Chapter 13
Dependency Parsing

13.1 Introduction

Parsing dependencies consists of finding links between heads (also called governors) and modifiers (or dependents) – one word being the root of the sentence (Fig. 13.1). In addition, each link can be annotated with a grammatical function.

There is a large array of techniques to parse dependencies. In this chapter, we introduce some of them in order of increasing complexity. We begin with an extension of shift–reduce to parse dependencies, and we describe how to use symbolic and machine-learning techniques to guide the parser. We then present other parsing strategies using constraint satisfaction and statistical lexical dependencies.

13.2 Evaluation of Dependency Parsers

Before we review dependency parsing techniques, let us first describe how we will evaluate them. As for constituents, we will compare the output of an automatic parser with its corresponding manual annotation. Most evaluation metrics follow Lin (1995), who proposed to consider the links between each word in the sentence and its head. The error count is then the number of words that are assigned a wrong head (governor). Figure 13.2 shows a manually annotated dependency tree of *Bring the meal to the table* and a possible parse. The error count is 1 out of 6 links and corresponds to the wrong attachment of *to*. Lin (1995) also described a method to adapt this error count to constituent structures. This error count is probably simpler and more intuitive than the PARSEVAL metrics.

P.M. Nugues, *Language Processing with Perl and Prolog*, Cognitive Technologies,
DOI 10.1007/978-3-642-41464-0_13, © Springer-Verlag Berlin Heidelberg 2014

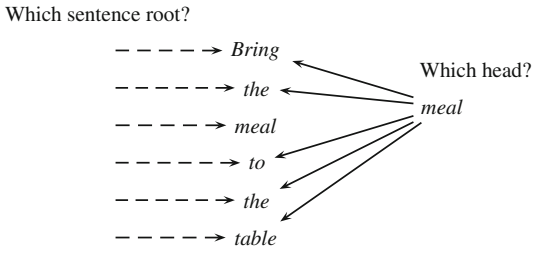

Fig. 13.1 Possible roots in the sentence *Bring the meal to the table* and heads for word *meal*. There are N possible roots, and each remaining word has theoretically $N - 1$ possible heads

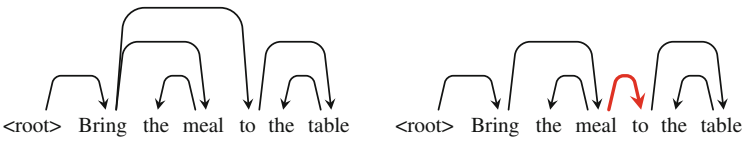

Fig. 13.2 Evaluation of dependency trees: the reference dependency tree (*left*), and a possible parse output (*right*)

13.3 Nivre's Parser

13.3.1 Extending the Shift–Reduce Algorithm to Parse Dependencies

Nivre (2003) proposed a dependency parser that creates a graph that he proved to be both projective and acyclic. The parser is an extension to the shift–reduce algorithm that we saw in Sect. 12.2. As with the regular shift–reduce, Nivre's parser uses a stack S and a list of input words W. However, instead of finding constituents, it builds a set of arcs A representing the graph of dependencies.

Nivre's parser uses two operations in addition to shift and reduce, left-arc and right-arc. Using the notation (head, modifier) to denote an arc from a head to a modifier:

Shift pushes the next input word onto the stack.

Reduce pops the top of the stack. We also add a constraint to the reduce operation to check that the top of the stack has a head and then ensures that the graph is connected.

Left-arc adds an arc (n', n) from the next input word n' to the top of the stack n and reduces n from the top of the stack. We set a condition to this operation: there must not be an arc (n'', n) already in the graph. Without it, the top of the stack would have two or more heads.

Right-arc adds an arc (n, n') from the top of the stack n to the next input word n' and pushes n' on the top of the stack.

Table 13.1 The parser transitions, where W is the initial word list; I, the current input word list; A, the graph of dependencies; and S, the stack. The triple $\langle S, I, A\rangle$ represents the parser state. n, n', and n'' are lexical tokens. The pair (n', n) represents an arc from the head n' to the modifier n

	Parser transitions		
Actions	**State before**	**State after**	**Conditions**
Initialization		$\langle nil, W, \emptyset\rangle$	
Termination	$\langle S, nil, A\rangle$		
Left-arc	$\langle n\|S, n'\|I, A\rangle$	$\langle S, n'\|I, A \cup \{(n', n)\}\rangle$	$\nexists n''(n'', n) \in A$
Right-arc	$\langle n\|S, n'\|I, A\rangle$	$\langle n'\|n\|S, I, A \cup \{(n, n')\}\rangle$	
Reduce	$\langle n\|S, I, A\rangle$	$\langle S, I, A\rangle$	$\exists n'(n', n) \in A$
Shift	$\langle S, n\|I, A\rangle$	$\langle n\|S, I, A\rangle$	

Table 13.1 shows the start and final parser states as well as the four transitions and their conditions.

13.3.2 Parsing an Annotated Corpus

Nivre (2006) proved that for each sentence with a projective dependency graph, there is a transition sequence that enables his parser to generate this graph. He called this procedure gold-standard parsing, because it corresponds to the sequence of parsing transitions taken in the set {left-arc, right-arc, reduce, shift} that produces the manually-obtained gold-standard graph. The parser has a linear complexity, and the number of transitions needed to parse a sentence is at most $2n - 1$, where n is the length of the sentence.

To parse a manually-constructed graph, we could set the graph as a search goal and use the Prolog backtracking mechanism to find the transitions. Although this is possible, there are more efficient methods. Given a dependency graph, we can formulate simple conditions on the stack and the current input list to execute left-arc, right-arc, shift, or reduce. The two first conditions on left-arc and right-arc are obvious:

- We execute a left-arc if the top of the stack and the next word in the list are linked by a left arc in the gold-standard graph.
- We execute a right-arc if the top of the stack and the next word in the list are linked by a right arc in the gold-standard graph.

The reduce condition is slightly more complex. We execute it when the gold-standard graph contains an arc in either direction between the next word in the list and a word in the stack, below the top. Finally, we execute a shift when no other action can be carried out. The operation we have just explained that determines which transition to apply given a certain parser state is called an **oracle**. The oracle at a given step of the parsing process is summarized by the condition below, where *TOP* is the top of the stack, *FIRST*, the first token of the input list, and *arc*, the relation holding between a head and a dependent:

Table 13.2 The transition sequence to apply to the sentence *The waiter brought the meal* to produce its dependency graph. The graph uses *left* or *right arrows* to give the direction of the dependency

Trans.	Stack	Queue	Graph
start	∅	ROOT the waiter brought the meal	{}
sh			
	ROOT	the waiter brought the meal	{}
sh			
	the ROOT	waiter brought the meal	{}
la			
	ROOT	waiter brought the meal	{the ← waiter}
sh			
	waiter ROOT	brought the meal	{the ← waiter}
la			
	ROOT	brought the meal	{the ← waiter, waiter ← brought}
ra			
	brought ROOT	the meal	{the ← waiter, waiter ← brought, ROOT → brought}
sh			
	the brought ROOT	meal	{the ← waiter, waiter ← brought, ROOT → brought}
la			
	brought ROOT	meal	{the ← waiter, waiter ← brought, ROOT → brought, the ← meal}
ra			
end	meal brought ROOT	[]	{the ← waiter, waiter ← brought, ROOT → brought, the ← meal, brought → meal}

1. **if** $arc(TOP, FIRST) \in A$, **then** right-arc;
2. **else if** $arc(FIRST, TOP) \in A$, **then** left-arc;
3. **else if** $\exists k \in Stack, arc(FIRST, k) \in A$ or $arc(k, FIRST) \in A$, **then** reduce;
4. **else** shift.

As an example, let us examine the oracle in action with the sentence:

The waiter brought the meal.

The dependency graph can be represented by the set of links:

$$\{\text{the} \leftarrow \text{waiter}, \text{waiter} \leftarrow \text{brought}, \text{ROOT} \rightarrow \text{brought}, \text{the} \leftarrow \text{meal}, \text{brought} \rightarrow \text{meal}\},$$

where we use left or right arrows to give the direction of the dependency, and the dummy word ROOT to indicate the root of a sentence. Knowing this graph, we can invoke the oracle to find the transition sequence. Table 13.2 shows the parser

states in the form of a stack, an input list, and the graph in construction. We start with an empty list of actions, and we ask the oracle at each step of the parsing procedure: What is the next action? When we reach the end of the input list, we have the complete transition sequence,

```
[sh, sh, la, sh, la, ra, sh, la, ra]
```

and we can check that the parser has built a graph that is equal to the one we constructed manually.

13.3.3 Nivre's Parser in Prolog

Nivre's parser is simple to understand, and its implementation in Prolog is easy. Let us call the top-level predicate

```
nivre_parser(+Sentence, ?Operations, ?RefGraph)
```

where `Sentence` is the input sentence, `RefGraph`, the manually-annotated dependency graph, and `Operations`, the sequence of transitions that produces this graph. We will design the program to work with two modes. The first one will use `Sentence` and `RefGraph` as input, and for each projective graph, the program will output a list of transitions. The second mode will use `Sentence` and `Operations` as input to produce a graph.

We represent the input sentences as lists of words with their corresponding position, form, and part of speech, `w([id=ID, form=FORM, postag=POSTAG])`, and we use a variation of the CONLL format described in Sect. 11.9.2 to encode dependencies, `w([id=ID, form=FORM, head=HEAD, deprel=DEPREL])`. The input sentence *The waiter brought the meal* is represented as:

```
[w([id=0, form=root, postag='ROOT']),
w([id=1, form=the, postag='DT']),
w([id=2, form=waiter, postag='NN']),
w([id=3, form=brought, postag='VBD']),
w([id=4, form=the, postag='DT']),
w([id=5, form=meal, postag='NN'])]
```

and its dependency graph as:

```
[w([id=1, form=the, head=2, deprel=det]),
w([id=2, form=waiter, head=3, deprel=sub]),
w([id=3, form=brought, head=0, deprel=root]),
w([id=4, form=the, head=5, deprel=det]),
w([id=5, form=meal, head=3, deprel=obj])]
```

Finally, the `Operations` list will consist of symbols corresponding to the initials of the transitions: {`la`, `ra`, `re`, `sh`}. For example, the sequence, shift, right-arc, left-arc, reduce, will be represented by `[sh, ra, la, re]`.

The Four Transitions

The Prolog program uses four transitions shown in Table 13.1 that we transcribe as four predicates: `left_arc/6`, `right_arc/6`, `shift/4`, and `reduce/3`.

The `left_arc/6` predicate adds an arc to the graph linking the first word of the list to the top of the stack with the condition that the first word has no head in the current graph.

```
% left_arc(+Words, +Stack, -NewStack, +Graph,
    -NewGraph)
left_arc([W | _], [T | Stack], Stack, Graph,
    [w([id=IDT, form=FORMT, head=IDW, deprel=_]) |
    Graph]) :-
  W = w([id=IDW | _]),
  T = w([id=IDT, form=FORMT | _]),
  \+ member(w([id=IDT, form=FORMT | _]), Graph).
```

The `right_arc/6` predicate adds an arc to the graph linking the top of the stack to the first word of the list with the condition that the top of the stack has no head in the current graph.

```
% right_arc(+Words, -NewWords, +Stack, -NewStack,
%  +Graph, -NewGraph)
right_arc([W | Words], Words, [T | Stack],
    [W, T | Stack], Graph, [w([id=IDW, form=FORMW,
      head=IDT, deprel=_]) | Graph]) :-
  W = w([id=IDW, form=FORMW | _]),
  T = w([id=IDT | _]),
  \+ member(w([id=IDW, form=FORMW | _]), Graph).
```

The `reduce/3` predicate reduces the `Stack` provided that the word has a head already in the graph.

```
% reduce(+Stack, -NewStack, +Graph)
reduce([T | Stack], Stack, Graph) :-
  T = w([id=IDT, form=FORMT | _]),
  member(w([id=IDT, form=FORMT | _]), Graph).
```

The `shift/4` predicate removes the next word from the list currently being parsed and pushes it on the top the stack. Here we set it as the head of the `Stack` list to produce a `NewStack`:

```
% shift(+Words, -NewWords, +Stack, -NewStack)
shift([First | Words], Words, Stack, [First | Stack]).
```

The Oracle

The oracle determines which transition to apply given a certain parser state. The `oracle/4` predicate uses the algorithm described in Sect. 13.3.2 and unifies `Operation` with a value in the set {`la`, `ra`, `re`, `sh`}. However, the first Prolog rule checks whether `Operation` is already instantiated when the predicate is called and if yes, it does not execute the algorithm. This will enable the program to work with two modes: determine a transition sequence from a graph, and also build a graph from a given transition sequence.

```
% oracle(+Words, +Stack, +Graph, -Operation)
% Predicts the next transition from the
   manually-annoted graph
oracle(_, _, _, Op) :-
  nonvar(Op),
  !.
oracle([W | _], [T | _], Graph, la) :-
  T = w([id=IDT, form=FORMT | _]),
  W = w([id=IDW | _]),
  member(w([id=IDT, form=FORMT, head=IDW | _]),
   Graph),
  !.
oracle([W | _], [T | _], Graph, ra) :-
  T = w([id=IDT | _]),
  W = w([id=IDW, form=FORMW | _]),
  member(w([id=IDW, form=FORMW, head=IDT | _]),
   Graph),
  !.
oracle([W | _], [_ |Stack], Graph, re) :-
  member(K, Stack),
  K = w([id=IDK, form=FORMK | _]),
  W = w([id=IDW, form=FORMW | _]),
  (
  member(w([id=IDK, form=FORMK, head=IDW | _]), Graph)
  ;
  member(w([id=IDW, form=FORMW, head=IDK | _]), Graph)
  ),
  !.
oracle(_, _, _, sh).
```

The Top-Level Predicate

The top-level predicate calls an auxiliary predicate that stores the stack and the current graph. If it fails to produce a transition sequence, it returns a list

containing the atom `fail`. We can use the program in two ways with the modes: `nivre_parser(+Sentence, -Transitions, +Graph)` and `nivre_parser(+Sentence, +Transitions, -Graph)`.

```
nivre_parser(Sentence, Ops, RefGraph) :-
  nivre_parser(Sentence, [], Ops, [], RefGraph).
nivre_parser(_, [fail], _).
```

The auxiliary predicate consists of three rules, where the first two represent terminal conditions, and the third is the general recursive case. The terminal conditions correspond respectively to the two possible modes. The first rule is used when the program produces a transition sequence from a reference graph. The terminal condition is met when the queue is empty. We check additionally that the two graphs – the reference graph and the graph built by the parser – are equal. This is always the case when the reference graph is well-formed and projective. The second rule corresponds to the application of a transition sequence, and the output is the graph. Finally, the recursive rule calls the oracle and executes the predicted transition.

```
nivre_parser([], _, [], CurGraph, RefGraph) :-
  nonvar(RefGraph),
  !,
  subset(RefGraph, CurGraph),
  subset(CurGraph, RefGraph).
nivre_parser([], _, [], Graph, Graph).
nivre_parser(Words, Stack, [Op | Ops], Graph,
  RefGraph) :-
  oracle(Words, Stack, RefGraph, Op),
  execute_action(Op, Words, NWords, Stack, NStack,
    Graph,NGraph),
  nivre_parser(NWords, NStack, Ops, NGraph,
    RefGraph).
```

The `execute_action/7` predicate calls the predicted operation and produces a new parser state. Its purpose is merely to select the arguments of the transitions.

```
% execute_action(+Op, +Words, -NewWords, +Stack,
% -NewStack, +Graph, NewGraph)
execute_action(la, Words, Words, Stack, NStack, Graph,
    NGraph) :-
  left_arc(Words, Stack, NStack, Graph, NGraph).
execute_action(ra, Words, NWords, Stack, NStack,
    Graph,NGraph) :-
  right_arc(Words, NWords, Stack, NStack, Graph,
    NGraph).
execute_action(re, Words, Words, Stack, NStack, Graph,
```

```
    Graph) :-
  reduce(Stack, NStack, Graph).
execute_action(sh, Words, NWords, Stack, NStack,
    Graph,Graph) :-
  shift(Words, NWords, Stack, NStack).
execute_action(Op, _, _, _, _, _, _) :-
  \+ member(Op, [la, ra, re, sh]),
  write('Illegal action. Returning'), nl.
```

Running the Parser

Applying the parser to *The waiter brought the meal* yields:

```
?- nivre_parser([w([id=0, form=root, postag='ROOT']),
    w([id=1, form=the, postag='DT']), ...], A,
        [w([id=1, form=the, head=2, deprel=det]),
          w([id=2, form=waiter, ...]).
A = [sh, sh, la, sh, la, ra, sh, la, ra] .
```

Conversely, given this transition sequence, the parser produces the original dependency graph.

13.4 Guiding Nivre's Parser

So far, we gave the parser a solution in the form of a reference graph to find the transition sequence. In most practical cases, this is not what we want. We generally have a sentence as input, and our goal is to automatically build a graph. We will now introduce techniques to carry this out. We will start with symbolic rules that resemble phrase-structure rules for dependencies: dependency rules. We will then move on to consider machine-learning techniques so that we can train our parser on manually-annotated corpora. The combination of machine learning and Nivre's parser produces highly efficient systems.

13.4.1 Parsing with Dependency Rules

Dependency Rules

Writing dependency rules or *D*-rules consists in describing possible dependency relations between word categories (Covington 1990): typically a head part of speech to a dependent part of speech (Fig. 13.3).

Fig. 13.3 Examples of D-rules

```
1. determiner ← noun.   4. noun ← verb.
2. adjective ← noun.    5. preposition ← verb.
3. preposition ← noun.  6. verb ← root.
```

These rules mean that a determiner can depend on a noun (1) (or that a noun can be the head of a determiner), an adjective can depend on a noun (2), and a noun can depend on a verb (4). The rules express ambiguity. A preposition can depend either on a verb (5) as in *Bring the meal to the table* or on a noun (3) as in *Bring the meal of the day*. Finally, rule 6 means that a verb can be the root of the sentence.

In our example, D-rules use parts of speech, which means that before parsing, we must use a POS tagger to annotate each word of the input list. D-rules can also involve the lexical value of the words or their semantic category or as, for instance,

of ← noun.

which means that the word *of* depends on a noun. When the rules involve word values, they are said to be **lexicalized**.

D-rules are often related to one or more functions. The first rule in Fig. 13.3 expresses the determinative function, the second one is an attributive function, and the third rule can be a subject, an object, or an indirect object function. Using a unification-based formalism, rules can encapsulate functions, as in:

$$\begin{bmatrix} category : noun \\ number : N \\ person : P \\ case : nominative \end{bmatrix} \leftarrow \begin{bmatrix} category : verb \\ number : N \\ person : P \end{bmatrix}$$

which indicates that a noun marked with the nominative case can depend on a verb. In addition, the noun and verb share the person and number features. Unification-based D-rules are valuable because they can easily pack properties into a compact formula: valence, direction of dependency relation (left or right), lexical values, etc. (Covington 1989; Koch 1993).

Using D-Rules with Nivre's Parser

We can use D-rules to guide Nivre's parser. We need first to write a grammar, a set of rules, for the language we want to process and write a guide predicate so that at a given point of the analysis it will examine the grammar and select a transition among the four possible ones.

Most languages have directional constraints for the dependencies. For instance, a determiner is always before the noun in English, French, and German. We can formulate these constraints using oriented D-rules to represent left, $POS(n') \leftarrow POS(n)$, and right, $POS(n) \rightarrow POS(n')$, dependencies. In the first case, the head is to the left of the dependent, and in the second one, it is the opposite. Oriented D-rules reduce ambiguity significantly.

Table 13.3 Conditions on the left-arc and right-arc transitions to run Nivre's parser with D-rules

Actions	Parser transitions	Conditions
Left-arc	$\langle n\|S, n'\|I, A\rangle \rightarrow \langle S, n'\|I, A \cup \{(n', n)\}\rangle$	$POS(n) \leftarrow POS(n') \in R$ $\nexists n''(n'', n) \in A$
Right-arc	$\langle n\|S, n'\|I, A\rangle \rightarrow \langle n'\|n\|S, I, A \cup \{(n, n')\}\rangle$	$POS(n) \rightarrow POS(n') \in R$

The guide links the D-rules to the parsing actions. To carry out a left-arc, the grammar must contain the rule $POS(n) \leftarrow POS(n')$ and to carry a right-arc, the grammar must contain the rule $POS(n) \rightarrow POS(n')$. Table 13.3 shows the new conditions to run Nivre's parser with D-rules.

The D-rule constraints are not sufficient to write a complete guide as the transition selection is still ambiguous. We can always apply a shift, and the grammar may contain conflicting rules; for instance, a preposition can depend on a noun and a noun can depend on a preposition. When two or more transitions are applicable, we need a procedure to select a unique one. In its original article, Nivre (2003) experimented with three parsing strategies that depended on the transition priorities. The first two are:

- The parser uses the constant priorities for the transitions: left-arc > right-arc > reduce > shift.
- The second parser uses the constant priorities left-arc > right-arc and a rule to resolve shift/reduce conflicts. If the top of the stack can be a transitive head of the next input word according to the grammar, then shift; otherwise reduce.

Both strategies are easy to implement.

D-Rules and Nivre's Parser in Prolog

Before we start writing a guide, we need to write a grammar. We represent the D-rules with a `drule/4` predicate, where each rule contains the parts of speech of the head and the dependent, the function, and the dependency direction. A simple grammar of English using the Penn Treebank tagset will be:

```
%drule(+HeadPOS, +DepPOS, +Function, +Direction)
drule('ROOT', 'VBD', root, right).
drule('NN', 'DT', determinative, left).
drule('NN', 'JJ', attribute, left).
drule('VBD', 'NN', subject, left).
drule('VBD', 'PRP', subject, left).
drule('VBD', 'NN', object, right).
drule('VBD', 'PRP', object, right).
drule('VBD', 'IN', adv, _).
drule('NN', 'IN', pmod, right).
drule('IN', 'NN', pcomp, right).
```

Let us now write a guide predicate that resembles those proposed by Nivre (2003). It consists of five rules that describe transition priorities. The first rule tries a left-arc. It looks for a D-rule that links the top of the stack to the first word in the queue. If this fails, the second rule tries a right-arc. It looks for a D-rule that links the first word in the queue to the top of the stack. If we cannot execute these actions, either because there is no D-rule or because their conditions are not met, we try shift with a lookahead condition that the top of the stack can be the head of the second-next word in the queue. We look for a corresponding D-rule. If this does not work, we try reduce and then shift.

```
% guide(+Words, +Stack, -Operation)
guide([W | _], [T | _], la) :-
  T = w([id=_, form=_, postag=POST | _]),
  W = w([id=_, form=_, postag=POSW | _]),
  drule(POSW, POST, _, left).
guide([W | _], [T | _], ra) :-
  T = w([id=_, form=_, postag=POST | _]),
  W = w([id=_, form=_, postag=POSW | _]),
  drule(POST, POSW, _, right).
guide([_, W | _], [T | _], sh) :-
  T = w([id=_, form=_, postag=POST | _]),
  W = w([id=_, form=_, postag=POSW | _]),
  drule(POST, POSW, _, right).
guide(_, _, re).
guide(_, _, sh).
```

We need to slightly modify `nivre_parser/5` to run it with the guide:

```
nivre_parser([], _, [], Graph, Graph).
nivre_parser(Words, Stack, [Op | Ops], Graph,
  RefGraph) :-
  guide(Words, Stack, Op),
  execute_action(Op, Words, NWords, Stack, NStack,
    Graph,NGraph),
  nivre_parser(NWords, NStack, Ops, NGraph,
    RefGraph).
```

Applying the parser to *The waiter brought the meal*, where each word was tagged with its part of speech, yields the correct graph:

```
[w([id=5, form=meal, head=3, deprel=_G1015]),
 w([id=4, form=the, head=5, deprel=_G915]),
 w([id=3, form=brought, head=0, deprel=_G775]),
 w([id=2, form=waiter, head=3, deprel=_G675]),
 w([id=1, form=the, head=2, deprel=_G535])]
```

with the transition sequence `[sh, sh, la, sh, la, ra, sh, la, ra]`.

Evaluating the Parser

Using annotated corpora, it is possible to derive automatically a grammar of D-rules, then run and assess the parser. There are a few freely available corpora that we can download. Talbanken05 in Swedish (Einarsson 1976; Nilsson et al. 2005) is one example. It contains annotated Swedish sentences and was used in the CoNLL 2006 shared task (Buchholz and Marsi 2006). It comes with a training set and a test set as well as an evaluation procedure that is comparable to the one described in Sect. 13.2.

We can use the training set to extract dependency rules and then run the parser on the test set. Nivre's parser does not guarantee to produce connected graphs. This means that some words from the input sentence will have no head in the graph. In this case, we assign them a `ROOT` head. The result score will depend on the number of rules we use. With the 100 most frequent ones in the CoNLL 2006 training set, we reach an attachment score of about 57 % in the test set.

Although this result is far from the state of the art, we obtained it with a minimal guide. There are many ways to improve it, and this means that using Nivre's parser with D-rules is a viable strategy. Here are a list of possible improvements: introduce lexicalized D-rules, constrain the parser to produce connected graphs, and constrain the graph to have only one root. We leave these improvements as an exercise (Exercise 13.1).

13.4.2 Using Machine-Learning Techniques

D-rules provide a simple way to control a dependency parser. However, they use limited information to make a decision: the part of speech of two words, the top of the stack, and the first word in the queue. Most current implementations of Nivre's parser use a richer set of features and hence rely on machine-learning techniques to implement the guide.

We now describe techniques that are similar to those we used with chunking (see Sect. 10.7.3) and that fit the sequential nature of Nivre's parser. We modify the guide so that before each transition it extracts features from the parser state. The features represent a sort of context to the transition and, using them as an input to a classifier, the guide predicts the next transition.

We build the classifier from a data set using a machine-learning algorithm. We collect a set of transition contexts from an annotated corpus using gold-standard parsing. This produces a list of feature values and the corresponding transition. We then automatically train a classifier such using ID3, logistic regression, or support vector machines (Chap. 4) that we have embedded in the guide.

Table 13.4 Feature vectors extracted while parsing the sentence *The waiter brought the meal*. In the *left* part of the table, the parser prints the parts of speech of the top of the stack and next in the queue before each transition. In the *middle* part, the parser prints two words from the stack and the queue

Stack	**Queue**	**Stack**		**Queue**		
POS(T_0)	POS(Q_0)	POS(T_0)	POS(T_{-1})	POS(Q_0)	POS(Q_{+1})	**Transitions**
nil	ROOT	nil	nil	ROOT	DT	sh
ROOT	DT	ROOT	nil	DT	NN	sh
DT	NN	DT	ROOT	NN	VBD	la
ROOT	NN	ROOT	nil	NN	VBD	sh
NN	VBD	NN	ROOT	VBD	DT	la
ROOT	VBD	ROOT	nil	VBD	DT	ra
VBD	DT	VBD	ROOT	DT	NN	sh
DT	NN	DT	VBD	NN	nil	la
VBD	NN	VBD	ROOT	NN	nil	ra

Predicting the Parser Transitions

We saw that gold-standard parsing could produce a transition sequence from a well-formed projective graph, and that the output sequence would exactly generate the input graph. While parsing, each transition has a corresponding parser state from which we can extract information in the form of feature vectors. To train a classifier, the idea is to associate a feature vector with the next transition.

The simplest features correspond to words' parts of speech on the top of the stack, POS(T_0), and the next word in the input queue, POS(Q_0). This is more or less the information provided by D-rules. A generalization is straightforward, and we can use more data from the stack or the queue and extend the size of the vector from two to four or more. We can also lexicalize the features and use the word forms in addition to their parts of speech. Table 13.4 shows feature vectors extracted while parsing the sentence:

> The waiter brought the meal

annotated as

> root/ROOT the/DT waiter/NN brought/VBD the/DT meal/NN.

The vectors use the part of the speech of the words and their dimension is two, POS(T_0), POS(Q_0), and four, POS(T_0), POS(T_{-1}), POS(Q_0), POS(Q_{+1}).

After the data is collected from an annotated corpus, we can apply a training procedure to create a 4-class classifier. Once trained, given a feature vector the classifier will choose a transition in the set {la, ra, sh, re}. We can then embed this classifier in the guide of Nivre's parser. When parsing a sentence, the parser extracts the current context after each transition and asks the guide to predict the next one. Support vector machines are among the most effective training algorithms we can use. In the next section, we use the C4.5 implementation of Weka (Sect. 4.3.2). The C4.5 training procedure is fast and produces decision trees that are easy to understand.

Table 13.5 Parsing performance using different feature sets on Swedish data. The training data was extracted from projective sentences in the CoNLL 2006 training set. The decision trees were trained using the C4.5 implementation available from the Weka environment. Words that had no head after parsing were assigned the ROOT head. The evaluation was carried out on the test set using the CoNLL 2006 script, which uses the attachment score. See Sect. 13.2 for a definition. The annotated data as well as the evaluation script are available from the CoNLL 2006 web page (After data provided by Buchholz and Marsi (2006))

Stack		**Queue**			**Constraints**			**Graph**	
POS(T_0)	POS(T_{-1})	POS(Q_0)	POS(Q_{+1})	POS(Q_{+2})	LA	RA	RE	LMS	Score
•	–	•	–	–	–	–	–	–	64.57
•	–	•	–	–	–	–	–	•	67.42
•	–	•	–	–	•	•	•	–	72.83
•	–	•	–	–	•	•	•	•	73.41
•	•	•	•	•	–	–	–	–	78.05
•	•	•	•	•	–	–	–	•	78.71
•	•	•	•	•	•	•	•	–	81.30
•	•	•	•	•	•	•	•	•	81.64

Feature Vectors and Parsing Performance

The feature set is a key factor in the parsing performance, and designing a good set is a very significant issue in practice. To realize it, we will experiment with our parser with two words in the stack and three words in the queue, and we will compare the results with a context involving just one word. We will also include the leftmost dependent of the top of the stack.

To help the guide, we will extract three additional Boolean parameters to check that the transition conditions are met: "can do left-arc", "can do right-arc", and "can do reduce". These features are intended to have the classifier model constraints on transitions. Hopefully, it will learn them and, as far as possible, avoid predicting illegal transitions. If this nevertheless happens and the classifier predicts a transition that violates the constraints, the guide will fall back to the priorities `la` > `ra` > `re` > `sh`. In total, the sets will have from two to eight parameters.

Table 13.5 shows the scores and hints that larger feature sets yield better results. This rule is true in general until we reach a ceiling in the size of the set. Using, say, 20 words from the queue would probably not improve the parser performance. In addition, there is a danger with larger sets to be overadapted to the training data. The classifier will then tend to learn specific properties of the data it is trained on and be biased toward it. An overadapted feature set will yield superior scores on training data, but possibly have inferior results on a different corpus. We call this an **overfit**.

The best score we obtain in Table 13.5 is 81.64. This is below the top score of the CoNLL 2006 shared task, which was 89.54. However, it is a good result given that we have only used parts of speech and that C4.5 is not as efficient as support vector machines. We can easily improve the performance using lexical features, such as the word form or the lemma. In fact, lexicalization has a strong impact on the parsing performance. Many attachment ambiguities can only be resolved by the knowledge

Table 13.6 Model for the feature set, where *top* denotes the top of the stack and *next*, the next in the queue. The feature names follow the CoNLL format described in Table 11.12 (After Nivre et al. (2006))

Location	Description	FORM	LEMMA	CPOS	POS	FEATS	DEPREL
Stack	*top*	•	•	•	•	•	•
Stack	*top* − 1				•		
Input	*next*	•	•	•	•	•	
Input	*next* + 1	•			•		
Input	*next* + 2				•		
Input	*next* + 3				•		
Graph	Head of *top*	•					
Graph	Leftmost dependent of *top*						•
Graph	Rightmost dependent of *top*						•
Graph	Leftmost dependent of *next*						•

of the lexical values. As an example of possible extensions, Nivre et al. (2006) proposed an effective feature set shown in Table 13.6.

Finding Grammatical Functions

We saw in Chap. 11 that the grammatical functions form an important layer in the description of dependency relations. In addition, the analysis of functions is essential in many applications. However, so far, our parser has only created the arcs of the dependency graphs. The time has now come to see how we can identify functions.

From a technical viewpoint, functions are just labels to add to the arcs, and there are two main ways to assign them:

1. We can first modify the guide and use a two-step classifier: the first one predicts the next transition as before and if it is a left-arc or a right-arc, a second classifier predicts the function. This second classifier will have as many classes as there are functions and can be trained from the functions extracted from gold-standard parsing.
2. A second possibility is to extend the left-arc and right-arc transitions to include the function. The transition sequence to parse *The waiter brought the meal* is shown in Table 13.2: `[sh, sh, la, sh, la, ra, sh, la, ra]`. The augmented transitions to assign the functions will be:
   `[sh, sh, la-det, sh, la-sub, ra-root, sh, la-det, ra-obj]`.
   We will just need one classifier to predict both the transition and the function. We train this classifier from the new transitions we collect with the gold-standard parsing procedure.

The annotation format of CoNLL 2006 and 2007 makes provision for grammatical functions. This means that we can train our transition and function classifiers on the same corpus. Moreover, classifiers can more or less use the same features. See Table 13.6 for an example. The evaluation is carried out the same way as for the bare dependency graphs, but this time it measures the labeled attachment score: the proportion of arcs that have both a correct head and a correct label. This label is called DEPREL in the CoNLL format.

Parsing Nonprojective Links

Identifying functions has an interesting side effect: it enables Nivre's parser to handle nonprojectivity. By construction, this parser is limited to projective graphs. However, we saw in Sect. 11.10.2 that is it possible to projectivize nonprojective graphs. We also saw that we can recover the original nonprojective graphs if we mark the arc labels with the projectivization operations. This, of course, assumes that the arcs in the graphs have labels (functions).

Now we can build a nonprojective system. It uses a projective parser that is preceded by a preprocessing step that projectivizes the training corpus and followed by a postprocessing step that recovers nonprojective arcs from the parsed sentences.

The preprocessing step corresponds to the procedure explained in Sect. 11.10.2 (Kunze 1967; Nivre and Nilsson 2005). It creates a new set of function labels, where the new labels annotate the projectivized arcs. Once this preprocessing step is applied to the training set, we can train the classifiers using a projective parser. When the classifiers are trained, we can run the parser on sentences. It labels the arcs with the original functions as well as with nonprojective markers. We finally apply the postprocessing step to identify these markers and create nonprojective arcs.

13.5 Finding Dependencies Using Constraints

Using constraints is a symbolic strategy that can be an alternative to D-rules. Although it is not as widely used as machine-learning techniques, it can be of interest when no annotated corpus is available. The parsing algorithm is then framed as a constraint satisfaction problem.

Constraint dependency parsing annotates words with dependencies and function tags. It then applies a set of constraints to find a tag sequence consistent with all the constraints. Some methods generate all possible dependencies and then discard inconsistent ones (Harper et al. 1999; Maruyama 1990). Others assign one single dependency per word and modify it (Tapanainen and Järvinen 1997).

Let us exemplify a method inspired by Harper et al. (1999) with the sentence *Bring the meal to the table*. Table 13.7 shows simplified head and function assignments compatible with a word's part of speech.

Table 13.7 Possible functions according to a word's part of speech

Parts of speech	Possible heads	Possible functions
Determiner	Noun	det
Noun	Verb	object, iobject
Noun	Prep	pcomp
Verb	Root	root
Prep	Verb, noun	mod, loc

The first step generates all possible head and function tags. Using Table 13.7, tagging yields:

Words	*Bring*	*the*	*meal*	*to*	*the*	*table*
Position	1	2	3	4	5	6
Part of speech	verb	det	noun	prep	det	noun
Possible tags	<nil, root>	<3, det>	<4, pcomp>	<3, mod>	<3, det>	<4, pcomp>
		<6, det>	<1, object>	<1, loc>	<6, det>	<1, object>
			<1, iobject>			<1, iobject>

Then, a second step applies and propagates the constraint rules. It checks that the constraints do not conflict and enforces the consistency of tag sequences. Rules for English describe, for instance, adjacency (links must not cross), function uniqueness (there is only one subject, one object, one indirect object), and topology:

- A determiner has its head to its right-hand side.
- A subject has its head to its right-hand side when the verb is at the active form.
- An object and an indirect object have their head to their left-hand side (active form).
- A prepositional complement has its head to its left-hand side.

Applying this small set of rules discards some wrong tags but leaves some ambiguity.

Words	*Bring*	*the*	*meal*	*to*	*the*	*table*
Position	1	2	3	4	5	6
Part of speech	verb	det	noun	prep	det	noun
Possible tags	<nil, root>	<3, det>	<1, object>	<3, mod>	<6, det>	<4, pcomp>
			<1, iobject>	<1, loc>		

13.6 Covington's Parser

In the previous sections, we saw that Nivre's parser was restricted to the class of projective graphs. Although projective structures are by far the most frequent in English or French, this restriction impairs the expressivity of the parser, notably in languages where nonprojectivity is not as anecdotal: German, Latin, Russian, etc.

We now introduce Covington's parser (1990, 1994a, 2001), which extends parsing to nonprojective dependencies. Covington's parser is, in fact, a family of algorithms. It starts with a brute-force search that considers all the word pair combinations on which we can gradually set constraints to comply with some generally accepted properties of dependency graphs: head uniqueness and possibly projectivity.

The parser is relatively easy to implement, and it can use D-rules or machine learning techniques. We first introduce a nonprojective version of it and we refine it to produce projective graphs. Like the other parsers we have already seen, the parser input will be a tokenized sentence, where words are tagged with their part of speech.

13.6.1 Covington's Nonprojective Parser

The brute-force version of Covington's algorithm examines each pair of words in the sentence and tries to set a link between them if the grammar or any kind of guide permits it. It can be implemented as a left-to-right pass with the following code:

Algorithm 1 The Covington algorithm

```
1: procedure PARSE(w_1, w_2, ..., w_n)
2:     for i ← 1, n do
3:         for j ← i − 1, 1 do
4:             if PERMISSIBLE(w_i, w_j) then
5:                 LINK(w_i, w_j)
```

that scans the word list and attempts to create links with the words to the left of the current word, w_i. The link operation can either create a left-arc $w_j \leftarrow w_i$, a right-arc $w_j \rightarrow w_i$, or do nothing.

A widely accepted property of dependency graphs is that each word must have a unique head. We will now use lists to represent data and enforce uniqueness. We will store the sentence's words in an input list: `InputList`. The parser accepts one word at a time and maintains two other lists: `HeadList` and `WordList`. `WordList` contains the words already read in a decreasing index order. `HeadList` contains the words that have not been assigned a head yet, also in a decreasing index order. At the beginning of the parse, both `HeadList` and `WordList` are empty.

1. Accept a word `W` from the input list. Add it to `WordList`.
2. Search `HeadList` to find dependents of `W` starting with the most recently added. Words found are removed from `HeadList`.
3. Search the elements of `WordList` to find at most one head for `W`. If no head is found, add `W` to `HeadList`.

At the end of the parse, `HeadList` should contain the head of the sentence.

At each step of the traversal of both `HeadList` and `WordList`, the parser has to decide whether or not it sets a link between the current word and the word in the list. We carry this out with a guide using the term we introduced in Sect. 13.4. Classifiers in the form of decision trees, logistic regression, or support vector machines will predict the existence or absence of a link and the relation holding between two words (if any). To train the classifiers, we need to collect data from a hand-annotated corpus. We extract features using a gold-standard parsing. In the rest of this section, we will exemplify the guide with dependency rules. Extending Covington's parser with machine-learning techniques is left as exercises (Exercises 13.4 and 13.5).

Let us now implement this algorithm in Prolog. We will use the dependency rules below. They have three arguments: the head part of speech, the dependent part of speech, and the grammatical function linking the words. We use a `guide/3` predicate between the rules and the parser to make the linking decision easier to port a classifier.

```
% drule(HeadPOS, DependentPOS, Function)
drule(noun, det, det).
drule(noun, adj, attribute).
drule(verb, noun, subject).
drule(verb, noun, object).
```

Let us now write the parsing algorithm. The input–output format will follow the CoNLL tabular structure. The input will be a sentence in the form of a list of words, where each word will have an index, a form, and a part of speech:

```
w([id=Inx, form=Word, postag=POS])
```

The output will be the dependency graph. We will use the same format as for the input, and we will add a head index and a grammatical function to the words:

```
w([id=Inx, form=Word, postag=POS, head=HdInx,
  deprel=Funct]).
```

The `parse/2` predicate needs an auxiliary `parse/4` to store the lists, `WordList` and `HeadList`. Instead of pushing the current word in `WordList`, we push the whole `w/1` fact as it is being parsed. If we can find a head to it in `WordList`, `w/1` will be fully instantiated. Otherwise, we will leave `HdInx` and `Funct` as variables until we find a word to be a suitable head. `WorldList` will store the parse result. At the end of the process, we attach the words remaining in `HeadList` to the root and we reverse `WordList`.

```
parse(InputL, Result) :-
  parse(InputL, [], [], ReversedResult),
  reverse(ReversedResult, Result).

% parse(+InputL, +HeadList, +WordList, -Result)
% We search dependents of W in HeadList and a head for
```

```
% W in WordList
parse([w([Inx, W, POS]) | InputL], HeadL, WordL,
  Result) :-
  search_headlist(w([Inx, W, POS]), HeadL, NewHeadL,
  WordL),
  search_wordlist(w([Inx, W, POS]), WordL, NewHeadL,
    NextHeadL, HeadInx, Function),
  parse(InputL, NextHeadL, [w([Inx, W, POS,
    head=HeadInx,deprel=Function])  | WordL], Result).

%The remaining headless words are assigned the ROOT
head parse([], HeadL, WordL, WordL) :-
  assign_root(HeadL, WordL).

% assign_root(+HeadList, -WordList)
% assigns roots to the words remaining in WordList
assign_root([], _).
assign_root([w([id=Inx, form=W, postag=POS]) | Rest],
    WordL) :-
  member(
    w([id=Inx, form=W, postag=POS, head=0,
    deprel='ROOT']),WordL),
  assign_root(Rest, WordL).


% search_headlist(+CurWord, +HeadL, -NewHeadL,
  -WordL).
% Searches dependency links in HeadL, and possibly
% assigns them the current word as their head
search_headlist(w([id=Inx1, W, postag=POS]),
  [w([id=Inx2, D, postag=POS_D]) | HeadL],
  NewHeadL, WordL) :-
  % Create left arc
  drule(POS, POS_D, F),
  % We instantiate the relation in WordL
  member(w([id=Inx2, D, postag=POS_D, head=Inx1,
    deprel=F]),WordL),
  search_headlist(w([id=Inx1, W, postag=POS]), HeadL,
    NewHeadL, WordL).
search_headlist(CurWord, [Word | HeadL],
    [Word | NewHeadL],WordL) :-
  % Do nothing
  search_headlist(CurWord, HeadL, NewHeadL, WordL).
search_headlist(_, [], [], _).
```

```
% search_wordlist(+CurWord, +WordL, +HeadL,
%   -NextHeadL, -Head, -Deprel),
% Tries to find a head in WordL
% We look for the first word that can be the head
% of the current word
% Nothing has been done. Shift CurWord to HeadList
search_wordlist(CurWord, [], HeadL, [CurWord | HeadL],
     _, _).
search_wordlist(w([_, _, postag=POS]),
    [w([id=Inx2, _, postag=POS_H, _, _]) | _], HeadL,
    HeadL,Inx2, F) :-
  % Create right arc
  drule(POS_H, POS, F).
% We have not found it and we go on
search_wordlist(CurWord, [_ | WordL], HeadL,
    NextHeadL,H, F) :-
  % Do nothing
  search_wordlist(CurWord, WordL, HeadL, NextHeadL,
    H, F).
```

This algorithm, which is nondeterministic, can successfully parse the sentences:

The waiter ran
The waiter brought a meal

However, the good parse is not the first one delivered by Prolog. Notably, the first parse attaches *a* to *waiter* and assigns the dependency `brought -> meal` the subject function (Fig. 13.4). Prolog must backtrack to find the correct solution.

```
?- parse([w([id=1, form=the, postag=det]),
w([id=2, form=waiter, postag=noun]),
w([id=3, form=brought, postag=verb]),
w([id=4, form=a, postag=det]),
w([id=5, form=meal, postag=noun])], R).

R = [w([id=1, form=the, postag=det, head=2,
    deprel=det]),
w([id=2, form=waiter, postag=noun, head=3,
    deprel=subject]),
w([id=3, form=brought, postag=verb, head=0,
    deprel='ROOT']),
w([id=4, form=a, postag=det, head=2, deprel=det]),
w([id=5, form=meal, postag=noun, head=3,
    deprel=subject])]
```

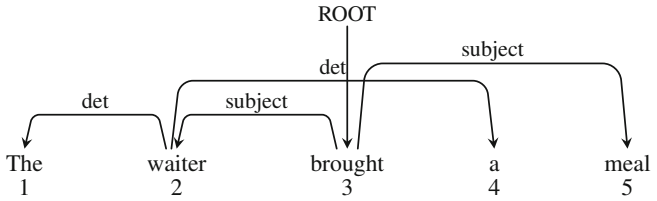

Fig. 13.4 The first parse

Table 13.8 Executing Covington's parser with *The waiter brought a meal*

Word	Procedure	Headlist	Wordlist	Actions
Init.		[]	[]	
w_1	Headlist			
		[]	[]	
	Wordlist			
		$[w_1]$	$[w_1 \leftarrow ?]$	
w_2	Headlist			la
		[]	$[w_1 \leftarrow w_2]$	
	Wordlist			noact
		$[w_2]$	$[w_2 \leftarrow ?, w_1 \leftarrow w_2]$	
w_3	Headlist			la
		[]	$[w_2 \leftarrow w_3, w_1 \leftarrow w_2]$	
	Wordlist			noact noact
		$[w_3]$	$[w_3 \leftarrow ?, w_2 \leftarrow w_3, w_1 \leftarrow w_2]$	
w_4	Headlist			noact
		$[w_3]$	$[w_3 \leftarrow ?, w_2 \leftarrow w_3, w_1 \leftarrow w_2]$	
	Wordlist			noact ra
		$[w_3]$	$[w_4 \leftarrow w_2, w_3 \leftarrow ?, w_2 \leftarrow w_3, w_1 \leftarrow w_2]$	
w_5	Headlist			noact
		$[w_3]$	$[w_4 \leftarrow w_2, w_3 \leftarrow ?, w_2 \leftarrow w_3, w_1 \leftarrow w_2]$	
	Wordlist			noact ra
		$[w_3]$	$[w_4 \leftarrow w_3, w_4 \leftarrow w_2, w_3 \leftarrow ?, w_2 \leftarrow w_3, w_1 \leftarrow w_2]$	

13.6.2 Relations Between Nivre's and Covington's Parsers

We can reframe Covington's parser and describe the link operations in terms of actions. Using this presentation, we will discover its similarities with Nivre's parser and we will be able to apply to it the same machine-learning techniques. When scanning `Headlist`, each time we set a link, it corresponds to a left-arc action of Nivre's parser. When scanning `WordList`, we create at most one link from a word to the current word, which corresponds to a right-arc action of Nivre's parser. To be compatible with the transition-based framework, we need to introduce a no-action that corresponds to list traversal steps, either `Headlist` or `WordList`, where we set no link to or from the current word (Table 13.8).

13.6.3 Covington's Projective Parser

Covington (1990) also provided a projective version of this parser. Recall the definition of a projective graph: if there is a sequence of words $w_i \ldots w_j \ldots w_k$ in which there is a dependency between w_i and w_k, in either direction, then w_j does not have a link preceding w_i or following w_k. To enforce projectivity, we modify the rules:

- When searching `HeadList`, use only the most recent elements and do not skip a word.
- When searching `WordList` for a head: first, skip all the words that are direct or indirect dependents of the current word w. Then, look at the preceding word, its head, its head's head, and so on, following a chain of dependency links.

```
search_headlist_for_dependents(w(W, POS),
    [w(D, POS_D) | HeadList], NewHeadList,
    WordList) :-
  drule(POS, POS_D, F),
  % We instantiate the relation in WordList
  member(w(D, POS_D, W, F), WordList),
  search_headlist_for_dependents(w(W, POS), HeadList,
    NewHeadList, WordList).
search_headlist_for_dependents(_, HL, HL, WordList).

% search_wordlist_for_a_head(+Word, +WordList, -Head,
% -Function),
% Tries to find a head in WordList

% We look for the first word that does not depend
% on the current word
search_wordlist_for_a_head(w(W, POS), WordList, H, F) :-
  next_with_no_link(w(W, POS), WordList, [],
    [w(H, POS_H, _, _) | NewWordList]),
  drule(POS_H, POS, F).

% next_with_no_link(+W, +WordList, -InvWordList,
  -Result)
% We go to the next word that has no link with the
% current word

next_with_no_link(w(W, POS), [w(H, POS_H, HH, F)
  | WordList],InvWordList, Result) :-
  link(w(W, POS), w(H, POS_H, HH, F), InvWordList),
  next_with_no_link(w(W, POS), WordList,
    [w(H, POS_H, HH, F) | InvWordList], Result).
```

```
next_with_no_link(w(W, POS), Result, _, Result).

% link(Word1, Word2, +InvWordList)
% Checks whether there is a link between Word1 and
% Word2 within InvWordList

link(w(W1, POS1), w(W2, POS2, H2, F2),
  InvWordList) :-
  H2 == W1.
link(w(W1, POS1), w(W2, POS2, H2, F2),
     [w(W3, POS3, H3, F3) | InvWordList]) :-
  W3 == H2,
  link(w(W1, POS1), w(W3, POS3, H3, F3),
    InvWordList).
link(w(W1, POS1), w(W2, POS2, H2, F2),
  [_ | InvWordList]) :-
  link(w(W1, POS1), w(W2, POS2, H2, F2),
    InvWordList).
```

Using this new version of the parser, we get better dependencies (Table 13.9):

```
?- parse([the, waiter, brought, a, meal], L).

L = [w(the, determiner, waiter, determinative),
w(waiter, noun, brought, subject),
w(brought, verb, _G861, _G862),
w(a, determiner, meal, determinative),
w(meal, noun, brought, subject)]
```

but *meal* is still the subject. Backtracking assigns the right function:

```
L = [w(the, determiner, waiter, determinative),
w(waiter, noun, brought, subject),
w(brought, verb, _G861, _G862),
w(a, determiner, meal, determinative),
w(meal, noun, brought, object)]
```

We could again improve the parser by enforcing some topology constraints, such as the object is after the verb when at the active form.

13.7 Eisner's Parser

Independently, Sleator and Temperley (1993) and Eisner (1996) developed chart-based dependency parsers that have an $O(n^3)$ complexity. Eisner's parser applies to projective graphs and can be combined with statistical or machine-learning methods. In CoNLL 2007 and 2008, this class of parsers delivered the highest accuracy figures for English.

Table 13.9 Executing Covington's parser with projectivity constraints with *The waiter brought a meal*

Word	Procedure	Headlist	Wordlist	Actions
Init.		[]	[]	
w_1	Headlist			
		[]	[]	
	Wordlist			
		$[w_1]$	$[w_1 \leftarrow ?]$	
w_2	Headlist			la
		[]	$[w_1 \leftarrow w_2]$	
	Wordlist			skip
		$[w_2]$	$[w_2 \leftarrow ?, w_1 \leftarrow w_2]$	
w_3	Headlist			la
		[]	$[w_2 \leftarrow w_3, w_1 \leftarrow w_2]$	
	Wordlist			skip skip
		$[w_3]$	$[w_3 \leftarrow ?, w_2 \leftarrow w_3, w_1 \leftarrow w_2]$	
w_4	Headlist			noact
		$[w_3]$	$[w_3 \leftarrow ?, w_2 \leftarrow w_3, w_1 \leftarrow w_2]$	
	Wordlist			noact
		$[w_4, w_3]$	$[w_4 \leftarrow ?, w_3 \leftarrow ?, w_2 \leftarrow w_3, w_1 \leftarrow w_2]$	
w_5	Headlist			la noact
		$[w_3]$	$[w_4 \leftarrow w_5, w_3 \leftarrow ?, w_2 \leftarrow w_3, w_1 \leftarrow w_2]$	
	Wordlist			skip skip ra
		$[w_3]$	$[w_4 \leftarrow w_3, w_4 \leftarrow w_5, w_3 \leftarrow ?, w_2 \leftarrow w_3, w_1 \leftarrow w_2]$	

Eisner's parser builds on an adaption of the CYK algorithm that it modifies to lower its complexity. We first describe the initial adaptation and then how to alter it to recreate the final parser.

13.7.1 Adapting the CYK Parser to Dependencies

Alshawi (1996) introduced a parser that resembles the CYK parser (Sect. 12.5.2) for dependencies. This parser uses the concept of dotted subtree (Eisner 2000): a sequence of words $w_s..w_t$ corresponding to the range $[s, t]$ and a root w_i inside this range: $s \leq i \leq t$, where all the words in the range are descendants of the root. We denote this subtree: (s, t, i).

A dotted subtree may not be complete; that is, the projection of the head may extend beyond the subtree in the final tree. Figure 13.5 shows an example of a dotted subtree spanning $w_3..w_5$ with *brought* as the head: $(3, 5, 3)$. This dotted subtree is not a real subtree as the projection of *brought* corresponds to the whole sentence.

As with the CYK algorithm, the parser combines dotted subtrees through a bottom up analysis. It initializes the chart with constituents of length 0 consisting of the individual words, and merges pairs of adjacent dotted subtrees (s_1, t_1, i) and

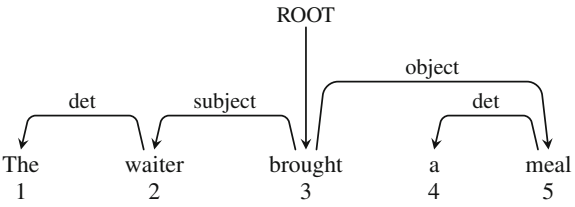

Fig. 13.5 Dependency graph of *The waiter brought a meal*

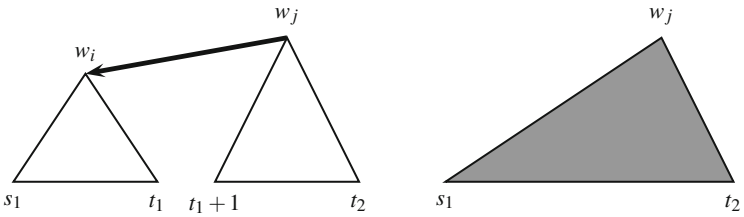

Fig. 13.6 **attach_left** operation: Merge two adjacent dotted subtrees rooted, respectively, at w_i and w_j, where w_j becomes the head of the resulting subtree

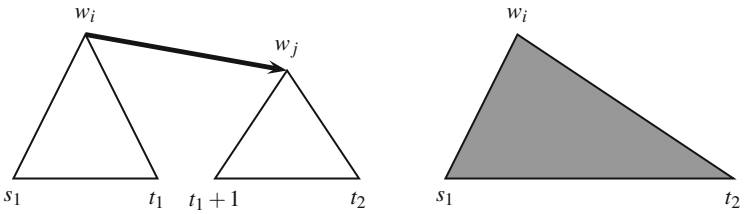

Fig. 13.7 **attach_right** operation: Merge two adjacent dotted subtrees rooted, respectively, at w_i and w_j, where w_i becomes the head of the resulting subtree

$(t_1 + 1, t_2, j)$ into a larger tree using two operations: **attach_left**$((s_1, t_1, i), (t_1 + 1, t_2, j))$ and **attach_right**$((s_1, t_1, i), (t_1+1, t_2, j))$. Their respective definitions are:

Initialization: $(0, 0, 0), (1, 1, 1), (2, 2, 2), \ldots, (i, i, i), \ldots, (n, n, n)$.

attach_left: The head of the right part, w_j, becomes the head of the resulting tree: (s_1, t_2, j) (Fig. 13.6) and

attach_right: This time the head of the left part, w_i, becomes the head of the new tree: (s_1, t_2, i) (Fig. 13.7).

Table 13.10 Chart corresponding to a sentence of length n: The chart is filled with constituents of increasing length, where the cells are filled with dotted subtrees

	Chart					
Length	0	1	2	$\ldots i$	$\ldots n-1$	n
0	$(0,0,0)$	$(1,1,1)$	$(2,2,2)$	$\ldots (i,i,i)$	$\ldots (n-1,n-1,n-1)$	(n,n,n)
1	$(0,1,0)$	$(1,2,1)$	$(2,3,2)$	$\ldots (i,i+1,i)$	$\ldots (n-1,n,n-1)$	–
	$(0,1,1)$	$(1,2,2)$	$(2,3,3)$	$\ldots (i,i+1,i+1)$	$\ldots (n-1,n,n)$	–
2	$(0,2,0)$	$(1,3,1)$	$(2,4,2)$	$\ldots (i,i+2,i)$	$\ldots -$	–
	$(0,2,1)$	$(1,3,2)$	$(2,4,3)$	$\ldots (i,i+2,i+1)$	$\ldots -$	–
	$(0,2,2)$	$(1,3,3)$	$(2,4,4)$	$\ldots (i,i+2,i+2)$	$\ldots -$	–
3	$\ldots$					
$\ldots$	$\ldots$					
$n-1$	$(0,n-1,0)$	$(1,n,1)$	–	$- -$	$- -$	–
	$(0,n-1,1)$	$(1,n,2)$	–	$- -$	$- -$	–
	$(0,n-1,2)$	$(1,n,3)$	–	$- -$	$- -$	–
	$\ldots$					
	$(0,n-1,n-1)$	$(1,n,n)$	–	$- -$	$- -$	–
n	$(0,n,0)$	–	–	$- -$	$- -$	–

Table 13.11 Chart corresponding to the dependency graph shown in Fig. 13.5. For instance, the triple $(0,3,0)$ corresponds to the subtree ranging from the root to word *brought*

Length\Word index	0	1	2	3	4	5
0	$(0,0,0)$	$(1,1,1)$	$(2,2,2)$	$(3,3,3)$	$(4,4,4)$	$(5,5,5)$
1		$(1,2,2)$			$(4,5,5)$	–
2		$(1,3,3)$		$(3,5,3)$	–	–
3	$(0,3,0)$			–	–	–
4			–	–	–	–
5	$(0,5,0)$	–	–	–	–	–
	root	the	waiter	brought	the	meal

After the initialization and similarly to CYK, the parser fills the cells of the chart with constituents (i,j,k) of increasing length: $j-i$ equals to 1, then 2, 3, ..., until the tree covers the range $[0,n]$ with the root at index 0: $(0,n,0)$. Each cell is the combination of two adjacent subtrees (Table 13.10).

Table 13.11 shows an example of the chart with the sentence *The waiter brought the meal*, where the cells are filled with triples leading to the complete tree.

We have presented a conversion of the CYK algorithm that enables us to parse dependencies. Using it, we can associate probabilities to the triples as with the CYK parser for constituents (Sect. 12.5.3), score the subtrees, and have a unique result for each sentence.

This version is far from optimal, however, as its time complexity is in $O(n^5)$: there are $O(n^3)$ triples to fill in the chart, and each triple (i,j,k) needs to examine $O(n^2)$ pairs, (i,l,k) and $(l+1,j,m)$, with varying l and m, to build it. We will see in the next section that we can bring small changes that lower its complexity to $O(n^3)$.

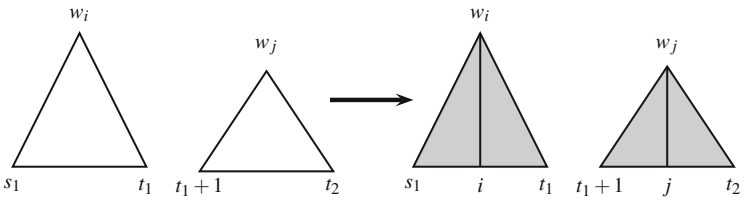

Fig. 13.8 Splitting the two trees into two parts on the head index. This results in four spans whose head is to the left or right of the span (*light gray*)

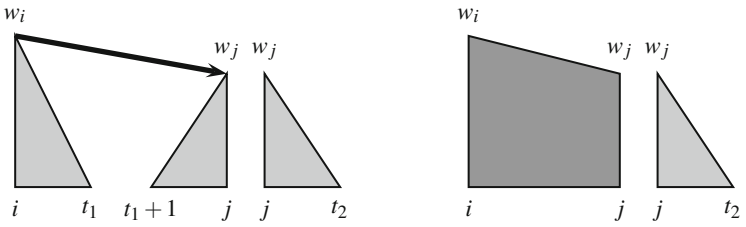

Fig. 13.9 **attach_right** creates an arc from w_i to w_j and a span that contains all the dependents to the right of w_i and to the left of w_j (*gray*)

13.7.2 A More Efficient Version

Eisner's parser is an improvement of the CYK conversion that constrains the position of the headword in the triple (i, j, k) to be either w_i or w_j. It thus eliminates the two variables representing the head positions and reduces the parsing complexity to $O(n^3)$.

To put this idea into practice, we start from the trees in Figs. 13.6 and 13.7 that we split into two parts to the left and right of their headword. Each of these parts is called a span. Figure 13.8 shows a partition that creates four spans from the two trees. To fit the new structure, we need to modify and complement the operations of the previous section. We use four functions:

1. **attach_right** that creates a right-arc between two spans rooted at w_i, respectively w_j, with its dependents to the right, respectively to the left, and results in a new span from i to j. Figure 13.9 shows this operation.
2. **complete_right** that gathers the right dependents of w_j (Fig. 13.10).
3. **attach_left** and **complete_left** that are the mirrors of the two preceding functions.

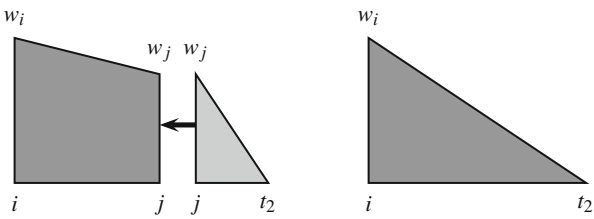

Fig. 13.10 complete_right gathers the dependents to the right of w_j and creates a span from i to t_2 (*gray*)

13.7.3 Implementation

Representing the Spans

Eisner's parser uses a chart, where we fill each cell with spans of increasing length. Their representation is similar to that of the triples from the previous section: each span is bounded by two indices, i and j, and has a head, either i or j. However, the spans need to distinguish whether they have gathered all their dependents to the left, respectively to the right, and thus whether they need to carry out a **complete_left**, respectively **complete_right** operation. We need then one more parameter to mark if a span is available for a complete operation.

Using the notation proposed by McDonald (2006), we model a span by a quadruple (i, j, d, c), where:

- i and j are the indices of the start and the end of the span,
- $d \in \{\rightarrow, \leftarrow\}$ marks the head of the span, either w_i or w_j, and
- $c \in \{0, 1\}$ is a flag that reflects if the span is either incomplete (0) or complete (1), meaning that it has dependents to acquire to the left, in the case of a head to the right, or to the right, in the case of a head to the left.

As examples, Fig. 13.9 shows five spans from left to right before and after an **attach_right** operation. We represent them by the following quadruples:

Before: $(i, t_1, \rightarrow, 1)$, $(t_1 + 1, j, \leftarrow, 1)$, and $(j, t_2, \rightarrow, 1)$,
After: $(i, j, \rightarrow, 0)$ and $(j, t_2, \rightarrow, 1)$.

Parsing Algorithm

As with the CYK parser, Eisner's parser incrementally fills the cells of the chart with spans of increasing length, where each span results from the composition of two subspans. Given a span of length k, this composition can be done in $k - 1$ different ways. To make this analysis possible, we associate a score with each span and span construction. We will then fill the chart with the maximal scoring spans and discard the others.

procedure EISNERPARSER($w_0, w_1, w_2, \ldots, w_n$)
  $C(s,s,d,c) \leftarrow 0, \forall s \in \{0,..,n\}, d \in \{\leftarrow, \rightarrow\}, c \in \{0,1\}$
  for $k \leftarrow 1, n$ **do** ▷ We build spans of increasing length
    for $s \leftarrow 1, n$ **do**
      $t \leftarrow s+k$
      if $t > n$ **then**
        break
      $C(s,t,\leftarrow,0) \leftarrow \max_{s \leq r < t} (C(s,r,\rightarrow,1) + C(r+1,t,\leftarrow,1) + s(t,s))$ ▷ Attach left
      $C(s,t,\rightarrow,0) \leftarrow \max_{s \leq r < t} (C(s,r,\rightarrow,1) + C(r+1,t,\leftarrow,1) + s(s,t))$ ▷ Attach right
      $C(s,t,\leftarrow,1) \leftarrow \max_{s \leq r < t} (C(s,r,\leftarrow,1) + C(r,t,\leftarrow,0))$ ▷ Complete left
      $C(s,t,\rightarrow,1) \leftarrow \max_{s < r \leq t} (C(s,r,\rightarrow,0) + C(r,t,\rightarrow,1))$ ▷ Complete right

Fig. 13.11 Eisner parser (After McDonald (2006))

We denote $C(i, j, d, c)$ the score of span (i, j, d, c), and we define an attachment score $s(h, d)$ between two words, w_h and w_d, where w_h is the head, and w_d, the dependent.

We initialize the chart with spans of length 0 and a score of 0, and we apply a sequence of **attach** and **complete** operations to spans of increasing length from 1 to n. We fill the quadruples with the optimal spans so that each span score corresponds to the maximal sum of the scores of two spans it can be created from plus, in the case of an **attach** operation, the attachment score. The score $C(i, j, d, c)$ of span (i, j, d, c) will then be the sum of all the scores of the individual links involved in its construction. Figure 13.11 shows the complete algorithm after McDonald (2006)'s implementation.

Let us denote W, the sequence of words $w_0, w_1, \ldots, w_n$, and G its dependency graph. The score of G is the sum of the individual scores $s(i, j)$, where $(i, j) \in G$, w_i is the head and w_j, the dependent:

$$s(W, G) = \sum_{(i,j) \in G} s(i, j).$$

This final score is given once the chart is filled by the score of the span ranging from the root w_0 to the end of the sentence w_n: $C(0, n, \rightarrow, 1)$. The dependency tree can then be extracted from the chart using a mirror chart of back pointers that store for each span the two spans it originates from.

13.7.4 Learning Graphs with the Perceptron

As we have seen in the previous section, Eisner's parser requires an attachment score $s(h, d)$ between the words w_h and w_d to carry out an **attach** operation. We define such a score as the dot product of a feature vector $\mathbf{f}(w_h, w_d)$ representing the link and a weight vector **weight**:

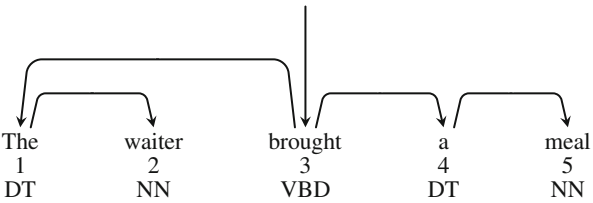

Fig. 13.12 An incorrect dependency analysis of *The waiter brought a meal*

$$s(h,d) = \textbf{weight} \cdot \mathbf{f}(w_h, w_d)$$

and we learn automatically the weight vector from manually parsed corpora using online learning algorithms such as the perceptron that we saw in Sect. 4.7. The final score of the dependency graph G of a sentence W is the sum of all the scores:

$$s(W,G) = \sum_{(h,d)\in G} \textbf{weight} \cdot \mathbf{f}(w_h, w_d)$$

This dot product may seem somewhat abstract. Let us make it concrete with a simple example. We have seen in Sect. 13.4 that D-rules indicate the possibility of a link. Let us use then the part of speech of the head and the part of speech of the dependent as elementary features. To apply the perceptron, we must represent these two parts of speech in the form of a vector of binary digits as introduced in Sect. 4.10. Given that there are about 50 parts of speech in the Penn Treebank, we need to create a vector of 100 dimensions to represent a link. Each vector will have 2 bits that are set to 1; the rest being zeros.

Now let us extract the feature vectors representing the links from graph G in Fig. 13.5, which shows the dependency tree of the sentence *The waiter brought the meal*, and from G' in Fig. 13.12 that shows a wrong analysis of the same sentence. For each link, we build pairs consisting of the parts of speech of the head and the dependent that we convert into binary vector. Table 13.12 shows these vectors with parts of speech limited to the set $\{ROOT, DT, NN, VBD\}$.

In real parsers, such as that of McDonald (2006), there are many more features, including the parts of speech and lexical values of the surrounding words and combinations of them.

The training procedure uses a corpus $\tau = \{(W_t, G_t)\}_{t=1}^{T}$ of T sentences, where W_t is a sentence, and G_t, its associated hand-annotated dependency graph. We initialize the weight vector $\textbf{weight}_{(0)}$ to $\mathbf{0}$ and we apply successive updates using the perceptron. At iteration k, we parse a sentence W_t of the corpus using the weight vector $\textbf{weight}_{(k)}$ to compute the score $s(W_t, G_t)$. The parser returns the graph $\hat{G}_{t(k)}$. The perceptron learns the next weight vector $\textbf{weight}_{(k+1)}$ from differences between G_t and $\hat{G}_{t(k)}$: we compute its update by subtracting $\mathbf{f}(W_t, \hat{G}_{t(k)})$ from $\mathbf{f}(W_t, G_t)$. And we do so until the weight vector converges or we have reached a preset epoch number. Figure 13.13 shows this algorithm modified from McDonald (2006).

Table 13.12 Feature vectors extracted from the dependency graph G shown in Fig. 13.5 and G' in Fig. 13.12. The parts of speech are encoded as Boolean values

		Head POS				Dependent POS			
Feature vectors		ROOT	DT	NN	VBD	ROOT	DT	NN	VBD
		Graph G							
$\mathbf{f}(waiter, The)$	(NN, DT)	0	0	1	0	0	1	0	0
$\mathbf{f}(brought, waiter)$	(VBD, NN)	0	0	0	1	0	0	1	0
$\mathbf{f}(ROOT, brought)$	(ROOT, VBD)	1	0	0	0	0	0	0	1
$\mathbf{f}(meal, the)$	(NN, DT)	0	0	1	0	0	1	0	0
$\mathbf{f}(brought, meal)$	(VBD, NN)	0	0	0	1	0	0	1	0
$\mathbf{f}(W, G) = \sum_{(h,d)\in G} \mathbf{f}(w_h, w_d)$		1	0	2	2	0	2	2	1
		Graph G'							
$\mathbf{f}(brought, The)$	(VBD, NN)	0	0	0	1	0	1	0	0
$\mathbf{f}(The, waiter)$	(DT, NN)	0	1	0	0	0	0	1	0
$\mathbf{f}(ROOT, brought)$	(ROOT, VBD)	1	0	0	0	0	0	0	1
$\mathbf{f}(brought, the)$	(VBD, DT)	0	0	0	1	0	1	0	0
$\mathbf{f}(the, meal)$	(DT, NN)	0	1	0	0	0	0	1	0
$\mathbf{f}(W, G') = \sum_{(h,d)\in G'} \mathbf{f}(w_h, w_d)$		1	2	0	2	0	2	2	1

1: **function** PERCEPTRON(τ)
2: $\quad$ **weight**$_{(0)} \leftarrow \mathbf{0}$
3: $\quad k \leftarrow 0$
4: $\quad$ **for** $n \leftarrow 1, N$ **do** ▷ We select the epoch number
5: $\quad\quad$ **for** $t \leftarrow 1, T$ **do** ▷ We iterate over the corpus
6: $\quad\quad\quad \hat{G}_{t(k)} \leftarrow$ EISNERPARSER(W_t, **weight**$_{(k)}$) ▷ We parse W_t with **weight**$_{(k)}$
7: $\quad\quad\quad$ **if** $\hat{G}_{t(k)} \neq G_t$ **then**
8: $\quad\quad\quad\quad$ **weight**$_{(k+1)} \leftarrow$ **weight**$_{(k)} + \mathbf{f}(W_t, G_t) - \mathbf{f}(W_t, \hat{G}_{t(k)})$ ▷ Perceptron update
9: $\quad\quad\quad\quad k \leftarrow k+1$
10: $\quad$ **return weight**$_{(k)}$

Fig. 13.13 Learning the weight vector **weight** using the perceptron (Modified from McDonald (2006))

13.8 Further Reading

While most research in English has been done using the constituency formalism – and many computational linguists still use it – dependency inspires much of the present work. Early implementations of dependency theories include Link Grammar (Sleator and Temperley 1993) and the Functional Dependency Grammar (Järvinen and Tapanainen 1997) that uses constraint rules and produces a dependency structure where links are annotated with functions. Covington (1990) described an algorithm that could parse discontinuous constituents. Constant (1991), El Guedj (1996), and Vergne (1998) provide accounts in French; Hellwig (1980, 1986) was among the pioneers in German.

Some authors reformulated parsing a constraint satisfaction problem (CSP), sometimes combining it with a chart. Constraint handling rules (CHR) is a simple, yet powerful language to define constraints (Frühwirth 1998). Constraint handling rules are available in some Prologs, notably SWI Prolog.

In 2006 and 2007, the Conference on Computational Natural Language Learning (CoNLL) organized its shared task on multilingual dependency parsing (Buchholz and Marsi 2006; Nivre et al. 2007). The conference site provides background literature, data sets, and an evaluation scheme (http://www.cnts.ua.ac.be/conll/). As a result, two main classes of parsing methods emerged from these shared tasks: the first one being transition-based, as is Nivre's parser, and the second one based on Eisner's parser. They still dominate the world of dependency parsing today. See McDonald (2006) and Kübler et al. (2009) for details as well as well as for a third technique based on maximum spanning trees.

The performance of a parser depends in a large measure on the feature set it uses. In this chapter, we reviewed relatively simple sets. Table 11.12 shows a baseline set for transition-based parsing that needs to be experimentally tuned for each language to analyze. While feature sets can be created manually, Nilsson and Nugues (2010) describe an automatic procedure to discover features for transition-based parsers. In Eisner's parser, we considered first-order features involving a single arc between a head and a dependent. It is possible to extend the set to second-order features representing two links, between a head and two adjacent dependents or between a head, a dependent, and a dependent of the dependent, as well as higher-order features. Eisner's parser needs then to be extended to accommodate these features. See Carreras (2007) for a description. Readers interested in building a high-performance parser should refer to the original papers, from CoNLL for instance, that describe the complete feature sets.

Finally, we trained Eisner's parser using the perceptron. This online learning technique can also be applied to transition-based parsing combined with beam search; the features are then extracted from the parser states and transitions (Johansson and Nugues 2007b). Although, this combination initially yielded results inferior to those obtained with local classifiers, they are now on a par with the best published performances (Zhang and Nivre 2011).

Exercises

13.1. Improve Nivre's parser with D-rules. Use the suggestions proposed in Sect. 13.4.1.

13.2. Add features to Nivre's parser and evaluate their contribution. Download an annotated corpus from the CoNLL 2006 web site and use the evaluation script to measure the attachment score. You can use features proposed in Table 13.6.

13.3. Implement a function classifier to Nivre's parser. Download an annotated corpus from the CoNLL 2006 web site and use the evaluation script to measure the labelled attachment score. You can use features proposed in Table 13.6.

13.4. Rewrite Covington's parser to parse an annotated corpus using either the projective or nonprojective version. Download a corpus from the CoNLL 2006 web site, for example, and extract features. You can use features proposed in Table 13.6.

13.5. Extend Covington's parser (projective or nonprojective) with classifiers. Train classifiers from the features you extracted in Exercise 13.4. You can use either decision trees or support vector machines. Apply your parser on a corpus from the CoNLL 2006 web site and use the evaluation script to measure the labeled attachment score.

13.6. Implement Eisner's parser.