4

Inductive Dependency Parsing

Machine learning based on various forms of inductive inference has been used for a wide range of problems in natural language processing, with syntactic parsing being one of the more prominent problems during the last decade. In particular, methods for parsing unrestricted natural language text under requirements of robustness and disambiguation have to an increasing extent been characterized by the data-driven approach, sometimes in combination with a grammar-based strategy.

In this chapter, we will examine how the parsing methods developed in the previous chapter can be combined with the data-driven approach using a form of guided parsing. In this scheme, we use inductive machine learning to construct parser guides from treebank data. These guides are essentially classifiers that predict the next transition given the current configuration at each nondeterministic choice point. In this way, we can maintain the high efficiency of deterministic processing while developing more and more accurate guides in order to improve parsing accuracy.

An empirical evaluation of this methodology, with respect to accuracy as well as efficiency, will be presented in chapter 5. Here we will focus instead on the model of inductive inference, which belongs to the class of conditional history-based models, and the way in which this model can be combined with a deterministic parsing strategy and with discriminative machine learning. We will also discuss the different kinds of contextual features that can be used as a basis for prediction, and we will introduce memory-based learning, which is the learning method that will be used in the experiments reported in the next chapter. Finally, we will give a very brief description of the implementation of all these elements in a system called MaltParser, which has been used to carry out all the experiments reported in this book.

4.1 A Framework for Inductive Dependency Parsing

In order to situate our approach within the larger context of data-driven parsing methods, we begin by recapitulating our analysis of the data-driven approach in terms of inductive inference based on a formal model of syntactic representations and a representative sample of the relevant text language (cf. section 2.3.2). We then define our model of inductive inference as a conditional history-based model, combine it with a deterministic parsing strategy and derive a learning problem that can be solved using discriminative classifier induction. Finally, we show how training data for this learning problem can be derived from a treebank using a variant of the deterministic parsing algorithm.

4.1.1 Data-Driven Text Parsing

In the data-driven approach to text parsing, the mapping from input strings to output analyses is defined by an inductive mechanism applied to a text sample $T_t = (x_1, \ldots, x_n)$ from the language L to be analyzed. As explained in section 2.3.2, we can generally understand a data-driven text parsing system as consisting of three essential components:

1. A formal model M defining permissible analyses for sentences in L.
2. A sample of text $T_t = (x_1, \ldots, x_n)$ from L, with or without the correct analyses $A_t = (y_1, \ldots, y_n)$.
3. An inductive inference scheme I defining actual analyses for the sentences of any text $T = (x_1, \ldots, x_n)$ in L, relative to M and T_t (and possibly A_t).

The model M may be a formal grammar, defining an exact string language, in which case permissible representations will be restricted to this language. In our case, the model M is given by the formal framework for dependency parsing defined in section 3.3, which does not impose any restriction on the strings being analyzed. Given a set of dependency types R, the permissible representations for a sentence $x = (w_1, \ldots, w_n)$ is the set of all well-formed projective dependency graphs $G = (V_x, E, L)$ with node set $V_x = \mathbf{Z}_{n+1}$ and labeling function $L : E \rightarrow R$. Alternatively, in terms of the parsing method defined in section 3.4, we can characterize the permissible representations as the set of dependency graphs defined by some terminating transition sequence corresponding to x. Since the only requirement is that the set of token nodes in V_x^+ are in a one-to-one mapping with the tokens in the sentence, and that the dependency type labels are restricted to a given set R, it is clear that this imposes no restriction on the set of strings that can be parsed by the system. Hence, this is not a grammar-driven approach to text parsing.

The sample of text T_t, which is our basis for inductive generalization, will normally be called the *training corpus*. Although there exist *unsupervised* learning methods that apply to raw, unannotated text, such as the Inside-Outside algorithm for estimating the parameters of a PCFG, we will follow the mainstream tradition in data-driven parsing and use *supervised* learning, which requires the text sample to be annotated with representations defined by the model M. This means that the training corpus will be a *treebank* of the language L, i.e., a text corpus where each sentence is annotated with its correct analysis relative to the given model M (Abeillé, 2003b; Nivre, forthcoming). There are several methodological problems connected to the use of treebank data for inductive learning, having to do with the representativity of the data, the validity and reliability of the annotation, and the problem of converting annotations from one type of representation to another. These problems will be dealt with in chapter 5, where we report the experiments performed to evaluate the framework of inductive dependency parsing. For the remainder of this chapter, we will simply assume that we have available a sample of text $T_t = (x_1, \ldots, x_n)$ from the language L that we want to analyze and that every sentence $x_i \in T_t$ has been annotated with its correct analysis $y_i \in A_t$, where y_i is a well-formed dependency graph as defined in the preceding chapter.

While the formal model M has been described in detail in chapter 3 and problems connected to the training sample T_t will be dealt with in chapter 5, the inductive inference scheme I is the central topic of this chapter. This is the heart of the data-driven approach to text parsing, since it defines the way in which a parsing system can generalize from sentences found in the training corpus T_t to previously unseen sentences encountered in new texts.

4.1.2 Inductive Inference

As seen in section 2.3.2, an inductive inference scheme can be conceptualized and implemented in many different ways but can generally be decomposed into three main elements:

1. A parameterized *stochastic model* M_Θ assigning a score $S(x, y)$ to each permissible analysis y of a sentence x, relative to a set of parameters Θ.
2. A *parsing method*, i.e., a method for computing the best analysis y for a sentence x according to $S(x, y)$ (given an instantiation of Θ).
3. A *learning method*, i.e., a method for instantiating Θ based on inductive inference from the training sample T_t.

It is important to remember that this is a conceptual decomposition, which does not always correspond to system components or temporal processes. For example, while it is usually possible to divide the work done by the learning method and the parsing method into a *training phase*, which is applied once to the training corpus, and a *parsing phase*, which is applied to every new sentence without reprocessing the training corpus, the exact division of labor between these phases is dependent on the learning strategy. With an *eager* learning method, all the inference is performed during the training phrase; with a *lazy* learning method, most of the inductive inference is postponed until the parsing phase.

For the time being we will disregard the practical implementation of both parsing and learning and concentrate on the characterization of models and parameters for inductive dependency parsing. We will return to the parsing problem in section 4.1.4 and to the learning problem in section 4.1.5.

4.1.3 History-Based Models

The general approach of inductive dependency parsing is compatible with a variety of different models and parameterizations, but in this book we will limit our attention to conditional history-based models (cf. section 2.3.2), which are easily combined with the parsing methods developed in the previous chapter. In a history-based model, the parameterization essentially involves three steps (cf. Collins, 1999):

1. Define a one-to-one mapping between syntactic analyses y and decision sequences $D = (d_1, \ldots, d_m)$ such that D is a canonical derivation of y.
2. Define the score $S(x, y)$, for every sentence x and analysis y, in terms of each decision d_i in the corresponding decision sequence $D = (d_1, \ldots, d_m)$, conditioned on the history $H = (d_1, \ldots, d_{i-1})$.
3. Define a function Φ that groups histories (and decision sequences) into equivalence classes, thereby reducing the number of parameters in Θ to make the learning problem manageable.

In a conditional history-based model, the score $S(x, y)$ defined by the model is the conditional probability $P(y \,|\, x)$ of the analysis y given the sentence x, which means that the input sentence is a conditioning variable for each decision in the decision sequence:

$$P(y \,|\, x) = P(d_1, \ldots, d_m \,|\, x) = \prod_{i=1}^{m} P(d_i \,|\, d_1, \ldots, d_{i-1}, x) \qquad (4.1)$$

In order to get a manageable learning problem, it is normally necessary to introduce a function Φ, which defines an equivalence relation among histories. This gives us the final form of the parameterized model:

$$P(y \,|\, x) = P(d_1, \ldots, d_m \,|\, x) = \prod_{i=1}^{m} P(d_i \,|\, \Phi(d_1, \ldots, d_{i-1}, x)) \qquad (4.2)$$

The parameters of this model are simply the conditional probabilities $P(d \,|\, H)$, for all possible decisions d and non-equivalent histories H.

In the framework investigated in this book, the mapping from analyses to decision sequences is given by the transition system defined in section 3.4.2, where every terminating transition sequence $C_{0,m} = (c_0, \ldots, c_m)$ defines exactly one dependency graph G (definition 3.16). The inverse mapping from dependency graphs to transition sequences will be discussed in section 4.1.5 below, because it will be needed to derive training instances for the inductive learner, and we will also demonstrate that any well-formed dependency graph can be mapped to a transition sequence. It is important to remember that, although we will only consider deterministic parsing strategies in this book, the transition system itself is nondeterministic, which means that it associates a set of transition sequences, and dependency graphs, with any given sentence. And other parsing strategies might explore a larger part of this space of alternatives.

Given that an analysis in our framework is a dependency graph G defined by a transition sequence $C_{0,m} = (c_0, \ldots, c_m)$, where the transition t_i defines the mapping from configuration c_{i-1} to c_i, i.e., $c_i = t_i(c_{i-1})$, the conditional history-based model can now be expressed as follows:

$$P(G \,|\, x) = P(c_0, \ldots, c_m \,|\, x) = \prod_{i=1}^{m} P(t_i \,|\, c_0, \ldots, c_{i-1}, x) \tag{4.3}$$

However, when considering the input sentence x as a conditioning variable, we want to be able to take into account not only the sequence of tokens but any information that is available about the sentence as a result of preprocessing. Therefore, we replace x with the set of annotation functions A_x, which also includes the function w_x mapping string positions to tokens (cf. section 3.3.1):

$$P(G \,|\, A_x) = P(c_0, \ldots, c_m \,|\, A_x) = \prod_{i=1}^{m} P(t_i \,|\, c_0, \ldots, c_{i-1}, A_x) \tag{4.4}$$

In order to reduce the number of model parameters, we first group together all histories that end in the same configuration. In other words, we make a kind of Markov assumption to the effect that the probability of a transition from configuration c_i is independent of all earlier configurations in the transition sequence. However, this assumption is not as drastic as it may seem, since a configuration $c = (\sigma, \tau, h, d)$ records almost all relevant information about the preceding transition sequence in the state of σ, τ, h and d. Therefore, we simply use the function Φ to define equivalence classes of pairs (c, A_x), consisting of a configuration c and an input sentence x represented by its annotation functions A_x:

$$P(G \,|\, A_x) = P(c_0, \ldots, c_m \,|\, A_x) = \prod_{i=1}^{m} P(t_i \,|\, \Phi(c_{i-1}, A_x)) \tag{4.5}$$

We will use the term *parser condition* to refer to a pair of the form (c, A_x), where c is a parser configuration and A_x is the set of annotation functions for a sentence x, and we will use the term *parser state* for an equivalence class of parser conditions defined by the function Φ. We will also say that Φ is a function from parser conditions to parser states, although we will continue to write Φ as a function of two arguments, one configuration and one set of annotation functions.

The model parameters are the conditional probabilities $P(t \,|\, \Phi(c, A_x))$, for all possible transitions $t \in T_R$ and distinct parser states $\Phi(c, A_x)$. The cardinality of the parameter set therefore depends on two factors. The first is the number of distinct transitions, which is $|T_R| = 2|R| + 2$ (where $|R|$ is the number of distinct dependency types), since there are $|R|$ different instances of LEFT-ARC(r) and RIGHT-ARC(r) plus REDUCE and SHIFT. The second is the number of distinct parser states $\Phi(c, A_x)$, which depends on the definition of Φ but which will normally be many orders of magnitude greater than $|T_R|$.

In section 4.2 we will show how the parameterization Φ can be defined by a set of feature functions $\{\phi_1, \ldots, \phi_p\}$ that extract relevant features from the current parser condition. The definition of these feature functions will in turn determine the exact number of parser states and model parameters.

4.1.4 Parsing Methods

Given a conditional history-based model, the conditional probability $P(y_j \,|\, x)$ of analysis y_j given input x can be used to rank a set of alternative analyses $\{y_1, \ldots, y_k\}$ of the input sentence x, derived by a nondeterministic parser. If the model allows a complete search of the analysis space, we can in this way be sure to find the analysis y_j that maximizes the probability $P(y_j \,|\, x)$ according to the model:

$$\arg\max_{y_j} P(y_j \,|\, x) = \arg \max_{(d_1,\ldots,d_m)} \prod_{i=1}^{m} P(d_i \,|\, \Phi(d_1, \ldots, d_{i-1}, x)) \qquad (4.6)$$

With a deterministic parsing strategy, we instead try to find the most probable analysis y_j without exploring more than one decision sequence, based on the following approximation:

$$\arg\max_{y_j} P(y_j|x) \approx (d_1^*, \ldots, d_m^*) : d_i^* = \arg\max_{d_i} P(d_i|\Phi(d_1, \ldots, d_{i-1}, x)) \quad (4.7)$$

A deterministic parsing strategy is in this context a *greedy algorithm*, making a locally optimal choice in the hope that this will lead to a globally optimal solution (Cormen et al., 1990). The main problem with the greedy strategy is that it may not lead to a globally optimal solution. The main advantage is that it improves parsing efficiency by avoiding an exhaustive search of the analysis space. An additional advantage is that it reduces the effective number of parameters of the stochastic model, since only the mode of the distribution $P(d_i \,|\, \Phi(d_1, \ldots, d_{i-1}, x))$ needs to be estimated for each distinct condition $\Phi(d_1, \ldots, d_{i-1}, x)$. This also means that a larger class of learning methods can be used, including discriminative methods as well as methods for estimating generative and conditional probability models.

In section 3.4.3, we gave the following specification of the deterministic algorithm for dependency parsing:

```
PARSE($x = (w_1, \ldots, w_n)$)
1 $c \leftarrow (\epsilon, (1, \ldots, n), h_0, d_0)$
2 while $c = (\sigma, \tau, h, d)$ is not terminal
3     if $\sigma = \epsilon$
4         $c \leftarrow$ SHIFT($c$)
5     else
6         $c \leftarrow [o(c, A_x)](c)$
7 $G \leftarrow (V_x, E_c, L_c)$
8 return $G$
```

This algorithm assumes the existence of an oracle function o predicting the next transition for a given nondeterministic configuration c in the correct transition sequence for the sentence x (cf. section 3.4.3). In inductive dependency parsing, we replace the oracle o with a *guide* g, which predicts the next transition for a given configuration c, based on the parser state $\Phi(c, A_x)$. This is a form of *guided parsing* (Boullier, 2003), where the parser is guided by the function g at each nondeterministic choice point, which gives us the following parsing algorithm:

```
GUIDED-PARSE($x = (w_1, \ldots, w_n)$)
1 $c \leftarrow (\epsilon, (1, \ldots, n), h_0, d_0)$
2 while $c = (\sigma, \tau, h, d)$ is not terminal
3     if $\sigma = \epsilon$
4         $c \leftarrow$ SHIFT($c$)
5     else
6         $c \leftarrow [g(c, A_x)](c)$
7 $G \leftarrow (V_x, E_c, L_c)$
8 return $G$
```

The only difference with respect to the original algorithm is that we have replaced the oracle function $o(c, A_x)$ with a guide function $g(c, A_x)$, which means that we no longer assume that the guide always returns the correct answer. In *inductive* dependency parsing, as opposed to other forms of guided dependency parsing, we use inductive machine learning to construct the guide. Given training data from a dependency treebank, we induce a classifier g that maps every distinct parser state $\Phi(c, A_x)$ to a transition. We overload notation by using the symbol g to refer both to the *classifier*, which is a function from parser states to transitions, and for the *guide*, which is a function from parser conditions to transitions and which respects the condition that the transition returned is applicable to the configuration included in the parser condition. More precisely, the guide g returns the transition returned by the classifier g if this is a permissible transition, but returns SHIFT otherwise (since SHIFT is applicable to any non-terminal transition). Formally:

$$g(c, A_x) = \begin{cases} g(\Phi(c, A_x)) & \text{if } g(\Phi(c, A_x)) \text{ is applicable to } c \\ \textsc{Shift} & \text{otherwise} \end{cases} \tag{4.8}$$

where $g(\Phi(c, A_x))$ is the value of the classifier for the parser state $\Phi(c, A_x)$.

Given the conditional history-based model underlying our parsing strategy, the *optimal classifier* g can be characterized as follows:

$$g(\Phi(c, A_x)) = \arg\max_{t_i} P(t_i \,|\, \Phi(c, A_x)) \tag{4.9}$$

Since there is no guarantee that the most probable transition, given $\Phi(c, A_x)$, is also the transition required by the correct analysis of x, it may be the case that $g(\Phi(c, A_x)) \neq o(c, A_x)$, which means that the guide g defined in terms of the optimal classifier is not a true oracle even in theory. In practice, we will have to use an estimated approximation $\hat{g}$ of the optimal classifier g, which means that it may also be the case that $\hat{g}(\Phi(c, A_x)) \neq g(\Phi(c, A_x))$. Thus, finding the best possible estimate $\hat{g}$ for the optimal classifier g, given a sample T_t of treebank data with analyses A_t, is the central learning problem in this framework.

Going back to our characterization of the inductive inference scheme for history-based models, we can say that choosing a deterministic form of guided parsing as our parsing method reduces the complexity of the model M_Θ in two ways. First of all, the parameter set Θ only contains the modes of the conditional distributions $P(t_i|\Phi(c, A_x))$:

$$\Theta = \{t_i \,|\, \arg\max_{t_i} P(t_i|\Phi(c, A_x))\} \tag{4.10}$$

Secondly, the score $S(x, y)$ assigned to an analysis y of sentence x by the model M_Θ is binary:

$$S(x, y) = \begin{cases} 1 \text{ if } y = \textsc{Guided-Parse}(x) \\ 0 \text{ otherwise} \end{cases} \tag{4.11}$$

However, it is important to remember that the history-based model defined in section 4.1.3 is also compatible with many other parsing methods, which do not reduce the model complexity in this way.

Before we turn from parsing methods to learning methods, it is worth noting that the linear time complexity of the deterministic parsing algorithm is based on the assumption that computing the oracle function $o(c, A_x)$ is a constant-time operation. If we want to preserve this complexity for inductive dependency parsing, we therefore have to ensure that the computation of the guide function $g(c, A_x)$ can also be performed in constant time. We will return to this issue when we discuss the definition of feature functions in section 4.2.

4.1.5 Learning Methods

The learning problem that we have derived from the conditional history-based model, in combination with the deterministic parsing algorithm, consists in the induction of a classifier that can be used to construct a guide, as defined in the previous section. In terms of machine learning, this is an instance of *function approximation* (Mitchell, 1997), where the *target function* is the optimal classifier g while the *learned function* $\hat{g}$ is an approximation of g.

There are many different learning methods that could be used to solve this problem. Since the target function g is defined in terms of a conditional probability, it may seem natural to use a probabilistic learning method. In a *generative* model we could estimate the joint probability $P(\Phi(c, A_x), t)$, for every parser state $\Phi(c, A_x)$ and transition $t \in T_R$, and then derive the required conditional probability by conditioning and marginalizing:[1]

$$\hat{P}(t \,|\, \Phi(c, A_x)) = \frac{\hat{P}(\Phi(c, A_x), t)}{\sum_{t_i \in T_R} \hat{P}(\Phi(c, A_x), t_i)} \tag{4.12}$$

When only the conditional probability is needed, we may be able to make more efficient use of the training data by estimating the conditional distribution $P(t \,|\, \Phi(c, A_x))$ directly. Thus, an early version of inductive dependency parsing was based on conditional maximum likelihood estimation (Nivre, 2004b). However, given that it is only the *mode* of the conditional distribution that is needed, i.e., the transition t that maximizes $P(t \,|\, \Phi(c, A_x))$, we can take this argument one step further and argue that a *discriminative* learning method might be even more efficient.

One way of relating discriminative learning to our conditional model is to say that, instead of estimating the complete conditional distribution $P(t \,|\, \Phi(c, A_x))$, discriminative methods try to optimize the mapping from inputs $\Phi(c, A_x)$ to outputs t by only estimating the *mode* of this distribution (Jebara, 2004). For example, memory-based learning tries to find the optimal output by extrapolating from the most similar inputs seen previously, but without explicitly estimating a conditional probability (Daelemans and Van den Bosch, 2005). Other discriminative learning methods are artificial neural networks (Bishop, 1996) and support vector machines (Vapnik, 1995).

Using a discriminative learning method means that we can formulate the learning problem as a pure classification problem, where an *input instance* is a parser state $\Phi(c, A_x)$ and an *output class* is a transition $t \in T_R$. Using a supervised learning method, our task is then to induce a classifier $\hat{g}$ given a set of training instances D_t. Ideally, we would like the training set to be a sample of the function that we want to approximate:

$$D_g = \{(\Phi(c, A_x), t) \,|\, g(\Phi(c, A_x)) = t\} \tag{4.13}$$

However, since the function g is not known, it is not clear how such a sample could be established. What we have instead is a training corpus T_t, from which we can obtain a sample of training instances defined in terms of the oracle function o and the parameterization function Φ. For every sentence $x = (w_1, \ldots, w_n)$, let $C^{o,x}_{0,m} = (c_0, \ldots, c_m)$ be the unique transition sequence such that $c_0 = (\epsilon, (1, \ldots, n), h_0, d_0)$ and $c_i = [o(c_{i-1}, A_x)](c_{i-1})$ (for $i > 0$). The sample of training instances for Φ given T_t is:

[1] We use the notation $\hat{P}(\cdot)$ to denote an estimate of $P(\cdot)$.

$$D_\Phi = \{(\Phi(c, A_x), t) \,|\, o(c, A_x) = t, c \in C^{o,x}_{0,m}, x \in T_t\} \tag{4.14}$$

This set can be defined in a two-step process, where we first extract a set of pairs (c, t) from the training corpus T_t:

$$D_t = \{(c, t) \,|\, o(c, A_x) = t, c \in C^{o,x}_{0,m}, x \in T_t\} \tag{4.15}$$

This set is independent of the parameterization function Φ and can be reused with different parameterizations to define proper training sets:

$$D_\Phi = \{(\Phi(c, A_x), t) \,|\, (c, t) \in D_t\} \tag{4.16}$$

In order to construct a specific instance of the inductive dependency parser, we therefore have to solve three independent subproblems:

1. Derive the set D_t from the training corpus T_t.
2. Define the parameterization Φ and derive the training set D_Φ from D_t.
3. Induce a classifier $\hat{g}$ from the training set D_Φ using inductive learning.

The first problem can be solved using a form of guided parsing, using an oracle defined by the gold standard dependency graph from the treebank, as we will show in the next section. The second problem will be the topic of section 4.2, where we discuss the way in which different types of features can be defined in terms of parser conditions. The third problem will be addressed in section 4.3, where we introduce memory-based learning, which is the family of learning methods that will be used in the experiments reported in chapter 5.

4.1.6 Oracle Parsing

Given a training corpus $T_t = (x_1, \ldots, x_n)$, we want to extract the set of training instances $D_t = \{(c, t) \,|\, o(c, A_x) = t, c \in C^{o,x}_{0,m}, x \in T_t\}$. If we let D_x be the set of instances derived from a particular sentence x, i.e., $D_x = \{(c, t) \,|\, o(c, A_x) = t, c \in C^{o,x}_{0,m}\}$, then we can construct D_t by taking the union of D_x for all the sentences $x \in T_t$:

$$D_t = \bigcup_{x \in T_t} D_x \tag{4.17}$$

For each sentence $x \in T_t$, let $G_g = (V_x, E_g, L_g)$ be the dependency graph assigned to x by the gold standard annotation, and let $h_g : V_x^+ \rightarrow V_x$ and $d_g : V_x^+ \rightarrow R$ be defined as follows:

1. $h_g(i) = j$ if and only if $(j, i) \in E_g$
2. $d_g(i) = r$ if and only if $\exists j : ((j, i), r) \in L_g$

We can then derive the set D_x of instances for each sentence x by the following algorithm, which is a variant of the deterministic parsing algorithm defined in section 3.4.3:

```
ORACLE-PARSE($x = (w_1, \ldots, w_n), h_g, d_g$)
 1 $c \leftarrow (\epsilon, (1, \ldots, n), h_0, d_0)$
 2 $D_x \leftarrow \emptyset$
 3 while $c = (\sigma, \tau, h, d)$ is not terminal
 4     if $\sigma = \epsilon$
 5         $c \leftarrow$ SHIFT($c$)
 6     else
 7         $t \leftarrow$ ORACLE($c, h_g, d_g$)
 8         $D \leftarrow D_x \cup (c, t)$
 9         $c \leftarrow t(c)$
10 return $D_x$
```

The main difference, apart from the fact that we accumulate pairs (c, t) in the variable D_x, is that the oracle function o is replaced by a call to the function ORACLE, which predicts the next transition using the gold standard functions h_g and d_g:

```
ORACLE($c = (\sigma|i, j|\tau, h, d), h_g, d_g$)
1 if $h_g(i) = j$
2     return LEFT-ARC($d_g(i)$)
3 else if $h_g(j) = i$
4     return RIGHT-ARC($d_g(j)$)
5 else if $\exists k \in \sigma$ ($h_g(j) = k$ or $h_g(k) = j$)
6     return REDUCE
7 else
8     return SHIFT
```

For any configuration $c = (\sigma, \tau, h, d)$ that is passed as an argument to the function ORACLE, we know that both the stack σ and the input sequence τ are non-empty, because c is non-terminal (τ) and nondeterministic (σ). Hence, we can always assume that there is a token i on top of the stack σ and a token j at the head of the input list τ. Now, if i and j are linked by a dependency arc according to h_g, then the correct transition is LEFT-ARC(r) or RIGHT-ARC(r), with the dependency type r specified by d_g. If there is no arc between i and j, then REDUCE is the correct choice if and only if j is linked to a token to the left of i (below i in the stack σ); otherwise, the correct transition is SHIFT.

To check whether j is linked to a token to the left of i, we need to check if there is a token $k \in \sigma$ that is either the head of j ($h_g(j) = k$) or a dependent of j ($h_g(k) = j$). The first of these conditions can be checked simply by inspecting $h_g(j)$, since it holds if and only if $0 < h_g(j) < i$. The second condition may in a naive implementation require searching the entire stack σ. However, if there is a token k to the left of i such that $h_g(k) = j$, then we must already have encountered k in a previous configuration. And if we use an auxiliary stack to store tokens that have their head to the right, according to h_g, then we only need to compare the top of this stack with j.

It is worth pointing out that, whereas every transition sequence $C_{0,m} = (c_1, \ldots, c_m)$ for a sentence x defines a unique dependency graph $G_m = (V_x, E_m, L_m)$, the inverse relation is strictly speaking not a function, since there are a limited number of situations where two distinct transition sequences define the same dependency graph. This happens in configurations where the smallest arc, according to the gold standard graph G_g, that spans both the top token i and the next token j does not involve either i or j but tokens k and l, such that $k < i$ and $j < l$. In this case, both i and j must be popped from the stack before the arc connecting k and l can be added, but the order in which i and j are reduced is immaterial. In other words, either REDUCE or SHIFT is a possible transition and both of them will lead to the correct dependency graph. The algorithm defined above always prefers SHIFT in this situation, which means that j will be reduced before i. In other words, the ORACLE-PARSE algorithm constructs a canonical transition sequence for every dependency graph, consistently choosing SHIFT in cases of harmless SHIFT-REDUCE conflicts.

We conclude the discussion of oracle parsing with a correctness proof for the algorithm ORACLE-PARSE. Theorem 4.1 says that the functions h_m and d_m derived by the algorithm are identical to the input functions h_g and d_g for any sentence x and projective dependency graph $G_g = (V_x, E_g, L_g)$, which entails that the transition sequence $C_{0,m}$ used to construct the training data set D_x assigns G_g to x. As promised in section 3.4.4, this indirectly proves also theorem 3.22, since we can use ORACLE-PARSE to constructively prove that there exists a corresponding transition sequence for any projective dependency graph.

Theorem 4.1. For every sentence $x = (w_1, \ldots, w_n)$ with dependency graph $G_g = (V_x, E_g, L_g)$, if $c_m = (\sigma_m, \epsilon, h_m, d_m)$ is the terminal configuration in the computation of ORACLE-PARSE(x, h_g, d_g), then $h_g = h_m$ and $d_g = d_m$.

Proof. We need to show that $h_g(i) = h_m(i)$ and $d_g(i) = d_m(i)$ for every $i \in V_x^+$. We begin by noting that the following conditions hold for every token $i \in V_x^+$, in virtue of the transition system used by ORACLE-PARSE:

1. In the initial configuration c_0, $h_0(i) = 0$ and $d_0(i) = r_0$ (definition 3.9).
2. Before the terminal configuration c_m is reached, i must be shifted onto the stack, i.e., there exist p $(0 \leq p < m)$ and q $(p < q \leq m)$ such that $c_p = (\sigma_p, i|\tau_{p^-}, h_p, d_p)$ and $c_q = (\sigma_{q^-}|i, \tau_{p^-}, h_q, d_q)$ (definition 3.10).
3. The values of $h(i)$ and $d(i)$ can only change in a transition from a configuration c where i occurs at the head of the input list, i.e., $c = (\sigma, i|\tau, h, d)$, or on top of the stack, i.e., $c = (\sigma|i, \tau, h, d)$. In the former case, $h(i)$ and $d(i)$ are modified only by a RIGHT-ARC(r) transition; in the latter case, only by a LEFT-ARC(r) transition (definition 3.12).

There are three cases to consider, based on the value of $h_g(i)$:[2]

[2] Line numbers in this proof refer to the ORACLE algorithm defined on page 97.

1. If $h_g(i) = 0$ (and $d_g(i) = r_0$), we only need to show that $h_0(i) = h_m(i)$ and $d_0(i) = h_m(i)$, i.e., that i cannot be involved as the dependent in a LEFT-ARC(r) or RIGHT-ARC(r) transition. Given condition 3 above, this reduces to two subcases:

   a) In a configuration $c = (\sigma|i, \tau, h, d)$, LEFT-ARC($r$) is excluded because $h_g(i) = 0$ (contradicting the condition in line 1).

   b) In a configuration $c = (\sigma, i|\tau, h, d)$, RIGHT-ARC($r$) is excluded because $h_g(i) = 0$ (contradicting the condition in line 3).

   We conclude that $h_g(i) = h_0(i) = h_m(i)$ and $d_g(i) = d_0(i) = d_m(i)$.

2. If $h_g(i) \neq 0$ and $h_g(i) < i$, we need to show that there is some transition where $h(i)$ and $d(i)$ are changed from $h_0(i)$ and $d_0(i)$ to $h_g(i)$ and $d_g(i)$. (Together with Lemma 3.20, this entails that $h_g(i) = h_m(i)$ and $d_g(i) = d_m(i)$.) In virtue of condition 2 above, we know that there exists a configuration $c_p = (\sigma_p, i|\tau_{p^-}, h_p, d_p)$. There are three cases to consider for the transition out of this configuration:

   a) If $c_p = (\sigma_{p^-}|j, i|\tau_{p^-}, h_p, d_p)$, $h_g(j) = i$ and $d_g(j) = r$, ORACLE returns LEFT-ARC(r), so $c_{p+1} = (\sigma_{p^-}, i|\tau_{p^-}, h_p[j \mapsto i], d_p[j \mapsto r])$ (line 1–2).

   b) If $c_p = (\sigma_{p^-}|j, i|\tau_{p^-}, h_p, d_p)$, $h_g(i) = j$ and $d_g(i) = r$, ORACLE returns RIGHT-ARC(r), so $c_{p+1} = (\sigma_{p^-}|j|i, \tau_{p^-}, h_p[i \mapsto j], d_p[i \mapsto r])$ (line 3–4).

   c) If $c_p = (\sigma_{p^-}|j, i|\tau_{p^-}, h_p, d_p)$, $h_g(i) \neq j$, $h_g(j) \neq i$ and $h_g(i) \in \sigma_{p^-}$, ORACLE returns REDUCE, so $c_{p+1} = (\sigma_{p^-}, i|\tau_{p^-}, h_p, d_p)$ (line 5–6).

   In case (b), the goal has been reached. In cases (a) and (c), i remains at the head of the input list, while the size of the stack decreases by 1. Hence, as long as $h_g(i) \in \sigma_{p^-}$, there will eventually be a configuration $c_{q-1} = (\sigma_{q-1^-}|h_g(i), i|\tau_{p^-}, h_{q-1}, d_{q-1})$ followed by a RIGHT-ARC(r) transition, where $d_g(i) = r$. Assume $h_g(i) \notin \sigma_{p^-}$. Then $h_g(i)$ must have been popped from the stack in an earlier transition, which entails that there is a node k such that $h_g(i) < k < i$ and either $h_g(h_g(i)) = k$ (if $h_g(i)$ was popped in a LEFT-ARC(r') transition) or there is a token l such that $l < h_g(i)$ and k is linked to l (if $h_g(i)$ was popped in a REDUCE transition). But in either case this is a contradiction, since G_g is projective. Hence, we may conclude that $h_g(i) \in \sigma_{p^-}$. It follows that $h_g(i) = h_q(i) = h_m(i)$ and $d_g(i) = d_q(i) = d_m(i)$.

3. If $h_g(i) > i$, we again need to show that there is some transition where $h(i)$ and $d(i)$ are changed from $h_0(i)$ and $d_0(i)$ to $h_g(i)$ and $d_g(i)$. In virtue of condition 2 above, we know that there exists a configuration $c_q = (\sigma_{q^-}|i, \tau_{p^-}, h_q, d_q)$. Moreover, since $h_g(i) > i$, $h_q(i) = 0$ and $d_q(i) = r_0$. We know that $h_g(i) \in \tau_{p-}$ (since $h_g(i) > i$) and that $h_g(i)$ must eventually be pushed onto the stack (condition 2). We now show that this can only happen in a configuration where $h(i) = h_g(i)$ and $d(i) = d_g(i)$ (which

together with Lemma 3.20 entails that $h_g(i) = h_m(i)$ and $d_g(i) = d_m(i)$). More precisely, this follows from the following two propositions:

a) The node i can only be popped from the stack in a configuration of the form $c_r = (\sigma_{q^-}|i, j|\tau_{r^-}, h_r, d_r)$ $(r \geq q)$, where $h_g(i) = j$. Assume $h_g(i) \neq j$. Then Left-Arc(r) is obviously excluded (line 1). And Reduce is excluded because this would entail that there exists some k such that $h_g(j) = k$ or $h_g(k) = j$ and $k < i < j < h_g(i)$ (line 5), which contradicts the assumption that G_g is projective.

b) The node $h_g(i)$ can only be pushed onto the stack in a configuration of the form $c_s = (\sigma_s, h_g(i)|\tau_{s^-}, h_s, d_s)$ $(s > q)$, where $i \notin \sigma_s$. Assume $i \in \sigma_s$. Then Shift is excluded because there exists some k (namely i) such that $k \in \sigma_s$ and $h_g(k) = h_g(i)$ (line 5). And Right-Arc(r) is excluded because this would entail that there exists some k such that $k = h_g(h_g(i))$ and $i < k < h_g(i)$ (line 3), which contradicts the assumption that G_g is projective.

Hence, $h_g(i) = h_{r+1}(i) = h_m(i)$ and $d_g(i) = d_{r+1}(i) = d_m(i)$.

This concludes the proof of theorem 4.1. □

4.2 Features and Models

One of the key elements in the model of inductive dependency parsing defined in the previous section is the function Φ that defines an equivalence relation on the set of parser conditions and thereby defines what properties of a condition are relevant for the prediction of the next transition. This is reflected in the definition of the learning problem, where the set of possible input instances is simply the range of the function Φ, which we call the set of parser states.

In this section, we will discuss how Φ can be defined in terms of a set of feature functions, each of which extracts a relevant feature of the current parser condition. We will begin by defining a formal model for the specification of feature functions and move on to discuss the specific feature functions that will be used in the experiments reported in chapter 5. Finally, we will introduce the concept of a feature model, which corresponds to an instantiation of the function Φ, defined by a specific set of feature functions.

The formalization of feature functions is necessary for the implementation of feature models in the MaltParser system, described briefly in section 4.4, but it is not essential for the experimental study of memory-based inductive dependency parsing reported in chapter 5, where only a subset of the definable features will be used. Readers who are not interested in the formal aspects of feature functions can therefore skip most of the technical discussion in section 4.2.1 without missing anything that will be important later on.

4.2.1 Feature Functions

The role of the function Φ in our model is to determine which properties of a parser condition are relevant for the prediction of the next transition. In general, Φ can be defined by a set of simpler functions ϕ_i, which we call *feature functions*. Although the order of these functions is normally irrelevant, we will assume that they are ordered in a sequence $\Phi_{1,p} = (\phi_1, \ldots, \phi_p)$, which will save us the trouble of introducing a special name for each feature function ϕ_i, since it can be identified by its position in the sequence.

If $\Phi_{1,p} = (\phi_1, \ldots, \phi_p)$, then each function ϕ_i corresponds to a *feature*, or *attribute*, of a parser condition (c, A_x). Applying Φ to (c, A_x) is equivalent to applying each feature function ϕ_i to (c, A_x) in turn, which means that $\Phi(c, A_x) = (v_1, \ldots, v_p)$ if and only if $\phi_i(c, A_x) = v_i$ for every $\phi_i \in \Phi$. In this way, the value of $\Phi(c, A_x)$ corresponds to the standard representation of an *instance* as a sequence of *features*, often called a *feature vector*, which is widely used in machine learning (Mitchell, 1997).

Recall from section 3.3.1 that every function f in the set A_x of annotation functions for a sentence $x = (w_1, \ldots, w_n)$ is a function from the set of token nodes $V^+ = \{1, \ldots, n\}$ to some set of values V_f, where V_w is the set of possible word forms and V_p is the set of permissible part-of-speech categories, etc. Using A_f to denote the set of possible annotation functions, the notion of a feature function can be characterized as follows:

Definition 4.2. Given a set of configurations C, a set of annotation functions A_f, and a set of values V_ϕ, a *feature function* is a function $\phi : (C \times 2^{A_f}) \rightarrow V_\phi$.

We can then define parameterization functions in terms of feature functions:

Definition 4.3. Given a sequence of feature functions $\Phi_{1,p} = (\phi_1, \ldots, \phi_p)$, the corresponding *parameterization function* is the function $\Phi : (C \times 2^{A_f}) \rightarrow (V_{\phi_1} \times \cdots \times V_{\phi_p})$ such that $\Phi(c, A_x) = (v_1, \ldots, v_p)$ if and only if $\phi_i(c, A_x) = v_i$ (for $1 \leq i \leq p$, $c \in C$ and $A_x \in 2^{A_f}$).

The number of distinct parser states $\Phi(c, A_x)$ can now be defined as $|V_\Phi| = |V_{\phi_1}| \cdot \ \ldots \ \cdot |V_{\phi_p}|$, and the total number of model parameters in M_Θ is $|\Theta| = |T_R| \cdot |V_\Phi|$ for the general model and $|\Theta| = |V_\Phi|$ for the reduced model with a deterministic parsing strategy (cf. section 4.1.4). But even for the general model, it is normally the case that $|\Theta|$ is $O(|V_\Phi|)$.

The definition of a feature function is very general and compatible with many different ways of specifying such functions. In this study, we will restrict our attention to feature functions ϕ that can be defined in terms of the the composition of two simpler functions a_ϕ and f_ϕ as follows:

$$\phi(c, A_x) = v \ \Leftrightarrow \ [f_\phi \circ a_\phi](c) = v \tag{4.18}$$

where a_ϕ and f_ϕ satisfy the following conditions:

$$a_\phi : C \rightarrow V^+ \tag{4.19}$$

$$f_\phi \in A_x \cup \{d_c\} \tag{4.20}$$

The basic idea is that a_ϕ is an *address function*, mapping the configuration c to a specific token $i \in V^+$, and that f_ϕ is an *attribute function*, picking out a specific attribute v of i. This attribute may be given by one of the annotation functions in A_x or by the dependency type function d_c. Note that the function d_c is parameterized for the current configuration c, since this function is updated dynamically from one configuration to the next, whereas the annotation functions in A_x remain constant during the analysis of a given sentence x.

The reason for restricting the class of feature functions in this way is twofold. First, we want to ensure that feature functions can be computed efficiently, so that overall parsing efficiency is not compromised. Computing the value of each feature function must be done for every new parser condition, both in the construction of training instances during the training phase (cf. section 4.1.5) and in the construction of instances for the classifier during the parsing phase (cf. section 4.1.4). Secondly, we want to define a formal specification language for feature functions, so that implementations of inductive dependency parsing do not need to rely on hard-coded feature functions but can allow users to specify arbitrary feature functions within the space of permissible functions. The MaltParser system described in section 4.4 implements this functionality.

In the remainder of this section we will discuss the formal specification of feature functions, in particular the specification of the address function a_ϕ. Part of this discussion will be rather technical, but we will try to illustrate all the formal definitions with concrete examples. We will use a configuration from the transition sequence in figure 3.5 as our running example, more precisely the configuration c_{14}, resulting from a REDUCE transition. Figure 4.1 shows the relevant properties of this configuration, together with the annotation functions w_x and p_x for the sentence in question (cf. figure 3.3).

First of all, we define functions that extract a token from the stack σ_c or the input sequence τ_c of the current configuration.

Definition 4.4. For every configuration $c = (\sigma_c, \tau_c, h_c, d_c)$ and $i \geq 0$:

1. $\sigma_i(c) = \sigma_c[i]$
2. $\tau_i(c) = \tau_c[i]$

where $x[i]$ returns the ith element of the list x (starting from 0).

Note that σ_i and τ_i (for $i \geq 0$) are partial functions, which are undefined if the length of the relevant list is less than or equal to i. For example, the function σ_0, when applied to a configuration c, returns the top token (if any), while the function τ_1 returns the token following the next token in the input sequence τ_c (if τ_c has length two or more). For our example in figure 4.1, $\sigma_0(c_{14}) = 5$, while $\tau_1(c_{14}) = \perp$ (because τ_{14} has length one).

Given the basic functions σ_i and τ_i, we can construct complex address functions by composition with functions that map tokens to tokens according to their relations in the dependency graph, as defined by the function h_c in

$$\begin{aligned} c_{14} &= \ ((3,5),\ (9),\ h_7,\ d_7) \\ \sigma_{14} &= (3,5) \\ \tau_{14} &= (9) \end{aligned}$$

$h_7(1) = 2$	$d_7(1) =$ NMOD	$w_x(1) =$ Economic	$p_x(1) =$ JJ
$h_7(2) = 3$	$d_7(2) =$ SBJ	$w_x(2) =$ news	$p_x(2) =$ NN
$h_7(3) = 0$	$d_7(3) =$ ROOT	$w_x(3) =$ had	$p_x(3) =$ VBD
$h_7(4) = 5$	$d_7(4) =$ NMOD	$w_x(4) =$ little	$p_x(4) =$ JJ
$h_7(5) = 3$	$d_7(5) =$ OBJ	$w_x(5) =$ effect	$p_x(5) =$ NN
$h_7(6) = 5$	$d_7(6) =$ NMOD	$w_x(6) =$ on	$p_x(6) =$ IN
$h_7(7) = 8$	$d_7(7) =$ NMOD	$w_x(7) =$ financial	$p_x(7) =$ JJ
$h_7(8) = 6$	$d_7(8) =$ PMOD	$w_x(8) =$ markets	$p_x(8) =$ NNS
$h_7(9) = 0$	$d_7(9) =$ ROOT	$w_x(9) =$.	$p_x(9) =$ PU

Fig. 4.1. Configuration c_{14} with functions w_x and p_x (cf. figures 3.3 and 3.5)

the current configuration. To this end we define three higher-order functions, that map an arbitrary address function to a new address function by composing it with a function returning the head, leftmost dependent or rightmost dependent of a token.

Definition 4.5. For every function $a : C \rightarrow V^+$:

1. $h(a) = h_c \circ a$
2. $l(a) = l_c \circ a$
3. $r(a) = r_c \circ a$

where $l_c(i)$ and $r_c(i)$ are partial functions returning the leftmost and rightmost dependent, respectively, of a token $i \in V^+$.

Equipped with the basic functions σ_i and τ_i and the higher-order functions h, l and r, we can now give an inductive definition of the class of address functions.

Definition 4.6. The set of *address functions* is the smallest set A satisfying the following conditions:

1. For every $i \geq 0$, $\sigma_i, \tau_i \in A$.
2. For every $a \in A$, $h(a), l(a), r(a) \in A$.

It is worth pointing out again that all address functions are partial and fail to return a token as soon as one of the underlying functions (σ_i, τ_i, h_c, l_c or r_c) is undefined.

In order to exemplify the use of complex address functions, we consider the functions $h(\sigma_0)$ and $r(r(h(\sigma_0)))$, which return the head of the top token and the rightmost dependent of the rightmost dependent of the head of the top token, respectively. For the example in figure 4.1, these functions return the

Function	**Description**
σ_0	The top token
$\sigma_n (n > 0)$	The nth stack token (not counting the top token)
τ_0	The next token
$\tau_n (n > 0)$	The nth input token (not counting the next token)
$h(\sigma_0)$	The head of the top token
$l(\sigma_0)$	The leftmost dependent of the top token
$r(\sigma_0)$	The rightmost dependent of the top token
$l(\tau_0)$	The leftmost dependent of the next token

Fig. 4.2. Commonly used address functions

tokens 3 and 6, respectively, since $h(\sigma_0)(c_{14}) = h_7(\sigma_0(c_{14})) = h_7(5) = 3$ and $r(r(h(\sigma_0)))(c_{14}) = r_7(r_7(h_7(\sigma_0(c_{14})))) = r_7(r_7(h_7(5))) = r_7(r_7(3)) = r_7(5) = 6$.

Although the framework allows address specifications of almost arbitrary complexity, most of the features considered in this study will be based on a relatively small number of functions, in combination with different attribute functions. The most commonly used address functions are listed with explanations in figure 4.2. The list does not include the functions $h(\tau_0)$ and $r(\tau_0)$, since the parsing algorithm precludes the possibility of the next token having a head (other than 0) or a right dependent in the current configuration.

Having considered the construction of address functions at some length, we are now in a position to define a set of feature functions, using higher-order functions that map an address function a to a new function ϕ from parser conditions (c, A_x) to values v of an attribute function f, such that $\phi(c, A_x) = f(a(c))$. Formally:

Definition 4.7. If a is an address function, then for any configuration c and set of annotation functions A_x:

1. $f(a)(c, A_x) = f_x(a(c))$ for every $f_x \in A_x$
2. $d(a)(c, A_x) = d_c(a(c))$

The attribute function is either one of the annotation functions $f_x \in A_x$ (including the token function w_x) or the function d_c belonging to the current configuration c. In the former case, we have a *static feature*, since the value of the attribute function $f_x(i)$, for a given token i, remains constant during the parsing of a sentence x. In the latter case, we have a *dynamic feature*, because $d_c(i)$ will change dynamically between the different configurations of a transition sequence. Static features will be discussed further in section 4.2.2 below, while dynamic features are treated in section 4.2.3.

Finally, a short note on the implementation of feature functions. As noted in section 4.1.4, the linear time complexity of the inductive parsing algorithm is dependent on the assumption that the guide function $g(c, A_x)$ can be computed in constant time. A naive implementation of the functions l_c and r_c, using only the function h_c in the current configuration c, would require an exhaustive search of the set of input tokens to find the leftmost or rightmost dependent. However, this can easily be avoided by adding an explicit representation of the functions l_c and r_c. These functions can be updated in constant time for any transition t as follows:

- If $t = \text{Left-Arc}(r)$ and $c = (\sigma|i, j|\tau, h, d)$ then $l_c(j) \leftarrow i$.
- If $t = \text{Right-Arc}(r)$ and $c = (\sigma|i, j|\tau, h, d)$ then $r_c(i) \leftarrow j$.
- If $t = \text{Reduce}$ or $t = \text{Shift}$ then no update is needed.

Given these functions, any component function of a complex address function can be computed in constant time. We can therefore conclude that an address function constructed from k component functions can be computed in time which is $O(k)$ regardless of the length of the input sentence. Moreover, since the application of an attribute function to the value returned by the address function is a constant time operation, it is clear that the time required to compute a feature function is constant in the length of the input.

4.2.2 Static Features

A static feature function has the form $f(a)$, where a is an address function and f refers to one of the annotation functions in A_x. In other words, static features are based on information available as input, which remains constant throughout the parsing process. On the other hand, since the address defined by a is relative and not absolute, the actual value of a static feature function will of course vary in the course of a transition sequence.

One important class of static features are those with the attribute function w_x, which we call *lexical features*, since they are defined in terms of the actual word form w_i of a token i, where $w_x(i) = w_i$. As discussed in section 2.3.2, the importance of lexical features for disambiguation has been a dominant theme in research on natural language parsing over the last ten to fifteen years. And despite studies such as Gildea (2001), Dubey and Keller (2003), Klein and Manning (2003) and Bikel (2004), which can be taken to show that the significance of lexicalization has been overstated, it remains a fact that all state-of-the-art systems for robust disambiguation make use of lexical information in some way. The benefit of using lexical features in the inductive dependency parsing was demonstrated in Nivre et al. (2004) and is further investigated in the experiments in chapter 5.

In the previous section, we introduced the most commonly occurring address functions. In a similar fashion, figure 4.3 introduces the lexical features that will be used in the experiments later on. The most central features are $w(\sigma_0)$ and $w(\tau_0)$, which extract the word form of the top token and the next token, respectively. But we will also make use of lexical features for lookahead

Function	**Description**
$w(\sigma_0)$	Word form of the top token
$w(\tau_0)$	Word form of the next token
$w(\tau_n)(n > 0)$	Word form of the nth input token
$w(h(\sigma_0))$	Word form of the head of the top token

Fig. 4.3. Lexical features

tokens, i.e., tokens occurring n positions after the next token, denoted by $w(\tau_n)$, and for the head of the top token, symbolized by $w(h(\sigma_0))$.

Returning to our example configuration in figure 4.1, we get the following values for some of the features defined in figure 4.3:

$$\begin{aligned} w(\sigma_0)(c_{14}, A_x) &= w_x(5) = \text{effect} \\ w(\tau_0)(c_{14}, A_x) &= w_x(9) = . \\ w(\tau_1)(c_{14}, A_x) &= w_x(\perp) = \perp \\ w(h(\sigma_0))(c_{14}, A_x) &= w_x(3) = \text{had} \end{aligned} \tag{4.21}$$

Besides lexical features, static features can be defined in terms of any kind of annotation introduced as a result of preprocessing and encoded in a function $f : V^+ \rightarrow V_f$ included in A_x. The only kind of preprocessing that will be used in our experiments is part-of-speech tagging, which means that the only annotation function that will be used in features is the function p_x that maps each token to its part-of-speech (as defined by the part-of-speech tagger applied in preprocessing). We call these features *part-of-speech features.*

If the role of lexicalization in syntactic parsing has recently been the matter of some debate, the role of part-of-speech tagging is even more of a moot point. In early work on treebank parsing it was more or less standard practice to have a separate tagging phase prior to parsing proper (Charniak, 1996), but with the emergence of lexicalized models it was found that better parsing accuracy could often be obtained if the part-of-speech analysis was integrated in the parsing process (Charniak, 1997a; Collins, 1997). More recently, it has been argued that the main reason for using parts-of-speech in data-driven parsing is that they provide a back-off model for lexical features and thereby counteract the sparse data problem (Charniak, 2000; Van den Bosch and Buchholz, 2002).

In a study of memory-based shallow parsing, Van den Bosch and Buchholz (2002) showed that a model incorporating words but no parts-of-speech, while inferior with small training data sets, outperforms a model involving parts-of-speech but no words for training sets over a certain size (which in their experiments was around 50 000 sentences). However, it was still the case that a model incorporating *both* words and parts-of-speech gave the best overall

Function	**Description**
$p(\sigma_0)$	Part-of-speech of the top token
$p(\sigma_n)(n > 0)$	Part-of-speech of the nth stack token
$p(\tau_0)$	Part-of-speech of the next token
$p(\tau_n)(n > 0)$	Part-of-speech of the nth input token

Fig. 4.4. Part-of-speech features

performance, which indicates that the smoothing effect obtained by including parts-of-speech is beneficial also with large training sets.

In the experiments reported in the next chapter we make use of part-of-speech features for the two target tokens, i.e., the top token and the next token, as well as neighboring tokens both on the stack and in the sequence of remaining input tokens. We use the term *stack tokens* to refer to tokens that occur below the top token on the stack and the term *lookahead tokens* to refer to tokens that occur after the next token in the input sequence. Figure 4.4 shows the notational conventions that will be used to refer to part-of-speech features. By way of example, here are the values of a sample of part-of-speech features for the configuration in figure 4.1:

$$\begin{aligned} p(\sigma_0)(c_{14}, A_x) &= p_x(5) &&= \text{NN} \\ p(\sigma_1)(c_{14}, A_x) &= p_x(3) &&= \text{VBD} \\ p(\tau_0)(c_{14}, A_x) &= p_x(9) &&= \text{PU} \\ p(\tau_1)(c_{14}, A_x) &= p_x(\perp) &&= \perp \end{aligned} \tag{4.22}$$

4.2.3 Dynamic Features

A dynamic feature function has the form $d(a)$, where a is an address function and d denotes the dependency type function d_c that belong to the current parser configuration c and that is updated dynamically during the parsing process. One of the differences between the parsing methods investigated in this book and many other approaches to dependency parsing is that the parser produces labeled dependency graphs directly, rather than first producing an unlabeled dependency graph and then assigning labels to dependency arcs. This fact can be exploited when defining relevant feature functions, since the labels of previously added arcs are available in the state of the function d_c. We call these features *dependency type features*, or *dependency features* for short.

Figure 4.5 introduces the dependency features that will be used in our experiments. There are three features defined in relation to the top token,

Function	**Description**
$d(\sigma_0)$	Dependency type of the top token
$d(l(\sigma_0))$	Dependency type of the leftmost dependent of the top token
$d(r(\sigma_0))$	Dependency type of the rightmost dependent of the top token
$d(l(\tau_0))$	Dependency type of the leftmost dependent of the next token

Fig. 4.5. Dependency features

extracting the dependency types relating this token to its head ($d(\sigma_0)$), its leftmost dependent ($d(l(\sigma_0))$) and its rightmost dependent ($d(r(\sigma_0))$). In addition, we consider the leftmost dependent of the next input token ($d(l(\tau_0))$). We exemplify these dependency features by applying them to the configuration in figure 4.1:

$$\begin{aligned}
d(\sigma_0)(c_{14}, A_x) &= d_7(5) &&= \text{OBJ} \\
d(l(\sigma_0))(c_{14}, A_x) &= d_7(4) &&= \text{NMOD} \\
d(r(\sigma_0))(c_{14}, A_x) &= d_7(6) &&= \text{NMOD} \\
d(l(\tau_0))(c_{14}, A_x) &= d_7(\bot) &&= \bot
\end{aligned} \tag{4.23}$$

While dependency features are the only dynamic features used in this study, it would also be possible to define features based on the function h_c that records the index of a token's head. Although the exact numerical index is unlikely to be a useful feature, the comparison of features could be used to define *distance-based features*, which have been used with some success in other data-driven approaches to syntactic parsing (Collins, 1999), although these functions are seldom based on a purely quantitative notion of distance. Moreover, features that compare the relative position of two tokens would require a more complex definition of feature functions, and we will therefore leave this as a possible topic for future research.

4.2.4 Feature Models

In this section, we have shown how the parameterization function Φ can be defined by a sequence of feature functions $\Phi_{1,p} = (\phi_1, \ldots, \phi_p)$, where each feature function has the form $f(a)$ for some address function a and attribute function $f \in \{w, p, d\}$. Since each of the functions ϕ_i defines a feature of the current parser parser condition, we will say that the complex function Φ defines a *feature model.*

One of the questions posed in the experiments reported in chapter 5 is how different features influence the performance of an inductive dependency parser, with respect to accuracy as well as efficiency. We will address this question by a series of experiments, where we vary the feature model Φ while keeping other things constant. In this context, it is often convenient to be able to define a complex model in terms of two or more simpler models. For this purpose, we define the *concatenation* of two models in the obvious way:

Definition 4.8. Let Φ_1 and Φ_2 be two parameterization functions (or feature models), defined by two sequences of feature functions $\Phi^1_{1,p} = (\phi^1_1, \ldots, \phi^1_p)$ and $\Phi^2_{1,q} = (\phi^2_1, \ldots, \phi^2_q)$. The *concatenation* of Φ_1 and Φ_2, denoted $\Phi_1 + \Phi_2$, is the function Φ defined by $\Phi_{1,p+q} = (\phi^1_1, \ldots, \phi^1_p, \phi^2_1, \ldots, \phi^2_q)$.

The models that will be examined in chapter 5 can be seen as concatenations of three types of models, based on the three types of features discussed in this section:

1. Part-of-speech models
2. Dependency models
3. Lexical models

Before we conclude the discussion of features and models in this chapter, we will define the three types of models and introduce the notational conventions that will be used to designate these models in the experiments in chapter 5.

Part-of-speech models will be designated Φ^p_{ij} $(i, j \geq 0)$. All part-of-speech models include the features $p(\sigma_0)$ and $p(\tau_0)$. The parameter i specifies how many successive stack tokens will be included in addition to the top token. That is, every feature $p(\sigma_n)$, for $n \leq i$, is included. In a similar fashion, the parameter j specifies how many lookahead tokens will be included over and above the next input token. Thus, every feature $p(\tau_n)$, for $n \leq j$, is included. We illustrate this class of models by applying the model Φ^p_{01} to the configuration in figure 4.1:

$$\begin{aligned} \Phi^p_{01}(c_{14}, A_x) &= (p(\sigma_0)(c_{14}, A_x), p(\tau_0)(c_{14}, A_x), p(\tau_1)(c_{14}, A_x)) \\ &= (\mathrm{NN}, \mathrm{PU}, \perp) \end{aligned} \tag{4.24}$$

Dependency models will be designated Φ^d_{ijk} $(i, j, k \in \{0, 1\})$. All dependency models include the feature $d(\sigma_0)$. In addition, it may include some or all of the features $d(l(\sigma_0))$, $d(r(\sigma_0))$ and $d(l(\tau_0))$, and the indices i, j and k are basically boolean variables indicating the presence or absence of these features (in the order just listed). We illustrate this class of models by applying the model Φ^d_{011} to the configuration in figure 4.1:

$$\begin{aligned} \Phi^d_{011}(c_{14}, A_x) &= (d(\sigma_0)(c_{14}, A_x), d(r(\sigma_0))(c_{14}, A_x), d(l(\tau_0))(c_{14}, A_x)) \\ &= (\mathrm{OBJ}, \mathrm{NMOD}, \perp) \end{aligned} \tag{4.25}$$

Lexical models, finally, will be designated Φ^w_{ij} $(i, j \geq 0)$. The parameter i specifies how many lexical features are extracted from the stack, starting with the top token $(i \geq 1)$ and possibly including the head of the top token $(i = 2)$. The parameter j specifies how many successive input tokens will be included, starting with the next input token $(i \geq 1)$ and possibly adding an extra lookahead token ($j = 2$). We illustrate this class of models by applying the model Φ^w_{11} to the configuration in figure 4.1:

$$\begin{aligned}\Phi^w_{11}(c_{14}, A_x) &= (w(\sigma_0)(c_{14}, A_x), w(\tau_0)(c_{14}, A_x)) \\ &= (\text{effect}, .)\end{aligned} \tag{4.26}$$

4.3 Memory-Based Learning

In the deterministic version of inductive dependency parsing investigated in this book, the central learning problem is to induce a mapping from parser states to parser transitions. This problem can be solved using *memory-based learning*, a discriminative machine learning method that has been successfully applied to a wide range of problems in natural language processing (Daelemans and Van den Bosch, 2005). Although the general approach of inductive dependency parsing is not directly committed to any particular method for inductive learning, memory-based learning seems well suited for the task, with a local approximation of the target function that is potentially sensitive to subregularities and exceptional instances (Daelemans et al., 2002).

In this section we introduce the basic concepts of memory-based learning and discuss the different algorithms and parameters that can be used in the implementation of this approach. For the experiments reported in chapter 5 we rely on the software package TiMBL (Tilburg Memory-Based Learner) developed by Walter Daelemans, Antal van den Bosch and their colleagues at Tilburg University and the University of Antwerp (Daelemans et al., 2004), and our presentation of memory-based learning is deeply influenced by their work, which is presented comprehensively in Daelemans and Van den Bosch (2005). We close the section by relating our use of memory-based learning in dependency parsing to previous work on memory-based language processing, in particular memory-based parsing.

4.3.1 Memory-Based Learning and Classification

Memory-based learning and problem solving is based on two fundamental principles: learning is the simple storage of experiences in memory, and solving a new problem is achieved by reusing solutions from similar previously solved problems (Daelemans and Van den Bosch, 2005). It is inspired by the *nearest neighbor* approach in statistical pattern recognition and artificial intelligence (Fix and Hodges, 1952), as well as the *analogical modeling* approach in linguistics (Skousen, 1989, 1992). In machine learning terms, it can be characterized as a *lazy* learning method, since it defers processing of input until needed and processes input by combining stored data (Aha, 1997).[3]

[3] Memory-based learning is also known as instance-based learning, exemplar-based learning and case-based learning.

In contrast to *eager* learning methods, such as the family of generative probabilistic methods that are used in many data-driven parsers, memory-based learning performs generalization without abstraction. In addition, it uses similarity-based reasoning as an implicit smoothing method to deal with low-frequency events (Daelemans and Van den Bosch, 2005). Both of these properties make the method potentially well suited for problems in natural language processing, which are often characterized by distributions containing a long tail of low-frequency events, where it is notoriously difficult to distinguish noise from significant exceptions (Daelemans et al., 2002).

Conceptually, memory-based learning algorithms can be seen as variants of the k-nearest neighbor algorithm (k-NN) (Cover and Hart, 1967; Devijver and Kittler, 1982; Aha et al., 1991). Given the task of inducing a classifier $\hat{g} : S \rightarrow T$ from a set of training instances $D_t = \{(s_1, t_1), \ldots (s_n, t_n)\}$, where $s_i \in S$ is an input instance and $t_i \in T$ is its class, this type of algorithm can be described as follows:

- Learning consists in storing the set D_t of training instances in memory.
- Classifying a new instance r is performed in two steps:
  1. Compare r to every stored input instance s_i ($(s_i, t_i) \in D_t$):
     a) Compute the distance $\Delta(r, s_i)$ between r and s_i.
     b) Update the set of k closest instances (nearest neighbors).
  2. Take the majority class t of the k nearest neighbors as the class of r.

Even though the basic memory-based strategy remains the same, there are many parameters that can be varied to modify the resulting classifier. The most obvious parameter is perhaps the value of k, which can be varied from 1 to n (where n is the number of training instances in D_t). A small k value leads to a very local approximation of the function g, which is more sensitive to local exceptions and subregularities but also less robust when faced with noisy data. A large k value gives a more global approximation, which is less sensitive to local variations, whether due to noise or to significant exceptions. For the task of predicting the next parser transitions, a k value of about 5 has turned out to be optimal for many feature models (Nivre et al., 2004), although this is something that is investigated further in the experiments reported in chapter 5.

Another important parameter is the distance metric Δ, which can be varied in many different ways. For example, *feature weighting* can be used to give different weights to features in the representation of instances; *value weighting* can be used to differentiate penalties for mismatches between feature values; and *exemplar weighting* can be used to weight stored instances differently. Exemplar weighting will not be exploited in the investigations in this book, but both feature weighting and value weighting will be discussed in section 4.3.2 below and studied experimentally in chapter 5. A third kind of parameter is the voting procedure, where the main alternative to a simple majority vote is to use a weighting scheme that gives more weight to closer instances, using so-called *distance-weighted class voting* (Dudani, 1976).

In the following section we will discuss the parameters of memory-based learning that are relevant to our study of inductive dependency parsing and to the experiments reported in chapter 5. We will focus on the way that these parameters are implemented in TiMBL, since this is the software that is used in all our experiments. On the other hand, we will only deal with a small subset of all the features that are available in this software package. For more information about TiMBL, see the TiMBL Reference Guide (Daelemans et al., 2004); see also Daelemans and Van den Bosch (2005).

4.3.2 Learning Algorithm Parameters

Let $D_t = \{(s_1, t_1), \ldots, (s_n, t_n)\}$ be the set of training instances, where each input instance is represented as a vector of feature values $s_i = (s_1^i, \ldots, s_p^i)$ and each output t_i is a class taken from some set T. We recall that in the case of inductive dependency parsing, an input instance is a parser state $\Phi(c, A_x) = (\phi_1(c, A_x), \ldots, \phi_p(c, A_x)) = (v_1, \ldots, v_p)$ while the output class is a transition $t \in T_R$. We will assume that all features are symbolic, i.e., that their values are not numeric, a restriction that holds for all the features considered in this book, where feature values are word forms, parts-of-speech or dependency types. However, memory-based learning as such is not restricted to symbolic features.

We will begin by discussing the implementation of the k-NN algorithm in TiMBL, which differs in two ways from the standard formulation (Aha et al., 1991). First of all, the TiMBL version of k-NN considers the k nearest *distances*, rather than the k nearest instances. Since the training set may contain several instances at the same distance from a given instance, the number m of instances included by TiMBL may therefore be greater than k.

The second difference concerns the method used for tie-breaking, i.e., for deciding which class to choose in case there is no majority class in the nearest neighbor set. The default method in TiMBL, which will be used in all the experiments in chapter 5, is to use a three-step procedure:

1. Increase the value of k by 1 and choose the majority class in the larger neighbor set, if such a class exists.
2. Otherwise, choose the majority class in the entire training set D_t, if such a class exists.
3. Otherwise, choose the class t_1 of the first instance (s_1, t_1) encountered in the training set D_t.

The next parameter to discuss is the choice of the distance metric Δ. When dealing with symbolic features, the most straightforward metric is the Overlap metric, also referred to as Hamming distance, Manhattan metric, city-block distance, and L1 metric (Daelemans and Van den Bosch, 2005). The distance between two input instances $r = (r_1, \ldots, r_p)$ and $s = (s_1, \ldots, s_p)$ according to this metric is simply the number of mismatching features. Formally:

$$\Delta(r,s) = \sum_{i=1}^{p} \delta(r_i, s_i) \tag{4.27}$$

The δ function in this definition is a 0-1 mismatch function:

$$\delta(r_i, s_i) = \begin{cases} 0 \text{ if } r_i = s_i \\ 1 \text{ if } r_i \neq s_i \end{cases} \tag{4.28}$$

More sophisticated distance metrics can usually be understood as variations on the Overlap metric. One common variation is to associate a weight w_i with each feature ϕ_i and calculate the distance as a sum of weighted mismatches:

$$\Delta(r,s) = \sum_{i=1}^{p} w_i \, \delta(r_i, s_i) \tag{4.29}$$

Although it is possible to assign weights to features manually, based on some kind of *a priori* knowledge, it is much more common to derive weights automatically from training data using information-theoretic concepts which are also used in decision tree learning. Thus, Information Gain (IG) weighting considers the average amount of information about the correct class label contributed by each feature:

$$w_i = H(T) - \sum_{v \in V_i} P(v) H(T|v) \tag{4.30}$$

In this equation, T is the set of class labels, V_i is the set of values for feature ϕ_i, and $H(T)$ and $H(T|v)$ is the entropy of the class labels, *a priori* and conditioned on the value v, respectively:

$$H(T) = -\sum_{t \in T} P(t) \log_2 P(t) \tag{4.31}$$

$$H(T|v) = -\sum_{t \in T} P(t,v) \log_2 P(t|v) \tag{4.32}$$

One problem with IG weighting is that it tends to overestimate the relevance of features with large value sets. Quinlan (1986) has therefore introduced a normalized version, called Gain Ratio (GR), where IG is divided by the entropy of the value set:

$$w_i = \frac{H(T) - \sum_{v \in V_i} P(v) H(T|v)}{H(V_i)} \tag{4.33}$$

Although GR weighting still has a bias towards features with large value sets, it often gives good performance in practice. Other weighting schemes, which attempt to correct the bias of IG and GR, have been proposed based on the χ^2 statistic (White and Liu, 1994).

Another way of modifying the Overlap metric is to use a more sophisticated mismatch function, which differentiates the penalty of a mismatch based on the similarity of the feature values involved. This is the idea behind the (Modified) Value Difference Metric (MVDM), proposed by Stanfill and Waltz (1986) and refined by Cost and Salzberg (1993), which quantifies the distance between two feature values v_j and v_k belonging to the same value set V_i by considering their cooccurrence with target classes:

$$\delta(v_j, v_k) = \sum_{t \in T} |P(t|v_j) - P(t|v_k)| \tag{4.34}$$

Although MVDM introduces a kind of *value* weighting, rather than feature weighting, it will also have an indirect feature weighting effect, since $\delta(v_j, v_k)$ will on average be larger for informative features that have a more skewed conditional class distributions than for less informative features with more uniform distributions (Daelemans and Van den Bosch, 2005). One problem with MVDM is that it is sensitive to sparse data. TiMBL therefore offers the possibility of setting a frequency threshold l, so that MVDM is applied only if both of the values compared occur at least l times in the training data; otherwise the 0-1 mismatch function is used instead.

All of the modifications to the distance metric considered so far have the potential drawback that they increase the complexity of distance computations and thereby compromise the efficiency of classification. However, in the TiMBL implementation both feature weights w_i (for $1 \leq i \leq p$) and value distances $\delta(v_j, v_k)$ (for $v_j, v_k \in V_i$) can be computed and stored at learning time, which means that only table lookup is required at classification time.

In addition to feature weighting and value weighting, a weighting scheme can also be applied in the voting procedure that determines the majority class. Dudani (1976) proposed a voting rule in which the vote of each instance s_i is weighted by a function w_i of its distance to the new instance r:

$$w_i = \begin{cases} \frac{d_k - d_i}{d_k - d_1} & \text{if } d_k \neq d_1 \\ 1 & \text{if } d_k = d_1 \end{cases} \tag{4.35}$$

In this equation, d_i is the distance of s_i to r, d_1 is the distance of the nearest neighbor, and d_k is the distance of the most distant instance in the neighbor set. In addition to this inverse-linear (IL) weighting scheme, Dudani (1976) proposed the inverse distance (ID) weight:

$$w_i = \frac{1}{d_i} \text{ if } d_i \neq 0 \tag{4.36}$$

In order to make the weighting applicable also to neighbors with zero distance, it is customary to add a small constant ϵ to the denominator (Wettschereck, 1994):

$$w_i = \frac{1}{d_i + \epsilon} \tag{4.37}$$

In chapter 5 we will investigate how the different parameters of memory-based learning affect the performance of inductive dependency parsing. The parameters that will be varied are the following:

1. Number of nearest distances: k
2. Distance metric: Overlap or MVDM (with frequency threshold l)
3. Feature weighting: IG, GR, or none
4. Distance-weighted class voting: IL, ID, or none

In experiments where other parameters are varied, such as feature models or training sets, we will usually keep the learning algorithm parameters constant. However, rather than using the default values of the TiMBL system, which usually gives suboptimal performance, we will use the following settings, which have been shown to give good performance in previous experiments (Nivre et al., 2004; Nivre and Scholz, 2004):

1. Number of nearest distances: $k = 5$
2. Distance metric: MVDM with $l = 3$
3. Feature weighting: None
4. Distance-weighted class voting: ID

Finally, a remark on the efficiency of memory-based learning and classification. Given the lazy learning approach, training a memory-based classifier is usually very efficient, given that this basically consists in storing instances in memory, in a so-called instance base, and precomputing metrics such as feature weights and value distances for MVDM. By contrast, classification is less efficient, with a worst-case complexity of $O(n)$, where n is the number of instances in the instance base. The TiMBL software package implements a tree-based indexing scheme that speeds up classification in practice, although the worst-case complexity remains the same. It also offers the possibility of compressing the instance base, e.g., by constructing a decision tree based on feature weights. This decision tree yields an approximation of the exhaustive k-NN search, which improves efficiency but usually has a negative effect on classification accuracy. TiMBL also offers several hybrid solutions, that exploit the trade-off between efficiency and accuracy in different ways. These alternative algorithms will not be explored in this book, mainly because previous experiments have shown that classification performance degrades considerably especially for more complex feature models. We refer the reader to the TiMBL Reference Guide for more information on the tree-based indexing used by TiMBL as well as alternatives to the k-NN algorithm.

4.3.3 Memory-Based Language Processing

Memory-based learning and classification has been applied to a wide range of problems in natural language processing, exemplified in the following list (see also Daelemans and Van den Bosch, 2005):

- Hyphenation and syllabification (Daelemans and Van den Bosch, 1992)
- Assignment of word stress (Daelemans et al., 1994)
- Grapheme-to-phoneme conversion (Stanfill and Waltz, 1986; Lehnert, 1987; Weijters, 1991; Daelemans and Van den Bosch, 1996)
- Morphological analysis (Van den Bosch and Daelemans, 1999)
- Part-of-speech tagging (Cardie, 1993; Daelemans et al., 1996; Zavrel and Daelemans, 1997)
- Prepositional phrase attachment (Zavrel et al., 1997)
- Word sense disambiguation (Ng and Lee, 1996; Fujii et al., 1998; Dagan et al., 1999; Veenstra and Daelemans, 2000; Escudero et al., 2000)
- Named entity recognition (De Meulder and Daelemans, 2003; Hendrickx and Van den Bosch, 2003)
- Semantic role labeling (Van den Bosch et al., 2004; Kouchnir, 2004)
- Text categorization and filtering (Masand et al., 1992; Yang and Chute, 1994; Riloff and Lehnert, 1994)

Most of these problems have a natural formulation as a classification problem, where some kind of linguistic entity, such as a grapheme, a syllable, a word, or an entire document, is mapped to a finite set of discrete categories. This formulation of the problem makes memory-based learning a natural choice.

However, syntactic parsing is *prima facie* not a classification problem of this kind, especially not if we consider full parsing. Even though the input is a linguistic entity such as a sentence, the output usually comes from an infinite set of complex structures, such as constituency trees or dependency graphs. In order to apply the memory-based approach to syntactic parsing, it is therefore necessary to reformulate the problem so that it can be solved using discriminative learning. Broadly speaking, there are three different reformulations that have been proposed in the literature, which we may call holistic parsing, cascaded partial parsing, and history-based parsing.

The holistic approach is in a way the most straightforward application of the memory-based approach to full syntactic parsing and is based on the idea of storing complete sentences with their analyses in the instance base. Parsing a new sentence is performed by finding the most similar sentences in the instance base and adapting their analyses to the input sentence, possibly backing off to smaller fragments if necessary. This approach is most clearly exemplified in the work of Streiter (2001a,b) and Kübler (2004), but the DOP framework (Bod, 1995, 1998, 2003) is essentially based on the same idea, especially in its non-probabilistic incarnation where priority is given to analyses composed of large fragments (Bod, 2000). A variation on this theme is De Pauw (2003), who uses a memory-based model to score analyses in a parse forest derived using a grammar-driven parsing method.

The cascaded approach starts from a partial parsing or chunking analysis, which can be cast as a classification problem using the so-called BIO representation[4] (Ramshaw and Marcus, 1995) and which has been performed successfully with memory-based methods by, among others, Veenstra (1998) and Tjong Kim Sang and Veenstra (1999). One way of extending this partial analysis to a more complete syntactic analysis is to use a cascade of partial parsers, where the input of each parser includes the output of previous parsers, in combination with methods for identifying grammatical relations holding between chunks. Memory-based approaches to cascaded partial parsing and grammatical relation finding include Argamon et al. (1998), Buchholz et al. (1999), Daelemans et al. (1999), Krymolowski and Dagan (2000), Kübler and Hinrichs (2001), Buchholz (2002) and Dagan and Krymolowski (2003).

The history-based approach is the most indirect way of performing parsing through classification, since the input instances are not linguistic entities but states of some parsing system, and the classes are not linguistic categories or structures but actions of this parsing system (cf. sections 2.3.2 and 4.1.3). In this way, Veenstra and Daelemans (2000) used memory-based learning to predict the actions of a shift-reduce parser, although this method was only tested on an artificially created corpus. Inductive dependency parsing using memory-based learning to guide a deterministic parser is exactly the same idea, although combined with a different kind of syntactic representation and a different parsing algorithm.

4.4 MaltParser

Using memory-based learning and classification to guide a deterministic parser is one instantiation of the general approach of inductive dependency parsing. In this section, we describe a system called MaltParser, which has been used to perform all the experiments on memory-based dependency parsing reported in chapter 5, but which is designed as a more general framework for inductive dependency parsing.

The system can be described as a data-driven parser-generator framework. While a traditional parser generator constructs a parser given a grammar, a data-driven parser generator constructs a parser given a treebank. However, MaltParser also takes as input the specification of a feature model, as defined in section 4.2, which means that different parsers can be induced from the same treebank without recompiling the system. Moreover, the design of the system is intended to facilitate the variation not only of feature models but also of parsing algorithms and learning methods, although these parameters will be kept constant in the experiments reported in this book.

[4] For a given phrase or chunk type, each token is tagged as **B**eginning, being **I**nside, or being **O**utside a constituent of that type.

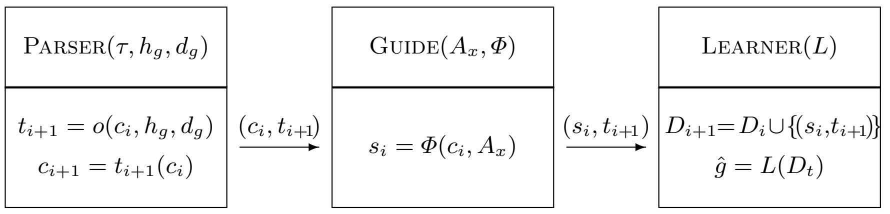

Fig. 4.6. Architecture for training

4.4.1 Architecture

In the data-driven approach to text parsing, we can usually distinguish two different phases, the *training phase* and the *parsing phase* (cf. section 4.1.2). Although these phases are different in nature, they can often be decomposed into very similar or even identical subtasks. For the framework investigated in this book, the training phase consists of two steps. The first step involves parsing every sentence x of the training corpus T_t using the oracle parsing algorithm, extracting the feature vector $\Phi(c_i, A_x)$ for every nondeterministic configuration c_i, and storing the pair $(\Phi(c_i, A_x), t_{i+1})$ in the set of training instances D_t. The second step is the induction of a classifier $\hat{g}$ from D_t using a particular learning method. The parsing phase consists in parsing every sentence x of the input text T using the inductive parsing algorithm, extracting the feature vector $\Phi(c_i, A_x)$ for every nondeterministic configuration c_i, and querying the classifier for $\hat{g}(\Phi(c_i, A_x)) = t_{i+1}$.

Comparing these two phases, we note that the extraction of feature vectors is performed in exactly the same way during training and parsing, although the vectors are used for learning in one case and for prediction in the other. Moreover, we have previously seen that the parsing algorithms used for training and parsing differ only minimally from each other. This suggests that a data-driven parsing system should be designed in such a way that the same basic components for parsing and feature extraction can be used both in the training phase and in the parsing phase. In addition, since we want to be able to vary parsing methods, feature models and learning methods independently of each other, these components should be encapsulated and separated from each other. This gives rise to an architecture with three main components (in addition to input/output modules and overall control structure):

1. Parser
2. Guide
3. Learner

In this architecture, the Parser constructs dependency graphs by applying transitions to parser configurations, the Guide extracts feature vectors from

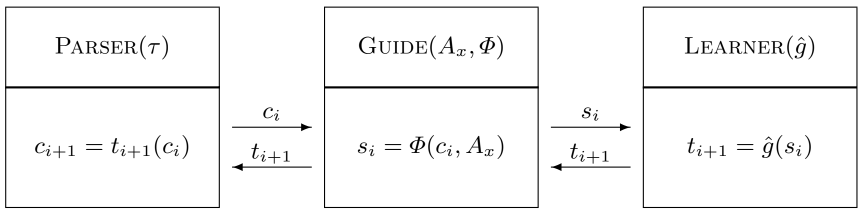

Fig. 4.7. Architecture for parsing

parser conditions and passes data between the Parser and Learner, and the Learner handles the mapping from feature vectors to transitions. In practice, the Learner will normally be an interface to a standard machine learning package such as TiMBL.

Figure 4.6 depicts the data flow in the architecture during the training phase. For a given sentence $x_i \in T_t$, the Parser takes as input the token sequence τ and the gold standard functions (h_g, d_g). If the current configuration c_i is nondeterministic, the Parser derives the correct transition t_{i+1} from (h_g, d_g) using the oracle function o, passes (c_i, t_{i+1}) to the Guide as a training instance, and derives the next configuration c_{i+1} by applying t_{i+1} to c_i. The Guide takes as input a feature model Φ (constant for all the sentences of the training corpus) and the annotation functions A_x corresponding to the current sentence x. When receiving the instance (c_i, t_{i+1}) from the Parser, the Guide uses the feature model Φ to extract the parser state s_i, represented as a feature vector, and passes the training instance (s_i, t_{i+1}) to the Learner. The Learner, parameterized by an inductive learning algorithm L, collects instances in the training set D_t and finally applies L to induce a classifier $\hat{g}$ when the entire training corpus has been parsed.

Figure 4.7 shows the data flow during the parsing phase. In this case, there is no gold standard analysis to guide the Parser, which only takes the input sequence as input. If the current configuration c_i is nondeterministic, the Parser requests a prediction of the next transition by passing c_i to the Guide. However, once the predicted transition t_{i+1} is returned by the Guide, the Parser applies t_{i+1} to the current configuration c_i in exactly the same way as during training. Furthermore, the Guide extracts the parser state $s_i = \Phi(c_i, A_x)$ in exactly the same way during parsing and training, using the model Φ defined by the current feature specification s_Φ. The only difference is that, instead of passing an instance (s_i, t_{i+1}) to the Learner as training data, during parsing it sends the state s_i and receives the predicted transition t_{i+1}, which is then passed on to the Parser. The Learner, finally, uses the function $\hat{g}$, induced in the training phase, to map the state s_i to the transition $t_{i+1} = \hat{g}(s_i)$.

One of the advantages of this architecture is that parsing is completely separated from learning, which makes it possible to vary parsing methods and learning methods independently. The Parser has no knowledge of the feature model Φ and behaves in exactly the same way regardless of which features are used for learning and prediction. The Learner only has to learn a mapping from feature vectors to transitions, without knowing either how the features are extracted or how the transitions are to be used. Finally, the Guide has no knowledge about either parsing algorithms or learning algorithms, but only handles the abstraction from configuration to states and passes data between the Parser and the Learner. The Learner can also encapsulate the interface to an external machine learning package, converting the feature vector constructed by the Guide to whatever special format is required by the external module. In this way, different machine learning packages can be plugged in without modifying the Guide module.

4.4.2 Implementation

The architecture presented above is realized in MaltParser (Nivre and Hall, 2005), a version of which is freely available for research and educational purposes, together with a suite of tools for data conversion and evaluation.[5] The version of MaltParser used for the experiments in this book supports the following functionality:

- **Parser:** The Parser implements the deterministic parsing algorithm in two versions: ORACLE-PARSE for training and GUIDED-PARSE for parsing.
- **Guide:** The Guide accepts specifications of arbitrary feature models, but features are limited to dependency features, part-of-speech features and lexical features.
- **Learner:** The Learner supports memory-based learning via an interface to TIMBL.

The most recent version of the system extends this functionality by providing alternative parsing algorithms, notably Covington's incremental algorithms for non-projective dependency parsing (Covington, 2001), and alternative learning methods, such as support vector machines using the LIBSVM tools (Wu et al., 2004).

[5] URL: http://www.msi.vxu.se/users/nivre/research/MaltParser.html