quadruple $\langle S, I, T, A\rangle$, where S is the stack of past tokens, I is the list of (remaining) input tokens, T is a stack of temporary tokens and A is the arc relation for the dependency graph.

Given an input string W, the parser is initialized to $\langle (), W, (), ()\rangle$, and terminates when it reaches a configuration $\langle S, (), (), A\rangle$.

The three basic parsing rule schemas are as follows:

$$\textit{Shift}\quad \frac{\langle S, n|I, T, A\rangle}{\langle n|S, I, T, A\rangle}$$

$$\textit{Right}_d\quad \frac{\langle s|S, n|I, T, A\rangle}{\langle S, n|I, T, A\cup\{(s, d, n)\}\rangle}$$

$$\textit{Left}_d\quad \frac{\langle s|S, n|I, T, A\rangle}{\langle S, s|I, T, A\cup\{(n, d, s)\}\rangle}$$

The schemas for the *Left* and *Right* rules are instantiated for each dependency type $d \in D$, for a total of $2|D| + 1$ rules. These rules perform both attachment and labeling.

At each step the parser uses classifiers trained on a treebank corpus in order to predict which action to perform and which dependency label to assign given the current configuration.

4 Non-Projective Relations

For handling non-projective relations, Nivre and Nilsson (2005) suggested applying a preprocessing step to a dependency parser, which consists in lifting non-projective arcs to their head repeatedly, until the tree becomes pseudo-projective. A post-processing step is then required to restore the arcs to the proper heads.

In DeSR non-projective dependencies are handled in a single step by means of the following additional parsing rules, slightly different from those in (Attardi, 2006):

$$\textit{Right2}_d\quad \frac{\langle s_1|s_2|S, n|I, T, A\rangle}{\langle S, s_1|n|I, T, A\cup\{(s_2, d, n)\}\rangle}$$

$$\textit{Left2}_d\quad \frac{\langle s_1|s_2|S, n|I, T, A\rangle}{\langle s_2|S, s_1|I, T, A\cup\{(n, d, s_2)\}\rangle}$$

$$\textit{Right3}_d\quad \frac{\langle s_1|s_2|s_3|S, n|I, T, A\rangle}{\langle S, s_1|s_2|n|I, T, A\cup\{(s_3, d, n)\}\rangle}$$

$$\textit{Left3}_d\quad \frac{\langle s_1|s_2|s_3|S, n|I, T, A\rangle}{\langle s_2|s_3|S, s_1|I, T, A\cup\{(n, d, s_3)\}\rangle}$$

$$\textit{Extract}\quad \frac{\langle s_1|s_2|S, n|I, T, A\rangle}{\langle n|s_1|S, I, s_2|T, A\rangle}$$

$$\textit{Insert}\quad \frac{\langle S, I, s_1|T, A\rangle}{\langle s_1|S, I, T, A\rangle}$$

Left2, *Right2* are similar to *Left* and *Right*, except that they create links crossing one intermediate node, while *Left3* and *Right3* cross two intermediate nodes. Notice that the *RightX* actions put back on the input the intervening tokens, allowing the parser to complete the linking of tokens whose processing had been delayed. *Extract*/*Insert* generalize the previous rules by moving one token to the stack T and reinserting the top of T into S.

5 Perceptron Learning and 2nd-Order Feature Maps

The software architecture of the DeSR parser is modular. Several learning algorithms are available, including SVM, Maximum Entropy, Memory-Based Learning, Logistic Regression and a few variants of the perceptron algorithm.

We obtained the best accuracy with a multiclass averaged perceptron classifier based on the ultraconservative formulation of Crammer and Singer (2003) with uniform negative updates. The classifier function is:

$$F(x) = \arg\max_k \{\alpha_k \cdot x\}$$

where each parsing action k is associated with a weight vector α_k. To regularize the model the final weight vectors are computed as the average of all weight vectors posited during training. The number of learning iterations over the training data, which is the only adjustable parameter of the algorithm, was determined by cross-validation.

In order to overcome the limitations of a linear perceptron, we introduce a feature map Φ: $\mathbb{R}^d \rightarrow \mathbb{R}^{d(d+1)/2}$ that maps a feature vector $\boldsymbol{x}$ into a higher dimensional feature space consisting of all unordered feature pairs:

$$\Phi(\boldsymbol{x}) = \langle x_i x_j \mid i = 1, \ldots, d, j = i, \ldots, d\rangle$$

In other words we expand the original representation in the input space with a feature map that generates all second-order feature combinations from each observation. We call this the 2nd-order model, where the inner products are computed as $\alpha_k \cdot \Phi(\boldsymbol{x})$, with α_k a vector of dimension $d(d+1)/2$. Applying a linear perceptron to this feature space corresponds to simulating a polynomial kernel of degree two.

A polynomial kernel of degree two for SVM was also used by Yamada and Matsumoto (2003). However, training SVMs on large data sets like those arising from a big training corpus was too