TRANSITION-BASED PARSING

JOAKIM NIVRE

Contents

1. Transition Systems 1
2. Greedy Classifier-Based Parsing 4
3. Beam Search and Structured Learning 6
4. Non-Projective Parsing 7
References 10
Appendix A. Transition Sequence for Arc-Standard System 11
Appendix B. Transition Sequence for Arc-Eager System 12

Transition-based models for dependency parsing use a factorization defined in terms of a transition system, or abstract state machine. In this lecture, I will introduce the arc-eager and arc-standard transition systems for dependency parsing (§1) and discuss two different approaches to learning and decoding with these models: greedy classifier-based parsing (§2) and beam search and structured learning (§3). Finally, I will discuss different techniques for non-projective transition-based parsing (§4).

1. Transition Systems

A transition system for dependency parsing is a quadruple $S = (C, T, c_s, C_t)$, where

1. $C$ is a set of configurations,
2. $T$ is a set of transitions, each of which is a (partial) function $t : C \rightarrow C$,
3. $c_s$ is an initialization function, mapping a sentence $x$ to its initial configuration $c_s(x)$,
4. $C_t \subseteq C$ is a set of terminal configurations.

A configuration for a sentence $x$ is a triple $c = (\Sigma, B, A)$, where $\Sigma$ is a list of nodes in $V_x$, known as the stack, $B$ is a list of nodes in $V_x$, known as the buffer, and $A$ is a set of dependency arcs in $V_x \times L \times V_x$ (for some set $L$ of dependency labels). A transition sequence for a sentence $x$ in transition system $S = (C, T, c_s, C_t)$ is a sequence $C_{0,m} = (c_0, c_1, \ldots, c_m)$ of configurations, such that

1. $c_0 = c_s(x)$,
2. $c_m \in C_t$,
3. for every $i (1 \leq i \leq m), c_i = t(c_{i-1})$ for some $t \in T$.

The parse assigned to $x$ by $C_{0,m}$ is the dependency graph $G_{c_m} = (V_x, A_{c_m})$, where $A_{c_m}$ is the set of dependency arcs in $c_m$. More generally, the dependency graph associated with any configuration $c_i$ for $x$ is $G_{c_i} = (V_x, A_{c_m})$.

Date: 2013-03-15.

We restrict ourselves here to the class of stack-based transition systems. For a more general characterisation of transition systems for dependency parsing, see Nivre (2008).
Initialization: \( c_s(x = x_1, \ldots, x_n) = ([0], [1, \ldots, n], \emptyset) \)

Terminal: \( C_t = \{c \in C | c = ([0], [], A)\} \)

Transitions:
1. \((\sigma, [i|\beta], A) \Rightarrow ([\sigma|i], \beta, A)\) (Shift)
2. \(([\sigma|i|j], B, A) \Rightarrow ([\sigma|j], B, A \cup \{(j, l, i)\})\) (Left-Arc\(_l\))
3. \(([\sigma|i|j], B, A) \Rightarrow ([\sigma|i], B, A \cup \{(i, l, j)\})\) (Right-Arc\(_l\))

1 Permitted only if \(i \neq 0\).

Figure 1. The arc-standard stack-based transition system for projective dependency parsing. The notation \([\sigma|i]\) (for the stack) denotes a right-headed list with head \(i\) and tail \(\sigma\); the notation \([j|\beta]\) (for the buffer) denotes a left-headed list with head \(j\) and tail \(\beta\).

The first transition system we will consider is the so-called arc-standard system for projective dependency parsing, which is specified in Figure 1. In this system, the initial configuration for a sentence \(x\) is a configuration where the stack contains the artificial root node (0), the buffer contains all the nodes corresponding to the real words of the sentence (1, \ldots, \(n\)), and where the arc set is empty (because no structure has been built yet). A terminal configuration is any configuration where the stack again contains only the artificial root node and where the buffer is empty (meaning that we have consumed all the input). Whatever arcs have been accumulated in the arc set \(A\) at that point defines the parse for that transition sequence. Finally, there are three types of transitions for getting from one configuration to the next:

1. **Shift** removes the first node \(i\) in the buffer and pushes it on top of the stack.
2. **Left-Arc\(_l\)** (for any dependency label \(l\)) adds a dependency arc \((j, l, i)\) to \(A\), where \(j\) is the first and \(i\) is the second node from the top of the stack. In addition, it pops the stack twice and then pushes \(j\) back on to the stack so that only the head node \(j\) remains on the stack after the transition. Left-Arc\(_l\) is only permitted if \(i \neq 0\), because we do not want to add arcs going into the artificial root node.
3. **Right-Arc\(_l\)** (for any dependency label \(l\)) adds a dependency arc \((i, l, j)\) to \(A\), where \(i\) is the second and \(j\) is the first node from the top of the stack. In addition, it pops the stack once so that only the head node \(i\) remains on the stack after the transition.

An illustration of how these transitions work can be found in Appendix A, which shows the complete transition sequence needed to parse one of our standard English example sentences.

The following theoretical results can be shown to hold for the arc-eager system:
- Every transition sequence outputs a projective dependency tree (soundness).
- Every projective dependency tree is output by some transition sequence (completeness).
- There are exactly \(2n\) transitions in a transition sequence for a sentence of length \(n\).

The arc-standard system considered so far builds a dependency tree strictly bottom-up, meaning that a dependency arc can only be added between two nodes if the dependent node has already found all its dependents. As a consequence, it is often necessary to postpone the attachment of right dependents (as shown in the example in Appendix A), which introduces a tricky kind of nondeterminism. This problem is avoided in the alternative arc-eager system, which always adds...
Initialization: \[ c_s(x = x_1, \ldots, x_n) = ([0], [1, \ldots, n], \emptyset) \]

Terminal: \[ C_t = \{ c \in C | c = (\Sigma, [], A) \} \]

Transitions:

1. \[ (\sigma, [i|\beta], A) \Rightarrow ([\sigma|i], \beta, A) \] (Shift)
2. \[ ([\sigma|i], [j|\beta], A) \Rightarrow ([\sigma][j\beta], A \cup \{(j, l, i)\}) \] (Left-Arc\_l)
3. \[ ([\sigma|i], [j|\beta], A) \Rightarrow ([\sigma][j\beta], A \cup \{(i, l, j)\}) \] (Right-Arc\_l)
4. \[ ([\sigma|i], B, A) \Rightarrow (\sigma, B, A) \] (Reduce)

\[ \text{1 Permitted only if } i \neq 0 \text{ and there are no } k, l' \text{ such that } (k, l', i) \in A. \]
\[ \text{2 Permitted only if there are } k, l' \text{ such that } (k, l', i) \in A. \]

Figure 2. The arc-standard stack-based transition system for projective dependency parsing. The notation \([\sigma|i]\) (for the stack) denotes a right-headed list with head \(i\) and tail \(\sigma\); the notation \([j|\beta]\) (for the buffer) denotes a left-headed list with head \(j\) and tail \(\beta\).

an arc at the earliest possible opportunity and which therefore builds parts of the tree top-down instead of bottom-up.

The arc-eager system, described in Figure 2, has the same initialization function as the arc-standard system, but the set of terminal configurations is different because the arc-eager system terminates as soon as the buffer is empty (regardless of the state of the stack). Moreover, in order to accommodate the arc-eager parsing strategy, we need four transitions instead of three:

1. **Shift** removes the first node \(i\) in the buffer and pushes it on top of the stack just as in the arc-standard system.
2. **Left-Arc\_l** (for any dependency label \(l\)) adds a dependency arc \((j, l, i)\) to \(A\), where \(j\) is the first node in the buffer and \(i\) is the node on top of the stack. In addition, it pops the stack so that only \(j\) remains after the transition (still at the head of the buffer). This is analogous to **Left-Arc\_l** in the arc-standard system except that the new arc combines one node from the stack and one node from the buffer instead of the two top nodes on the stack. As before, we do not allow this transition if the leftmost node \(i\) is the artificial root node, but in addition we have to check that \(i\) has not already been assigned a head in the dependency structure, a possibility that could never arise in the arc-standard system.
3. **Right-Arc\_l** (for any dependency label \(l\)) adds a dependency arc \((i, l, j)\) to \(A\), where \(i\) is the node on top of the stack and \(j\) is the first node in the buffer. In addition, we push the node \(j\) on to the stack and do not remove any of the nodes. This is different from **Right-Arc\_l** in the arc-standard system not only because the new arc combines one node from the stack and one node from the buffer but also because we retain both nodes in the new configuration. The reason we do this is to allow the dependent \(j\) to later pick up right-dependents of its own, which is a necessary consequence of the arc-eager strategy.
4. **Reduce** removes the node on top of the stack subject to the condition that it has already been assigned a head in the dependency structure. This transition is needed to remove a node previously pushed onto the stack in a **Right-Arc\_l** transition after it has found all its own right-dependents.
The different parsing order enforced by this system is exemplified in Appendix B, which shows the complete transition sequence for the same example sentence as in Appendix A. Theoretical results for the arc-eager system are similar to but slightly different from those holding for the arc-standard system:

- Every transition sequence outputs a projective dependency forest (soundness).
- Every projective dependency tree is output by some transition sequence (completeness).
- There are at most \(2n\) transitions in a transition sequence for a sentence of length \(n\).

In particular, the arc-eager system has a weaker soundness result than the arc-standard system in that it does not guarantee the output to be a dependency tree, only a sequence of (unconnected) trees (a forest). In the best case, this is a sequence of length 1, meaning that the output is in fact a dependency tree. In the worst case, it is a sequence of length \(n\), meaning that every node is its own tree (the output resulting from \(n\) consecutive Shift transitions). However, any output graph can be trivially converted to a dependency tree by adding arcs from the artificial root node to all the other nodes that do not have an incoming arc.

2. Greedy Classifier-Based Parsing

In a transition-based model, dependency trees are represented by the transition sequences that derive them. However, the transition system itself is nondeterministic and does not impose any preference among possible transition sequences for a given sentence. In order to turn this into a parser, we must therefore add a model for scoring transition sequences and a method for finding the highest-scoring sequence under our current model. In the greedy classifier-based approach to transition-based parsing (also known as deterministic dependency parsing), we assume that the problem of scoring transition sequences can be reduced to the simpler problem of scoring single transitions from one configuration to the next and that a globally optimal sequence can be found by making a sequence of locally optimal transitions in a completely greedy fashion. Under this assumption, transition-based dependency parsing can be performed using the simple algorithm in Figure 3.

After initializing the parser to the initial configuration, the algorithm consists of a single loop that just repeatedly applies the highest-scoring transition out of the current configuration until a terminal configuration is reached. It is easy to see that, as long as the computation of transition scores and the application of the highest-scoring transition to the current configuration can be performed in some constant time, the worst-case complexity of parsing is linear in the length of the sentence. This follows from the linear bound on transition sequence length observed for both the arc-standard and the arc-eager systems in §1. Linear complexity is arguably as good as it gets for syntactic parsing, and greedy transition-based parsers are therefore optimally efficient. In addition, since only the highest-scoring transition is considered for each configuration, the scoring model can in fact be reduced to a simple (unstructured) classifier mapping each configuration to its optimal transition. This is why we refer to this approach as greedy classifier-based parsing.
Table 1. Feature templates for the arc-eager transition system. The symbols $\Sigma_i$ and $B_i$ refer to the $i$th node in the stack and buffer, respectively, with indexing starting at 0. The functors ldep, rdep and hd return the leftmost dependent, rightmost dependent and syntactic head, respectively, of a node with respect to the partially built dependency graph. Attributes: word = word form; pos = part-of-speech tag; lab = dependency label. Note that we need one complete set of binarized features for each possible transition $t$ out of a configuration $c$.

In actual practice, a wide range of different models have been used to score transitions, but the most common approach is (not surprisingly) a linear model:

$$
\text{Score}(c, t) = \sum_{k=1}^{K} f_k(c, t) \cdot w_k
$$

Typical feature templates used in a model for the arc-eager transition system can be found in Table 1. Comparing these feature templates to those considered for graph-based models in the previous two lectures, the most obvious difference is that features are now defined relative to the stack and buffer rather than with respect to graph factors. But it is also worth noting that the transition-based parser can make use of features defined over the partially built dependency graph without any penalty in terms of complexity. For example, the last of the trigram features corresponds to a second-order factor with siblings on opposite side of the head.

Learning the weights of the transition scoring model can be done using any of a multitude of machine learning algorithms for simple unstructured classification, including the simple perceptron, logistic regression and multiclass support vector machines. Given a treebank, training data for such classifiers can be generated by, for every sentence $x'$ with dependency tree $y'$ in the treebank, finding a transition sequence $c_0, \ldots, c_m$ that outputs $y'$ for $x'$, and creating one training instance $(c_i, t_i)$ for each transition $t_i(c_i) = c_{i+1}$ in that sequence. One drawback of this approach is that the scoring model is only trained on configurations resulting from a sequence of optimal transitions, which may lead to error propagation when incorrect transitions are predicted at parsing time.

Greedy classifier-based dependency parsing was pioneered by Yamada & Matsumoto (2003) and Nivre (2003) among others and was shown to give state-of-the-art accuracy for a number of languages (Buchholz & Marsi, 2006; Nivre et al., 2007). As shown by McDonald & Nivre (2007),
Figure 4. Algorithm for transition-based dependency parsing with beam search.

however, the greedy classifier-based approach is sensitive to search errors and subsequent error propagation and these parsers cannot quite compete with contemporary higher-order graph-based models in terms of accuracy. Nevertheless, with their linear time complexity, they remain the most efficient parsers available and are commonly used for web-scale parsing and other applications requiring very high throughput.

3. BEAM SEARCH AND STRUCTURED LEARNING

One way of mitigating the effect of search errors and error propagation in transition-based parsing is to use beam search in combination with structured learning. First of all, this requires that we switch to a model that explicitly scores complete transition sequences:

\[ \text{Score}(c_0, m, x) = \sum_{i=0}^{m-1} \text{Score}(c_i, t_i) \quad \text{[where } t_i(c_i) = c_{i+1} \text{]} \]

We then give up greedy search and instead use beam search, a heuristic search algorithm that explores the \( q \) most promising hypotheses at each step, \( q \) being referred to as the beam size. Greedy search can be thought of as beam search with a beam size of 1, and by increasing the beam size we can explore a larger part of the search space. As long as we use a constant beam size, parsing time only increases by a constant factor and the worst-case complexity remains linear. Figure 4 gives pseudo-code for beam search parsing with a transition-based model.

The overall algorithm is very similar to the greedy algorithm, but instead of initializing the parser to a single configuration, we initialize the beam to a set containing the single start configuration with a score of 0.0. As long as we have at least one nonterminal configuration in the beam, we apply every possible transition \( t \) to every configuration \( c \) in the beam and store the resulting configuration with a score obtained by adding the old score \( s \) of \( c \) to the score \( \text{Score}(c, t) \) of the new transition. After expanding all the hypotheses and storing them in a temporary set (NewBeam), we keep only the \( q \) highest-scoring ones in the new beam. Once we terminate, we return the highest-scoring configuration in the beam (or, rather, the dependency graph defined by this configuration).

We still assume that transition scores are based on a linear model \( \sum_{k=1}^{K} f_k(c, t) \cdot w_k \), and in principle we could learn the weights for this model using the same approach as for the classifier-based parsers described in §2. However, research has shown that better results can be obtained if we use a structured learning algorithm, such as the structured perceptron, because this allows us to learn weights that perform well also after non-optimal transitions have been applied. Figure 5 outlines the structured perceptron algorithm as it applies to transition-based parsing.

This algorithm is very similar to the structured perceptron for the arc-factored graph-based model, considered in a previous lecture, and differs only in three respects. First of all, dependency
Training data: \( \mathcal{T} = \{ (x^i, c^i_{0,m}) \}_{i=1}^{|\mathcal{T}|} \)

1. \( w \leftarrow 0 \)
2. for \( n : 1..N \)
3.     for \( i : 1..|\mathcal{T}| \)
4.         \( c^*_0,m \leftarrow \text{Parse}(x^i, w) \)
5.         if \( c^*_0,m \neq c_i^0,m \)
6.             \( w \leftarrow \text{Update}(w, c^*_0,m, c_i^0,m) \)
7. return \( w \)

**Figure 5.** The structured perceptron algorithm for transition-based dependency parsing.

trees are represented indirectly by their transition sequences (both in the training data and in the parser output). Secondly, the weight updates iterate over the transitions of a transition sequence (instead of over the arcs of a dependency tree). Finally, the call to the parsing algorithm (\( \text{Parse}(x^i, w) \)) uses beam search instead of a dynamic programming algorithm.

The use of beam search and structured learning for transition-based dependency parsing was pioneered by Zhang & Clark (2008), who also found that accuracy could be further improved by using so-called early updates during training, following Collins & Roark (2004). With early updates, parsing is interrupted as soon as the gold standard transition sequence falls out of the beam, and weights are then updated with the partial transition sequences obtained up to this point. Later research has shown that transition-based parsers with beam search and structured learning can accommodate much richer feature models than greedy parsers and thereby attain the same level of parsing accuracy as higher-order graph-based models (Zhang & Nivre, 2011). Research has also demonstrated that they are less sensitive to search errors and subsequent error propagation (Zhang & Nivre, 2012).

### 4. Non-Projective Parsing

The two transition systems introduced in §1 can only derive projective dependency trees. Attardi (2006) proposed an extension to the arc-standard system with additional transitions for adding arcs between nodes that are not adjacent on the stack, thereby creating non-projective dependencies, as shown in Figure 6. Although this system cannot handle arbitrary non-projective trees, it is sufficient to handle nearly all non-projective dependencies found in natural language data while maintaining the linear bound on transition sequence length. It is possible to generalize this idea to non-projective dependencies of arbitrary length, using an open list instead of a stack to store partially processed nodes, but then the worst-case complexity becomes quadratic instead of linear (Nivre, 2007, 2008).

A tree is non-projective only with respect to a particular word order, and it is always possible to reorder the words of a sentence to make the tree projective. This is the idea underlying the notion of online reordering, where instead of adding arcs between nodes that are not adjacent
on the stack we allow the parser to reorder the nodes so that nodes that should be linked by an arc are always adjacent. In this way, we can create a transition system that can handle arbitrary non-projective dependency trees with only four transitions, as shown in Figure 7, whereas an Attardi-style extension would require an infinite number of transitions to get full coverage. The new system, first proposed by Nivre (2009), uses a transition called Swap to reorder nodes by moving the second node on the stack back to the buffer (so that the node on top of the stack is the same before and after the transition). The Swap transition is not allowed if the node $i$ to be moved back is the artificial root node or if the two nodes $i$ and $j$ have already been swapped. It is easy to show that the Swap transition allows us to make arbitrary permutations of the original word order (constituting a very simple sorting algorithm), which means that we can always find a reordering that makes the target tree projective. This technique has been shown to give good accuracy for non-projective dependency parsing especially in combination with beam search and structured learning (Bohnet & Nivre, 2012). The worst-case complexity is $O(n^2)$ – because we could in theory reverse the entire sentence, which would take $O(n^2)$ time, but since non-projective trees found in natural language tend to be very nearly projective, the average running time turns out to be linear (Nivre, 2009).

One final technique for handling non-projective trees in transition-based parsing is what is known as pseudo-projective parsing (Nivre & Nilsson, 2005). The basic idea here is that we can transform any non-projective tree to a projective tree by reattaching words higher in the tree (in the worst case at the artificial root). The pseudo-projective transform of a non-projective tree is the tree where each non-projective arc $(i, l, j)$ in the original tree is replaced by $(k, l, j)$ such that $k$ is the closest ancestor of $i$ that does not violate the projectivity constraint. A non-projective tree and its pseudo-projective transform is shown in Figure 8. In pseudo-projective parsing, we apply the pseudo-projective transform to every dependency tree in the training set and then train a projective dependency parsers as usual (typically using the arc-standard or arc-eager transition.

| Initialization: | $c_a(x = x_1, \ldots, x_n) = ([0], [1, \ldots, n], \emptyset)$ |
| Terminal: | $C_t = \{c \in C | c = ([0], [], A)\}$ |
| Transitions: | $(\sigma, [i|\beta], A) \Rightarrow ([\sigma|i], \beta, A)$ (Shift) |
| | $([\sigma|i], B, A) \Rightarrow ([\sigma|j], B, A \cup \{(j, l, i)\})$ (Left-Arc1) |
| | $([\sigma|i], B, A) \Rightarrow ([\sigma|i], B, A \cup \{(i, l, j)\})$ (Right-Arc1) |
| | $([\sigma|i], B, A) \Rightarrow ([\sigma|k], B, A \cup \{(i, l, j)\})$ (Right-Arc2) |
| | $([\sigma|i], B, A) \Rightarrow ([\sigma|k_1], B, A \cup \{(j, l, i)\})$ (Left-Arc2) |
| | $([\sigma|i], B, A) \Rightarrow ([\sigma|k_1], B, A \cup \{(i, l, j)\})$ (Right-Arc3) |

1 Permitted only if $i \neq 0$. 
Initialization: \( c_s(x = x_1, \ldots, x_n) = ([0], [1, \ldots, n], \emptyset) \)

Terminal: \( C_t = \{ c \in C| c = ([0], [\ ], A) \} \)

Transitions:

\[
\begin{align*}
(\sigma, [i|\beta], A) &\Rightarrow ([\sigma|i], \beta, A) \quad \text{(Shift)} \\
([\sigma|i]|j, B, A) &\Rightarrow ([\sigma|j], B \cup \{(j, l, i)\}) \quad \text{(Left-Arc)}^1 \\
([\sigma|i]|j, B, A) &\Rightarrow ([\sigma|i], B \cup \{(i, l, j)\}) \quad \text{(Right-Arc)} \\
([\sigma|i]|j, \beta, A) &\Rightarrow ([\sigma|j], [i|\beta], A) \quad \text{(Swap)}^2
\end{align*}
\]

1 Permitted only if \( i \neq 0 \).
2 Permitted only if \( i \neq 0 \) and \( i < j \).

Figure 7. Non-projective parsing with online reordering (swap) (Nivre, 2009). The notation \([\sigma|i]\) (for the stack) denotes a right-headed list with head \( i \) and tail \( \sigma \); the notation \([j|\beta]\) (for the buffer) denotes a left-headed list with head \( j \) and tail \( \beta \).

Figure 8. A non-projective dependency tree (top) and its pseudo-projective transform (bottom).

system), thus learning to produce trees that are close projective approximations of the correct non-projective trees. In addition, we can use arc labels to encode information about the true (non-projective) head and apply heuristic post-processing to try to recover the underlying non-projective tree (by essentially computing the inverse of the pseudo-projective transform). This approach has the advantage that it keeps the parsing process strictly projective, thus preserving linear time complexity, but nevertheless allows us to recover a subset of the non-projective dependencies in post-processing.
References


Transition Configuration

\[ c_s(x) = ( [0], [1, \ldots, 9], \emptyset ) \]

Shift,\( \Rightarrow \)
\[ ( [0], [1, \ldots, 9], \emptyset ) \]

Shift,\( \Rightarrow \)
\[ ( [0, 1, 2], [3, \ldots, 9], \emptyset ) \]

Left-Arc,\( \Rightarrow \)
\[ ( [0, 2], [3, \ldots, 9], A_1 = \{ (2, \text{ATT}, 1) \} ) \]

Shift,\( \Rightarrow \)
\[ ( [0, 2, 3], [4, \ldots, 9], A_1 ) \]

Left-Arc,\( \Rightarrow \)
\[ ( [0, 3], [4, \ldots, 9], A_2 = A_1 \cup \{ (3, \text{SBJ}, 2) \} ) \]

Shift,\( \Rightarrow \)
\[ ( [0, 3, 4], [5, \ldots, 9], A_2 ) \]

Shift,\( \Rightarrow \)
\[ ( [0, \ldots, 5], [6, \ldots, 9], A_2 ) \]

Left-Arc,\( \Rightarrow \)
\[ ( [0, 3, 5], [6, \ldots, 9], A_3 = A_2 \cup \{ (5, \text{ATT}, 4) \} ) \]

Shift,\( \Rightarrow \)
\[ ( [0, \ldots, 6], [7, 8, 9], A_3 ) \]

Shift,\( \Rightarrow \)
\[ ( [0, \ldots, 7], [8, 9], A_3 ) \]

Shift,\( \Rightarrow \)
\[ ( [0, \ldots, 8], [9], A_3 ) \]

Left-Arc,\( \Rightarrow \)
\[ ( [0, \ldots, 8], [9], A_4 = A_2 \cup \{ (8, \text{ATT}, 7) \} ) \]

Right-Arc,\( \Rightarrow \)
\[ ( [0, \ldots, 6], [9], A_5 = A_4 \cup \{ (6, \text{PC}, 8) \} ) \]

Right-Arc,\( \Rightarrow \)
\[ ( [0, 3, 5], [9], A_6 = A_5 \cup \{ (5, \text{ATT}, 6) \} ) \]

Right-Arc,\( \Rightarrow \)
\[ ( [0, 3], [9], A_7 = A_6 \cup \{ (3, \text{OBJ}, 5) \} ) \]

Shift,\( \Rightarrow \)
\[ ( [0, 3, 9], [\cdot], A_7 ) \]

Right-Arc,\( \Rightarrow \)
\[ ( [0, 3], [\cdot], A_8 = A_7 \cup \{ (3, \text{PU}, 9) \} ) \]

Right-Arc,\( \Rightarrow \)
\[ ( [0, \cdot], A_9 = A_8 \cup \{ (0, \text{ROOT}, 3) \} ) \]
### Appendix B. Transition Sequence for Arc-Eager System

**Transition Configuration**

\[
\begin{align*}
\text{Shift} &\Rightarrow (0, 1, [2, \ldots, 9], \emptyset) \\
\text{Left-Arc}_{\text{att}} &\Rightarrow (0, 2, [3, \ldots, 9], A_1 = \{(2, \text{att}, 1)\}) \\
\text{Shift} &\Rightarrow (0, 3, [5, \ldots, 9], A_3) \\
\text{Left-Arc}_{\text{root}} &\Rightarrow (0, 3, [5, \ldots, 9], A_4 = A_3 \cup \{(3, \text{root}, 2)\}) \\
\text{Right-Arc}_{\text{obj}} &\Rightarrow (0, 3, [6, \ldots, 9], A_5 = A_4 \cup \{(3, \text{obj}, 5)\}) \\
\text{Right-Arc}_{\text{att}} &\Rightarrow (0, \ldots, 6, [7, 8, 9], A_6 = A_5 \cup \{(5, \text{att}, 6)\}) \\
\text{Shift} &\Rightarrow (0, \ldots, 7, [8, 9], A_6) \\
\text{Left-Arc}_{\text{att}} &\Rightarrow (0, \ldots, 6, [8, 9], A_7 = A_6 \cup \{(8, \text{att}, 7)\}) \\
\text{Right-Arc}_{\text{pc}} &\Rightarrow (0, \ldots, 8, [9], A_8 = A_7 \cup \{(6, \text{pc}, 8)\}) \\
\text{Reduce} &\Rightarrow (0, \ldots, 6, [9], A_8) \\
\text{Reduce} &\Rightarrow (0, 3, [9], A_8) \\
\text{Reduce} &\Rightarrow (0, 3, [9], A_8) \\
\text{Right-Arc}_{\text{pu}} &\Rightarrow (0, 3, 9, [9], A_9 = A_8 \cup \{(3, \text{pu}, 9)\})
\end{align*}
\]

<table>
<thead>
<tr>
<th>Transition</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shift</td>
<td>(0, 1, [2, \ldots, 9], \emptyset)</td>
</tr>
<tr>
<td>Left-Arc_att</td>
<td>(0, 2, [3, \ldots, 9], A_1 = {(2, att, 1)})</td>
</tr>
<tr>
<td>Shift</td>
<td>(0, 3, [5, \ldots, 9], A_3)</td>
</tr>
<tr>
<td>Left-Arc_root</td>
<td>(0, 3, [5, \ldots, 9], A_4 = A_3 \cup {(3, root, 2)})</td>
</tr>
<tr>
<td>Right-Arc_obj</td>
<td>(0, 3, [6, \ldots, 9], A_5 = A_4 \cup {(3, obj, 5)})</td>
</tr>
<tr>
<td>Right-Arc_att</td>
<td>(0, \ldots, 6, [7, 8, 9], A_6 = A_5 \cup {(5, att, 6)})</td>
</tr>
<tr>
<td>Shift</td>
<td>(0, \ldots, 7, [8, 9], A_6)</td>
</tr>
<tr>
<td>Left-Arc_att</td>
<td>(0, \ldots, 6, [8, 9], A_7 = A_6 \cup {(8, att, 7)})</td>
</tr>
<tr>
<td>Right-Arc_pc</td>
<td>(0, \ldots, 8, [9], A_8 = A_7 \cup {(6, pc, 8)})</td>
</tr>
<tr>
<td>Reduce</td>
<td>(0, \ldots, 6, [9], A_8)</td>
</tr>
<tr>
<td>Reduce</td>
<td>(0, 3, [9], A_8)</td>
</tr>
<tr>
<td>Reduce</td>
<td>(0, 3, [9], A_8)</td>
</tr>
<tr>
<td>Right-Arc_pu</td>
<td>(0, 3, 9, [9], A_9 = A_8 \cup {(3, pu, 9)})</td>
</tr>
</tbody>
</table>

- **Diagram:**
  - PRED \(\rightarrow\) ATT \(\rightarrow\) OBJ
  - ROOT \(\rightarrow\) SBJ \(\rightarrow\) ATT
  - ATT \(\rightarrow\) ATT \(\rightarrow\) ATT
  - ATT \(\rightarrow\) ATT

- **Text:**

  Economic news had little effect on financial markets.