CHAPTER 1

Introduction

Dependency parsing is an approach to automatic syntactic analysis of natural language inspired by the theoretical linguistic tradition of dependency grammar. After playing a rather marginal role in natural language processing for many years, dependency parsing has recently attracted considerable interest from researchers and developers in the field. One reason for the increasing popularity is the fact that dependency-based syntactic representations seem to be useful in many applications of language technology, such as machine translation and information extraction, thanks to their transparent encoding of predicate-argument structure. Another reason is the perception that dependency grammar is better suited than phrase structure grammar for languages with free or flexible word order, making it possible to analyze typologically diverse languages within a common framework. But perhaps the most important reason is that this approach has led to the development of accurate syntactic parsers for a number of languages, particularly in combination with machine learning from syntactically annotated corpora, or treebanks. It is the parsing methods used by these systems that constitute the topic of this book.

It is important to note from the outset that this is a book about dependency parsing, not about dependency grammar, and that we will in fact have very little to say about the way in which dependency grammar can be used to analyze the syntax of a given natural language. We will simply assume that such an analysis exists and that we want to build a parser that can implement it to automatically analyze new sentences. In this introductory chapter, however, we will start by giving a brief introduction to dependency grammar, focusing on basic notions rather than details of linguistic analysis. With this background, we will then define the task of dependency parsing and introduce the most important methods that are used in the field, methods that will be covered in depth in later chapters. We conclude, as in every chapter, with a summary and some suggestions for further reading.

1.1 DEPENDENCY GRAMMAR

Dependency grammar is rooted in a long tradition, possibly going back all the way to Pāṇini's grammar of Sanskrit several centuries before the Common Era, and has largely developed as a form for syntactic representation used by traditional grammarians, in particular in Europe, and especially for Classical and Slavic languages. The starting point of the modern theoretical tradition of dependency grammar is usually taken to be the work of the French linguist Lucien Tesnière, published posthumously in the late 1950s. Since then, a number of different dependency grammar frameworks have been proposed, of which the most well-known are probably the Prague School's Functional Generative Description, Mel'čuk's Meaning-Text Theory, and Hudson's Word Grammar.

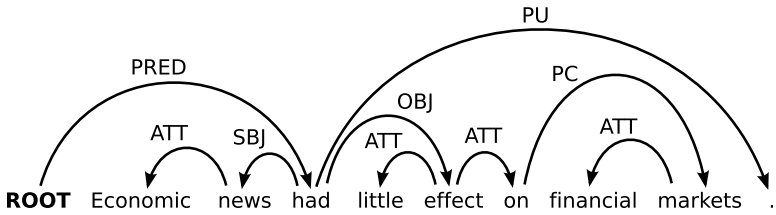

Figure 1.1: Dependency structure for an English sentence.

The basic assumption underlying all varieties of dependency grammar is the idea that syntactic structure essentially consists of *words* linked by binary, asymmetrical relations called *dependency relations* (or *dependencies* for short). A dependency relation holds between a syntactically subordinate word, called the *dependent*, and another word on which it depends, called the *head*.[1] This is illustrated in figure 1.1, which shows a dependency structure for a simple English sentence, where dependency relations are represented by arrows pointing from the head to the dependent.[2] Moreover, each arrow has a label, indicating the *dependency type*. For example, the noun *news* is a dependent of the verb *had* with the dependency type *subject* (SBJ). By contrast, the noun *effect* is a dependent of type *object* (OBJ) with the same head verb *had*. Note also that the noun *news* is itself a syntactic head in relation to the word *Economic*, which stands in the *attribute* (ATT) relation to its head noun.

One peculiarity of the dependency structure in figure 1.1 is that we have inserted an artificial word ROOT before the first word of the sentence. This is a mere technicality, which simplifies both formal definitions and computational implementations. In particular, we can normally assume that every real word of the sentence should have a syntactic head. Thus, instead of saying that the verb *had* lacks a syntactic head, we can say that it is a dependent of the artificial word ROOT. In chapter 2, we will define dependency structures formally as labeled directed graphs, where *nodes* correspond to words (including ROOT) and *labeled arcs* correspond to typed dependency relations.

The information encoded in a dependency structure representation is different from the information captured in a *phrase structure* representation, which is the most widely used type of syntactic representation in both theoretical and computational linguistics. This can be seen by comparing the dependency structure in figure 1.1 to a typical phrase structure representation for the same sentence, shown in figure 1.2. While the dependency structure represents head-dependent relations between *words*, classified by *functional* categories such as subject (SBJ) and object (OBJ), the phrase structure represents the grouping of words into *phrases*, classified by *structural* categories such as noun phrase (NP) and verb phrase (VP).

[1] Other terms that are found in the literature are *modifier* or *child*, instead of *dependent*, and *governor*, *regent* or *parent*, instead of *head*. Note that, although we will not use the noun *modifier*, we will use the verb *modify* when convenient and say that a dependent *modifies* its head.

[2] This is the notational convention that we will adopt throughout the book, but the reader should be warned that there is a competing tradition in the literature on dependency grammar according to which arrows point from the dependent to the head.

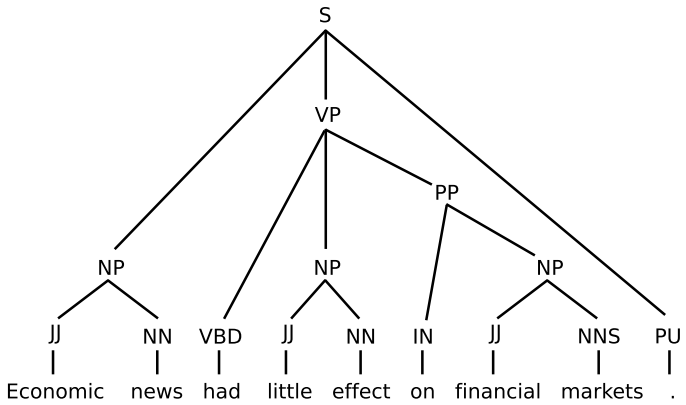

Figure 1.2: Phrase structure for an English sentence.

However, it is important to bear in mind that these differences only concern what is explicitly encoded in the respective representations. For example, phrases can be distinguished in a dependency structure by letting each head word represent a phrase consisting of the word itself and all the words that are dependent on it (possibly in several steps). Conversely, functional relations like subject and object can be identified in a phrase structure in terms of structural configurations (e.g., "NP under S" and "NP under VP"). Nevertheless, practical experience has shown that it is a non-trivial task to perform an automatic conversion from one type of representation to the other (cf. section 6.3). It is also worth noting that many syntactic theories make use of hybrid representations, combining elements of dependency structure with elements of phrase structure. Hence, to describe dependency grammar and phrase structure grammar as two opposite and mutually exclusive approaches to natural language syntax is at best an over-simplification.

If we assume that dependency structure captures an essential element of natural language syntax, then we need some criteria for establishing dependency relations, and for distinguishing the head and the dependent in these relations. Such criteria have been discussed not only in the dependency grammar tradition, but also within other frameworks where the notion of syntactic head plays an important role, including many theories based on phrase structure. Here is a list of some of the more common criteria that have been proposed for identifying a syntactic relation between a head H and a dependent D in a linguistic construction C:[3]

1. H determines the syntactic category of C and can often replace C.

2. H determines the semantic category of C; D gives semantic specification.

3. H is obligatory; D may be optional.

[3]The term *construction* is used here in a non-technical sense to refer to any structural complex of linguistic expressions.

4. *H* selects *D* and determines whether *D* is obligatory or optional.

5. The form of *D* depends on *H* (agreement or government).

6. The linear position of *D* is specified with reference to *H*.

It is clear that this list contains a mix of different criteria, some syntactic and some semantic, and one may ask whether there is a single coherent notion of dependency corresponding to all the different criteria. Some theorists therefore posit the existence of several layers of dependency structure, such as morphology, syntax and semantics, or surface syntax and deep syntax. Others have pointed out the need to have different criteria for different kinds of syntactic constructions, in particular for *endocentric* and *exocentric* constructions.

In figure 1.1, the attribute relation (ATT) holding between the noun *markets* and the adjective *financial* is an endocentric construction, where the head can replace the whole without disrupting the syntactic structure:

Economic news had little effect on [financial] markets.

Endocentric constructions may satisfy all of the criteria listed above, although number 4 is usually considered less relevant, since dependents in endocentric constructions are taken to be optional and not selected by their heads. By contrast, the prepositional complement relation (PC) holding between the preposition *on* and the noun *markets* is an exocentric construction, where the head cannot readily replace the whole:

Economic news had little effect on [markets].

Exocentric constructions, by their definition, fail on criterion number 1, at least with respect to substitutability of the head for the whole, but may satisfy the remaining criteria. Considering the rest of the relations exemplified in figure 1.1, the subject and object relations (SBJ, OBJ) are clearly exocentric, and the attribute relation from the noun *news* to the adjective *Economic* clearly endocentric, while the remaining attribute relations (effect → little, effect → on) have a less clear status.

The distinction between endocentric and exocentric constructions is also related to the distinction between *head-complement* and *head-modifier* (or *head-adjunct*) relations found in many contemporary syntactic theories, since head-complement relations are exocentric while head-modifier relations are endocentric. The distinction between complements and modifiers is often defined in terms of *valency*, which is a central notion in the theoretical tradition of dependency grammar. Although the exact characterization of this notion differs from one theoretical framework to the other, valency is usually related to the semantic predicate-argument structure associated with certain classes of lexemes, in particular verbs but sometimes also nouns and adjectives. The idea is that the verb imposes requirements on its syntactic dependents that reflect its interpretation as a semantic predicate. Dependents that correspond to arguments of the predicate can be obligatory or optional in surface syntax but can only occur once with each predicate. By contrast, dependents that do not correspond to arguments can have more than one occurrence with a single predicate and tend to be

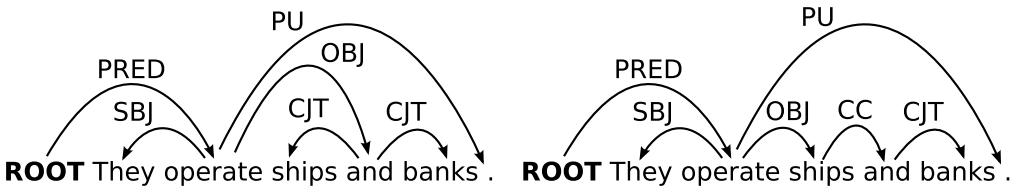

Figure 1.3: Two analyses of coordination in dependency grammar.

optional. The *valency frame* of the verb is normally taken to include argument dependents, but some theories also allow obligatory non-arguments to be included. Returning to figure 1.1, the subject and the object would normally be treated as valency-bound dependents of the verb *had*, while the adjectival modifiers of the nouns *news* and *markets* would be considered valency-free. The prepositional modification of the noun *effect* may or may not be treated as valency-bound, depending on whether the entity undergoing the effect is supposed to be an argument of the noun *effect* or not. Another term that is sometimes used in connection with valency constraints is *arity*, which primarily refers to the *number* of arguments that a predicate takes (without distinguishing the types of these arguments).

While most head-complement and head-modifier structures have a straightforward analysis in dependency grammar, there are also constructions that have a more unclear status. This group includes constructions that involve grammatical function words, such as articles, complementizers and auxiliary verbs, but also structures involving prepositional phrases. For these constructions, there is no general consensus in the tradition of dependency grammar as to whether they should be analyzed as dependency relations at all and, if so, what should be regarded as the head and what should be regarded as the dependent. For example, some theories regard auxiliary verbs as heads taking lexical verbs as dependents; other theories make the opposite assumption; and yet other theories assume that verb chains are connected by relations that are not dependencies in the usual sense.

Another kind of construction that is problematic for dependency grammar (as for most theoretical traditions) is *coordination*. According to the structuralist tradition, coordination is an endocentric construction, since it contains not only one but several heads that can replace the whole construction syntactically. However, this raises the question of whether coordination can be analyzed in terms of binary relations holding between a head and a dependent. Consider the following simple examples:

They operate ships and banks.
She bought and ate an apple.

In the first example, it seems clear that the phrase *ships and banks* functions as a direct object of the verb *operate*, but it is not immediately clear how this phrase can be given an internal analysis that is compatible with the basic assumptions of dependency grammar, since the two nouns *ships* and *banks* seem to be equally good candidates for being heads. Similarly, in the second example, the noun *apple*

is the object of the coordinated verb group *bought and ate*, where in some sense both verbs function as the head of the noun. The most popular treatments of coordination in dependency grammar are illustrated for the first example in figure 1.3, where the analysis to the left treats the conjunction as the head, an analysis that may be motivated on semantic grounds, while the analysis on the right treats the conjunction as the head only of the second conjunct and analyzes the conjunction as a dependent of the first conjunct. The arguments for the latter analysis are essentially the same as the arguments for an asymmetric right-branching analysis in phrase structure grammar.

To sum up, the theoretical tradition of dependency grammar is united by the assumption that syntactic structure essentially consists of dependency relations between words. Moreover, there is a core of syntactic constructions for which the analysis given by different frameworks agree in all important respects, notably predicate-argument and head-modifier constructions. However, there are also constructions for which there is no clear consensus, such as verb groups and coordination. Finally, it is worth pointing out that the inventory of dependency types used to classify dependency relations vary from one framework to the other. Besides traditional grammatical functions (such as predicate, subject, and object), semantic roles (such as agent, patient, and goal) are commonly used, especially in representations of deep syntax and semantics. Another dimension of variation is the number of representational levels, or strata, assumed in different theories. Although we will concentrate in this book on mono-stratal representations, using a single dependency structure for syntactic analysis, many theoretical frameworks make use of multi-stratal representations, often with different levels for syntax and semantics.

1.2 DEPENDENCY PARSING

Having introduced the basic notions of dependency grammar, we will now turn to the problem of *dependency parsing*, that is, the task of automatically analyzing the dependency structure of a given input sentence. Throughout this book we will consider a number of different methods for solving this problem, some based on inductive machine learning from large sets of syntactically annotated sentences, others based on formal grammars defining permissible dependency structures. Common to all of these methods is that they do not make any specific assumptions about the kind of dependency types used, be they grammatical functions or semantic roles, nor about the specific analysis of different linguistic constructions, such as verb groups or coordination.

All that is assumed is that the task of the parser is to produce a labeled dependency structure of the kind depicted in figure 1.1, where the words of the sentence (including the artificial word ROOT) are connected by typed dependency relations. This will be made more precise in chapter 2, where we define dependency structures as labeled directed graphs – called *dependency graphs* – and discuss a number of formal properties of these structures. But for the time being we can define the parsing problem as that of mapping an input sentence S, consisting of words $w_0 w_1 \ldots w_n$ (where $w_0 = \text{ROOT}$), to its dependency graph G. In the remainder of this chapter, we will give an overview of the different approaches to this problem that are covered in the book.

Broadly speaking, these approaches can be divided into two classes, which we will call *data-driven* and *grammar-based*, respectively. An approach is data-driven if it makes essential use of *machine learning* from linguistic data in order to parse new sentences. An approach is *grammar-based* if it relies on a *formal grammar*, defining a formal language, so that it makes sense to ask whether a given input sentence is in the language defined by the grammar or not. It is important to note that these categorizations are orthogonal, since it is possible for a parsing method to make essential use of machine learning *and* use a formal grammar, hence to be both data-driven and grammar-based. However, most of the methods that we cover fall into one of these classes only.

The major part of the book, chapters 3–4 to be exact, is devoted to data-driven methods for dependency parsing, which have attracted the most attention in recent years. We focus on *supervised* methods, that is, methods presupposing that the sentences used as input to machine learning have been annotated with their correct dependency structures. In supervised dependency parsing, there are two different problems that need to be solved computationally. The first is the *learning problem*, which is the task of learning a *parsing model* from a representative sample of sentences and their dependency structures. The second is the *parsing problem*, which is the task of applying the learned model to the analysis of a new sentence.[4] We can represent this as follows:

- **Learning:** Given a training set $\mathcal{D}$ of sentences (annotated with dependency graphs), induce a parsing model M that can be used to parse new sentences.
- **Parsing:** Given a parsing model M and a sentence S, derive the optimal dependency graph G for S according to M.

Data-driven approaches differ in the type of parsing model adopted, the algorithms used to learn the model from data, and the algorithms used to parse new sentences with the model. In this book, we focus on two classes of data-driven methods, which we call *transition-based* and *graph-based*, respectively. These classes contain most of the methods for data-driven dependency parsing that have been proposed in recent years.

Transition-based methods start by defining a transition system, or state machine, for mapping a sentence to its dependency graph. The learning problem is to induce a model for predicting the next state transition, given the transition history, and the parsing problem is to construct the optimal transition sequence for the input sentence, given the induced model. This is sometimes referred to as shift-reduce dependency parsing, since the overall approach is inspired by deterministic shift-reduce parsing for context-free grammars. Transition-based approaches are treated in chapter 3.

Graph-based methods instead define a space of candidate dependency graphs for a sentence. The learning problem is to induce a model for assigning scores to the candidate dependency graphs for a sentence, and the parsing problem is to find the highest-scoring dependency graph for the input sentence, given the induced model. This is often called maximum spanning tree parsing, since the problem of finding the highest-scoring dependency graph is equivalent, under certain assumptions,

[4]The parsing problem is sometimes referred to as the *inference problem* or *decoding problem*, which are the general terms used in machine learning for the application of a learned model to new data.

to the problem of finding a maximum spanning tree in a dense graph. Graph-based approaches are treated in chapter 4.

Most data-driven approaches, whether transition-based or graph-based, assume that any input string is a valid sentence and that the task of the parser is to return the most plausible dependency structure for the input, no matter how unlikely it may be. Grammar-based approaches, by contrast, make use of a formal grammar that only accepts a subset of all possible input strings. Given our previous characterization of the parsing problem, we may say that this formal grammar is an essential component of the model M used to parse new sentences. However, the grammar itself may be hand-crafted or learned from linguistic data, which means that a grammar-based model may or may not be data-driven as well. In chapter 5, we discuss selected grammar-based methods for dependency parsing, dividing them into two classes, which we call *context-free* and *constraint-based*, respectively.

Context-free dependency parsing exploits a mapping from dependency structures to context-free phrase structure representations and reuses parsing algorithms originally developed for context-free grammar. This includes chart parsing algorithms, which are also used in graph-based parsing, as well as shift-reduce type algorithms, which are closely related to the methods used in transition-based parsing.

Constraint-based dependency parsing views parsing as a constraint satisfaction problem. A grammar is defined as a set of constraints on well-formed dependency graphs, and the parsing problem amounts to finding a dependency graph for a sentence that satisfies all the constraints of the grammar. Some approaches allow soft, weighted constraints and score dependency graphs by a combination of the weights of constraints violated by that graph. Parsing then becomes the problem of finding the dependency graph for a sentence that has the best score, which is essentially the same formulation as in graph-based parsing.

We can sum up our coverage of dependency parsing methods as follows:

- Data-driven dependency parsing
  - Transition-based dependency parsing (chapter 3)
  - Graph-based dependency parsing (chapter 4)
- Grammar-based parsing (chapter 5)
  - Context-free dependency parsing
  - Constraint-based dependency parsing

In chapter 6, we discuss issues concerning evaluation, both the evaluation of dependency parsers and the use of dependencies as a basis for cross-framework evaluation, and in chapter 7, we compare the approaches treated in earlier chapters, pointing out similarities and differences between methods, as well as complementary strengths and weaknesses. We conclude the book with some reflections on current trends and future prospects of dependency parsing in chapter 8.

1.3 SUMMARY AND FURTHER READING

In this chapter, we have introduced the basic notions of dependency grammar, compared dependency structure to phrase structure, and discussed criteria for identifying dependency relations and syntactic heads. There are several textbooks that give a general introduction to dependency grammar but most of them in other languages than English, for example, Tarvainen (1982) and Weber (1997) in German and Nikula (1986) in Swedish. For a basic introduction in English we refer to the opening chapter of Mel'čuk (1988). Open issues in dependency grammar, and their treatment in different theories, are discussed in chapter 3 of Nivre (2006b).

Tesnière's seminal work was published posthumously as Tesnière (1959). (The French text has been translated into German and Russian but not into English.) Other influential theories in the dependency grammar tradition include Functional Generative Description (Sgall et al., 1986); Meaning-Text Theory (Mel'čuk, 1988; Milicevic, 2006); Word Grammar (Hudson, 1984, 1990, 2007); Dependency Unification Grammar (Hellwig, 1986, 2003); and Lexicase (Starosta, 1988). Constraint-based theories of dependency grammar have a strong tradition, represented by Constraint Dependency Grammar, originally proposed by Maruyama (1990) and further developed by Harper and Helzerman (1995) and Menzel and Schröder (1998) into Weighted Constraint Dependency Grammar (Schröder, 2002); Functional Dependency Grammar (Tapanainen and Järvinen, 1997; Järvinen and Tapanainen, 1998), largely developed from Constraint Grammar (Karlsson, 1990; Karlsson et al., 1995); and finally Topological Dependency Grammar (Duchier and Debusmann, 2001), later evolved into Extensible Dependency Grammar (Debusmann et al., 2004).

In the second half of the chapter, we have given an informal introduction to dependency parsing and presented an overview of the most important approaches in this field, both data-driven and grammar-based. A more thorough discussion of different approaches can be found in chapter 3 of Nivre (2006b). Grammar-based dependency parsing originates with the work on context-free dependency parsing by Gaifman and Hays in the 1960s (Hays, 1964; Gaifman, 1965), and the constraint-based approach was first proposed by Maruyama (1990). Data-driven dependency parsing was pioneered by Eisner (1996b), using graph-based methods, and the transition-based approach was first explored by Matsumoto and colleagues (Kudo and Matsumoto, 2002; Yamada and Matsumoto, 2003). The terms *graph-based* and *transition-based* to characterize the two classes of data-driven methods were first used by McDonald and Nivre (2007), but essentially the same distinction was proposed earlier by Buchholz and Marsi (2006), using the terms *all pairs* and *stepwise*.

Although we concentrate in this book on supervised methods for data-driven parsing, there is also a considerable body of work on *unsupervised* parsing, which does not require annotated training data, although the results are so far vastly inferior to supervised approaches in terms of parsing accuracy. The interested reader is referred to Yuret (1998), Klein (2005), and Smith (2006).

Dependency parsing has recently been used in a number of different applications of natural language processing. Relevant examples include language modeling (Chelba et al., 1997), information extraction (Culotta and Sorensen, 2004), machine translation (Ding and Palmer, 2004; Quirk et al.,

2005), textual entailment (Haghighi et al., 2005), lexical ontology induction (Snow et al., 2005), and question answering (Wang et al., 2007).

CHAPTER 2

Dependency Parsing

In this chapter we formally introduce dependency graphs and dependency parsing, as well as the primary notation used throughout the rest of the book.

2.1 DEPENDENCY GRAPHS AND TREES

As mentioned in the previous chapter, dependency graphs are syntactic structures over sentences.

Definition 2.1. A *sentence* is a sequence of tokens denoted by:

$$S = w_0 w_1 \dots w_n$$

We assume that the tokenization of a sentence is fixed and known at parsing time. That is to say that dependency parsers will always operate on a pre-tokenized input and are not responsible for producing the correct tokenization of an arbitrary string. Furthermore, $w_0 = \text{ROOT}$ is an artificial root token inserted at the beginning of the sentence and does not modify any other token in the sentence. Each token w_i typically represents a word and we will use *word* and *token* interchangeably. However, the precise definition of w_i is often language dependent and a token can be a morpheme or a punctuation marker. In particular, it is not uncommon in highly inflected languages to tokenize a sentence aggressively so that w_i can be either a lemma or the affix of a word.

For simplicity we assume that a sentence is a sequence of *unique* tokens/words. Consider the sentence:

Mary saw John and Fred saw Susan.

This sentence contains two different instances of the word *saw* and we assume each to be distinct from the other. It is straight-forward to ensure this by simply storing an index referencing the position of every word in the sequence. We assume such indices exist, even though we do not explicitly mark their presence.

Definition 2.2. Let $R = \{r_1, \dots, r_m\}$ be a finite set of possible *dependency relation types* that can hold between any two words in a sentence. A relation type $r \in R$ is additionally called an *arc label*.

For example, the dependency relation between the words *had* and *effect* in figure 1.1 is labeled with the type $r = \text{OBJ}$. As stated earlier, we make no specific assumptions about the nature of R except that it contains a fixed inventory of dependency types.

With these two definitions in hand, we can now define dependency graphs.

Definition 2.3. A *dependency graph* $G = (V, A)$ is a labeled directed graph (digraph) in the standard graph-theoretic sense and consists of nodes, V, and arcs, A, such that for sentence $S = w_0 w_1 \ldots w_n$ and label set R the following holds:

1. $V \subseteq \{w_0, w_1, \ldots, w_n\}$
2. $A \subseteq V \times R \times V$
3. if $(w_i, r, w_j) \in A$ then $(w_i, r', w_j) \notin A$ for all $r' \neq r$

The arc set A represents the labeled dependency relations of the particular analysis G. Specifically, an arc $(w_i, r, w_j) \in A$ represents a dependency relation from head w_i to dependent w_j labeled with relation type r. A dependency graph G is thus a set of labeled dependency relations between the words of S.

Nodes in the graph correspond directly to words in the sentence and we will use the terms node and word interchangeably. A standard node set is the *spanning node set* that contains all the words of the sentence, which we sometimes denote by $V_S = \{w_0, w_1, \ldots, w_n\}$.

Without the third restriction, dependency graphs would be *multi-digraphs* as they would allow more than one possible arc between each pair of nodes, i.e., one arc per label in R. This definition of dependency graphs is specific to mono-stratal theories of syntactic dependencies, where the entire dependency analysis is relative to a single graph over the words of the sentence. In contrast, multi-stratal theories like Functional Generative Description, Meaning-Text Theory or Topological Dependency Grammar assume that the true dependency analysis consists of multiple dependency graphs, each typically representing one layer of the analysis such as the morphological, syntactic, or semantic dependency analysis.

To illustrate this definition, consider the dependency graph in figure 1.1, which is represented by:

1. $G = (V, A)$
2. $V = V_S = \{$ROOT, Economic, news, had, little, effect, on, financial, markets, .$\}$
3. $A = \{$(ROOT, PRED, had), (had, SBJ, news), (had, OBJ, effect),(had, PU, .), (news, ATT, Economic), (effect, ATT, little), (effect, ATT, on), (on, PC, markets), (markets, ATT, financial)$\}$

As discussed in the first chapter, the nature of a dependency (w_i, r, w_j) is not always straight-forward to define and differs across linguistic theories. For the remainder of this book we assume that it is fixed, being either specified by a formal grammar or implicit in a labeled corpus of dependency graphs.

Finally, having defined sentences, dependency relation types and dependency graphs, we can now proceed to a central definition,

Definition 2.4. A *well-formed dependency graph* $G = (V, A)$ for an input sentence S and dependency relation set R is any dependency graph that is a *directed tree originating out of node* w_0 and has the spanning node set $V = V_S$. We call such dependency graphs *dependency trees*.

Notation 2.5. For an input sentence S and a dependency relation set R, denote the space of all well-formed dependency graphs as $\mathcal{G}_S$.

The dependency graphs in figures 1.1 and 1.3 are both trees. For the remainder of the book we only consider parsing systems that produce dependency trees, that is, parsing systems that produce a tree from the set $\mathcal{G}_S$ for a sentence S.

The restriction of well-formed dependency graphs to dependency trees may seem rather strong at first given the flexibility of language. However, most mono-stratal dependency theories make this assumption (a notable exception being Hudson's Word Grammar) as do most multi-stratal theories for each individual layer of the analysis. In the next section we break down the various properties of dependency trees and examine each from a linguistic or computational point of view. Many of these properties are generally agreed upon across different dependency theories and will help to motivate the restriction of well-formed dependency graphs to trees.

2.1.1 PROPERTIES OF DEPENDENCY TREES

First, we will define a few notational conventions that will assist in our analysis of dependency trees.

Notation 2.6. The notation $w_i \rightarrow w_j$ indicates the *unlabeled dependency relation* (or *dependency relation* for short) in a tree $G = (V, A)$. That is, $w_i \rightarrow w_j$ if and only if $(w_i, r, w_j) \in A$ for some $r \in R$.

Notation 2.7. The notation $w_i \rightarrow^* w_j$ indicates the *reflexive transitive closure of the dependency relation* in a tree $G = (V, A)$. That is, $w_i \rightarrow^* w_j$ if and only if $i = j$ (reflexive) or both $w_i \rightarrow^* w_{i'}$ and $w_{i'} \rightarrow w_j$ hold (for some $w_{i'} \in V$).

Notation 2.8. The notation $w_i \leftrightarrow w_j$ indicates the *undirected dependency relation* in a tree $G = (V, A)$. That is, $w_i \leftrightarrow w_j$ if and only if either $w_i \rightarrow w_j$ or $w_j \rightarrow w_i$.

Notation 2.9. The notation $w_i \leftrightarrow^* w_j$ indicates the *reflexive transitive closure of the undirected dependency relation* in a tree $G = (V, A)$. That is, $w_i \leftrightarrow^* w_j$ if and only if $i = j$ (reflexive) or both $w_i \leftrightarrow^* w_{i'}$ and $w_{i'} \leftrightarrow w_j$ hold (for some $w_{i'} \in V$).

With this notation in hand, we can now examine a set of dependency tree properties that are always true. These properties are true of any directed tree, but we examine them from the perspective of their linguistic motivation.

Property 2.10. A dependency tree $G = (V, A)$ always satisfies the *root property*, which states that there does not exist $w_i \in V$ such that $w_i \rightarrow w_0$.

Property 2.10 holds from the definition of dependency trees as rooted directed trees originating out of w_0. This property is artificial since we have already indicated the presence of the word ROOT and defined its unique nature in the definition of dependency trees. The addition of an artificial root node may seem spurious, but as we discuss subsequent properties below, it will become clear that the artificial root provides us with both linguistic and algorithmic generalization ability.

Property 2.11. A dependency tree $G = (V, A)$ always satisfies the *spanning property* over the words of the sentence, which states that $V = V_S$.

Property 2.11 is also explicitly stated in the definition of dependency trees and therefore must hold for all dependency trees. The spanning property is widely accepted in dependency theories since a word in a sentence almost always has some relevance to the dependency analysis and in particular the syntactic analysis of the sentence. This property is sometimes relaxed for punctuation, for example words like periods or other sentence boundary markers that play no role in the dependency analysis of the sentence. The property may be further relaxed for additional punctuation such as hyphens and brackets – as well as some comma usage – that implicitly participate in the analysis by providing cues for the intended reading but again play no explicit role in the analysis. When considering semantic dependencies the spanning property is less universal as many words simply facilitate the reader's understanding of the true semantic interpretation and do not actually have an explicit semantic function.

In practice it is irrelevant if linguistic theories agree on whether a dependency analysis should be spanning over all the words in the sentence. This is because the artificial root node allows one to be theory general with respect to the spanning property as we can simply create an arc from the root word to all $w_i \in V$ that do not participate in the analysis. The result is always a dependency tree where the spanning property holds.

Property 2.12. A dependency tree $G = (V, A)$ satisfies the *connectedness property*, which states that for all $w_i, w_j \in V$ it is the case that $w_i \leftrightarrow^* w_j$. That is, there is a path connecting every two words in a dependency tree when the direction of the arc (dependency relation) is ignored. This notion of connectedness is equivalent to a *weakly connected directed graph* from graph theory.

Property 2.12 holds due to the fact that all nodes in a directed tree are weakly connected through the root. The connectedness property simply states that all words in the sentence interact with one another in the dependency analysis, even if at a distance or through intermediate words. This

property is not universally accepted, as a sentence may be fragmented into a number of disjoint units. However, we can again use the artificial root word and make this property universal by simply creating a dependency relation from the root to some word in each of the dependency fragments. Thus, the artificial root word again allows one to be theory-neutral, this time with respect to dependency analysis connectedness. Furthermore, we also gain a computational generalization through the artificial root node. As we will see in Chapter 3, some dependency parsing algorithms do not actually produce a single dependency tree but rather a set of disjoint dependency trees, commonly called a *dependency forest*. These algorithms can be trivially modified to return a dependency tree by adding a dependency arc from the artificial root word to the root of each disjoint tree.

Property 2.13. A dependency tree $G = (V, A)$ satisfies the *single-head property*, which states that for all $w_i, w_j \in V$, if $w_i \rightarrow w_j$ then there does not exist $w_{i'} \in V$ such that $i' \neq i$ and $w_{i'} \rightarrow w_j$. That is, each word in a dependency tree is the dependent of at most one head.

Property 2.13 holds due to the fact that a directed tree is specifically characterized by each node having a single incoming arc. The single-head property is not universal in dependency theories. The example from chapter 1 – *She bought and ate an apple* – is an instance where one might wish to break the single-head property. In particular, *she* and *apple* can be viewed as dependents of both verbs in the coordinated verb phrase and as a result should participate as the dependent in multiple dependency arcs in the tree. However, many formalisms simply posit that *she* and *apple* modify the head of the coordinate phrase (whether it is the conjunction or one of the verbs) and assume that this dependency is propagated to all the conjuncts.

Property 2.14. A dependency tree $G = (V, A)$ satisfies the *acyclicity property*, which states that for all $w_i, w_j \in A$, if $w_i \rightarrow w_j$, then it is not the case that $w_j \rightarrow^* w_i$. That is, a dependency tree does not contains cycles.

The acyclicity property also makes sense linguistically as any dependency tree not satisfying this property would imply that a word implicitly is dependent upon itself.

Property 2.15. A dependency tree $G = (V, A)$ satisfies the *arc size property*, which states that $|A| = |V| - 1$.

Property 2.15 falls out of the unique root and single-head properties. We listed this property as it can simplify both algorithm construction and analysis.

2.1.2 PROJECTIVE DEPENDENCY TREES

Up to this point we have presented properties that hold for all dependency trees. However, many computational systems restrict the class of well-formed dependency graphs even further. The most common restriction is to the set of *projective dependency trees*, which we examine here.

Definition 2.16. An arc $(w_i, r, w_j) \in A$ in a dependency tree $G = (V, A)$ is *projective* if and only if $w_i \rightarrow^* w_k$ for all $i < k < j$ when $i < j$, or $j < k < i$ when $j < i$.

That is to say, an arc in a tree is projective if there is a directed path from the head word w_i to all the words between the two endpoints of the arc.

Definition 2.17. A dependency tree $G = (V, A)$ is a *projective dependency tree* if (1) it is a dependency tree (definition 2.4), and (2) all $(w_i, r, w_j) \in A$ are projective.

A similar definition exists for *non-projective dependency trees*.

Definition 2.18. A dependency tree $G = (V, A)$ is a *non-projective dependency tree* if (1) it is a dependency tree (definition 2.4), and (2) it is not projective.

The trees in figures 1.1 and 1.3 are both projective dependency trees. Linguistically, projectivity is too rigid a restriction. Consider the sentence in figure 2.1. The dependency tree for this sentence is *non-projective* since the prepositional phrase *on the issue* that modifies the noun *hearing* is separated sequentially from its head by the main verb group. As a result, the dependency (hearing, PP, on) does not satisfy the projective arc definition, requiring a non-projective analysis to account for the syntactic validity of this sentence.

In English, non-projective constructions occur with little frequency relative to other languages that are highly inflected and, as a result, have less constraints on word order. In particular, sentences in languages like Czech, Dutch and Turkish frequently require non-projective dependency trees to correctly analyze a significant fraction of sentences. As a result, most linguistic theories of dependency parsing do not presume that dependency trees are projective. Thus, throughout most of this book we will not assume that dependency trees are projective and make it clear when we are referring to the set of all dependency trees, or the subset of projective dependency trees.

Notation 2.19. For an input sentence S and a dependency relation set R, denote the space of all projective dependency trees as $\mathcal{G}_S^p$.

Even though they are too restrictive, projective dependency trees have certain properties of interest, primarily from a computational perspective.

Property 2.20. A projective dependency tree $G = (V, A)$ satisfies the *planar property*, which states that it is possible to graphically configure all the arcs of the tree in the space above the sentence without any arcs crossing.

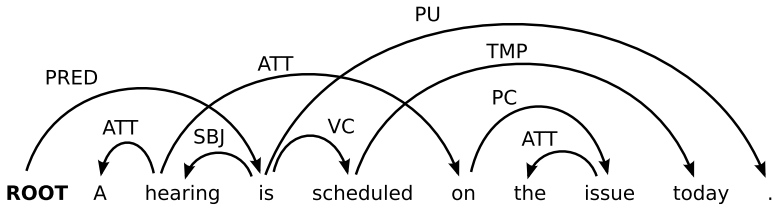

Figure 2.1: Non-projective dependency tree for an English sentence.

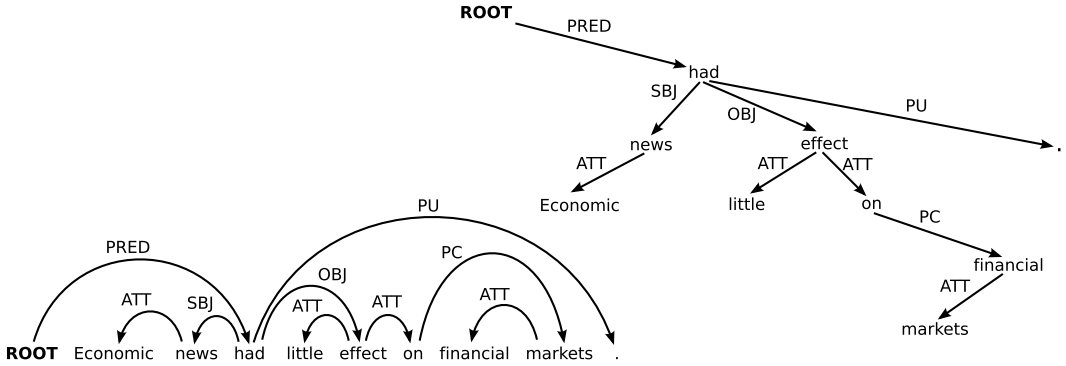

Figure 2.2: Projective dependency tree drawn in the standard way (left) and as a nested tree (right).

The left tree in figure 2.2 displays a projective tree drawn without arc crossings, whereas the tree in figure 2.1 shows a non-projective tree where it is impossible to configure the arcs so that none cross. The inverse of this property is true as well: all dependency trees that can be drawn so that no arcs cross are projective dependency trees. This direction of the equivalence specifically relies on the fact that the left-most word in the sentence is the root of the tree. Consider the case where $S = w_0 w_1 w_2$ with arcs $w_0 \rightarrow w_2$ and $w_1 \rightarrow w_0$ in a dependency tree G, i.e., w_1 is the root. Such a tree can be drawn with no arcs crossing, but is not projective.

Property 2.21. A projective dependency tree $G = (V, A)$ satisfies the *nested property*, which states that for all nodes $w_i \in V$, the set of words $\{w_j | w_i \rightarrow^* w_j\}$ is a contiguous subsequence of the sentence S.

The set $\{w_j | w_i \rightarrow^* w_j\}$ is often called the *yield* of w_i in G. Figure 2.2 illustrates both a projective dependency tree and its nested depiction. Proving that all projective dependency trees are nested trees is straight-forward. If we assume that the yield of w_i is not contiguous, that means that there is some node w_j between the end-points of the yield such that $w_i \rightarrow^* w_j$ does not hold. If we trace dependency arcs back from w_j we will eventually reach a node w_k between the end-points of the

yield of w_i such that $w_{k'} \rightarrow w_k$ is in the tree but $w_{k'}$ is not between the end-points of the yield of w_i. But such an arc would necessarily cross at least one other arc and thus the tree could not have been projective in the first place.

The nested tree property is the primary reason that many computational dependency parsing systems have focused on producing trees that are projective as it has been shown that certain dependency grammars enforcing projectivity are (weakly) equivalent in generative capacity to context-free grammars, which are well understood computationally from both complexity and formal power standpoints.

2.2 FORMAL DEFINITION OF DEPENDENCY PARSING

In this section, we aim to make mathematically precise the dependency parsing problem for both data-driven and grammar-based methods. This will include introducing notation and defining both the general problems of *learning*, which is required for data-driven methods, and *parsing*, which is required for both data-driven and grammar-based methods. To reiterate a point made in the previous chapter, data-driven and grammar-based methods are compatible. A grammar-based method can be data-driven when its parameters are learned from a labeled corpus.

As with our earlier convention, we use G to indicate a dependency tree and $\mathcal{G}$ to indicate a set of dependency trees. Similarly, $S = w_0 w_1 \dots w_n$ denotes a sentence and $\mathcal{S}$ denotes a set of sentences. For a given sentence S, we use $\mathcal{G}_S$ to indicate the space of dependency trees for that sentence, and we use $\mathcal{G}_S^p$ for the subset of projective dependency trees.

An important function that will be used at various points throughout the book is the feature function $\mathbf{f}(x) : \mathcal{X} \rightarrow \mathcal{Y}$ that maps some input x to a feature representation in the space $\mathcal{Y}$. Examples include mappings from an input sentence S or history of parsing decisions to a set of predictive symbolic or binary predicates. When $\mathcal{Y}$ is a collection of predicates (either symbolic or numeric), then we often refer to $\mathbf{f}$ as the *feature vector*. Possibly the most common mapping for $\mathbf{f}$ is to a high dimensional real valued feature vector, i.e., $\mathcal{Y} = \mathbb{R}^m$. The features used in a parsing system differ by the parsing scheme and will be discussed in further detail in later chapters.

Let us now proceed with an important definition:

Definition 2.22. A *dependency parsing model* consists of a set of constraints Γ that define the space of permissible dependency structures for a given sentence, a set of parameters $\boldsymbol{\lambda}$ (possibly null), and fixed parsing algorithm h. A model is denoted by $M = (\Gamma, \boldsymbol{\lambda}, h)$.

The constraints Γ are specific to the underlying formalism used by a system. Minimally the constraint set maps an arbitrary sentence S and dependency type set R to the set of well-formed dependency graphs $\mathcal{G}_S$, in effect restricting the space of dependency graphs to dependency trees. Additionally, Γ could encode more complex mechanisms such as context-free grammar or a constraint dependency grammar that further limit the space of dependency graphs.

The *learning* phase of a parser aims to construct the parameter set $\boldsymbol{\lambda}$, and it is specific to data-driven systems. The parameters are learned using a training set $\mathcal{D}$ that consists of pairs of sentences and their corresponding dependency trees:

$$\mathcal{D} = \{(S_d, G_d)\}_{d=0}^{|\mathcal{D}|}$$

Parameters are typically learned by optimizing some function over $\mathcal{D}$ and come from some predefined class of parameters $\boldsymbol{\Lambda}$. Common optimizations include minimizing training set parsing error or maximizing conditional probability of trees given sentences for examples in $\mathcal{D}$. The nature of $\boldsymbol{\lambda}$ and the optimization depend on the specific learning methods employed. For example, a single parameter might represent the likelihood of a dependency arc occurring in a dependency tree for a sentence, or it might represent the likelihood of satisfying some preference in a formal grammar. In the following chapters, these specifics will be addressed when we examine the major approaches to dependency parsing. For systems that are not data-driven, $\boldsymbol{\lambda}$ is either null or uniform rendering it irrelevant.

After a parsing model has defined a set of formal constraints and learned appropriate parameters, the model must fix a parsing algorithm to solve the *parsing* problem. That is, given the constraints, parameters and a new sentence S, how does the system find the single most likely dependency tree for that sentence:

$$G = h(S, \Gamma, \boldsymbol{\lambda})$$

The function h is a search over the set of well-formed dependency graphs $\mathcal{G}_S$ for input sentence S and produces a single tree or null if Γ defines a grammar in which S is not a member of the defined language. As we will see in the remaining chapters, h can take many algorithmic forms including greedy and recursive algorithms as well as those based on chart-parsing techniques. Furthermore, h can be exact or approximate relative to some objective function.

To give a quick illustration of the notation defined here, we can apply it to the well known case of a probabilistic context-free grammar (PCFG) for phrase structure parsing – a grammar-based and data-driven parsing system. In that case, $\Gamma = (N, \Sigma, \Pi, \text{START})$ is a standard CFG with non-terminals N, terminals Σ, production rules Π, and start symbol $\text{START} \in N$, all of which defines a space of nested phrase structures. $\boldsymbol{\lambda}$ is a set of probabilities, one for each production in the grammar. $\boldsymbol{\lambda}$ is typically set by maximizing the likelihood of the training set $\mathcal{D}$ relative to appropriate consistency constraints. The fixed parsing algorithm h can then be a number of context-free algorithms such as CKY (Younger, 1967) or Earley's algorithm (Earley, 1970).

2.3 SUMMARY AND FURTHER READING

In this chapter, we discussed the formal definition of dependency graphs, as well as a set of properties of these graphs that are common among many systems (both linguistic and computational). A key definition is that of a dependency tree, which is any well-formed dependency graph that is a directed spanning tree originating out of the root word w_0. There have been many studies of the structural properties of dependency graphs and trees that go beyond what is discussed here. Mentioned earlier

was work showing that certain projective dependency grammars are weakly equivalent to context-free grammars (Hays, 1964; Gaifman, 1965). Structural properties of dependency graphs that have been studied include: *planarity*, which is strongly correlated to projectivity (Kuhlmann and Nivre, 2006; Havelka, 2007); *gap-degree*, which measures the discontinuity of subgraphs (Bodirsky et al., 2005; Kuhlmann and Nivre, 2006); *well-nestedness*, which is a binary property on the overlap between subtrees of the graph (Bodirsky et al., 2005; Kuhlmann and Nivre, 2006); and *arc-degree*, which measures the number of disconnected subgraphs an arc spans in the graph (Nivre, 2006a; Kuhlmann and Nivre, 2006). Some interesting facts arise out of these studies. This includes the relation of dependency graph structural constraints to derivations in tree adjoining grammars (Bodirsky et al., 2005) as well as empirical statistics on how frequently certain constraints are obeyed in various dependency treebanks (Nivre, 2006a; Kuhlmann and Nivre, 2006; Havelka, 2007). In terms of projectivity, Marcus (1965) proves the equivalence of a variety of projectivity definitions and Havelka (2007) discusses many of the above properties in relation to the projective constraint.

The final section of this chapter introduced the formal definition of dependency parsing including the definition of a parsing model and its sub-components: the formal constraints, the parameters, and the parsing algorithm. These definitions, as well as those given for dependency trees, form the basis for the next chapters that delve into different parsing formalisms and their relation to one another.

CHAPTER 3

Transition-Based Parsing

In data-driven dependency parsing, the goal is to learn a good predictor of dependency trees, that is, a model that can be used to map an input sentence $S = w_0 w_1 \ldots w_n$ to its correct dependency tree G. As explained in the previous chapter, such a model has the general form $M = (\Gamma, \boldsymbol{\lambda}, h)$, where Γ is a set of constraints that define the space of permissible structures for a given sentence, $\boldsymbol{\lambda}$ is a set of parameters, the values of which have to be learned from data, and h is a fixed parsing algorithm. In this chapter, we are going to look at systems that parameterize a model over the transitions of an abstract machine for deriving dependency trees, where we learn to predict the next transition given the input and the parse history, and where we predict new trees using a greedy, deterministic parsing algorithm – this is what we call transition-based parsing. In chapter 4, we will instead consider systems that parameterize a model over sub-structures of dependency trees, where we learn to score entire dependency trees given the input, and where we predict new trees using exact inference – graph-based parsing. Since most transition-based and graph-based systems do not make use of a formal grammar at all, Γ will typically only restrict the possible dependency trees for a sentence to those that satisfy certain formal constraints, for example, the set of all projective trees (over a given label set). In chapter 5, by contrast, we will deal with grammar-based systems, where Γ constitutes a formal grammar pairing each input sentence with a more restricted (possibly empty) set of dependency trees.

3.1 TRANSITION SYSTEMS

A *transition system* is an abstract machine, consisting of a set of *configurations* (or *states*) and *transitions* between configurations. One of the simplest examples is a finite state automaton, which consists of a finite set of atomic states and transitions defined on states and input symbols, and which accepts an input string if there is a sequence of valid transitions from a designated *initial* state to one of several *terminal* states. By contrast, the transition systems used for dependency parsing have complex configurations with internal structure, instead of atomic states, and transitions that correspond to steps in the derivation of a dependency tree. The idea is that a sequence of valid transitions, starting in the initial configuration for a given sentence and ending in one of several terminal configurations, defines a valid dependency tree for the input sentence. In this way, the transition system determines the constraint set Γ in the parsing model, since it implicitly defines the set of permissible dependency trees for a given sentence, but it also determines the parameter set $\boldsymbol{\lambda}$ that have to be learned from data, as we shall see later on. For most of this chapter, we will concentrate on a simple stack-based transition system, which implements a form of shift-reduce parsing and exemplifies the most widely

used approach in transition-based dependency parsing. In section 3.4, we will briefly discuss some of the alternative systems that have been proposed.

We start by defining configurations as triples consisting of a stack, an input buffer, and a set of dependency arcs.

Definition 3.1. Given a set R of dependency types, a *configuration* for a sentence $S = w_0 w_1 \dots w_n$ is a triple $c = (\sigma, \beta, A)$, where

1. σ is a stack of words $w_i \in V_S$,
2. β is a buffer of words $w_i \in V_S$,
3. A is a set of dependency arcs $(w_i, r, w_j) \in V_S \times R \times V_S$.

The idea is that a configuration represents a partial analysis of the input sentence, where the words on the stack σ are partially processed words, the words in the buffer β are the remaining input words, and the arc set A represents a partially built dependency tree. For example, if the input sentence is

Economic news had little effect on financial markets.

then the following is a valid configuration, where the stack contains the words ROOT and *news* (with the latter on top), the buffer contains all the remaining words except *Economic*, and the arc set contains a single arc connecting the head *news* to the dependent *Economic* with the label ATT:

([ROOT, news]$_\sigma$, [had, little, effect, on, financial, markets, .]$_\beta$, {(news, ATT, Economic)}$_A$)

Note that we represent both the stack and the buffer as simple lists, with elements enclosed in square brackets (and subscripts σ and β when needed), although the stack has its head (or top) to the right for reasons of perspicuity. When convenient, we use the notation $\sigma|w_i$ to represent the stack which results from pushing w_i onto the stack σ, and we use $w_i|\beta$ to represent a buffer with head w_i and tail β.[1]

Definition 3.2. For any sentence $S = w_0 w_1 \dots w_n$,

1. the *initial* configuration $c_0(S)$ is $([w_0]_\sigma, [w_1, \dots, w_n]_\beta, \emptyset)$,
2. a *terminal* configuration is a configuration of the form $(\sigma, [\,]_\beta, A)$ for any σ and A.

Thus, we initialize the system to a configuration with w_0 = ROOT on the stack, all the remaining words in the buffer, and an empty arc set; and we terminate in any configuration that has an empty buffer (regardless of the state of the stack and the arc set).

[1]The operator | is taken to be left-associative for the stack and right-associative for the buffer.

Transition		Precondition
LEFT-ARC$_r$	$(\sigma \mid w_i, w_j \mid \beta, A) \Rightarrow (\sigma, w_j \mid \beta, A \cup \{(w_j, r, w_i)\})$	$i \neq 0$
RIGHT-ARC$_r$	$(\sigma \mid w_i, w_j \mid \beta, A) \Rightarrow (\sigma, w_i \mid \beta, A \cup \{(w_i, r, w_j)\})$	
SHIFT	$(\sigma, w_i \mid \beta, A) \Rightarrow (\sigma \mid w_i, \beta, A)$	

Figure 3.1: Transitions for shift-reduce dependency parsing.

Having defined the set of configurations, including a unique initial configuration and a set of terminal configurations for any sentence, we now define transitions between configurations. Formally speaking, a transition is a partial function from configurations to configurations, i.e., a transition maps a given configuration to a new configuration but may be undefined for certain configurations. Conceptually, a transition corresponds to a basic parsing action that adds an arc to the dependency tree or modifies the stack or the buffer. The transitions needed for shift-reduce dependency parsing are defined in figure 3.1 and contain three types of transitions:

1. Transitions LEFT-ARC$_r$ (for any dependency label r) add a dependency arc (w_j, r, w_i) to the arc set A, where w_i is the word on top of the stack and w_j is the first word in the buffer. In addition, they pop the stack. They have as precondition that both the stack and the buffer are non-empty and that $w_i \neq$ ROOT.[2]

2. Transitions RIGHT-ARC$_r$ (for any dependency label r) add a dependency arc (w_i, r, w_j) to the arc set A, where w_i is the word on top of the stack and w_j is the first word in the buffer. In addition, they pop the stack and replace w_j by w_i at the head of buffer.[3] They have as their only precondition that both the stack and the buffer are non-empty.

3. The transition SHIFT removes the first word w_i in the buffer and pushes it on top of the stack. It has as its only precondition that the buffer is non-empty.

We use the symbol $\mathcal{T}$ to refer to the set of permissible transitions in a given transition system. As noted above, transitions correspond to elementary parsing actions. In order to define complete parses, we introduce the notion of a *transition sequence*.

Definition 3.3. A *transition sequence* for a sentence $S = w_0 w_1 \ldots w_n$ is a sequence of configurations $C_{0,m} = (c_0, c_1, \ldots, c_m)$ such that

[2]The latter precondition guarantees that the dependency graph defined by the arc set always satisfies the root property.
[3]This may seem counterintuitive, given that the buffer is meant to contain words that have not yet been processed, but it is necessary in order to allow w_j to attach to a head on its left.

Transition	Configuration		
	([ROOT],	[Economic, . . . , .],	∅)
SH ⇒	([ROOT, Economic],	[news, . . . , .],	∅)
LA_{ATT} ⇒	([ROOT],	[news, . . . , .],	$A_1 = \{(news, ATT, Economic)\}$)
SH ⇒	([ROOT, news],	[had, . . . , .],	A_1)
LA_{SBJ} ⇒	([ROOT],	[had, . . . , .],	$A_2 = A_1 \cup \{(had, SBJ, news)\}$)
SH ⇒	([ROOT, had],	[little, . . . , .],	A_2)
SH ⇒	([ROOT, had, little],	[effect, . . . , .],	A_2)
LA_{ATT} ⇒	([ROOT, had],	[effect, . . . , .],	$A_3 = A_2 \cup \{(effect, ATT, little)\}$)
SH ⇒	([ROOT, had, effect],	[on, . . . , .],	A_3)
SH ⇒	([ROOT, . . . on],	[financial, markets, .],	A_3)
SH ⇒	([ROOT, . . . , financial],	[markets, .],	A_3)
LA_{ATT} ⇒	([ROOT, . . . on],	[markets, .],	$A_4 = A_3 \cup \{(markets, ATT, financial)\}$)
RA_{PC} ⇒	([ROOT, had, effect],	[on, .],	$A_5 = A_4 \cup \{(on, PC, markets)\}$)
RA_{ATT} ⇒	([ROOT, had],	[effect, .],	$A_6 = A_5 \cup \{(effect, ATT, on)\}$)
RA_{OBJ} ⇒	([ROOT],	[had, .],	$A_7 = A_6 \cup \{(had, OBJ, effect)\}$)
SH ⇒	([ROOT, had],	[.],	A_7)
RA_{PU} ⇒	([ROOT],	[had],	$A_8 = A_7 \cup \{(had, PU, .)\}$)
RA_{PRED} ⇒	([ ],	[ROOT],	$A_9 = A_8 \cup \{(ROOT, PRED, had)\}$)
SH ⇒	([ROOT],	[ ],	A_9)

Figure 3.2: Transition sequence for the English sentence in figure 1.1 (LA_r = LEFT-ARC$_r$, RA_r = RIGHT-ARC$_r$, SH = SHIFT).

1. c_0 is the initial configuration $c_0(S)$ for S,
2. c_m is a terminal configuration,
3. for every i such that $1 \leq i \leq m$, there is a transition $t \in \mathcal{T}$ such that $c_i = t(c_{i-1})$.

A transition sequence starts in the initial configuration for a given sentence and reaches a terminal configuration by applying valid transitions from one configuration to the next. The dependency tree derived through this transition sequence is the dependency tree defined by the terminal configuration, i.e., the tree $G_{c_m} = (V_S, A_{c_m})$, where A_{c_m} is the arc set in the terminal configuration c_m. By way of example, figure 3.2 shows a transition sequence that derives the dependency tree shown in figure 1.1 on page 2.

The transition system defined for dependency parsing in this section leads to derivations that correspond to basic shift-reduce parsing for context-free grammars. The LEFT-ARC$_r$ and RIGHT-ARC$_r$ transitions correspond to reduce actions, replacing a head-dependent structure with its head, while the SHIFT transition is exactly the same as the shift action. One peculiarity of the transitions, as defined here, is that the "reduce transitions" apply to one node on the stack and one node in the buffer, rather than two nodes on the stack. This simplifies the definition of terminal configurations and has become standard in the dependency parsing literature.

Every transition sequence in this system defines a dependency graph with the *spanning*, *root*, and *single-head* properties, but not necessarily with the *connectedness* property. This means that not every transition sequence defines a dependency tree, as defined in chapter 2. To take a trivial example, a transition sequence for a sentence S consisting only of SHIFT transitions defines the graph $G = (V_S, \emptyset)$, which is not connected but which satisfies all the other properties. However, since any transition sequence defines an *acyclic* dependency graph G, it is trivial to convert G into a dependency tree G' by adding arcs of the form (ROOT, r, w_i) (with some dependency label r) for every w_i that is a root in G. As noted in section 2.1.1, a dependency graph G that satisfies the spanning, root, single-head, and acyclic properties is equivalent to a set of dependency trees and is often called a *dependency forest*.

Another important property of the system is that every transition sequence defines a *projective* dependency forest,[4] which is advantageous from the point of view of efficiency but overly restrictive from the point of view of representational adequacy. In sections 3.4 and 3.5, we will see how this limitation can be overcome, either by modifying the transition system or by complementing it with pre- and post-processing.

Given that every transition sequence defines a projective dependency forest, which can be turned into a dependency tree, we say that the system is *sound* with respect to the set of projective dependency trees. A natural question is whether the system is also *complete* with respect to this class of dependency trees, that is, whether every projective dependency tree is defined by some transition sequence. The answer to this question is affirmative, although we will not prove it here.[5] In terms of our parsing model $M = (\Gamma, \boldsymbol{\lambda}, h)$, we can therefore say that the transition system described in this section corresponds to a set of constraints Γ characterizing the set $\mathcal{G}_S^p$ of projective dependency trees for a given sentence S (relative to a set of arc labels R).

3.2 PARSING ALGORITHM

The transition system defined in section 3.1 is nondeterministic in the sense that there is usually more than one transition that is valid for any non-terminal configuration.[6] Thus, in order to perform deterministic parsing, we need a mechanism to determine for any non-terminal configuration c, what is the correct transition out of c. Let us assume for the time being that we are given an *oracle*, that is, a function o from configurations to transitions such that $o(c) = t$ if and only if t is the correct transition out of c. Given such an oracle, deterministic parsing can be achieved by the very simple algorithm in figure 3.3.

We start in the initial configuration $c_0(S)$ and, as long as we have not reached a terminal configuration, we use the oracle to find the optimal transition $t = o(c)$ and apply it to our current configuration to reach the next configuration $t(c)$. Once we reach a terminal configuration, we simply return the dependency tree defined by our current arc set. Note that, while finding the

[4] A dependency forest is projective if and only if all component trees are projective.

[5] The interested reader is referred to Nivre (2008) for proofs of soundness and completeness for this and several other transition systems for dependency parsing.

[6] The notable exception is a configuration with an empty stack, where only SHIFT is possible.

h(S, Γ, o)
1 $c \leftarrow c_0(S)$
2 **while** c is not terminal
3 $\quad t \leftarrow o(c)$
4 $\quad c \leftarrow t(c)$
5 **return** G_c

Figure 3.3: Deterministic, transition-based parsing with an oracle.

optimal transition $t = o(c)$ is a hard problem, which we have to tackle using machine learning, computing the next configuration $t(c)$ is a purely mechanical operation.

It is easy to show that, as long as there is at least one valid transition for every non-terminal configuration, such a parser will construct exactly one transition sequence $C_{0,m}$ for a sentence S and return the dependency tree defined by the terminal configuration c_m, i.e., $G_{c_m} = (V_S, A_{c_m})$. To see that there is always at least one valid transition out of a non-terminal configuration, we only have to note that such a configuration must have a non-empty buffer (otherwise it would be terminal), which means that at least SHIFT is a valid transition.

The time complexity of the deterministic, transition-based parsing algorithm is $O(n)$, where n is the number of words in the input sentence S, provided that the oracle and transition functions can be computed in constant time. This holds since the worst-case running time is bounded by the maximum number of transitions in a transition sequence $C_{0,m}$ for a sentence $S = w_0 w_1 \ldots w_n$. Since a SHIFT transition decreases the length of the buffer by 1, no other transition increases the length of the buffer, and any configuration with an empty buffer is terminal, the number of SHIFT transitions in $C_{0,m}$ is bounded by n. Moreover, since both LEFT-ARC$_r$ and RIGHT-ARC$_r$ decrease the height of the stack by 1, only SHIFT increases the height of the stack by 1, and the initial height of the stack is 1, the combined number of instances of LEFT-ARC$_r$ and RIGHT-ARC$_r$ in $C_{0,m}$ is also bounded by n. Hence, the worst-case time complexity is $O(n)$.

So far, we have seen how transition-based parsing can be performed in linear time if restricted to projective dependency trees, and provided that we have a constant-time oracle that predicts the correct transition out of any non-terminal configuration. Of course, oracles are hard to come by in real life, so in order to build practical parsing systems, we need to find some other mechanism that we can use to approximate the oracle well enough to make accurate parsing feasible. There are many conceivable ways of approximating oracles, including the use of formal grammars and disambiguation heuristics. However, the most successful strategy to date has been to take a data-driven approach, approximating oracles by *classifiers* trained on treebank data. This leads to the notion of classifier-based parsing, which is an essential component of transition-based dependency parsing.

h$(S, \Gamma, \boldsymbol{\lambda})$
1 $c \leftarrow c_0(S)$
2 **while** c is not terminal
3 $t \leftarrow \lambda_c$
4 $c \leftarrow t(c)$
5 **return** G_c

Figure 3.4: Deterministic, transition-based parsing with a classifier.

3.3 CLASSIFIER-BASED PARSING

Let us step back for a moment to our general characterization of a data-driven parsing model as $M = (\Gamma, \boldsymbol{\lambda}, h)$, where Γ is a set of constraints on dependency graphs, $\boldsymbol{\lambda}$ is a set of model parameters and h is a fixed parsing algorithm. In the previous two sections, we have shown how we can define the parsing algorithm h as deterministic best-first search in a transition system (although other search strategies are possible, as we shall see later on). The transition system determines the set of constraints Γ, but it also defines the model parameters $\boldsymbol{\lambda}$ that need to be learned from data, since we need to be able to predict the oracle transition $o(c)$ for every possible configuration c (for any input sentence S). We use the notation $\lambda_c \in \boldsymbol{\lambda}$ to denote the transition predicted for c according to model parameters $\boldsymbol{\lambda}$, and we can think of $\boldsymbol{\lambda}$ as a huge table containing the predicted transition λ_c for every possible configuration c. In practice, $\boldsymbol{\lambda}$ is normally a compact representation of a function for computing λ_c given c, but the details of this representation need not concern us now. Given a learned model, we can perform deterministic, transition-based parsing using the algorithm in figure 3.4, where we have simply replaced the oracle function o by the learned parameters $\boldsymbol{\lambda}$ (and the function value $o(c)$ by the specific parameter value λ_c).

However, in order to make the learning problem tractable by standard machine learning techniques, we need to introduce an abstraction over the infinite set of possible configurations. This is what is achieved by the feature function $\mathbf{f}(x) : \mathcal{X} \rightarrow \mathcal{Y}$ (cf. section 2.2). In our case, the domain $\mathcal{X}$ is the set $\mathcal{C}$ of possible configurations (for any sentence S) and the range $\mathcal{Y}$ is a product of m feature value sets, which means that the feature function $\mathbf{f}(c) : \mathcal{C} \rightarrow \mathcal{Y}$ maps every configuration to an m-dimensional feature vector. Given this representation, we then want to learn a classifier $g : \mathcal{Y} \rightarrow \mathcal{T}$, where $\mathcal{T}$ is the set of possible transitions, such that $g(\mathbf{f}(c)) = o(c)$ for any configuration c. In other words, given a training set of gold standard dependency trees from a treebank, we want to learn a classifier that predicts the oracle transition $o(c)$ for any configuration c, given as input the feature representation $\mathbf{f}(c)$. This gives rise to three basic questions:

- How do we represent configurations by feature vectors?
- How do we derive training data from treebanks?
- How do we train classifiers?

We will deal with each of these questions in turn, starting with feature representations in section 3.3.1, continuing with the derivation of training data in section 3.3.2, and finishing off with the training of classifiers in section 3.3.3.

3.3.1 FEATURE REPRESENTATIONS

A feature representation $\mathbf{f}(c)$ of a configuration c is an m-dimensional vector of simple features $\mathbf{f}_i(c)$ (for $1 \leq i \leq m$). In the general case, these simple features can be defined by arbitrary attributes of a configuration, which may be either categorical or numerical. For example, "the part of speech of the word on top of the stack" is a categorical feature, with values taken from a particular part-of-speech tagset (e.g., NN for noun, VB for verb, etc.). By contrast, "the number of dependents previously attached to the word on top of the stack" is a numerical feature, with values taken from the set $\{0, 1, 2, \ldots\}$. The choice of feature representations is partly dependent on the choice of learning algorithm, since some algorithms impose special restrictions on the form that feature values may take, for example, that all features must be numerical. However, in the interest of generality, we will ignore this complication for the time being and assume that features can be of either type. This is unproblematic since it is always possible to convert categorical features to numerical features, and it will greatly simplify the discussion of feature representations for transition-based parsing.

The most important features in transition-based parsing are defined by attributes of words, or tree nodes, identified by their position in the configuration. It is often convenient to think of these features as defined by two simpler functions, an *address function* identifying a particular word in a configuration (e.g., the word on top of the stack) and an *attribute function* selecting a specific attribute of this word (e.g., its part of speech). We call these features *configurational word features* and define them as follows.

Definition 3.4. Given an input sentence $S = w_0 w_1 \ldots w_n$ with node set V_S, a function $(v \circ a)(c) : \mathcal{C} \rightarrow Y$ composed of

1. an address function $a(c) : \mathcal{C} \rightarrow V_S$,
2. an attribute function $v(w) : V_S \rightarrow Y$.

is a *configurational word feature*.

An address function can in turn be composed of simpler functions, which operate on different components of the input configuration c. For example:

- Functions that extract the kth word (from the top) of the stack or the kth word (from the head) of the buffer.
- Functions that map a word w to its parent, leftmost child, rightmost child, leftmost sibling, or rightmost sibling in the dependency graph defined by c.

Table 3.1: Feature model for transition-based parsing.

f_i	Address	Attribute
1	STK[0]	FORM
2	BUF[0]	FORM
3	BUF[1]	FORM
4	LDEP(STK[0])	DEPREL
5	RDEP(STK[0])	DEPREL
6	LDEP(BUF[0])	DEPREL
7	RDEP(BUF[0])	DEPREL

By defining such functions, we can construct arbitrarily complex address functions that extract, e.g., "the rightmost sibling of the leftmost child of the parent of the word on top of the stack" although the address functions used in practice typically combine at most three such functions. It is worth noting that most address functions are partial, which means that they may fail to return a word. For example, a function that is supposed to return the leftmost child of the word on top of the stack is undefined if the stack is empty or if the word on top of the stack does not have any children. In this case, any feature defined with this address function will also be undefined (or have a special null value).

The typical attribute functions refer to some linguistic property of words, which may be given as input to the parser or computed as part of the parsing process. We can exemplify this with the word *markets* from the sentence in figure 1.1:

- Identity of w_i = *markets*
- Identity of lemma of w_i = *market*
- Identity of part-of-speech tag for w_i = NNS
- Identity of dependency label for w_i = PC

The first three attributes are *static* in the sense that they are constant, if available at all, in every configuration for a given sentence. That is, if the input sentence has been lemmatized and tagged for parts of speech in preprocessing, then the values of these features are available for all words of the sentence, and their values do not change during parsing. By contrast, the dependency label attribute is *dynamic* in the sense that it is available only after the relevant dependency arc has been added to the arc set. Thus, in the transition sequence in figure 3.2, the dependency label for the word *markets* is undefined in the first twelve configurations, but has the value PC in all the remaining configurations. Hence, such attributes can be used to define features of the transition history and the partially built dependency tree, which turns out to be one of the major advantages of the transition-based approach.

$\mathbf{f}(c_0)$	=	(ROOT	Economic	news	NULL	NULL	NULL	NULL)
$\mathbf{f}(c_1)$	=	(Economic	news	had	NULL	NULL	NULL	NULL)
$\mathbf{f}(c_2)$	=	(ROOT	news	had	NULL	NULL	ATT	NULL)
$\mathbf{f}(c_3)$	=	(news	had	little	ATT	NULL	NULL	NULL)
$\mathbf{f}(c_4)$	=	(ROOT	had	little	NULL	NULL	SBJ	NULL)
$\mathbf{f}(c_5)$	=	(had	little	effect	SBJ	NULL	NULL	NULL)
$\mathbf{f}(c_6)$	=	(little	effect	on	NULL	NULL	NULL	NULL)
$\mathbf{f}(c_7)$	=	(had	effect	on	SBJ	NULL	ATT	NULL)
$\mathbf{f}(c_8)$	=	(effect	on	financial	ATT	NULL	NULL	NULL)
$\mathbf{f}(c_9)$	=	(on	financial	markets	NULL	NULL	NULL	NULL)
$\mathbf{f}(c_{10})$	=	(financial	markets	.	NULL	NULL	NULL	NULL)
$\mathbf{f}(c_{11})$	=	(on	markets	.	NULL	NULL	ATT	NULL)
$\mathbf{f}(c_{12})$	=	(effect	on	.	ATT	NULL	NULL	ATT)
$\mathbf{f}(c_{13})$	=	(had	effect	.	SBJ	NULL	ATT	ATT)
$\mathbf{f}(c_{14})$	=	(ROOT	had	.	NULL	NULL	SBJ	OBJ)
$\mathbf{f}(c_{15})$	=	(had	.	NULL	SBJ	OBJ	NULL	NULL)
$\mathbf{f}(c_{16})$	=	(ROOT	had	NULL	NULL	NULL	SBJ	PU)
$\mathbf{f}(c_{17})$	=	(NULL	ROOT	NULL	NULL	NULL	NULL	PRED)
$\mathbf{f}(c_{18})$	=	(ROOT	NULL	NULL	NULL	PRED	NULL	NULL)

Figure 3.5: Feature vectors for the configurations in figure 3.2.

Let us now try to put all the pieces together and examine a complete feature representation using only configurational word features. Table 3.1 shows a simple model with seven features, each defined by an address function and an attribute function. We use the notation STK[i] and BUF[i] for the ith word in the stack and in the buffer, respectively,[7] and we use LDEP(w) and RDEP(w) for the farthest child of w to the left and to the right, respectively. The attribute functions used are FORM for word form and DEPREL for dependency label. In figure 3.5, we show how the value of the feature vector changes as we go through the configurations of the transition sequence in figure 3.2.[8]

Although the feature model defined in figure 3.1 is quite sufficient to build a working parser, a more complex model is usually required to achieve good parsing accuracy. To give an idea of the complexity involved, table 3.2 depicts a model that is more representative of state-of-the-art parsing systems. In table 3.2, rows represent address functions, defined using the same operators as in the earlier example, while columns represent attribute functions, which now also include LEMMA (for

[7] Note that indexing starts at 0, so that STK[0] is the word on top of the stack, while BUF[0] is the first word in the buffer.
[8] The special value NULL is used to indicate that a feature is undefined in a given configuration.

Table 3.2: Typical feature model for transition-based parsing with rows representing address functions, columns representing attribute functions, and cells with + representing features.

	Attributes				
Address	**FORM**	**LEMMA**	**POSTAG**	**FEATS**	**DEPREL**
STK[0]	+	+	+	+	
STK[1]			+		
LDEP(STK[0])					+
RDEP(STK[0])					+
BUF[0]	+	+	+	+	
BUF[1]	+		+		
BUF[2]			+		
BUF[3]			+		
LDEP(BUF[0])					+
RDEP(BUF[0])					+

lemma or base form) and FEATS (for morphosyntactic features in addition to the basic part of speech). Thus, each cell represents a possible feature, obtained by composing the corresponding address function and attribute function, but only cells containing a + sign correspond to features present in the model.

We have focused in this section on configurational word features, i.e., features that can be defined by the composition of an address function and an attribute function, since these are the most important features in transition-based parsing. In principle, however, features can be defined over any properties of a configuration that are believed to be important for predicting the correct transition. One type of feature that has often been used is the *distance* between two words, typically the word on top of the stack and the first word in the input buffer. This can be measured by the number of words intervening, possibly restricted to words of a certain type such as verbs. Another common type of feature is the number of children of a particular word, possibly divided into left children and right children.

3.3.2 TRAINING DATA

Once we have defined our feature representation, we want to learn to predict the correct transition $o(c)$, for any configuration c, given the feature representation $\mathbf{f}(c)$ as input. In machine learning terms, this is a straightforward *classification* problem, where the instances to be classified are (feature representations of) configurations, and the classes are the possible transitions (as defined by the transition system). In a supervised setting, the training data should consist of instances labeled with their correct class, which means that our training instances should have the form $(\mathbf{f}(c), t)$ $(t = o(c))$. However, this is not the form in which training data are directly available to us in a treebank.

In section 2.2, we characterized a training set $\mathcal{D}$ for supervised dependency parsing as consisting of sentences paired with their correct dependency trees:

$$\mathcal{D} = \{(S_d, G_d)\}_{d=0}^{|\mathcal{D}|}$$

In order to train a classifier for transition-based dependency parsing, we must therefore find a way to derive from $\mathcal{D}$ a new training set $\mathcal{D}'$, consisting of configurations paired with their correct transitions:

$$\mathcal{D}' = \{(\mathbf{f}(c_d), t_d)\}_{d=0}^{|\mathcal{D}'|}$$

Here is how we construct $\mathcal{D}'$ given $\mathcal{D}$:

- For every instance $(S_d, G_d) \in \mathcal{D}$, we first construct a transition sequence $C_{0,m}^d = (c_0, c_1, \ldots, c_m)$ such that
  1. $c_0 = c_0(S_d)$,
  2. $G_d = (V_d, A_{c_m})$.
- For every non-terminal configuration $c_i^d \in C_{0,m}^d$, we then add to $\mathcal{D}'$ an instance $(\mathbf{f}(c_i^d), t_i^d)$, where $t_i^d(c_i^d) = c_{i+1}^d$.

This scheme presupposes that, for every sentence S_d with dependency tree G_d, we can construct a transition sequence that results in G_d. Provided that all dependency trees are projective, we can do this using the parsing algorithm defined in section 3.2 and relying on the dependency tree $G_d = (V_d, A_d)$ to compute the oracle function in line 3 as follows:

$$o(c = (\sigma, \beta, A)) = \begin{cases} \textsc{Left-Arc}_r & \text{if } (\beta[0], r, \sigma[0]) \in A_d \\ \textsc{Right-Arc}_r & \text{if } (\sigma[0], r, \beta[0]) \in A_d \text{ and, for all } w, r', \\ & \text{if } (\beta[0], r', w) \in A_d \text{ then } (\beta[0], r', w) \in A \\ \textsc{Shift}_r & \text{otherwise} \end{cases}$$

The first case states that the correct transition is $\textsc{Left-Arc}_r$ if the correct dependency tree has an arc from the first word $\beta[0]$ in the input buffer to the word $\sigma[0]$ on top of the stack with dependency label r. The second case states that the correct transition is $\textsc{Right-Arc}_r$ if the correct dependency tree has an arc from $\sigma[0]$ to $\beta[0]$ with dependency label r – but only if all the outgoing arcs from $\beta[0]$ (according to G_d) have already been added to A. The extra condition is needed because, after the $\textsc{Right-Arc}_r$ transition, the word $\beta[0]$ will no longer be in either the stack or the buffer, which means that it will be impossible to add more arcs involving this word. No corresponding condition is needed for the $\textsc{Left-Arc}_r$ case since this will be satisfied automatically as long as the correct dependency tree is projective. The third and final case takes care of all remaining configurations, where $\textsc{Shift}$ has to be the correct transition, including the special case where the stack is empty.

3.3.3 CLASSIFIERS

Training a classifier on the set $\mathcal{D}' = \{(\mathbf{f}(c_d), t_d)\}_{d=0}^{|\mathcal{D}'|}$ is a standard problem in machine learning, which can be solved using a variety of different learning algorithms. We will not go into the details of how to do this but limit ourselves to some observations about two of the most popular methods in transition-based dependency parsing: memory-based learning and support vector machines.

Memory-based learning and classification is a so-called lazy learning method, where learning basically consists in storing the training instances while classification is based on similarity-based reasoning (Daelemans and Van den Bosch, 2005). More precisely, classification is achieved by retrieving the k most similar instances from memory, given some similarity metric, and extrapolating the class of a new instance from the classes of the retrieved instances. This is usually called k nearest neighbor classification, which in the simplest case amounts to taking the majority class of the k nearest neighbors although there are a number of different similarity metrics and weighting schemes that can be used to improve performance. Memory-based learning is a purely discriminative learning technique in the sense that it maps input instances to output classes without explicitly computing a probability distribution over outputs or inputs (although it is possible to extract metrics that can be used to estimate probabilities). One advantage of this approach is that it can handle categorical features as well as numerical ones, which means that feature vectors for transition-based parsing can be represented directly as shown in section 3.3.1 above, and that it handles multi-class classification without special techniques. Memory-based classifiers are very efficient to train, since learning only consists in storing the training instances for efficient retrieval. On the other hand, this means that most of the computation must take place at classification time, which can make parsing inefficient, especially with large training sets.

Support vector machines are max-margin linear classifiers, which means that they try to separate the classes in the training data with the widest possible margin (Vapnik, 1995). They are especially powerful in combination with kernel functions, which in essence can be used to transform feature representations to higher dimensionality and thereby achieve both an implicit feature combination and non-linear classification. For transition-based parsing, polynomial kernels of degree 2 or higher are widely used, with the effect that pairs of features in the original feature space are implicitly taken into account. Since support vector machines can only handle numerical features, all categorical features need to be transformed into binary features. That is, a categorical feature with m possible values is replaced with m features with possible values 0 and 1. The categorical feature assuming its ith value is then equivalent to the ith binary feature having the value 1 while all other features have the value 0. In addition, support vector machines only perform binary classification, but there are several techniques for solving the multi-class case. Training can be computationally intensive for support vector machines with polynomial kernels, so for large training sets special techniques often must be used to speed up training. One commonly used technique is to divide the training data into smaller bins based on the value of some (categorical) feature, such as the part of speech of the word on top of the stack. Separate classifiers are trained for each bin, and only one of them is invoked for a given configuration during parsing (depending on the value of the feature

Transition		Preconditions
LEFT-ARC$_r$	$(\sigma \mid w_i, w_j \mid \beta, A) \Rightarrow (\sigma, w_j \mid \beta, A \cup \{(w_j, r, w_i)\})$	$(w_k, r', w_i) \notin A$ $i \neq 0$
RIGHT-ARC$_r$	$(\sigma \mid w_i, w_j \mid \beta, A) \Rightarrow (\sigma \mid w_i \mid w_j, \beta, A \cup \{(w_i, r, w_j)\})$	
REDUCE	$(\sigma \mid w_i, \beta, A) \Rightarrow (\sigma, \beta, A)$	$(w_k, r', w_i) \in A$
SHIFT	$(\sigma, w_i \mid \beta, A) \Rightarrow (\sigma \mid w_i, \beta, A)$	

Figure 3.6: Transitions for arc-eager shift-reduce dependency parsing.

used to define the bins). Support vector machines with polynomial kernels currently represent the state of the art in terms of accuracy for transition-based dependency parsing.

3.4 VARIETIES OF TRANSITION-BASED PARSING

So far, we have considered a single transition system, defined in section 3.1, and a single, deterministic parsing algorithm, introduced in section 3.2. However, there are many possible variations on the basic theme of transition-based parsing, obtained by varying the transition system, the parsing algorithm, or both. In addition, there are many possible learning algorithms that can be used to train classifiers, a topic that was touched upon in the previous section. In this section, we will introduce some alternative transition systems (section 3.4.1) and some variations on the basic parsing algorithm (section 3.4.2). Finally, we will discuss how non-projective dependency trees can be processed even if the underlying transition system only derives projective dependency trees (section 3.5).

3.4.1 CHANGING THE TRANSITION SYSTEM

One of the peculiarities of the transition system defined earlier in this chapter is that right dependents cannot be attached to their head until all their dependents have been attached. As a consequence, there may be uncertainty about whether a RIGHT-ARC$_r$ transition is appropriate, even if it is certain that the first word in the input buffer should be a dependent of the word on top of the stack. This problem is eliminated in the *arc-eager* version of this transition system, defined in figure 3.6. In this system, which is called arc-eager because all arcs (whether pointing to the left or to the right) are added as soon as possible, the RIGHT-ARC$_r$ is redefined so that the dependent word w_j is pushed onto the stack (on top of its head w_i), making it possible to add further dependents to this word. In addition, we have to add a new transition REDUCE, which makes it possible to pop the dependent word from the stack at a later point in time, and which has as a precondition that the word on the top of the stack already has a head, i.e., that the arc set contains an arc (w_k, r', w_i) for some k

Transition	Configuration		
	([ROOT],	[Economic, . . . , .],	∅)
SH ⇒	([ROOT, Economic],	[news, . . . , .],	∅)
LA_{ATT} ⇒	([ROOT],	[news, . . . , .],	$A_1 = \{(\text{news, ATT, Economic})\}$)
SH ⇒	([ROOT, news],	[had, . . . , .],	A_1)
LA_{SBJ} ⇒	([ROOT],	[had, . . . , .],	$A_2 = A_1 \cup \{(\text{had, SBJ, news})\}$)
RA_{PRED} ⇒	([ROOT, had],	[little, . . . , .],	$A_3 = A_2! \cup \{(\text{ROOT, PRED, had})\}$)
SH ⇒	([ROOT, had, little],	[effect, . . . , .],	A_3)
LA_{ATT} ⇒	([ROOT, had],	[effect, . . . , .],	$A_4 = A_3 \cup \{(\text{effect, ATT, little})\}$)
RA_{OBJ} ⇒	([ROOT, had, effect],	[on, . . . , .],	$A_5 = A_4 \cup \{(\text{had, OBJ, effect})\}$)
RA_{ATT} ⇒	([ROOT, . . . on],	[financial, markets, .],	$A_6 = A_5 \cup \{(\text{effect, ATT, on})\}$)
SH ⇒	([ROOT, . . . , financial],	[markets, .],	A_6)
LA_{ATT} ⇒	([ROOT, . . . on],	[markets, .],	$A_7 = A_6 \cup \{(\text{markets, ATT, financial})\}$)
RA_{PC} ⇒	([ROOT, . . . , markets],	[.],	$A_8 = A_7 \cup \{(\text{on, PC, markets})\}$)
RE ⇒	([ROOT, . . . , on],	[.],	A_8)
RE ⇒	([ROOT, . . . , effect],	[.],	A_8)
RE ⇒	([ROOT, had],	[.],	A_8)
RA_{PU} ⇒	([ROOT, . . . , .],	[ ],	$A_9 = A_8 \cup \{(\text{had, PU, .})\}$)

Figure 3.7: Arc-eager transition sequence for the English sentence in figure 1.1 (LA_r = LEFT-ARC$_r$, RA_r = RIGHT-ARC$_r$, RE = REDUCE, SH = SHIFT).

and r' (where w_i is the word on top of the stack).[9] To further illustrate the difference between the two systems, figure 3.7 shows the transition sequence needed to parse the sentence in figure 1.1 in the arc-eager system (cf. figure 3.2). Despite the differences, however, both systems are sound and complete with respect to the class of projective dependency trees (or forests that can be turned into trees, to be exact), and both systems have linear time and space complexity when coupled with the deterministic parsing algorithm formulated in section 3.2 (Nivre, 2008). As parsing models, the two systems are therefore equivalent with respect to the Γ and h components but differ with respect to the $\boldsymbol{\lambda}$ component, since the different transition sets give rise to different parameters that need to be learned from data.

Another kind of variation on the basic transition system is to add transitions that will allow a certain class of non-projective dependencies to be processed. Figure 3.8 shows two such transitions called NP-LEFT$_r$ and NP-RIGHT$_r$, which behave exactly like the ordinary LEFT-ARC$_r$ and RIGHT-ARC$_r$ transitions, except that they apply to the second word from the top of the stack and treat the top word as a context node that is unaffected by the transition. Unless this context node is later attached to the head of the new arc, the resulting tree will be non-projective. Although this system cannot

[9] Moreover, we have to add a new precondition to the LEFT-ARC$_r$ transition to prevent that it applies when the word on top of the stack already has a head, a situation that could never arise in the old system. The precondition rules out the existence of an arc (w_k, r', w_i) in the arc set (for any k and r').

Transition		Precondition
NP-LEFT$_r$	$(\sigma \mid w_i \mid w_k, w_j \mid \beta, A) \Rightarrow (\sigma \mid w_k, w_j \mid \beta, A \cup \{(w_j, r, w_i)\})$	$i \neq 0$
NP-RIGHT$_r$	$(\sigma \mid w_i \mid w_k, w_j \mid \beta, A) \Rightarrow (\sigma \mid w_i, w_k \mid \beta, A \cup \{(w_i, r, w_j)\})$	

Figure 3.8: Added transitions for non-projective shift-reduce dependency parsing.

cope with arbitrary non-projective dependency trees, it can process many of the non-projective constructions that occur in natural languages (Attardi, 2006).

In order to construct a transition system that can handle arbitrary non-projective dependency trees, we can modify not only the set of transitions but also the set of configurations. For example, if we define configurations with two stacks instead of one, we can give a transition-based account of the algorithms for dependency parsing discussed by Covington (2001). With an appropriate choice of transitions, we can then define a system that is sound and complete with respect to the class $\mathcal{G}_S$ of arbitrary dependency for a given sentence S. The space complexity for deterministic parsing with an oracle remains $O(n)$ but the time complexity is now $O(n^2)$. To describe this system here would take us too far afield, so the interested reader is referred to Nivre (2008).

3.4.2 CHANGING THE PARSING ALGORITHM

The parsing algorithm described in section 3.2 performs a greedy, deterministic search for the optimal transition sequence, exploring only a single transition sequence and terminating as soon as it reaches a terminal configuration. Given one of the transition systems described so far, this happens after a single left-to-right pass over the words of the input sentence. One alternative to this single-pass strategy is to perform multiple passes over the input while still exploring only a single path through the transition system in each pass. For example, given the transition system defined in section 3.1, we can reinitialize the parser by refilling the buffer with the words that are on the stack in the terminal configuration and keep iterating until there is only a single word on the stack or no new arcs were added during the last iteration. This is essentially the algorithm proposed by Yamada and Matsumoto (2003) and commonly referred to as Yamada's algorithm. In the worst case, this may lead to $n - 1$ passes over the input, each pass taking $O(n)$ time, which means that the total running time is $O(n^2)$, although the worst case almost never occurs in practice.

Another variation on the basic parsing algorithm is to relax the assumption of determinism and to explore more than one transition sequence in a single pass. The most straightforward way of doing this is to use *beam search*, that is, to retain the k most promising partial transition sequences after each transition step. This requires that we have a way of scoring and ranking all the possible transitions out of a given configuration, which means that learning can no longer be reduced to a pure classification problem. Moreover, we need a way of combining the scores for individual transitions

in such a way that we can compare transition sequences that may or may not be of the same length, which is a non-trivial problem for transition-based dependency parsing. However, as long as the size of the beam is bounded by a constant k, the worst-case running time is still $O(n)$.

3.5 PSEUDO-PROJECTIVE PARSING

Most of the transition systems that are used for classifier-based dependency parsing are restricted to projective dependency trees. This is a serious limitation given that linguistically adequate syntactic representations sometimes require non-projective dependency trees. In this section, we will therefore introduce a complementary technique that allows us to derive non-projective dependency trees even if the underlying transition system is restricted to dependency trees that are strictly projective. This technique, known as *pseudo-projective parsing*, consists of four essential steps:

1. Projectivize dependency trees in the training set while encoding information about necessary transformations in augmented arc labels.
2. Train a projective parser on the transformed training set.
3. Parse new sentences using the projective parser.
4. Deprojectivize the output of the projective parser, using heuristic transformations guided by augmented arc labels.

The first step relies on the fact that it is always possible to transform a non-projective dependency tree into a projective tree by substituting each non-projective arc (w_i, r, w_j) by an arc $(\text{ANC}(w_i), r', w_j)$, where $\text{ANC}(w_i)$ is an ancestor of w_i such that the new arc is projective. In a dependency tree, such an ancestor must always exist since the ROOT node will always satisfy this condition even if no other node does.[10] However, to make a minimal transformation of the non-projective tree, we generally prefer to let $\text{ANC}(w_i)$ be the *nearest* ancestor (from the original head w_i) such that the new arc is projective.

We will illustrate the projectivization transformation with respect to the non-projective dependency tree in figure 2.1, repeated in the top half of figure 3.9. This tree contains two non-projective arcs: *hearing* $\overset{\text{ATT}}{\rightarrow}$ *on* and *scheduled* $\overset{\text{TMP}}{\rightarrow}$ *today*. Hence, it can be projectivized by replacing these arcs with arcs that attach both *on* and *today* to *is*, which in both cases is the head of the original head. However, to indicate that these arcs do not belong to the true, non-projective dependency tree, we modify the arc labels by concatenating them with the label going into the original head: SBJ:ATT and VC:TMP. Generally speaking, a label of the form HEAD:DEP signifies that the dependent has the function DEP and was originally attached to a head with function HEAD. Projectivizing the tree with this type of encoding gives the tree depicted in the bottom half of figure 3.9.

Given that we have projectivized all the dependency trees in the training set, we can train a projective parser as usual. When this parser is used to parse new sentences, it will produce dependency

[10] I.e., for any arc (w_0, r, w_j) in a dependency tree, it must be true that $w_0 \rightarrow^* w_k$ for all $0 < k < j$.

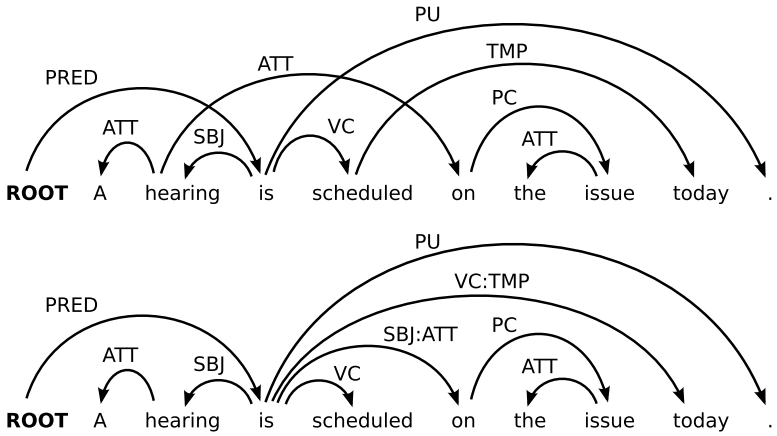

Figure 3.9: Projectivization of a non-projective dependency tree.

trees that are strictly projective as far as the tree structure is concerned, but where arcs that need to be replaced in order to recover the correct non-projective tree are labeled with the special, augmented arc labels. These trees, which are said to be pseudo-projective, can then be transformed into the desired output trees by replacing every arc of the form $(w_i, \text{HEAD:DEP}, w_j)$ by an arc $(\text{DESC}(w_i), \text{DEP}, w_j)$, where $\text{DESC}(w_i)$ is a descendant of w_i with an ingoing arc labeled HEAD. The search for $\text{DESC}(w_i)$ can be made more or less sophisticated, but a simple left-to-right, breadth-first search starting from w_i is usually sufficient to correctly recover more than 90% of all non-projective dependencies found in natural language (Nivre and Nilsson, 2005).

The main advantage of the pseudo-projective technique is that it in principle allows us to parse sentences with arbitrary non-projective dependency trees in linear time, provided that projectivization and deprojectivization can also be performed in linear time. Moreover, as long as the base parser is guaranteed to output a dependency tree (or a dependency forest that can be automatically transformed into a tree), the combined system is sound with respect to the class $\mathcal{G}_S$ of non-projective dependency trees for a given sentence S. However, one drawback of this technique is that it leads to an increase in the number of distinct dependency labels, which may have a negative impact on efficiency both in training and in parsing (Nivre, 2008).

3.6 SUMMARY AND FURTHER READING

In this chapter, we have shown how parsing can be performed as greedy search through a transition system, guided by treebank-induced classifiers. The basic idea underlying this approach can be traced back to the 1980s but was first applied to data-driven dependency parsing by Kudo and Matsumoto (2002), who proposed a system for parsing Japanese, where all dependencies are head-final. The approach was generalized to allow mixed headedness by Yamada and Matsumoto (2003), who applied

it to English with state-of-the-art results. The latter system essentially uses the transition system defined in section 3.1, together with an iterative parsing algorithm as described in section 3.4.2, and classifiers trained using support vector machines.

The arc-eager version of the transition system, described in section 3.4.1, was developed independently by Nivre (2003) and used to parse Swedish (Nivre et al., 2004) and English (Nivre and Scholz, 2004) in linear time using the deterministic, single-pass algorithm formulated in section 3.2. An in-depth description of this system, sometimes referred to as Nivre's algorithm, can be found in Nivre (2006b) and a large-scale evaluation, using data from ten different languages, in Nivre et al. (2007). Early versions of this system used memory-based learning but more accurate parsing has later been achieved using support vector machines (Nivre et al., 2006).

A transition system that can handle restricted forms of non-projectivity while preserving the linear time complexity of deterministic parsing was first proposed by Attardi (2006), who extended the system of Yamada and Matsumoto and combined it with several different machine learning algorithms including memory-based learning and logistic regression. The pseudo-projective parsing technique was first described by Nivre and Nilsson (2005) but is inspired by earlier work in grammar-based parsing by Kahane et al. (1998). Systems that can handle arbitrary non-projective trees, inspired by the algorithms originally described by Covington (2001), have recently been explored by Nivre (2006a, 2007).

Transition-based parsing using different forms of beam search, rather than purely deterministic parsing, has been investigated by Johansson and Nugues (2006, 2007b), Titov and Henderson (2007a,b), and Duan et al. (2007), among others, while Cheng et al. (2005) and Hall et al. (2006) have compared the performance of different machine learning algorithms for transition-based parsing. A general framework for the analysis of transition-based dependency-based parsing, with proofs of soundness, completeness and complexity for several of the systems treated in this chapter (as well as experimental results) can be found in Nivre (2008).