Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

BACHELORTHESIS

# Dependency Tree Schema Conversion with Tree Transducers

submitted by

Felix Hennig

MIN-Fakultät

Fachbereich Informatik

Natürlichsprachliche Systeme

Studiengang: Informatik

Matrikelnummer: 6540353

Erstgutachter: Prof. Dr-Ing. Wolfgang Menzel

Zweitgutachter: Arne Köhn

# Contents

# 1. Introduction

Treebanks are a fundamental resource for natural language processing as they are used for part-of-speech taggers and dependency parsers. They are also relevant in other areas of linguistic research. Knowledge about the syntactic structure of a language is made accessible to algorithms as empiric data. Treebanks can contain hundreds of thousands of sentences, annotated manually in a work-intensive procedure and as such they are a precious resource. The annotation schema of the treebank is just as important as the data itself, as it has to be decided in advance and is then used to annotate a significant number of sentences. It determines how syntactic structure is encoded, which relations between words are recognized and distinguished. The set of tags and labels used in the schema is the same as it is later in a parser or tagger trained on the treebank. The schema can already be geared towards a certain use, or involve certain assumptions which might change at some point in time. Because the annotation schema is applied to all sentences in the treebank involving a lot of manual labor, making changes to the schema can be difficult, as the changes have to be applied to all sentences in the treebank as well.

## 1.1. The Problem of Large-Scale Dependency Tree Conversion

Treebanks are occasionally modified to fix errors which have been made in the initial annotation, these changes involve only single trees. Large numbers of trees only have to be edited if the annotation schema used changes, which is the case if a schema is modified to include a distinction that was not recognized previously, certain structures are annotated differently or when a completely new schema should be used. While the annotations might have been created manually initially, updating each individual tree manually again is a disproportionate amount of work because of the sheer size of the data to be adapted. Depending on the changes to the schema, a lot of restructuring is repetitive and can be automated. In cases where the whole schema is changed, there is already syntactic information in each tree, the representation of this information is just changed, which does not require as much human intervention as a schema modification. Automatic conversion on the other hand can be difficult if the source schema is less detailed than the target schema, as it is the case when a schema is modified to include a previously untreated distinction. In the case of – for example – splitting up a label, the modification is introduced explicitly to provide information that was not available or inferable previously, making completely automatic conversion intrinsically difficult. But also in this case, a lot of the decisions can be automated based on context, and a human annotator should only be faced with the difficult, ambiguous decisions, reducing the workload as much as possible.

Conversion errors can be introduced either way. Human annotators might introduce inconsistencies, especially if the treebank is edited by multiple people, but errors made previously would be noted in the process. Automatic conversion is more consistent but can include systemic errors.

Because these schema conversions are applied only once, the software used for the conversion process is often developed ad-hoc, making it less reusable for other treebanks. Although every conversion process is distinct, the basic principles are reused. Because the resulting treebank is more important than the conversion process itself, the software supporting the conversion needs to be as reliable as possible; powerful, yet constrained enough to not introduce any errors, making the process safer. It should support an intuitive model of the conversion process and the work presented will show that large parts of the conversion process can be dealt with without programming knowledge. This allows for a wider range of users, including people with a background in linguistics, who might be more suited for the conversion task because of their linguistic knowledge, but are not able to automate the repetitive parts. The repetitive nature of the conversion process points to the use of a rule-based system, converting patterns found in the source treebank to corresponding patterns in the target treebank.

## 1.2. Example: Converting the Hamburg Dependency Treebank to Universal Dependencies

To develop such a tool, I used the *Hamburg Dependency Treebank* (HDT) (Foth et al. 2014) as an example of a large treebank which should be converted, in this case to the *Universal Dependencies* (UD) annotation schema (McDonald et al. 2013). The HDT is a very large treebank with over 200k sentences and a detailed annotation schema native to German. The UD annotation schema on the other hand seeks to be applicable to all languages and features less labels and tags. It is a content-head schema in contrast to the HDT schema which is function-head. This means that not only the set of tags and labels differs, but the structure also changes, from having function words as the heads of subtrees to making their content-bearing words the heads. Figure 1.1 shows a sentence from the HDT annotated with both schemes, illustrating the differences in tagset and structure. The first three words of the sentence form a prepositional phrase, in which originally the preposition is the head and in UD, the noun is the head, having the adjective and the preposition as dependents. Both schemes feature an 'auxiliary' relation between the two verbs of the sentence, but the direction of the edge is inverted in the UD tree. While at first sight a transformation based on multiple systematic local conversions seems difficult, I will show that systematic conversion steps can be found and formalized, based on the local subtree groups of the tree.

Taking a specific conversion task as a foundation for the development of a tool ensures that the tool will be tailored to the task at hand and it also provides an immediate testing situation and a basis for evaluation of the tool. However the present work focuses on the tool and the underlying formalism instead of the conversion of the treebank, the HDT will only be used as an example trough out the development of the tool.

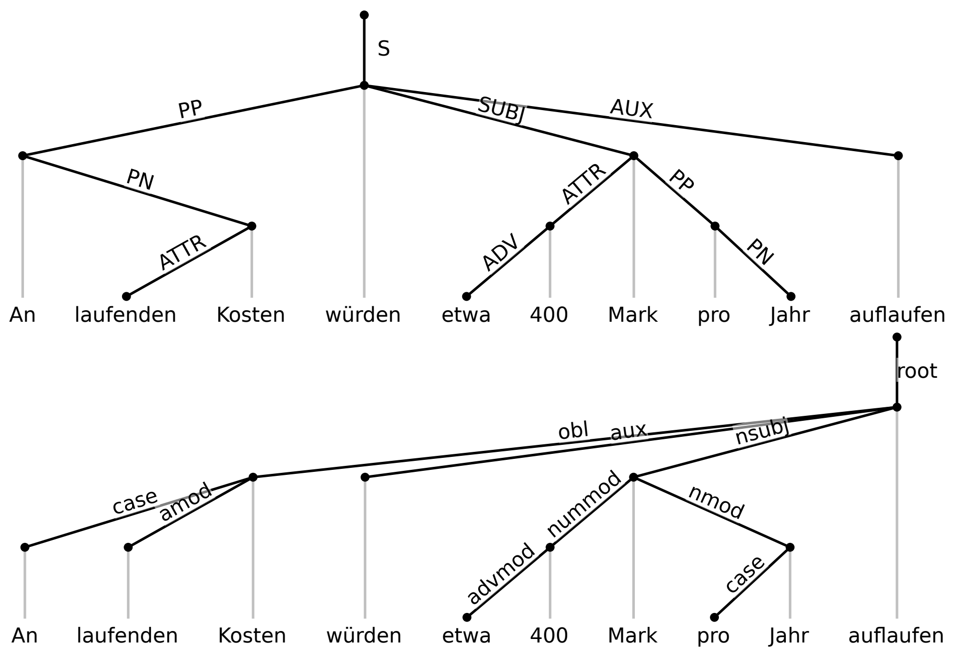

S
PP
SUBJ
AUX
PN
ATTR
ATTR
PP
ADV
PN
An laufenden Kosten würden etwa 400 Mark pro Jahr auflaufen
root
obl
aux
nsubj
case
amod
nummod
nmod
advmod
case
An laufenden Kosten würden etwa 400 Mark pro Jahr auflaufen

Figure 1.1.: An example sentence taken from the HDT and annotated manually with the UD tagset. The most notable changes include the different root node and the different structure of prepositional phrases (“An laufenden Kosten”, “pro Jahr”).

# 2. Related Work

The presented work is linked to different research areas. Immediately related are the many treebank conversion publications, published since the inception of the UD project. Various conversion techniques have been applied, although some overlaps can be observed. Existing supporting conversion software ranges from rigid rule based conversion systems to software libraries supporting the work with dependency trees. Some overlap can be found in other research areas concerned with transformations of tree structures like XML data structures. In natural language processing, machine translation is also concerned with transforming sentences and their syntactic structures; transformations based on syntax trees have been explored too, although machine translation frequently uses phrase structure trees instead of dependency trees which are used here.

A paper about the work presented here has been published at the Univeral Dependencies Workshop at the NoDaLiDa Conference 2017 (Hennig and Köhn 2017), from which the Figures 3.1, 3.7 and 4.1 are taken.

## 2.1. Treebank Conversion so Far

Annotated corpora as a foundation for parsers have been around for a while, but the modification of the schemes used is a more recent development. The Universal Dependencies project (McDonald et al. 2013) has led to a number of treebank conversions, where native treebanks were converted to Universal Dependencies (e.g. Danish (Johannsen, Alonso, and Plank 2015), Norwegian (Øvrelid and Hohle 2016), Swedish (Nivre 2014; Ahrenberg 2015) and Hindi (Tandon et al. 2016)). Treebanks previously had annotation schemes tailored specifically to the language of the treebank. UD strives to establish an annotation schema which applies to all languages universally to facilitate multi-language parsing and support other cross-lingual research areas such as machine translation. Most treebanks, like the HDT, are annotated with a function-head schema, because it directly reflects the syntactic structure of the language. Function-head means that function words introducing phrases are used as the head of the subtree annotating the phrase. While this approach seems sensible when looking at a single language, it translates poorly to a cross-lingual approach. UD applies a content-head schema instead, meaning that the content-bearing word of a subtree is used as its head, while words without important content are attached below, with relations indicating that these words are markers or particles. Some languages contain functions words, others work heavily based on inflections of the content word. All languages contain the content words, thus a tree using these words as the head will be more similar to other trees in different languages.

Since this is a recent development in the research area, there has not been any research in the direction of efficient and safe treebank conversion. The conversion process goes beyond the renaming of labels, as the function-head structures have to be transformed to content-head structures, which, depending on the source annotation schema can change the tree structure to a large extend. In the published research about treebanks that have been converted to Universal Dependencies, different approaches can be found, as it is expected since the underlying treebanks differ. As the software used to convert will not be used after the conversion is finished, there is little incentive to put a lot of work into the software, leading to many ad-hoc solutions. Some similarities can still be observed, while details of the process often remain unpublished, as they are not central to the result, which means that scripts are not published and the processes only outlined.

For the published software, conversion is usually done in a single script, divided into multiple sections. The script may rely on other tools to support the process, of which two will be discussed in the next section, or conversion is done on a low abstraction level, sometimes even working directly on the CoNLL files. All conversion rely on additional human annotation afterwards to treat edge cases in the annotations. The software can mark unusual structures, but tool support to modify the trees marked for manual adjustments is rare or not mentioned in the publications.

### 2.1.1. Supporting Tools

The Finnish Turku dependency treebank was converted to UD by Pyysalo et al. (2015), the used script involved dep2dep[1] a rule-based conversion tool with its own syntax. rules are converted to prolog code and applied to the dependency trees. The rules use constraints to match edges in the tree and convert them, it allows to match structures of multiple edges at once, edges can be matched anywhere in the tree.

Tyers and Sheyanova (2017) convert North Sámi to UD with a rule pipeline implemented as XSLT rules. XSLT is a programming language for the transformation of XML documents, which are also trees. XSLT is declarative and rule-based as well and the conversion rules are formulated in XML.

Ribeyre, Seddah, and Villemonte de la Clergerie (2012) outline a two step process where the rules to be applied are determined in the first step and are then iteratively applied in the second with meta-rules handling conflicting rule applications.

Neither of these approaches can inherently guarantee well-formed tree output, which is due to modifying the tree at arbitrary points, a directed approach inherently prevents loops, which is not the case here. All three tools use matching based rule systems, like the work presented here.

### 2.1.2. Universal Dependencies Internal Conversion

The Universal Dependency schema itself is also under active development and alterations have to be applied to treebanks in many different languages, allowing for little reliance on the words themselves in generic solutions. Solutions can include specialized software

[1]`https://github.com/TurkuNLP/dep2dep`

for each language, by adapting a base script to each language specifically and relying on manual work as well. For the conversion from version 1.3 to version 2 of the schema, multiple conversion scripts have been developed and published, also relying on the udapi framework[2]. Specific changes for this version change included simple renaming of labels, a label being split into two new labels, a complete rework of handling of ellipsis and various modifications to the usage guidelines of certain labels.

The published scripts execute simple translations of labels and basic transformations and mark words which require further human intervention. In the case of ambiguous label splitting, the label to be used can be chosen interactively, but no graphical interface supports the user in their decision.

The udapi framework supports the development of scripts working with CoNLL-U[3] data, by providing building blocks to read and write data, abstracting away the files and providing utility functionality for basic operations such as reattaching tokens elsewhere in the tree. A pipeline structure with different building blocks is suggested for conversion scripts, and a few blocks are already provided for common restructuring operations such as changing the direction of a `flat` edge. The framework takes a pragmatic approach to the conversion process and improves on the commonly chosen ad-hoc conversion process by supporting common operations and providing many common utility functions. It allows for more high-level programming than writing a whole conversion script from scratch, but when a new building block for the pipeline is developed, programming is still required.

The java implementation of the framework is also used in the implementation described in Section 4. It does not include as many features as the python version described above, it provides only basic access to data structures.

## 2.2. Tree Transducers in Machine Translation

Machine translation is also concerned with with the conversion of sentences. Word by word translation does not make sense when the grammatical structure of source and target language differs, some languages have specific grammatical cases which others lack, other languages contain a lot of inflections. It is also possible that the order of subject predicate and object differs in the target language. This is why translation works on phrase structure trees, where translation can work on phrases instead of words thus allowing to merge multiple words into one when translating the phrase or rotating subtrees before they are converted.

Tree transducers are widespread in machine translation, Maletti (2010) gives an overview of the different types of tree transducers which are used in machine translation as well as their properties in regards to the task of machine translation. The present work relies on the formalizations given in Malettis work, although a number of modifications have to me made to accommodate for the differences between phrase structure and dependency trees.

[2]`https://github.com/udapi/udapi-java`

[3]A dependency tree file format based on the CoNLL-X format

# 3. Tree Transducers for Dependency Trees

The use of tree transducers in machine translation was a starting point for their use in treebank conversion. To my knowledge, tree transducers have not been used with dependency trees so far. The tree transducer model makes certain assumptions about the type of tree it is applied to, which makes phrase structure trees a good fit and dependency trees seem less suited. In a phrase structure tree, there are inner nodes which consist only of a label. The edges are unlabeled and the words of the sentence are the leaves of the tree. A dependency tree on the other hand does not have inner nodes, all the nodes in the tree are words from the sentence and also have other morphological features attached. Before required modifications will be discussed in Section 3.2, the basic top-down tree transducer model is introduced in Section 3.1, which includes an outline of the fundamental components entailing a transducer and the basic process of a tree transformation.

When using the implementation of the tree transducer to convert dependency trees, a number of features were found to be helpful to formulate powerful and concise rules. Existing conversion procedures were also taken into account to incorporate useful features from other approaches as well. These extensions to the mechanism and how they affect the transformation process are discussed in Section 3.3.

## 3.1. Top-Down Tree Transducer Model

The top-down tree transducer model and basic notation will be introduced based on the example conversion of the number "six hundred twenty six" to its digit representation. A visualization of the transformation process can be observed in Figure 3.1, along with the definitions for the components in Figure 3.2. The conversion process shows the use of the different components throughout the conversion process. three different sets of nodes can be seen, coloured in yellow, orange and green, as well as the set of rules $R$, which is denoted at the arrows of the conversion steps and contains the transformations rules.

**Tree Components** The initial tree is composed of only yellow nodes, which are nodes from the input alphabet $\Sigma$. A node consists of a symbol and a rank, which is the number of children it has. The alphabet is therefore a set of symbols and a function mapping symbols to their ranks. In dependency trees as well as phrase structure trees, the symbols can be unranked as well, meaning that the number of child elements is not predefined. Without the rank only the set of symbols remains, therefore the set of symbols as well as the alphabet is denoted by the same symbol. The input alphabet is called $\Sigma$ and the

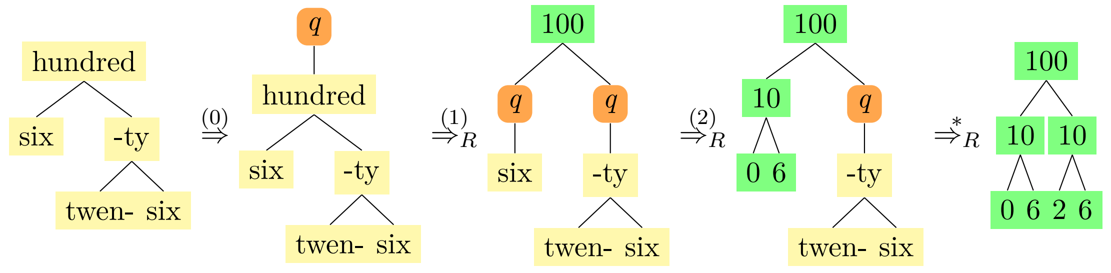


Figure 3.1.: An example of a tree transformation of the numeric expression "six hundred twenty six" to its digit form 626, using a top-down tree transducer. The yellow nodes are nodes from the input alphabet, the green nodes are from the output alphabet and the orange nodes are state nodes. Example based on Maletti (2010), the last step includes multiple transformation steps.

$$\Sigma = \{\text{one}, \text{two}, ..., \text{six}, ...\text{twen-}, \text{thir-}, ...\text{-ty}^2, ..., \text{hundred}^2\} \tag{3.1}$$

$$\Delta = \{0, 1, 2, ..., 6, ..., 10^2, ..., 100^2\} \tag{3.2}$$

$$Q = \{q, ...\}, \quad I = \{q\} \tag{3.3}$$

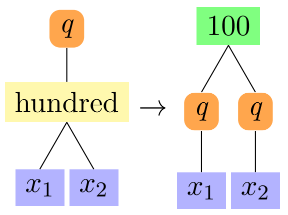
$$R \subset Q(\Sigma(X)) \times \Delta(Q(X)) \tag{3.4}$$

$$M = (Q, \Sigma, \Delta, I, R) \tag{3.5}$$

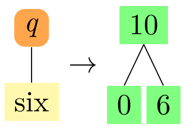
Figure 3.2.: Formal components of the transducer used in Figure 3.1, the transducer itself is a 5-tuple denoted by $M$.

output alphabet is called $\Delta$, its nodes in the example are highlighted in green. Figure 3.2.3.1 and 3.2.3.2 show the formal definition of these alphabets, where the ranks are denoted as superscript to the symbols. The set of possible trees which can be created from the input alphabet is called $T_\Sigma$, respectively $T_\Delta$ in the case of the output alphabet. The first tree in Figure 3.1 is an element of $T_\Sigma$ and the last tree, after all conversion steps are applied, is an element from the set of output trees $T_\Delta$.

The intermediate trees contain nodes from the set of state nodes $Q$ as well, which are coloured orange in the example. These trees are from the set $T_\Delta(Q(T_\Sigma))$, which means that the elements at the top are from $T_\Delta$, at the bottom there is $T_\Sigma$ and in between are the state nodes. Transformation steps are always anchored at a state node, this is why, in Step 0 of Figure 3.1 a state node $q$ is attached at the root of the tree. The figure shows how the transformation is started at the top and is progressing down the tree. Throughout the transformation process, the state nodes separate the already converted output nodes at the top from the nodes of the input alphabet at the bottom. Therefore the set of state nodes in the tree can be referred to as a *frontier*, as it splits the tree in a converted and a nonconverted part. Figure 3.2.3.3 shows the set $Q$ of all state nodes, as well as the set $I$ of initial state nodes, which can only be used in the first step as a



(a) Transformation rule used in Figure 3.1, Step 1. In text representation the rule can be expressed as $q(\text{hundred}(x_1, x_2)) \rightarrow 100(q(x_1), q(x_2))$.



(b) Transformation rule used in Figure 3.1, Step 2. In text representation this rule can be expressed as $q(\text{six}) \rightarrow 10(0, 6)$.

Figure 3.3.: Transformation rules used in Figure 3.1. Additionally to the input, output and state nodes, variable nodes are introduced with a blue highlighting.

root for the transformation process. Later on only a single type of state node will be used, which means $Q = I = \{q\}$, and therefore the concept of multiple state nodes is not elaborated further at this point. For additional information I refer to Maletti (2010).

**Transformation Rules** The last missing component in the 5-tuple is the ruleset $R$ which contains the transformation rules, the core of the transformation process.

Figure 3.1 shows rules used in step 1 and 2 of the transformation process in Figure 3.3. A rule consists of two trees, one on the left-hand side of the arrow and one on the right-hand side; a rule can also be expressed as a tuple: $r = (T_L, T_R)$. $T_L$ represents a subtree structure that is supposed to be matched to the tree that is currently converted. The $q$ and “hundred” node can structurally be matched to the root of the tree in the conversion example.

Below the “hundred” node a new type of node can be found, highlighted in blue. The nodes $x_1$ and $x_2$ are variable nodes from the set of variables $X$, which can match any type of node. $x_1$ and $x_2$ also appear in $T_R$, attached below a state node. Parts of the tree matched to variables are not converted, they are reattached below state nodes, below the converted parts of the tree to be matched in the next conversion step. In the example , the “six” node is matched to the $x_1$ variable, and is converted in the next step.

As mentioned previously the intermediate trees in the conversion process have are structured according to $T_\Delta(Q(T_\Sigma))$ and $T_L$ is always structured with $Q(\Sigma(X))$ thereby matching below the frontier denoted by the $Q$ nodes. The right hand side is then pushing the frontier down the tree, by converting a node. $T_R$ is structured as follows: $\Delta(Q(X))$. $\Sigma$ nodes are replaced with $\Delta$ nodes and the $Q$ nodes progress down the tree.

Figure 3.3b shows a rule without variables. It matches a state node but does not introduce new ones, thereby removing parts of the frontier entirely; it can be seen as a termination rule.

**Rule Properties** An essential factor for the use of transducers in the context of natural language processing is the handling of the variables. If a node or the subtree below it contains words from a sentence and it is currently being converted, those nodes should not be duplicated or removed, otherwise the sentence would change. The variables contain subtrees of unknown structure, and these subtrees are deleted or duplicated, without any information about their contents, if the variable would be used more or less than once on the right-hand side of the rule. A rule $r = (T_L, T_R)$ is *non-deleting* if all variables in $V \subseteq X$ in $T_L$ also appear in $T_R$. If each variable appears on most once in $T_R$, $r$ is *linear*. Reuse of the same variable multiple times on the left-hand side is not considered.

The rule shown in Figure 3.3a contains $x_1$ and $x_2$ on the left-hand side, and both variables also appear exactly once on the right-hand side. This means that no subtree is deleted, therefore this rule is non-deleting. Also, no variable appears twice, making this rule linear as well.

In the following, only non-deleting, linear rules are considered and the properties will be extended in Section 3.2.3.

## 3.2. Mapping the Tree Model to Dependency Trees

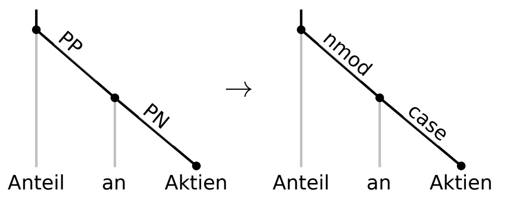
The first thing to look at when creating a tree transducer model for dependency trees is again the tree structure and how it maps to the transducer model, the previously discussed elements: input, output and state nodes. The nodes in the tree are the tokens in the sentence, they do not consist of only a label as it was shown in the example in Figure 3.1, they have an index, a lemma, morphological features, a dependency relation (the label on the edge to the parent) and a Part-of-Speech tag. Section 3.2.1 will treat these differences.

The order of the words in the sentence is defined by the indices on the tokens, the order in which the nodes are attached below their parent does not have any effect on the order of the nodes in the sentence, which means that the list of child nodes is actually unordered. Also in contrast to the previously introduced model, the number of child elements is not fixed based on the parent dependency relation or any other property of the parent node, i.e. the nodes are *unranked*. This too requires adaptions to the formalism, which will be described in Section 3.2.2.

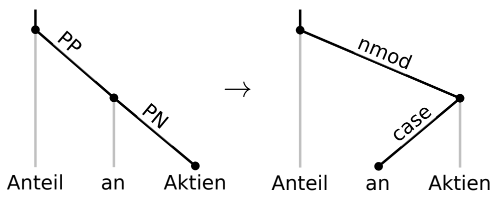
Section 3.2.3 introduces an additional property, extending the concept of non-deleting and linear transducers to *word-preserving* transducers on dependency trees.

### 3.2.1. Node Identity

The part that should be changed in the transofrmation process is the dependency relation and the structure, other token properties do not change, such as the word or its position in the sentence. To accomodate for this duality, we define that $\Sigma = L_i \times \mathbb{N}$ and $\Delta = L_o \times \mathbb{N}$, where $L_i$ and $L_o$ are the sets of input and output dependency labels. The second component of the tuple is the index of the token, used to attach the unmodified properties



(a) The first possibility of attaching the labels to the nodes. No restructuring is done, only the labels are renamed. The rule with node identity is `q(n1:PP(n2:PN))` → `n1:nmod(n2:case())`.



(b) The second possibility of attaching the labels to the nodes, here the edge between the nodes is inverted, the previously lower node is now the head of the subtree. The rule with node identity is `q(n1:PP(n2:PN))` → `n2:nmod(n1:case())`.

Figure 3.4.: Example of transformation ambiguity without node identity in the conversion rule `q(PP(PN))` → `nmod(case())`.

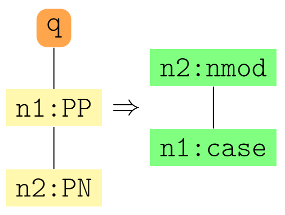
to the node. When using this node definition in transformation rules, the exact index is irrelevant, the index makes the nodes uniquely identifiable, so any unique identifier can be used. Nodes have to be identified on the left- and right-hand side of the rule to unambiguously attach new dependency labels to the nodes. The syntactic representation is `n1:PP`, where `n1` is the alpha-numeric identifier and `PP` is the dependency relation. `n1` is an arbitrary but fixed element $\in \mathbb{N}$, the second part of the tuple, while `PP` is an element from $L_i$, the first part of the tuple.

In Figure 3.4 the use of the node identity is shown. The figure illustrates two different rules, which can only be distinguished by the use of node identifiers. Either the nodes are only renamed, as it is shown in Figure 3.4a, or the nodes are restructured as well, which is shown in Figure 3.4b. The ambiguity is resolved by the node identity mechanism, in this case the intended meaning of the rule includes the swapping of the nodes, swapping the function and content word therefore moving from a function-head to a content-head structure.

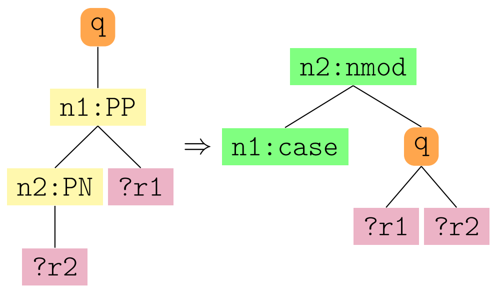
### 3.2.2. Unrankedness

The rank of the elements respectively nodes in $\Sigma$ and $\Delta$ has been left undefined so far because the dependency relations in a dependency tree do not have a rank. Each token in the sentence can have arbitrarily many dependents, for example in Figure 1.1 two nodes are attached with the `PN` relation, one does not have any child nodes, the other has an additional `ATTR` dependent. Nouns can have adjectives and adjectival clause dependents, predicates have adverbs and adverbial clauses.

Because of this, it is impossible to define a single fixed rank for each dependency relation, which means that a rank function as it was described in Section 3.1 cannot be defined. Formally, the alphabet is extended to include each dependency relation multiple times with different ranks, solving the problem of the rank function. Since the input and output alphabet do not need to be specified explicitly, extending them to contain



(a) `q(n1:PP(n2:PN()))` $\rightarrow$ `n2:nmod(n1:case())`
The rule without any variables.



(b) `q(n1:PP(n2:PN(?r2), ?r1))` $\rightarrow$ `n2:nmod(n1:case(), q(?r1, ?r2))`
The rule with catch-all variables matching potential dependents to either `n1` or `n2`. The `n1` node loses the dependents.

Figure 3.5.: Extending the `PP-PN` rule to account for additional unspecified dependents.

each node multiple times with different ranks is not a problem. Rules however need to be specified explicitly. Since there can be any number of dependents to any node in the tree, and the children of a node are unordered, the matching process has to be adapted to be order independent. The rule given in Figure 3.4b is the same rule used in the conversion of the `PP-PN` structure in Figure 1.1. However, currently it does not account for additional dependents of neither the `PP` nor the `PN` node.

In Figure 3.5 the extension is shown, variable dependents are added to the rule on both sides, thereby allowing the rule to be applied in the previously inapplicable case as well. In contrast to the previous variable concept, the variables used for dependency trees can match arbitrarily many nodes, including no nodes at all. These variables are called *catch-all* variables and they are represented by an identifier prefixed with a question mark. By introducing a variable below each node, any combination of additional nodes can be matched, which allows to have a single rule for all cases.

This extension does not change the properties of the transducer, as each rule with catch-all variables can be expanded into multiple rules with different numbers of variable dependents.

### 3.2.3. Word-Preserving

Based on the node identity in rules, a new property of a rule and consequently a ruleset and a transducer can be defined, in addition to the linearity and non-deleting property. Not only should variables not be duplicated or removed, but words should also be preserved. A rule similar to the one depicted in Figure 3.3b cannot be written in the context of a dependency tree, as the number of nodes on the left- and right-hand side of the rule is not the same. In a dependency tree, no nodes can be created and duplication or deletion should not happen as well. A rule $r = (T_l, T_r)$ is *word-preserving* if it meets the following conditions: First, $r$ is linear and non-deleting. Second, the left-hand side of $r$ cannot contain the same node index twice:

$n \neq m \; \forall (\cdot, n), (\cdot, m) \in T_l, (\cdot, n), (\cdot, m) \in \Sigma$. Third, $r$ neither deletes nor duplicates a matched word. This means that for $N_{in} := \{n \in \mathbb{N} | (\cdot, n) \in T_l\}$ each $n \in N_{in}$ appears exactly once in $T_r$ (Hennig and Köhn 2017). It follows that the number of nodes remains the same after conversion and the nodes also remain the same, but might have received different dependency relations and might be connected in a modified structure.

## 3.3. Extensions for Expressiveness

The additions layed out in the previous section are necessary when working with dependency trees, without these changes the formalism cannot be applied to the dependency tree structure. In this section further modifications to the formalism are discussed. These changes increase the expressiveness and flexibility of the rules, making them easier to work with.

### 3.3.1. Node Features

In Section 3.2.1 the nodes of the tree were defined to consist of their ID as well as the dependency label, to allow to identify nodes beyond their dependency relation. Nodes also have other features, the most prominent one being the PoS tag. The PoS tag can be included when formulating rules, nodes in the tree then need to have a matching PoS tag to make the rule applicable.

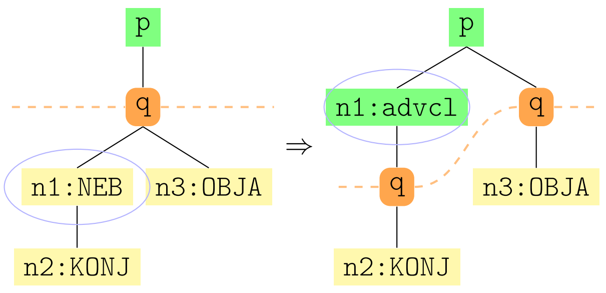
The UD schema features a `nummod` and a `amod` relation. Both are represented as `ATTR` in the HDT. To distinguish numeric modifiers from other modifiers the PoS tag can be used. Numbers carry the `CARD` PoS tag, thus we can use these two rules to distinguish `nummod` and `amod`:

```
n.CARD:ATTR() -> n:nummod();
     n:ATTR() -> n:amod();
```

Syntactically the PoS tag is prefixed by a period. It is important to test these rules in the correct order, the `nummod` rule needs to be tested first as it is more specific than the `amod` rule. The order in which rules are tested will be discussed in detail in Section 4.1.

### 3.3.2. Extended

A common modification is switching from a ruleset $\subset Q(\Sigma(X))$ to a ruleset $\subset Q(T_\Sigma(X))$. Instead of having single nodes below the state nodes ($\Sigma$), tree structures can be used ($T_\Sigma$). This is an intuitive and common extensions to tree transducers, transducers with this modification are called *extended*. A rule which is extended can transform a bigger context, which is particularly useful when transforming function-head structures to content-head structures, as it is done in Figure 3.4b. The figure shows a rule which converts a prepositional phrase from function-head to content-head. The HDT has the preposition as the head of the phrase and the noun phrase is attached as a complement to the preposition. In UD the noun phrase is the head and the preposition is attached



```
p(q(n1:NEB(n2:KONJ()), n3:OBJA())) → p(n1:advcl(q(n2())), q(n3()));
```

Figure 3.6.: Example of look-ahead usage to identify an adverbial clause.

as `case`. Since the new head is below the current head, the rules need to work with a depth of at least two levels, which is achieved by making them extended.

### 3.3.3. Look-Ahead

When including more context of the tree, sometimes not all of the matched nodes should be converted at once. It is possible that some dependent should be present to recognize a certain structure to constrain rule application to a specific context, but the conversion of that matched dependent in turn requires a certain context, which is why it is converted in a separate rule. Matching a node as part of the left-hand side rule context, but not converting it on the right-hand side is called *look-ahead*.

Figure 3.6 exemplifies this. A `NEB` dependency relation can be converted to `advcl`, `ccomp` or `xcomp` depending on its context and the rule shown in the Figure relies on the presence of a neighboring object to determine that this subclause is an adverbial clause. For the two other cases there are dedicated rules as well which are not shown here. The object can in turn be an indirect object if a second object is present; it has its own context dependent conversion variations, which is why it is not converted in this rule.

The `KONJ` dependent is also matched for context. This dependent does not have multiple cases where it is converted differently based on its context, it is always converted to a `mark` dependent. It is still converted in a separate rule, since the `KONJ` dependent is referenced in multiple rules concerned with the conversion of `NEB` dependents. Following the software development principle of separation of concerns, this dependent is converted in its own rule.

Using this principle one dependency relation in the source schema converted using a few rules, either just one if it has a direct correspondence in the target schema or multiple rules if a distinction needs to be made. By using the look-ahead a wider context can be used to formulate very specific rules with fine grained distinctions, and only a narrow transformation area. This approach leads to blocks of rules each concerned with one or two dependency relations, which can also be seen later on in the implementation (Section 4) where rules are declared in a file and the block structure contributes to keeping track of all conversion steps.

### 3.3.4. Look-Back

The *look-back* follows that same principle as the look-ahead, it matches nodes to constrain rule application to a specific context, the matched nodes are not modified. Look-back refers to looking at nodes which are above the frontier and thus already converted.

The PP-PN rule in Figure 3.5 needs to be extended to take the PoS tag of the governing node into account; in UD the noun phrase is attached differently depending on its regent. The nmod label is used for noun phrase regents and for predicates the obl label is used. Here are two PP-PN rules matching the parents PoS tag for context:

```
p.VVFIN(q(n1:PP(n2:PN(?r2), ?r1)))
    -> p(n2:obl (n1:case(), q(?r1, ?r2)));
p.NOUN(q(n1:PP(n2:PN(?r2), ?r1)))
    -> p(n2:nmod(n1:case(), q(?r1, ?r2)));
```

The p node is matched above the frontier; the first rule requires the VVFIN tag on the node, the second requires NOUN. If n2 is attached to a verb (VVFIN), the dependency relation should be obl, otherwise it should be nmod.

Information about converted parts of the tree can alternatively be encoded in state nodes. This requires the relevant information for one rule to be known in advance to the a different, encoding rule. These rule interactions increase coplexity, the look-back is a practical alternative with no changes to the formalism required, since the accessed information could be encoded in a state node.

### 3.3.5. Cross-Frontier Modifications

Converting a function-head tree to a content-head tree requires an inversion of edges whenever a function word is the head of a subtree. In the previous example of the PP-PN respectively case-nmod structure the position of the content word in relation to the function word was fixed, allowing to match the content-word together with the function word whenever the latter is encountered.

Verb structures also require an inversion of edges. In a sentence there is the main verb at the root of the tree and other verbs connected as auxiliary verbs. In the HDT schema the conjugated verb is the root of the tree and other verbs are chained below this verb, connected by AUX relations. If the sentence contains multiple verbs, the conjugated verb is usually a verb to express modality or tense, while the content-bearing verb is at the end of the AUX-chain. In UD the root of the tree is the content-bearing verb, which means the whole chain has to be inverted. However, in contrast to the prepositional phrase mentioned above, the depth of the chain is not known in advance and there is more than one dependency relation used for the token above the auxiliary verb. Also the dependency relations do not have to appear together with an auxiliary verb in contrast to the PP relation, which requires a PN dependent. The simplest chain is a basic two verb sentence with a root-aux structure, however verbs can also be in a subclause and there can be more than two verbs, which means advcl-aux-aux is also a potential chain that

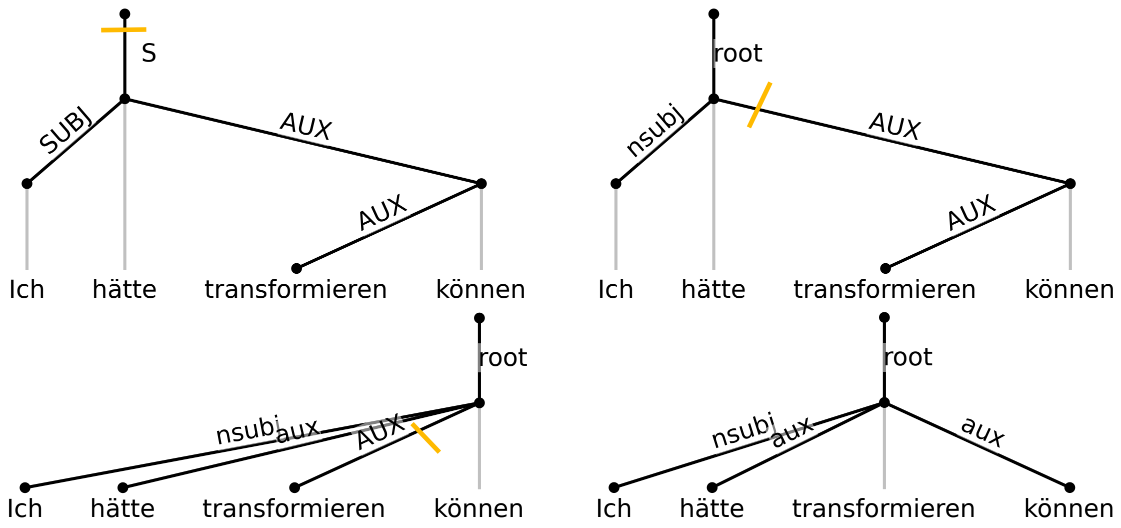


Figure 3.7.: Conversion of a dependency tree from a function-head to a content-head scheme. The *AUX* edges are inverted when they are converted. The rule used in step 2 and 3 is `p:$x(q(n:AUX(xsn), xsq), xsp)` → `n:$x(p:aux(), q(xsn, xsq), xsp)`.

needs to be covered by the ruleset. Instead of writing one rule per combination of labels that can occur, the first word in the chain is converted with a normal rule and when an auxiliary verb is found below it, the previously already converted token is modified again, even though it now already above the frontier.

This *cross-frontier modification* allows for a significantly smaller number of rules by separating the conversion of each of the labels in the chain into different rules; there is a rule to convert a `S` relation and a rule to convert an `AUX` relation which can be applied multiple times for longer `AUX`-chains.

Figure 3.7 illustrates how a sentence that requires a head inversion over three tree levels can be transformed with a rule that only matches one `AUX` relation at a time. First the `S` and `SUBJ` relations are converted, then the `AUX`-rule is applied for the first time. It matches the regent of the `AUX` edge into the `p` variable and for the dependency relation of the `p` node, the rule contains the expression `$x`. This is a mechanism to match the dependency relation of the node into variable which is distinct from the variables used to match arbitrary nodes. This `$x` variable is reused on the right-hand side of the rule to attach the dependency relation to a different node, while the `p` node receives the aux dependency relation. After the rule is applied once the previous head is now attached as an aux dependent while the previous `AUX` dependent received the dependency relation of the head, which is already in the UD schema due to being converted already with a different rule. The rule is then applied to the next `AUX` relation which is inverted as well, and the `root` relation is attached to the previously last verb in the chain, which is the content-bearing verb and now also the root of the sentence.

This mechanism was implemented with the intention of using it for edge cases which cannot be converted otherwise. Modifying previously converted nodes changes the context for applying rules further, which means the applicability of rules can change in unexpected ways, which is why this mechanism should be used sparingly.

# 4. Implementation

The transducer implementation operating on dependency trees was developed in parallel with the design of a ruleset for the conversion of the HDT. The implementation is written in Java and uses the udapi framework (See also Section 2.1.2).

The components of a transducer – input and output alphabet, state nodes and conversion rules – are specified in a file which is read by the software, the file only contains the conversion rules. The input and output alphabets are defined implicitly through the use of the labels in the rules; the set of state nodes is predefined, there is only a single type of state node. By defining the rules in a single file, the actual conversion logic is in a single place and separated from the rest of the code which is required to read in CoNLL files, apply the rules to the dependency trees or visualize the conversion process.

This section lays out the syntax used to specify rules, including details such as inference and rule application order. PoS tag expansions as another convenience feature are explained and lastly interactive conversion and the graphical user interface are described.

## 4.1. Rule Syntax

The rule syntax is largely identical to the syntax introduced in and extended throughout Section 3. Examples are shown in Figure 4.1.

Rule (a) and (b) demonstrate the basic rule syntax. A rule consists of two trees and an arrow in between. The trees consist of arbitrarily nested nodes, a node consists of at least an identifier and often of a dependency relation following the identifier, separated by a colon. In Rule (b) the catch-all variables introduced in Section 3.2.2 are shown, they are denoted by an identifier prefixed with a question mark. Rule (c) includes a variable $x instead of a dependency relation. The variable can match any dependency relation on the left-hand side and allows to reattach the matched dependency relation to a different node on the right-hand side as it was explained in Section 3.3.5.

**Inference** Rule (c) also includes an explicit frontier, while the previous two rules did not. The frontier node is denoted by curly braces, it does not need an identifier since there is only a single type of state node. If the frontier is not mentioned explicitly it is assumed to be above the root node on the left-hand side. On the right-hand side the frontier is also not included, here a frontier is inserted above the catch-all variables. Any node receiving a dependency relation on the right hand side will be above the frontier after the rule is applied.

Rule (d) is semantically identical to Rule (a) but it explicitly states all the parts which are inferred to the rule by the engine. Besides the already mentioned frontier nodes, it

```
(a) n:SUBJ() -> n:nsubj();

(b) n1:PP(n2:PN(?r2), ?r1) -> n2:nmod(n1:case(), ?r1, ?r2);

(c) parent:$x({n:AUX(?auxr), ?fr}, ?r) -> n:$x(parent:aux(), ?auxr, ?r, ?fr);

(d) p({n:SUBJ(?r), ?fr}, ?pr) -> p(n:nsubj({?r}), {?fr}, ?pr)

(e) p({n1:OBJA(), n2:OBJD()}) -> p(n1:obj(), n2:iobj());
    n:OBJA() -> n:obj();
    n:OBJD() -> n:obj();

(f) p.NE({n.NN:APP()}) -> p(n:appos());
    p.NN({n.NN:APP()}) -> p(n:compound());

(g) p({n:APP()}) -> p(n:compound()) :- {n.getOrd() < p.getOrd()};
```

Figure 4.1.: Rule (a) to (c) show rule syntax examples and (d) is a verbose version of (a). Rule (e) and (f) show rule combinations and (g) shows the use of groovy code to further constrain rule applicability.

can be seen that a technical root is inferred, as well as numerous catch-all variables, one for each node, also for the frontier node. This is necessary, since there can be neighboring nodes at every level of the tree. Sibling nodes which are already converted will be above the frontier and are matched into the `?pr` variable, sibling nodes which still need to be converted are matched into the `?fr` variable. This allows the annotator to focus on the structures of the tree which are relevant for the conversion while ignoring irrelevant tree context. Nodes which are not relevant for the conversion are matched into a variable and reattached to the same nodes on the right-hand side.

**Rule Application Order** Three rules are shown in example (e), an elaborate rule at the top followed by two simpler rules. The first rule matches two neighboring objects with different grammatical cases, `OBJA` is the *accusative* case and `OBJD` the *dative* case. The rules are tested for applicability from top to bottom. If both objects are present, the accusative object becomes the direct object and the dative object becomes the indirect object. If either object appears by itself, the second respectively third rule applies and the object becomes a direct object. All rules in the rule file are tested from top to bottom, thus more specific rules are at the top, while generic rules that do not use tree context are at the bottom. Each rule is tested at all frontier nodes in the tree before the next rule is tested. If a rule is applied, the first rule is tested again. The alternative execution order of taking a single frontier node and testing the whole ruleset only on that node until it is converted is not used as it could result in missing out on a constellation where a more specific rule could have been applied if another state node would have been tested first.

**PoS tags** Example (f) shows how a dependency relation can be converted based on PoS tags, the PoS tag follows the node identifier separated by a period. Both rules in (f) convert the `APP` relation but also match the PoS tag of the parent node. The source tag set uses the STTS PoS tag set (Schiller et al. 1995), where `NE` refers to proper nouns and `NN` refers to nouns in general. Following this logic, an apposition attached to a proper noun is more likely to be a UD `appos` while an apposition attached to a regular noun is more likely to be part of a `compound` relation. This is the same principle explained in Section 3.3.4, where the PoS tag is used to distinguish `obl` and `nmod` dependents. Besides the dependency relation the PoS tag is the most important feature of a token, which is why it has explicit syntactical support.

**Groovy** The previous examples have shown how the local tree context can be accessed and the dependency relation and PoS tag of a token can be incorporated into rules. The CoNLL format specifies additional features for each node, such as its ID, the so called "feats", the lemma and miscellaneous features. These were not found to be as important respectively not suited for direct matching. However, node order or other features of the node might still contain relevant information for some edges cases. These features cannot be accessed with specific rule syntax, but the rules can contain a body, which is shown in example (g). This rule body can contain *groovy code*, groovy is a java based scripting language[1]. In this groovy code, the whole data structure can be accessed. The matched nodes are available as objects under the same identifier as used in the rest of the rule. The software uses the udapi framework internally, so the nodes are objects from this framework. These objects allow to access all token features read from the CoNLL file. Also the parent, sibling and child nodes of a token can be accessed, allowing to inspect the whole tree. The last expression in the rule body should return a boolean, if it returns *true*, the rule is applied, if it returns *false* the rule is not applied. Example (g) shows how the direction of an edge is taken into account for rule applicability, i.e. the `n` dependent should be to the right of its `p` regent. The groovy code can also access the new tree. The variables containing the nodes from the new tree, i.e. after conversion, are prefixed with an underscore. This makes it possible to set additional features after conversion or set a flag in the miscellaneous section of a node, marking it for further manual inspection later on.

## 4.2. Part of Speech Tag Expansions

Many decisions based on PoS tags work based on word classes, such as all nouns or all predicates. To allow for matching of different PoS tags all from the same class, it is possible to define lists of PoS tags like this:

```
expansion predicatePos =
    [VVFIN, VVPP, VVINF, VVIZU, ADJD, ADJA];
```

[1]http://groovy-lang.org/

The identifier `predicatePos` can then be used instead of a PoS tag, allowing to formulate a single rule instead of one rule for each PoS tag. The PoS tag used in the dependency tree needs to be contained in the list to make the tree match the rule. Formally, rules containing these expansions are prototypes for a number of rules, one rule per PoS tag in the list.

## 4.3. User Interaction and Interface

It is also possible to trigger user interaction in the rule by calling specific methods in the groovy code in the rule body. The software includes a user interface, which allows to inspect a conversion process of a tree, which can be used to analyse the process or debug a ruleset. Interactive conversion can be used in ambiguous cases, where the context is not sufficient to distinguish between two or more cases. An example use case is the distinction between `nmod` and `obl` usage in latest version of Universal Dependencies. The `nmod` relation should be used for dependents below nominals while the `obl` relation should be used for dependents below predicates. For the remaining cases a catch-all rule can be introduced which triggers interaction with a human annotator to decide the case; the rule might look like this:

```
n1:PP(n2:PN(?r2), ?r1) -> n2:xxx(n1:case(), ?r1, ?r2) :-
    {interactive.decideLabel(_n2, "nmod", "obl")};
```

This method call would trigger a pop-up in the graphical user interface giving the user a choice between the `nmod` and `obl` label for the node `n2`, while highlighting the affected edge in the tree in the interface.

# 5. Applying the Concept to the Hamburg Dependency Treebank

To align the implementation of the transducer closely to the needs of treebank conversion, the design of the transducer was intertwined with the conversion of the HDT. To get familiar with the requirements for a treebank conversion tool, 45 sentences were chosen from the HDT in such a way that each label from the schema was present at least three times. The trees were then converted manually while notes were maintained about which source structures match to which target structures. After all the trees were converted, a conversion ruleset was extracted from the notes. At this point, the software was not fully implemented, and was updated according to the new found requirements which came up when the rules were created.

## 5.1. The Sample Rule Set

The resulting ruleset consists of 58 rules, although none of the rules use the groovy code feature, all of them are based solely on the transducer mechanism. The expansion feature (Section 4.2) was used to define a list of PoS tags found on predicates and a list of PoS tags indicating a sibling adverbial clause.

The file is structured in multiple blocks to make it easier to keep track of the file contents. Each block is concerned with the conversion of a single dependency relation or a small number of related dependency relations. The high-level relations concerning the root of the tree, predicates and subclauses are at the top of the file, these rules are also the most complex ones. Rules are similar to Example (c) from Figure 4.1. Then optional dependents are treated, like obliques (`obl`) and nominal modifiers (`nmods`), also adverbials (`ADV`), adjectives (`ATTR`) and appositions (`APP`). From the same figure Example (b) and (f) are from this part of the ruleset. Lastly the HDT labels having a direct correspondence in UD are converted, the rules look like Example (a) from the figure.

Inside a block which usually consists of one to four rules, the rules are ordered from most specific to most generic, which means that special cases are treated first and in the end a general fall-back rule is triggered if nothing before matched the tree. This is exemplified in (e).

Some aspects of the rule set are discussed in detail below, to the various use cases that were covered with the different techniques.

**No Transformation, No Target Differentiation** The easiest conversions do not involve any restructuring of the tree, and also do not require to distinguish between multiple target labels. Labels that can be converted this way are `VOK` → `vocative`, `PART` → `mark`, `GMOD` → `nmod`, `DET` → `det`, `EXPL` → `expl`, `AVZ` → `mark`, `SUBJ` → `nsubj`, `NP2` → `nsubj`. Slightly more complex is the conversion of `ATTR`, in UD this label becomes either `amod` or `nummod`. The `nummod` cases can be identified by the `CARD` PoS tag, which makes for a clear-cut distinction as well.

**Multiple Target Labels** The `APP` label is difficult to convert. The HDT uses the `APP` label for any consecutive words in a noun phrase which are not determiner or attributes. In most cases these are actual appositions and should be converted to the `appos` label in UD. Other uses include connecting names and titles, suffixes like version numbers or other composite expressions like product and organization names. These are covered by the `flat` and `compound` relation in UD, which means that the `APP` label is split into three different labels in the target schema. From the experiences in the rule writing process, it can be said that distinguishing two different target cases is fairly easy, while three or more target labels becomes difficult. In this case, a distinction was made based on the PoS tags of the tokens. Names are connected with `flat` and carry the `NE` PoS tag, in contrast to other nouns (`NN`), which makes a distinction easy. Differentiating between appositions and compound expressions is more difficult, the evaluation later on showed that the simple heuristics used here were not sufficient to get a good precision score, but still did not cover all occurrences of appositions which lead to a poor coverage score as well.

**Objects** There are seven labels for marking objects in the HDT schema, distinguishing different cases and other phenomena. Most of these are mapped to `obj` and `iobj`. If an object appears by itself, it becomes `obj`. If the sentence features a ditransitive verb with two objects, one needs to become the direct object (`obj`) and the other becomes the indirect object (`iobj`). This distinction can be made by matching two objects at once. For other labels such as the object clause (`OBJC`) or the prepositional object (`OBJP`), non-object relations are used in the UD. The object clause is attached as a clausal complement (`ccomp`) and the prepositional object becomes an oblique or nominal modifier, because of the syntactic structure consisting of a noun and a case marker. Here the use of UD label subtypes should be considered, as the prepositional object is different from other oblique nominals. In contrast to the prepositional phrase dependents which are also attached as obliques and nominal modifiers, the prepositional object is a core dependent even though the syntactic structure suggests otherwise. The UD schema does not specify how to handle these cases, however the use of a `obl:arg` label might be useful to mark core oblique dependents.

**Nominal Modifiers** A widespread dependent subtree in the german language is the combination of a preposition and an additional nominal dependent as a modifier to a predicate or other nominal. In the sentence "Das Buch liegt <u>auf dem Tisch</u>" (*The* *book lies* <u>*on the table*</u>) the underlined phrase is such a nominal modifier, it modifiers the predicate "liegt". In the HDT these are annotated as `PP-PN`, where `PP` is the preposition ("auf"), a function word attached directly to the predicate, and the `PN` is the noun below it ("Tisch"). The determiner ("dem") and potential modifiers to the noun are all attached below the noun.

In this structure the function-head and content-head difference becomes apparent. The rules concerned with the transformation of these structures invert the function-head structure to a content-head structure and therefore requires multiple multiple nodes at different depths in the tree to match, specifically the function word and the noun below it. In UD the preposition is attached below the nominal, with the `case` relation and the noun is attached either as an oblique (`obl`) if the regent is a predicate – as it is the case in the previous example – or as a a nominal modifier (`nmod`) if the regent is a noun. This distinction is easy to make based on the PoS tag of the regent in most cases, exceptional cases where the regent is neither a predicate nor a nominal require the decision of human annotator.

**Tree Context** The `NEB` dependency relation is used to attach subclauses in the HDT. In UD subclauses are divided into core and non-core clausal dependents, which are either clausal complements (`ccomp`) or adverbial clauses (`advcl`). The ruleset makes This distinction by matching adjacent nodes in the tree and heuristically deciding if the clause is a core part of the sentence based on the presence of other core dependents at the same level as the subclause. If there are objects at the same level, the subclause is assumed to be a non-core dependent, which works in a lot of cases.

**Cross-Frontier Modifications** The `AUX` and `PRED` relations are converted with the cross frontier modification mechanism outlined in Section 3.3.5. Due to not knowing about the presence of an aux dependent while the regent is still converted, the rules concerned with the conversion of aux dependents need to restructure the the previously converted regent again, which requires a dedicated mechanism. This mechanism allows for a smaller number of rules. It also separates the conversion of the `aux` relation from the conversion of any possible parent relation, following the software principle of *separation of concerns.*

## 5.2. Evaluation & Analysis

The relevant metrics for the evaluation of such a ruleset are the coverage and the precision. How much of the tokens in the treebank is covered by the rules in the ruleset and are the converted nodes converted correctly? The coverage can be evaluated fairly easily by applying the ruleset to the whole treebank and counting the nodes which were not converted. The precision is more difficult to evaluate, since it requires gold standard annotations to compare to. Each evaluation metric and its experimental setup will be discussed separately. For coverage as well as precision the software includes a command that computes relevant statistics about the generated trees in comparison to the original data respectively the manually annotated trees.

### 5.2.1. Coverage

To test the coverage the ruleset was applied to part B[1] of the treebank, which consists of 100k sentences and had not been used in the rule creation process. 91.5% of the tokens were converted successfully and out of the remaining 8.5%, 1.7% could not be converted because no matching rule was found, and four times as many (6.8%) were below another nonconvertible node. Because of the top down conversion, any node below a node that cannot be converted, cannot be converted as well, which means a few nonconvertible structures can block large subtrees from being converted. To increase coverage, the relevant dependency relations are only the ones causing the blocks in the process. For these dependency relations all rules in the ruleset require too much context which means rarer edge cases with uncovered context cannot be converted. While every dependency relation was found in the nonconverted subtrees, only 6 relations were responsible for the blocks.

Out of the 28.5k tokens which were responsible for stopping the conversion process, 59% were attached with an `APP` relation that was not converted. The `APP` label was converted in 51% of the cases, 21% were not converted because no applicable rule was found and the remaining 28% were in subtrees which were not converted. This can be explained by long apposition chains, of which the first element cannot be converted which leaves the remaining labels nonconverted as well. The rules concerned with `APP` conversion all include PoS tag constrains on the token and sometimes PoS tag constrains for the parent node as well; a generic fallback rule would drastically increase coverage. Also a rule covering the PoS tag for foreign words should be introduced, all the product and organization names with English names where the tokens are connected via `APP` relations are not converted, because the PoS tag does not match the other rules.

The second biggest blocker relation is the `KON` relation for conjunctions, with 18% followed by `CJ`, conjunct, with 9%. The ruleset only includes a rule for converting these two together, if they appear separately a conversion is not possible. `PN` was responsible for another 9% of conversion stops, and here the same problem is the cause, `PN` is only converted with a `PP` parent. The same holds true for `KOM` and `OBJC`, these two labels are responsible for the remaining 4% of blocks.

The table containing the previously mentioned numbers can be found in Appendix A.1.

### 5.2.2. Precision

To test the precision of the ruleset, a gold standard comparison was needed. 50 sentences were selected randomly from the treebank – excluding the trees which were used to create the ruleset – and annotated manually. The sample included a variety of sentences ranging from short sentence fragments with only three tokens to sentences with more than 20 tokens. While the trees were converted manually, annotation errors in the source trees were fixed as to not have source annotation errors influence the precision measure of the transducer. Fixing annotation errors is a different task than transforming the annotations. The transducer only transforms existing patterns into new patterns and

[1]Part A contains the sentences used to create the ruleset

is not responsible for deciding on the quality of the source annotation, the annotation errors would occur again in the target treebank, in transformed structures.

The manually created test set was then compared to the trees generated by the transducer to measure the correctness of the transducer. In this process some obvious errors in the manually annotated UD trees were found, such as confusing `nmod` and `obl`. Again, as the trees should function as a gold standard and not distort the actual performance measure of the transducer, these errors were fixed before the precision metrics were calculated.

It should be noted that for some phenomena the UD annotation guidelines are not clear. There are no guidelines for German in particular, which means it is not always obvious which annotation is the correct one. Particularly the guidelines regarding appositions are not as elaborate as they should be. The HDT uses the `APP` label to cover appositions as well as connecting names of persons and organizations. These structures are covered by `appos`, `flat` and `compound` in UD, but it is not always definite which label should be used. Many of the incorrect conversions were not completely wrong but both versions, the generated and the manually annotated one seemed plausible.

The 50 sentences forming the test set had a total token count of 782, which includes 84 punctuation nodes, which were not included in the transformation process. Out of the remaining 698, 662 were converted and 610 were converted correctly. This yields a coverage of 89% for this smaller test set as well as a precision of 92%, 23 sentences were entirely correct. Looking at individual dependency relations, all the very frequent ones have high precision scores, the relations having between 92% and 98% precision make up 78% of all converted tokens. `DET` and `PP` are always converted to the correct target label, but they are attached wrongly in rare instances. While `DET` is always converted to `det`, `PP` is converted to either `case` or `advmod` in some instances. They have a correctness of 96.4% and 97.5% respectively, together they make up over 30% of all tokens converted tokens in the test set. There are seven other labels in the same range, all of which common as well, such as `S`, the root of the tree, `SUBJ`, `OBJA` and `AUX`, essential parts of a sentence and `ATTR`, `GMOD` and `PN`, common modifiers to other tokens. The errors found here are the most difficult ones, as the structures which are not converted correctly are unusual exceptions to the rules.

The rarer labels are found at both ends of the precision spectrum, they are either converted entirely correctly or a few errors affect the metric heavily. Nine Labels were converted correctly in 100% of the cases, together they make up 14% of the converted tokens. Some of the labels are `EXPL`, `OBJC`, `OBJD` and `AVZ`. Based on the occurrences which were in the test data, the rules for these labels look sufficient.

The eight remaining labels have less than 90% precision and the labels occur less frequently as well. More examples have to be taken into account to find conversion patterns for these labels, which include `PART`, `NEB`, `SUBJC`. The `KONJ` relation appeared only eight times, it is always mapped to `mark` but was attached wrongly in one instance, yielding a correctness of 87.5%. Two more frequent labels which still end up with a bad precision score are `APP` and `PRED`. The `APP` label was already discussed above in relation with the coverage; the rules for that label have to be completely reworked. For this there needs to be a clear definition of when to use which target label. In the gold standard annotations, the `APP` and `PRED` relations were mapped to four different target labels each. For this many distinctions, more rules are needed.

The `KOM` relation for comparisons is never converted correctly. In the ruleset it is treated similar to the `KON` relation, which is for conjuncts, which only makes sense for phrases like “genauso hoch wie lang” (*As high as long*). In sentences like “[...] sollte benutzt werden wie ein Telefon.” (*[...] should be used like a telephone.*) it should be treated similarly to prepositional phrases in that it should be converted to `nmod` and `obl` structures, with the “wie” being attached as a case. Here it would make sense to use subclasses to the `nmod` and `obl` label to mark the comparison, such as `nmod:comp` and `obl:comp` respectively. The test data contains multiple instances of “mehr als” (*more than*), where the `KOM` relation was not converted because it does not appear together with a `CJ` dependent.

The word “wie” which is usually attached as `KOM` also points to multiword expressions (MWEs), which are not covered at all in the ruleset. In the manually annotated data, “darüber hinaus” and “mehr als” were annotated as MWEs by using the `fixed` label to connect the tokens. In the HDT the “als” of “mehr als” is connected as a `KOM` because it is a comparative expression. The HDT does not have a notion of MWEs, which makes it difficult to extract them from the source annotations. Introducing special treatment for MWEs in the ruleset would cover this issue, a possible solution would be lexicalization of rules by using groovy code.

Another construct which cannot be detected from context but could be covered with lexicalization is the inherently reflexive pronoun. The test set contains a single case of this, “sich beeilen” (*hurry up*). The guidelines state that here the `expl` label should be used to attach the pronoun to the predicate. In not inherently reflexive cases, the pronoun is attached as an object, but distinguishing inherently reflexive cases from not inherently reflexive cases based on tree context is not possible. Besides lexicalization relying on manual decisions would also be plausible, as the number of reflexive pronouns might be fairly low. This also prompts to further developments in the manual conversion process, to allow for fast and clear decisions.

Other issues include the use of the `S` relation *inside* the tree instead of just at the top, which can be detected and the `ccomp` label can be used instead of the `root` label. This is usually the case for direct speech, where the uttered phrase is a complete sentence in itself, but is also an argument to the predicate above it. This means it is unclear if it should be attached as `nsubj` or as `expl`.

The swapping, especially cross frontier modifications, makes it difficult to analyse the structures statistically. Some of the errors in other labels than the APP label are actually from one rule related to APP conversion which also does cross frontier modification.

# 6. Conclusion & Outlook

This work presented a framework for treebank annotation schema conversion consisting of a tree transducer implementation, a syntax for formulating rules as well as a number of extensions to the transducer formalism. The tree transducer is complimented by a graphical user interface for human annotation, facilitating a semi automatic conversion process which combines the strengths of automatic conversion with the human decision process in edge cases.

The experiments demonstrate that a relatively small ruleset can already achieve valuable results, analysis of the first conversion results shows that room for improvement exists and points to specific issues which can be tackled by means of the framework. Specifically, the coverage experiment shows that only a small number of dependency relations block the conversion process and only one fifth of the nonconverted nodes could not be converted because of missing rules. The coverage can therefore be increased with little effort, and the precision experiments also pointed to specific issues which can be solved with additional rules. The rules did not use the groovy code feature or interactive conversion, which should be used to tackle edge cases and exceptions.

The manual conversion of trees to create and test the ruleset surfaced the need for distinct German annotation guidelines for Universal Dependencies where German language constructs are covered specifically. The experiments also showed that manual conversion is error prone and automatic conversion should be used as much as possible. Annotation errors in the source treebank were found, as well as errors in the manually annotations were made which were discovered later on but show the fragility of the process. This tool will make it easier for people without programming knowledge to convert a treebank automatically.

This work was also intended to evaluate whether tree conversion based on local context only is viable, and the results show that it is. The locality of word groups in a sentence can easily be captured in a rule based system, making the transducer mechanism a good fit for this task. The top-down nature of the formalism helps thinking about the transformation process, in contrast to systems which match structures at arbitrary points in the tree.

Future work should focus on the conversion of a complete treebank to evaluate the effectiveness of the tool on an actual large-scale level. The ruleset used as a basis for the experiments can be extended and used to convert the HDT entirely. The rules should transfer as much information from the HDT schema as possible, the use of relation subtypes should be considered, for example an `obl:arg` label for prepositional objects (`OBJP`). In this regard, the tool needs to be extended to support subtyping of dependency relations, the subtypes are marked by colons which clashes with the current rule syntax. The use of more morphological features should be explored, in particular for distinguishing between `compound` and `appos`. For the rule bodies using groovy code, it should be possible to define common variables which can be used across all rule bodies, as to allow to define lists which can be reused in multiple rules or similar convenience definitions. A complete conversion of the HDT would also include the splitting of words into syntactic words, for example splitting “zum” into “zu” und “dem”. Also the PoS tags still need to be converted, both of these tasks need to be treated with additional tools. The tool could be embedded into a block in the udapi framework, easing its integration into existing conversion pipelines and giving it more exposure in the community.

After more extensive use of the tool, an iterative rule writing workflow should emerge which should then be supported by additional software developments, manual edits to the converted trees should be kept intact while still allowing to apply the transducer again after the ruleset was updated. The basic building blocks of the user interface and the existing statistical evaluation tools provide a basis to continue the development of the ruleset from its current state.

Introducing machine learning techniques into the conversion process is another potential area of future research the rules could be learned from a basis of manually converted trees, or rules could be learned from decisions made by a human annotator interactively.

Code and data is available under:
`http://nats.gitlab.io/truducer`

# Bibliography

Ahrenberg, Lars (2015). “Converting an English-Swedish Parallel Treebank to Universal Dependencies”. In: *Proceedings of the Third International Conference on Dependency Linguistics (Depling 2015).* Uppsala, Sweden: Uppsala University, Uppsala, Sweden, pp. 10–19. URL: `http://www.aclweb.org/anthology/W15-2103` (cit. on p. 5).

Foth, Kilian A. et al. (2014). “Because Size Does Matter: The Hamburg Dependency Treebank”. In: *Proceedings of the Language Resources and Evaluation Conference 2014.* LREC. European Language Resources Association (ELRA). URL: `http://http://nats-www.informatik.uni-hamburg.de/HDT/` (cit. on p. 2).

Hennig, Felix and Arne Köhn (2017). “Dependency Tree Transformation with Tree Transducers”. In: *Proceedings of the NoDaLiDa 2017 Workshop on Universal Dependencies, 22 May, Gothenburg Sweden.* 135. Linköping University Electronic Press, Linköpings universitet, pp. 58–66 (cit. on pp. 5, 15).

Johannsen, Anders, Héctor Martínez Alonso, and Barbara Plank (2015). “Universal Dependencies for Danish”. In: *Proceedings of the Fourteenth International Workshop on Treebanks and Linguistic Theories (TLT14).* Ed. by Markus Dickinson et al. Warsaw, Poland, pp. 157–167 (cit. on p. 5).

Maletti, Andreas (2010). “Survey: Tree Transducers in Machine Translation”. In: *Proc. 2nd Int. Workshop Non-Classical Models of Automata and Applications.* Ed. by Henning Bordihn et al. Vol. 263. `books@ocg.at`. Österreichische Computer Gesellschaft, pp. 11–32 (cit. on pp. 7, 10, 11).

McDonald, Ryan et al. (2013). “Universal Dependency Annotation for Multilingual Parsing”. In: *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers).* Sofia, Bulgaria: Association for Computational Linguistics, pp. 92–97. URL: `http://www.aclweb.org/anthology/P13-2017` (cit. on pp. 2, 5).

Nivre, Joakim (2014). “Universal Dependencies for Swedish”. In: *Proceedings of the Swedish Language Technology Conference (SLTC).* Uppsala, Sweden: Uppsala University, Uppsala, Sweden (cit. on p. 5).

Øvrelid, Lilja and Petter Hohle (2016). “Universal Dependencies for Norwegian”. In: *Proceedings of the Tenth International Conference on Language Resources and Evaluation LREC 2016, Portorož, Slovenia, May 23-28, 2016.* Ed. by Nicoletta Calzolari et al. European Language Resources Association (ELRA). URL: `http://www.lrec-conf.org/proceedings/lrec2016/summaries/462.html` (cit. on p. 5).

Pyysalo, Sampo et al. (2015). “Universal Dependencies for Finnish”. In: *Proceedings of the 20th Nordic Conference of Computational Linguistics (NODALIDA 2015).* Vilnius, Lithuania: Linköping University Electronic Press, Sweden, pp. 163–172. URL: `http://www.aclweb.org/anthology/W15-1821` (cit. on p. 6).

Ribeyre, Corentin, Djamé Seddah, and Éric Villemonte de la Clergerie (2012). “A linguistically-motivated 2-stage Tree to Graph Transformation”. In: *Proceedings of the 11th International Workshop on Tree Adjoining Grammars and Related Formalisms (TAG+11).* Paris, France, pp. 214–222. URL: `http://www.aclweb.org/anthology/W12-4625` (cit. on p. 6).

Schiller, Anne et al. (1995). *Guidelines für das Tagging deutscher Textcorpora mit STTS.* Tech. rep. Universität Stuttgart / Universität Tübingen (cit. on p. 23).

Tandon, Juhi et al. (2016). “Conversion from Paninian Karakas to Universal Dependencies for Hindi Dependency Treebank”. In: *Proceedings of the 10th Linguistic Annotation Workshop held in conjunction with ACL 2016, LAW@ACL 2016, August 11, 2016, Berlin, Germany.* Ed. by Katrin Tomanek and Annemarie Friedrich. The Association for Computer Linguistics. URL: `http://aclweb.org/anthology/W/W16/W16-1716.pdf` (cit. on p. 5).

Tyers, Francis M. and Mariya Sheyanova (2017). “Annotation schemes in North Sámi dependency parsing”. In: *Proceedings of the Third Workshop on Computational Linguistics for Uralic Languages.* St. Petersburg, Russia: Association for Computational Linguistics, pp. 66–75. URL: `http://www.aclweb.org/anthology/W17-0607` (cit. on p. 6).

# A. Appendix

## A.1. Coverage Table

| Dependency Relation | total cut | converted | blocking | remaining |
|---|---:|---:|---:|---:|
| APP | 4.78% | 51.42% | 20.52% | 28.06% |
| OBJC | 0.26% | 85.44% | 10.42% | 4.15% |
| KON | 3.28% | 79.44% | 9.25% | 11.31% |
| KOM | 0.69% | 86.42% | 6.89% | 6.70% |
| CJ | 3.23% | 85.79% | 4.72% | 9.49% |
| PN | 12.13% | 93.07% | 1.23% | 5.70% |
| GRAD | 0.05% | 90.51% | 0.00% | 9.49% |
| NP2 | 0.02% | 91.90% | 0.00% | 8.10% |
| REL | 0.97% | 91.96% | 0.00% | 8.04% |
| ADV | 8.10% | 92.44% | 0.00% | 7.56% |
| VOK | 0.00% | 92.65% | 0.00% | 7.35% |
| PAR | 0.04% | 93.05% | 0.00% | 6.95% |
| GMOD | 2.50% | 93.34% | 0.00% | 6.66% |
| ETH | 0.09% | 93.35% | 0.00% | 6.65% |
| PRED | 1.17% | 93.53% | 0.00% | 6.47% |
| ATTR | 8.13% | 93.66% | 0.00% | 6.34% |
| DET | 13.94% | 93.90% | 0.00% | 6.10% |
| AUX | 3.74% | 94.23% | 0.00% | 5.77% |
| EXPL | 0.10% | 94.29% | 0.00% | 5.71% |
| OBJG | 0.02% | 94.36% | 0.00% | 5.64% |
| PP | 12.03% | 94.45% | 0.00% | 5.55% |
| ZEIT | 0.37% | 94.52% | 0.00% | 5.48% |
| OBJD | 0.46% | 94.86% | 0.00% | 5.14% |
| KONJ | 0.98% | 94.97% | 0.00% | 5.03% |
| PART | 0.60% | 95.00% | 0.00% | 5.00% |
| SUBJ | 8.29% | 95.14% | 0.00% | 4.86% |
| OBJA | 4.55% | 95.22% | 0.00% | 4.78% |
| OBJA2 | 0.05% | 95.28% | 0.00% | 4.72% |
| NEB | 0.77% | 95.33% | 0.00% | 4.67% |
| OBJP | 0.46% | 95.33% | 0.00% | 4.67% |
| OBJI | 0.42% | 95.44% | 0.00% | 4.56% |
| SUBJC | 0.21% | 95.82% | 0.00% | 4.18% |
| AVZ | 0.68% | 96.94% | 0.00% | 3.06% |
| S | 6.88% | 99.91% | 0.00% | 0.09% |

# Erklärung der Urheberschaft

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Bachelorstudiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Ich stimme der Einstellung der Arbeit in die Bibliothek des Fachbereichs Informatik zu.

Ort, Datum Unterschrift