Section 14.5.3.

Before describing the algorithm it's useful to consider two intuitions about directed graphs and their spanning trees. The first intuition begins with the fact that every vertex in a spanning tree has exactly one incoming edge. It follows from this that every *connected component* of a spanning tree will also have one incoming edge. The second intuition is that the absolute values of the edge scores are not critical to determining its maximum spanning tree. Instead, it is the relative weights of the edges entering each vertex that matters. If we were to subtract a constant amount from each edge entering a given vertex it would have no impact on the choice of the maximum spanning tree since every possible spanning tree would decrease by exactly the same amount.

The first step of the algorithm itself is quite straightforward. For each vertex in the graph, an incoming edge (representing a possible head assignment) with the highest score is chosen. If the resulting set of edges produces a spanning tree then we're done. More formally, given the original fully-connected graph $G = (V,E)$, a subgraph $T = (V,F)$ is a spanning tree if it has no cycles and each vertex (other than the root) has exactly one edge entering it. If the greedy selection process produces such a tree then it is the best possible one.

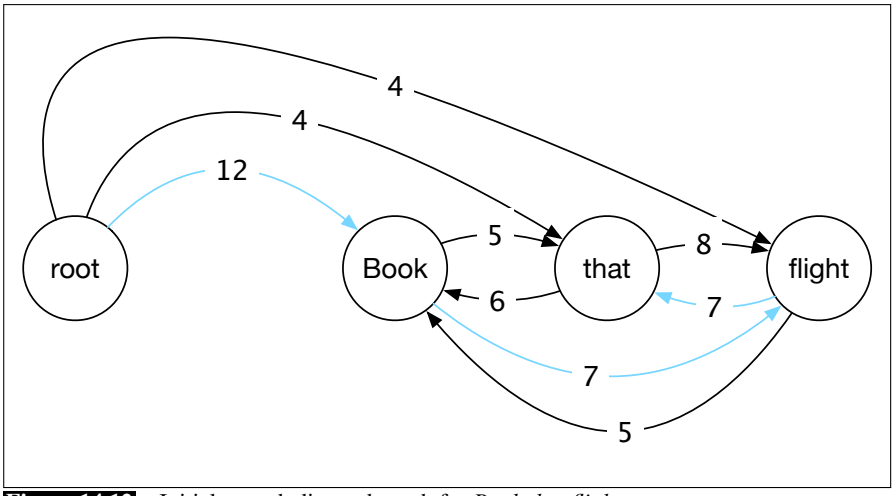


Figure 14.12 Initial rooted, directed graph for *Book that flight*.

Unfortunately, this approach doesn't always lead to a tree since the set of edges selected may contain cycles. Fortunately, in yet another case of multiple discovery, there is a straightforward way to eliminate cycles generated during the greedy selection phase. Chu and Liu (1965) and Edmonds (1967) independently developed an approach that begins with greedy selection and follows with an elegant recursive cleanup phase that eliminates cycles.

The cleanup phase begins by adjusting all the weights in the graph by subtracting the score of the maximum edge entering each vertex from the score of all the edges entering that vertex. This is where the intuitions mentioned earlier come into play. We have scaled the values of the edges so that the weight of the edges in the cycle have no bearing on the weight of *any* of the possible spanning trees. Subtracting the value of the edge with maximum weight from each edge entering a vertex results in a weight of zero for all of the edges selected during the greedy selection phase, *including all of the edges involved in the cycle*.

Having adjusted the weights, the algorithm creates a new graph by selecting a cycle and collapsing it into a single new node. Edges that enter or leave the cycle are altered so that they now enter or leave the newly collapsed node. Edges that do not touch the cycle are included and edges within the cycle are dropped.

Now, if we knew the maximum spanning tree of this new graph, we would have what we need to eliminate the cycle. The edge of the maximum spanning tree directed towards the vertex representing the collapsed cycle tells us which edge to delete to eliminate the cycle. How do we find the maximum spanning tree of this new graph? We recursively apply the algorithm to the new graph. This will either result in a spanning tree or a graph with a cycle. The recursions can continue as long as cycles are encountered. When each recursion completes we expand the collapsed vertex, restoring all the vertices and edges from the cycle *with the exception of the single edge to be deleted*.

Putting all this together, the maximum spanning tree algorithm consists of greedy edge selection, re-scoring of edge costs and a recursive cleanup phase when needed. The full algorithm is shown in Fig. 14.13.

Fig. 14.14 steps through the algorithm with our *Book that flight* example. The first row of the figure illustrates greedy edge selection with the edges chosen shown in blue (corresponding to the set F in the algorithm). This results in a cycle between

```
function MAXSPANNINGTREE(G=(V,E), root, score) returns spanning tree

  F ← []
  T' ← []
  score' ← []
  for each v ∈ V do
    bestInEdge ← argmax_{e=(u,v)∈ E} score[e]
    F ← F ∪ bestInEdge
    for each e=(u,v) ∈ E do
      score'[e] ← score[e] − score[bestInEdge]

  if T=(V,F) is a spanning tree then return it
  else
    C ← a cycle in F
    G' ← CONTRACT(G, C)
    T' ← MAXSPANNINGTREE(G', root, score')
    T ← EXPAND(T', C)
    return T

function CONTRACT(G, C) returns contracted graph

function EXPAND(T, C) returns expanded graph
```

Figure 14.13 The Chu-Liu Edmonds algorithm for finding a maximum spanning tree in a weighted directed graph.

that and *flight*. The scaled weights using the maximum value entering each node are shown in the graph to the right.

Collapsing the cycle between *that* and *flight* to a single node (labelled *tf*) and recursing with the newly scaled costs is shown in the second row. The greedy selection step in this recursion yields a spanning tree that links *root* to *book*, as well as an edge that links *book* to the contracted node. Expanding the contracted node, we can see that this edge corresponds to the edge from *book* to *flight* in the original graph. This in turn tells us which edge to drop to eliminate the cycle

On arbitrary directed graphs, this version of the CLE algorithm runs in $O(mn)$ time, where m is the number of edges and n is the number of nodes. Since this particular application of the algorithm begins by constructing a fully connected graph $m = n^2$ yielding a running time of $O(n^3)$. Gabow et al. (1986) present a more efficient implementation with a running time of $O(m + nlogn)$.

14.5.2 Features and Training

Given a sentence, S, and a candidate tree, T, edge-factored parsing models reduce the score for the tree to a sum of the scores of the edges that comprise the tree.

$$score(S,T) = \sum_{e \in T} score(S,e)$$

Each edge score can, in turn, be reduced to a weighted sum of features extracted from it.

$$score(S,e) = \sum_{i=1}^{N} w_i f_i(S,e)$$

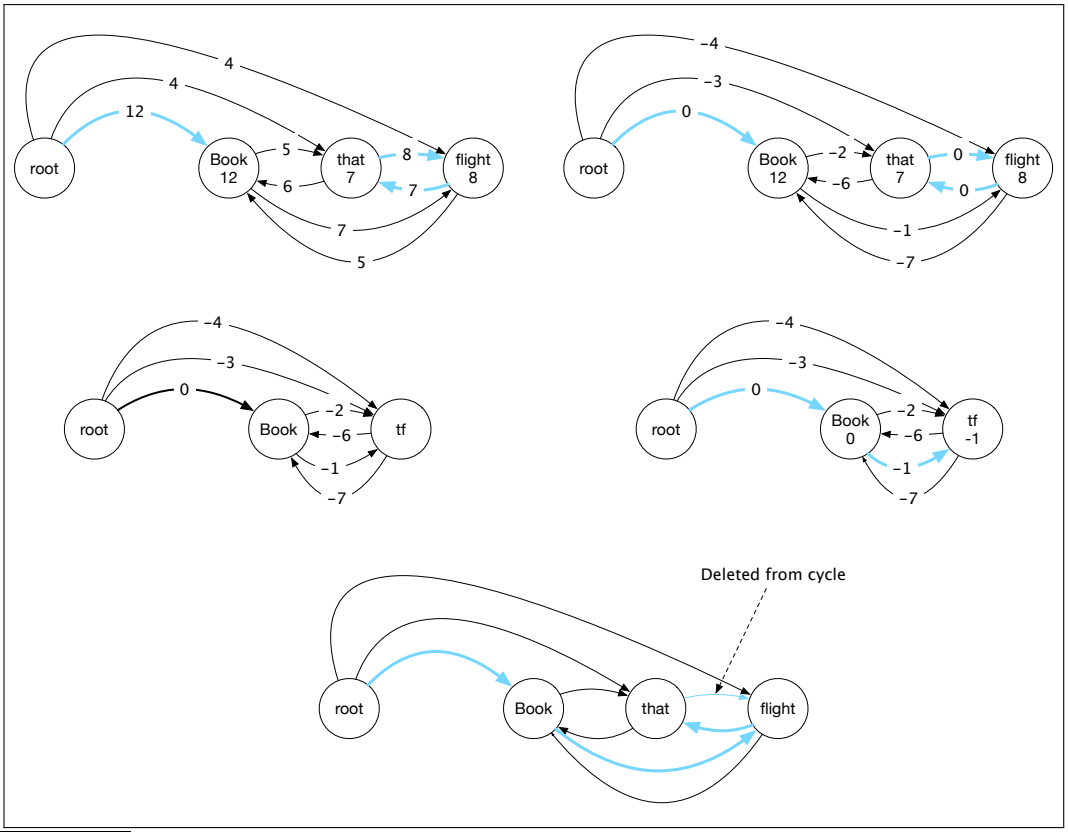


Figure 14.14 Chu-Liu-Edmonds graph-based example for *Book that flight*

Or more succinctly.

$$score(S,e) \;=\; w \cdot f$$

Given this formulation, we are faced with two problems in training our parser: identifying relevant features and finding the weights used to score those features.

The features used to train edge-factored models mirror those used in training transition-based parsers (as shown in Fig. 14.9). This is hardly surprising since in both cases we're trying to capture information about the relationship between heads and their dependents in the context of a single relation. To summarize this earlier discussion, commonly used features include:

- Wordforms, lemmas, and parts of speech of the headword and its dependent.
- Corresponding features derived from the contexts before, after and between the words.
- Word embeddings.
- The dependency relation itself.
- The direction of the relation (to the right or left).
- The distance from the head to the dependent.

As with transition-based approaches, pre-selected combinations of these features are often used as well.

Given a set of features, our next problem is to learn a set of weights corresponding to each. Unlike many of the learning problems discussed in earlier chapters,

inference-based learning

here we are not training a model to associate training items with class labels, or parser actions. Instead, we seek to train a model that assigns higher scores to correct trees than to incorrect ones. An effective framework for problems like this is to use **inference-based learning** combined with the perceptron learning rule. In this framework, we parse a sentence (i.e, perform inference) from the training set using some initially random set of initial weights. If the resulting parse matches the corresponding tree in the training data, we do nothing to the weights. Otherwise, we find those features in the incorrect parse that are *not* present in the reference parse and we lower their weights by a small amount based on the learning rate. We do this incrementally for each sentence in our training data until the weights converge.

State-of-the-art algorithms in multilingual parsing are based on recurrent neural networks (RNNs) (Zeman et al. 2017, Dozat et al. 2017).

14.5.3 Advanced Issues in Graph-Based Parsing

14.6 Evaluation

As with phrase structure-based parsing, the evaluation of dependency parsers proceeds by measuring how well they work on a test set. An obvious metric would be exact match (EM) — how many sentences are parsed correctly. This metric is quite pessimistic, with most sentences being marked wrong. Such measures are not fine-grained enough to guide the development process. Our metrics need to be sensitive enough to tell if actual improvements are being made.

For these reasons, the most common method for evaluating dependency parsers are labeled and unlabeled attachment accuracy. Labeled attachment refers to the proper assignment of a word to its head along with the correct dependency relation. Unlabeled attachment simply looks at the correctness of the assigned head, ignoring the dependency relation. Given a system output and a corresponding reference parse, accuracy is simply the percentage of words in an input that are assigned the correct head with the correct relation. These metrics are usually referred to as the labeled attachment score (LAS) and unlabeled attachment score (UAS). Finally, we can make use of a label accuracy score (LS), the percentage of tokens with correct labels, ignoring where the relations are coming from.

As an example, consider the reference parse and system parse for the following example shown in Fig. 14.15.

(14.11) Book me the flight through Houston.

The system correctly finds 4 of the 6 dependency relations present in the reference parse and receives an LAS of 2/3. However, one of the 2 incorrect relations found by the system holds between *book* and *flight*, which are in a head-dependent relation in the reference parse; the system therefore achieves a UAS of 5/6.

Beyond attachment scores, we may also be interested in how well a system is performing on a particular kind of dependency relation, for example NSUBJ, across a development corpus. Here we can make use of the notions of precision and recall introduced in Chapter 8, measuring the percentage of relations labeled NSUBJ by the system that were correct (precision), and the percentage of the NSUBJ relations present in the development set that were in fact discovered by the system (recall). We can employ a confusion matrix to keep track of how often each dependency type was confused for another.

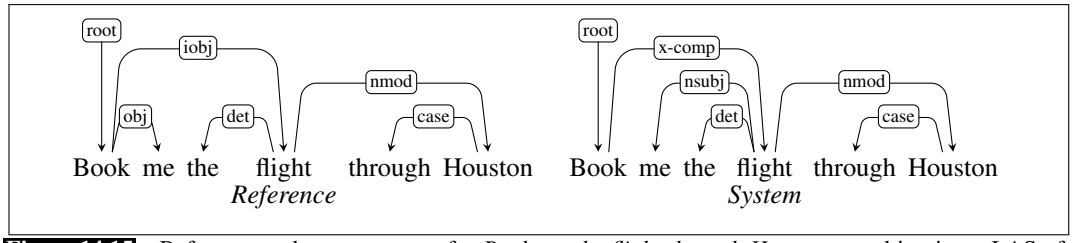


Figure 14.15 Reference and system parses for *Book me the flight through Houston*, resulting in an LAS of 2/3 and an UAS of 5/6.

14.7 Summary

This chapter has introduced the concept of dependency grammars and dependency parsing. Here's a summary of the main points that we covered:

- In dependency-based approaches to syntax, the structure of a sentence is described in terms of a set of binary relations that hold between the words in a sentence. Larger notions of constituency are not directly encoded in dependency analyses.
- The relations in a dependency structure capture the head-dependent relationship among the words in a sentence.
- Dependency-based analysis provides information directly useful in further language processing tasks including information extraction, semantic parsing and question answering.
- Transition-based parsing systems employ a greedy stack-based algorithm to create dependency structures.
- Graph-based methods for creating dependency structures are based on the use of maximum spanning tree methods from graph theory.
- Both transition-based and graph-based approaches are developed using supervised machine learning techniques.
- Treebanks provide the data needed to train these systems. Dependency treebanks can be created directly by human annotators or via automatic transformation from phrase-structure treebanks.
- Evaluation of dependency parsers is based on labeled and unlabeled accuracy scores as measured against withheld development and test corpora.

Bibliographical and Historical Notes

The dependency-based approach to grammar is much older than the relatively recent phrase-structure or constituency grammars that have been the primary focus of both theoretical and computational linguistics for years. It has its roots in the ancient Greek and Indian linguistic traditions. Contemporary theories of dependency grammar all draw heavily on the work of Tesnière (1959). The most influential dependency grammar frameworks include Meaning-Text Theory (MTT) (Mel'čuk, 1988), Word Grammar (Hudson, 1984), Functional Generative Description (FDG) (Sgall et al., 1986). These frameworks differ along a number of dimensions including the degree and manner in which they deal with morphological, syntactic,

semantic and pragmatic factors, their use of multiple layers of representation, and the set of relations used to categorize dependency relations.

Automatic parsing using dependency grammars was first introduced into computational linguistics by early work on machine translation at the RAND Corporation led by David Hays. This work on dependency parsing closely paralleled work on constituent parsing and made explicit use of grammars to guide the parsing process. After this early period, computational work on dependency parsing remained intermittent over the following decades. Notable implementations of dependency parsers for English during this period include Link Grammar (Sleator and Temperley, 1993), Constraint Grammar (Karlsson et al., 1995), and MINIPAR (Lin, 2003).

Dependency parsing saw a major resurgence in the late 1990's with the appearance of large dependency-based treebanks and the associated advent of data driven approaches described in this chapter. Eisner (1996) developed an efficient dynamic programming approach to dependency parsing based on bilexical grammars derived from the Penn Treebank. Covington (2001) introduced the deterministic word by word approach underlying current transition-based approaches. Yamada and Matsumoto (2003) and Kudo and Matsumoto (2002) introduced both the shift-reduce paradigm and the use of supervised machine learning in the form of support vector machines to dependency parsing.

Nivre (2003) defined the modern, deterministic, transition-based approach to dependency parsing. Subsequent work by Nivre and his colleagues formalized and analyzed the performance of numerous transition systems, training methods, and methods for dealing with non-projective language Nivre and Scholz 2004, Nivre 2006, Nivre and Nilsson 2005, Nivre et al. 2007, Nivre 2007.

The graph-based maximum spanning tree approach to dependency parsing was introduced by McDonald et al. 2005, McDonald et al. 2005.

The earliest source of data for training and evaluating dependency English parsers came from the WSJ Penn Treebank (Marcus et al., 1993) described in Chapter 12. The use of head-finding rules developed for use with probabilistic parsing facilitated the automatic extraction of dependency parses from phrase-based ones (Xia and Palmer, 2001).

The long-running Prague Dependency Treebank project (Hajič, 1998) is the most significant effort to directly annotate a corpus with multiple layers of morphological, syntactic and semantic information. The current PDT 3.0 now contains over 1.5 M tokens (Bejček et al., 2013).

Universal Dependencies (UD) (Nivre et al., 2016) is a project directed at creating a consistent framework for dependency treebank annotation across languages with the goal of advancing parser development across the world's languages. The UD annotation scheme evolved out of several distinct efforts including Stanford dependencies (de Marneffe et al. 2006, de Marneffe and Manning 2008, de Marneffe et al. 2014), Google's universal part-of-speech tags (Petrov et al., 2012), and the Interset interlingua for morphosyntactic tagsets (Zeman, 2008). Under the auspices of this effort, treebanks for over 90 languages have been annotated and made available in a single consistent format (Nivre et al., 2016).

The Conference on Natural Language Learning (CoNLL) has conducted an influential series of shared tasks related to dependency parsing over the years (Buchholz and Marsi 2006, Nilsson et al. 2007, Surdeanu et al. 2008, Hajič et al. 2009). More recent evaluations have focused on parser robustness with respect to morphologically rich languages (Seddah et al., 2013), and non-canonical language forms such as social media, texts, and spoken language (Petrov and McDonald, 2012).

Choi et al. (2015) presents a performance analysis of 10 dependency parsers across a range of metrics, as well as DEPENDABLE, a robust parser evaluation tool.

Exercises

Aho, A. V. and Ullman, J. D. (1972). *The Theory of Parsing, Translation, and Compiling*, Vol. 1. Prentice Hall.

Bejček, E., Hajičová, E., Hajič, J., Jínová, P., Kettnerová, V., Kolářová, V., Mikulová, M., Mírovský, J., Nedoluzhko, A., Panevová, J., Poláková, L., Ševčíková, M., Štěpánek, J., and Zikánová, Š. (2013). Prague dependency treebank 3.0. Tech. rep., Institute of Formal and Applied Linguistics, Charles University in Prague. LINDAT/CLARIN digital library at Institute of Formal and Applied Linguistics, Charles University in Prague.

Bhat, I., Bhat, R. A., Shrivastava, M., and Sharma, D. (2017). Joining hands: Exploiting monolingual treebanks for parsing of code-mixing data. *EACL*.

Buchholz, S. and Marsi, E. (2006). Conll-x shared task on multilingual dependency parsing. *CoNLL*.

Chen, D. and Manning, C. D. (2014). A fast and accurate dependency parser using neural networks. *EMNLP*.

Choi, J. D. and Palmer, M. (2011a). Getting the most out of transition-based dependency parsing. *ACL*.

Choi, J. D. and Palmer, M. (2011b). Transition-based semantic role labeling using predicate argument clustering. *Proceedings of the ACL 2011 Workshop on Relational Models of Semantics*.

Choi, J. D., Tetreault, J., and Stent, A. (2015). It depends: Dependency parser comparison using a web-based evaluation tool. *ACL*.

Chu, Y.-J. and Liu, T.-H. (1965). On the shortest arborescence of a directed graph. *Science Sinica 14*, 1396–1400.

Collins, M. (1999). *Head-Driven Statistical Models for Natural Language Parsing*. Ph.D. thesis, University of Pennsylvania, Philadelphia.

Collins, M. (2003). Head-driven statistical models for natural language parsing. *Computational Linguistics 29*(4), 589–637.

Covington, M. (2001). A fundamental algorithm for dependency parsing. *Proceedings of the 39th Annual ACM Southeast Conference*.

de Marneffe, M.-C., Dozat, T., Silveira, N., Haverinen, K., Ginter, F., Nivre, J., and Manning, C. D. (2014). Universal Stanford dependencies: A cross-linguistic typology. *LREC*.

de Marneffe, M.-C., MacCartney, B., and Manning, C. D. (2006). Generating typed dependency parses from phrase structure parses. *LREC*.

de Marneffe, M.-C. and Manning, C. D. (2008). The Stanford typed dependencies representation. *Coling 2008: Proceedings of the workshop on Cross-Framework and Cross-Domain Parser Evaluation*.

Dozat, T., Qi, P., and Manning, C. D. (2017). Stanford's graph-based neural dependency parser at the CoNLL 2017 shared task. *Proceedings of the CoNLL 2017 Shared Task*.

Edmonds, J. (1967). Optimum branchings. *Journal of Research of the National Bureau of Standards B 71*(4), 233–240.

Eisner, J. (1996). Three new probabilistic models for dependency parsing: An exploration. *COLING*.

Gabow, H. N., Galil, Z., Spencer, T., and Tarjan, R. E. (1986). Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica 6*(2), 109–122.

Hajič, J. (1998). *Building a Syntactically Annotated Corpus: The Prague Dependency Treebank*, 106–132. Karolinum.

Hajič, J., Ciaramita, M., Johansson, R., Kawahara, D., Martí, M. A., Màrquez, L., Meyers, A., Nivre, J., Padó, S., Štěpánek, J., Stranǎk, P., Surdeanu, M., Xue, N., and Zhang, Y. (2009). The conll-2009 shared task: Syntactic and semantic dependencies in multiple languages. *CoNLL*.

Hovy, E. H., Marcus, M. P., Palmer, M., Ramshaw, L. A., and Weischedel, R. (2006). Ontonotes: The 90% solution. *HLT-NAACL*.

Huang, L. and Sagae, K. (2010). Dynamic programming for linear-time incremental parsing. *ACL*.

Hudson, R. A. (1984). *Word Grammar*. Blackwell.

Karlsson, F., Voutilainen, A., Heikkilä, J., and Anttila, A. (Eds.). (1995). *Constraint Grammar: A Language-Independent System for Parsing Unrestricted Text*. Mouton de Gruyter.

Kudo, T. and Matsumoto, Y. (2002). Japanese dependency analysis using cascaded chunking. *CoNLL*.

Lin, D. (2003). Dependency-based evaluation of minipar. *Workshop on the Evaluation of Parsing Systems*.

Magerman, D. M. (1994). *Natural Language Parsing as Statistical Pattern Recognition*. Ph.D. thesis, University of Pennsylvania.

Marcus, M. P., Santorini, B., and Marcinkiewicz, M. A. (1993). Building a large annotated corpus of English: The Penn treebank. *Computational Linguistics 19*(2), 313–330.

McDonald, R., Crammer, K., and Pereira, F. C. N. (2005). Online large-margin training of dependency parsers. *ACL*.

McDonald, R. and Nivre, J. (2011). Analyzing and integrating dependency parsers. *Computational Linguistics 37*(1), 197–230.

McDonald, R., Pereira, F. C. N., Ribarov, K., and Hajič, J. (2005). Non-projective dependency parsing using spanning tree algorithms. *HLT-EMNLP*.

Mel'čuk, I. A. (1988). *Dependency Syntax: Theory and Practice*. State University of New York Press.

Nilsson, J., Riedel, S., and Yuret, D. (2007). The conll 2007 shared task on dependency parsing. *Proceedings of the CoNLL shared task session of EMNLP-CoNLL*. sn.

Nivre, J. (2007). Incremental non-projective dependency parsing. *NAACL-HLT*.

Nivre, J. (2003). An efficient algorithm for projective dependency parsing. *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT)*.

Nivre, J. (2006). *Inductive Dependency Parsing*. Springer.

Nivre, J. (2009). Non-projective dependency parsing in expected linear time. *ACL IJCNLP*.

Nivre, J., de Marneffe, M.-C., Ginter, F., Goldberg, Y., Hajič, J., Manning, C. D., McDonald, R., Petrov, S., Pyysalo, S., Silveira, N., Tsarfaty, R., and Zeman, D. (2016). Universal Dependencies v1: A multilingual treebank collection. *LREC*.

Nivre, J., Hall, J., Nilsson, J., Chanev, A., Eryigit, G., Kübler, S., Marinov, S., and Marsi, E. (2007). Maltparser: A language-independent system for data-driven dependency parsing. *Natural Language Engineering 13*(02), 95–135.

Nivre, J. and Nilsson, J. (2005). Pseudo-projective dependency parsing. *ACL*.

Nivre, J. and Scholz, M. (2004). Deterministic dependency parsing of english text. *COLING*.

Petrov, S., Das, D., and McDonald, R. (2012). A universal part-of-speech tagset. *LREC*.

Petrov, S. and McDonald, R. (2012). Overview of the 2012 shared task on parsing the web. *Notes of the First Workshop on Syntactic Analysis of Non-Canonical Language (SANCL)*, Vol. 59.

Seddah, D., Tsarfaty, R., Kübler, S., Candito, M., Choi, J. D., Farkas, R., Foster, J., Goenaga, I., Gojenola, K., Goldberg, Y., Green, S., Habash, N., Kuhlmann, M., Maier, W., Nivre, J., Przepiórkowski, A., Roth, R., Seeker, W., Versley, Y., Vincze, V., Woliński, M., Wróblewska, A., and Villemonte de la Clérgerie, E. (2013). Overview of the SPMRL 2013 shared task: cross-framework evaluation of parsing morphologically rich languages. *4th Workshop on Statistical Parsing of Morphologically-Rich Languages*.

Sgall, P., Hajičová, E., and Panevova, J. (1986). *The Meaning of the Sentence in its Pragmatic Aspects*. Reidel.

Sleator, D. and Temperley, D. (1993). Parsing English with a link grammar. *IWPT-93*.

Surdeanu, M., Johansson, R., Meyers, A., Màrquez, L., and Nivre, J. (2008). The conll-2008 shared task on joint parsing of syntactic and semantic dependencies. *CoNLL*.

Tesnière, L. (1959). *Éléments de Syntaxe Structurale*. Librairie C. Klincksieck, Paris.

Weischedel, R., Hovy, E. H., Marcus, M. P., Palmer, M., Belvin, R., Pradhan, S., Ramshaw, L. A., and Xue, N. (2011). Ontonotes: A large training corpus for enhanced processing. Olive, J., Christianson, C., and McCary, J. (Eds.), *Handbook of Natural Language Processing and Machine Translation: DARPA Global Automatic Language Exploitation*, 54–63. Springer.

Xia, F. and Palmer, M. (2001). Converting dependency structures to phrase structures. *HLT*.

Yamada, H. and Matsumoto, Y. (2003). Statistical dependency analysis with support vector machines. Noord, G. V. (Ed.), *IWPT-03*.

Zeman, D. (2008). Reusable tagset conversion using tagset drivers. *LREC*.

Zeman, D., Popel, M., Straka, M., Hajič, J., Nivre, J., Ginter, F., Luotolahti, J., Pyysalo, S., Petrov, S., Potthast, M., Tyers, F. M., Badmaeva, E., Gokirmak, M., Nedoluzhko, A., Cinková, S., Hajic, Jr., J., Hlavácová, J., Kettnerová, V., Uresová, Z., Kanerva, J., Ojala, S., Missilä, A., Manning, C. D., Schuster, S., Reddy, S., Taji, D., Habash, N., Leung, H., de Marneffe, M.-C., Sanguinetti, M., Simi, M., Kanayama, H., de Paiva, V., Droganova, K., Alonso, H. M., Çöltekin, Ç., Sulubacak, U., Uszkoreit, H., Macketanz, V., Burchardt, A., Harris, K., Marheinecke, K., Rehm, G., Kayadelen, T., Attia, M., El-Kahky, A., Yu, Z., Pitler, E., Lertpradit, S., Mandl, M., Kirchner, J., Alcalde, H. F., Strnadová, J., Banerjee, E., Manurung, R., Stella, A., Shimada, A., Kwak, S., Mendonça, G., Lando, T., Nitisaroj, R., and Li, J. (2017). Conll 2017 shared task: Multilingual parsing from raw text to universal dependencies. *Proceedings of the CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*.

Zhang, Y. and Clark, S. (2008). A tale of two parsers: investigating and combining graph-based and transition-based dependency parsing using beam-search. *EMNLP*.

Zhang, Y. and Nivre, J. (2011). Transition-based dependency parsing with rich non-local features. *ACL*.