**Julius–Maximilians–Universität Würzburg**
**Institut für Informatik**
Lehrstuhl für Künstliche Intelligenz und Wissenssysteme (Informatik VI)

# Automatic Correction of Syntax Errors with Methods of Parsers

# Automatische Korrektur von Syntaxfehlern mit Methoden von Parsern

Master's Thesis submitted by

**Jonas Buchwald**

**Julius–Maximilians–Universität Würzburg**
**Institut für Informatik**
Lehrstuhl für Künstliche Intelligenz und Wissenssysteme (Informatik VI)

# Automatic Correction of Syntax Errors with Methods of Parsers

# Automatische Korrektur von Syntaxfehlern mit Methoden von Parsern

Master's Thesis submitted by

**Jonas Buchwald**
geboren am 29. July 1997 in Straubing

Angefertigt am
Lehrstuhl für Künstliche Intelligenz und Wissenssysteme (Informatik VI)
Julius–Maximilians–Universität Würzburg

Gutachter:
Prof. Dr. Frank Puppe
Prof. Dr. Dietmar Seipel

Betreuer:
Dr. Markus Krug

Abgabe der Arbeit:
01.12.2021

## Erklärung

Ich versichere, die vorliegende Master's Thesis selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur verfasst zu haben. Darüber hinaus versichere ich, die Arbeit bisher oder gleichzeitig keiner anderen Prüfungsbehörde zur Erlangung eines akademischen Grades vorgelegt zu haben.

Regensburg, den 30.11.2021

(Jonas Buchwald)

**Abstract.** This thesis presents an error recovery system for common parsing technologies, which enable a user to detect and correct syntactical errors in sentences. In addition, a grammar extension system is introduced, enabling a semi-automatic generation of grammars using sentences of the desired domain. Furthermore, the work provides a brief overview of the most common bottom-up parser CYK and top-down parser Earley, which are the fundamental algorithms for the error recovery system. Parsing technologies are based on a given grammar and decide if a sentence is part of this grammar. The parser tries to match the words of the sentence with the literals in the grammar, forming rules and returning a parsing tree for the given sentence. If the sentence contains an unknown word or sequence, the parser rejects the sentence. Few papers are focusing on robust parsing technologies, and the implemented technologies mainly cover one major language. This thesis focuses on creating two algorithms, which are language independent. Nevertheless, this work covers the German language. The CYK parser is extended with a brute force error recovery system. Due to the powerfulness of the algorithm, pinpointing the exact position of an error is difficult. Once a sentence cannot be parsed entirely, the error recovery system is called and then adds, deletes, and swaps tokens in the sentence to find a possible improvement. For adding tokens, the Stuttgart-Tübigen-Tagset is used as it is common for a part-of-speech tagger for German. The tagset contains about 54 different tags, which are all added to the sentence. During the brute force algorithm, every possible improvement is computed. Each token is deleted, swapped to every possible position, and all tags are added. After each operation, the sentence is parsed again, and if an operation is successful, the error recovery system returns the improvement to the user. The algorithm is designed to find a maximum of two errors in the sentence but starts by computing only one error. If this is unsuccessful, the sentence is parsed again for two errors.

The Earley parser is extended with a similar error recovery system. First, the standard algorithm parses the sentence, and if the resulting Earley table is incomplete, the error recovery system is called. Then, the system identifies the corrupted chart in the Earley table and performs the three operations at this chart. The missing terminals are added to the chart, the current token is removed, or the token is swapped in the sentence. After these operations, the Earley parser is called again with the altered sequence. If a successful improvement is found, it is returned to the user.

At last, a semi-automatic system for expanding a grammar is presented. The CYK algorithm is used as the fundamental algorithm, and if it returns an incomplete parsing tree, the system is called. It then matches the incomplete parsing sequence with the rules of the grammar. The algorithm tries to find rules, which can be extended to cover the unknown sequences. If a possible rule is found, the suggestion is returned to the user.

In order to present the functionality of all three algorithms, they are tested and evaluated against a proof-of-concept domain. The evaluation shows the potential of the technologies as well as their flaws. Nevertheless, the algorithm meets the expectations and returns even unexpected correct improvements on several occasions.

The thesis provides a reasonable basis for further development and enhancement of the three algorithms.

# Contents

# 1 Introduction

With the rising globalization in the world, people need to speak foreign languages to communicate. Conducting a communication when one interlocutor is not fluent in the language can lead to problems. In Germany, around eleven million foreign workers are doing their job every day [1], which means around 13% of all workers are from another country [2]. Nevertheless, communication at the workplace cannot be compromised by language barriers. In a direct conversation, two parties may exchange opinions and overcome some barriers because they can correct each other or ask questions about the intended meaning. However, written communication can result in a more complex and challenging situation. The receiving person does not understand the text because of syntactically and semantically errors, or the person is not fluent enough to understand the meaning. The second problem can only be solved by asking coworkers about the meaning and expanding the vocabulary. Whereas a spell checker can tackle the first problem by the sender before the text is sent to the receiver.
Using technologies like that can enhance the writing process and the created text. However, most spell checkers only focus on the semantical part of the sentence. They analyze if the correct verb form is used or if the article is suited for the noun. Only a few also analyze the syntactical part, which means if the words are in the correct order or if the structure needs to be altered. In Germany, two significant problems meet: on the one hand, a large number of foreign speakers and a language with complex and challenging syntax. According to a UNESCO study, German is one of the most complex languages to learn [3] because the complex and unique syntax impede the learning process, and many people struggle with it. Using a spell checker and a syntax checker would help people master the language faster and more efficiently.
Some approaches focus on syntactical error correction in documents, but most are designed for significant languages. Unfortunately, German is not widely spread in the world, and only 130 million people have German as their mother tongue [4]. Due to the language's complexity and insignificance, the implemented approach does not cover German but more frequently spoken languages like English, Chinese, Japanese, or Spanish. Adapting a working technique might be a difficult task because the German syntax is unique and complex, and if the approach uses artificial intelligence, the algorithm learns the syntax of the desired language.
For a perfect syntax correction tool, the algorithm has to be language independent and yet efficient in helping the user correct the mistakes in the sentence. Parsing technologies can be used as the fundamental algorithms to become language-independent. A parser is designed to decide if a sentence is part of the language by using a grammar. The grammar can be designed and cover a wide variety of topics from different languages or domains. The parser is equipped with a grammar, and it can iterate over sentences deciding which are covered by the grammar. That results

in a language-independent system, which can be adapted to different domains as well. Parsing technologies have higher traceability than other artificial techniques because they follow a strict pattern during their parsing process, which a human can understand. Therefore, it is also possible to troubleshoot and analyze mistakes made by the parser.
Syntactical problems can be identified quickly using a parsing algorithm because the parser cannot complete the sentence and return an error. However, returning a meaningful and helpful message to the user is complex. Highlighting the exact position of an error can be an arduous task depending on the parsing technique. Nevertheless, there is always the possibility that the sentence is correct and the grammar is flawed or missing essential rules. The vital fact is returning good feedback highlighting the error in the sentence or grammar to the user.
This thesis addresses the question if it is possible to enable a parser to:

- mark problematic sequences in a sentence?
- give insightful feedback to the user about error correction?
- make possible suggestions for adding rules to the grammar

This work is structured as follows. Chapter 2 provides the background information for this thesis. The two iconic parsing technologies of natural language processing are presented and explained. The Related Work chapter summarizes the current state of the science and highlights both the importance of the work and the differences with the science. In Chapter 4, a proof-of-concept domain is presented, and a grammar covering this domain is created, which is later used for the evaluation of the algorithms. The next chapter explained the theory of the advanced parsing algorithm and visualized the techniques on several examples. Chapter 6 evaluates the created algorithm with the proof-of-concept domain and compares the algorithm based on their results. Finally, an outlook for future works is laid out.

# 2 Background

This chapter covers the basics of this thesis. Parsing techniques can be split into two major groups, top-down and bottom-up parsing. The workflow of the standard techniques from each group in the domain of natural language is presented. An example sentence is parsed to visualize the operating principle. At first, the preconditions needed for parsing will be explained.

## 2.1 Part-of-Speech Tagging

Neither parsing technologies nor neural networks can work directly on a raw document. The sentences in the document must be preprocessed. Usually, a parser processes one sentence until it is parsed correctly or reaches a dead end. Therefore, the document must be split into separate sentences.
The next step is to tokenize and tag the sentence. The tagger splits the sentence by the most frequently used delimiters like white spaces, commas, exclamation, and question marks. The delimiters themself are used as tokens since they perform a syntactical meaning in a sentence. Only the white spaces are removed.
Theoretically, the parser can work directly with these tokens. The grammar of the parser must cover each token, and then the sentence is parsable. This procedure might work with an example sentence and show the functionality of a parser. Nevertheless, covering all possible words in a language is impossible. Therefore, part-of-speech tagging (short: PoS tagging) [5] is used to determine the token type. During the procedure, every token is assigned a label based on its role in the sentence. The tagger uses the previous and next token in the sentence to predict the label of the current token. Usually, in this process, Hidden-Markow-Models [6] are used to achieve the results.
A PoS tagger needs a predetermined tag set to label the tokens correctly. In this work, the Stuttgart-Tübingen-Tagset (STTS) [7] is used to determine the type of a token. The tag set consists of 54 tags and is the most frequently used tag set in the German language.

## 2.2 Parsing

After preprocessing and tagging a sentence, a parser works with the tokens and returns a possible parsing tree for this sentence. This workflow is independent of the underlying parsing technique. The created parsing tree represents the found relations between the tokens of the sentence. Usually, the relations are in a hierarchical structure. The PoS tags from the tokens are at the bottom, and the higher the structure, the more general the relations become. These found connections are also presented in tags extracted from the grammar. Ideally, a sentence tag is returned after the parser is finished. Otherwise, the parser most likely failed to build a correct parsing tree. If the grammar is not precise enough, the parsing tree might be incomplete, and there is no sentence tag at all.
The focus of this section is to present the two dominant parsing techniques used as a foundation for the thesis.

### 2.2.1 Grammar

In the field of parsing, the algorithm itself is straightforward and follows a strict structure. The intelligence of the parser is configured with the used grammar. Based on it, the algorithm can decide if the sentence is part of the language or not. A grammar can be defined differently. In the manner of this work, a context-free grammar [8] is used by all parsing technologies. In context-free grammar, each rule has the form of $A \rightarrow \alpha$ with A as a single non-terminal. However, $\alpha$ can be a sequence of non-terminals and terminals or an empty sequence marked with $\epsilon$. The grammar is context-free because its right-hand side can replace the left-hand side at every fitting situation independent of the surrounding.
Additionally, the rules are extended with several operators. If a token of a rule has a "*"-operator behind it, the token may not appear at all or an arbitrary number of times. If the "+"-operator is used, the token has to appear at least once. The token might not appear or once using the "?"-operator. The two last operators reduce the number of rules needed to represent grammar and for a more precise presentation. The "|"-operator is a logical or-operator. For example, the rule $A \rightarrow B|C$ is completed if one B or C is the current token. Finally, brackets can be used to structure reoccurring sequences or keep the grammar neat.

### 2.2.2 Top-Down Parsing

The first technique of the two dominant parsers covers the domain of top-down parsing. The hierarchy of the parsing tree is built from top to bottom. At first, all rules resulting in the top-level tag are extracted from the grammar. Then the sentence is parsed while trying to achieve one of the starting rules. Once a starting rule is finished, the sentence is part of the language and is accepted. The parser identifies the rules in a prefix order.
In case the parser cannot find an applicable rule, the parsing process is stopped. Some parsers can use backtracking to try and find the error in the parsing. Back-

tracking [8] means the parser goes back in the already created parsing tree and searches for other possible solutions to label the tokens. If backtracking fails, the sentence is labeled as not parsable.
In the following section, the iconic Earley parser will be introduced and explained. In the field of natural language top-down parsing, this parser is the best-known.

### Earley Parser

The Earley parser was first introduced in 1968 by Jay Earley [9]. The algorithm is designed to parse input strings by using context-free grammars. The parser uses charts to determine the label of a token of a sentence. These charts are dynamically programmed and are altered during the parsing process. The author describes the Earley parser as a breadth-first top-down parser with a bottom-up recognition [8]. The parser is capable of handling left recursion. However, it struggles with $\epsilon$-rules. The given grammar should be adjusted in order to avoid $\epsilon$-rules.
During the parsing process, the Earley algorithm uses charts or sets. A chart contains different items (also called Earley items). An item is a grammar rule with a marker added to its right-hand side. The marker indicates which part of the rule is already identified. All literals to the maker's left-hand side have been recognized and parsed. The literals to its right-hand side still need to be processed. Usually, the marker is represented as a dot or star. For example, in the rule "A $\rightarrow$ a B $\cdot$ C," the literals "a" and "B" have been processed and recognized by the parser. However, the literal "C" has not been recognized yet but is the next in line to be parsed.
Depending on the position of the marker, the item has a different meaning for the parser: [8]

- Is the dot at the end, the right-hand side has been recognized, and the rule is parsed completely and marked as reducible. Therefore, the item is then called an *reduce item*
- If the marker is at the beginning directly after the arrow, the item is called *predicted item* because the item has not been parsed yet and just been predicted. The prediction process will be explained in the following
- A *shift item* has the dot in front of a terminal and allows a shift of the terminal.
- If the marker is directly before a terminal, it is called *prediction item* because it leads to a prediction.
- Is the marker at any other position, the item is called *kernel item* since parts have already been processed and recognized.

The general parsing process can be described in three phases the completer, scanner, and predictor. These three alter the charts of the parser and search the grammar for fitting rules. The algorithm iterates through all items of the chart. Depending on the item and position of the marker, one of the three phases is used. If the phases find applicable rules, they are added to the chart, and the iteration is expanded to process the newly added items. The Earley parser contains $n+1$ charts with n equals the number of tokens in the sentence. Each Earley item stores the chart index, in

which it has been added.
At the start of each iteration, the item is checked for completion. If the item is a reduced item and is recognized completely, the algorithm iterates through the chart in which the item was initially added. The left-hand side of the finished rule is used to find processed rules. Every rule with a marker directly in front of the finished left-hand side terminal is added to the current chart. The marker is then moved over the processed terminal. Once all rules of the old chart are checked, the iteration of the current chart continues with the added rules.
If the item is not completed yet and it is a shifting item, the scanner is called. The scanner verifies if the current terminal is equivalent to the current token of the sentence. If the two match, the marker is moved over the terminal. Finally, the changed rule is added to the following chart.
Otherwise, the predictor is called because the item then must be a prediction item. The predictor searches the grammar for every rule, in which the left-hand side is equivalent to the current terminal. All matching rules are added to the current chart as an Earley item, and the iteration is expanded with the newly added items.
If all phases cannot alter the charts anymore, the algorithm is finished. The parsing process is successful if the parser stops at the last chart and completes an item with a sentence literal on the left-hand side. If there is no sentence literal or the parser stops in an earlier chart, the parsing process fails, and the given sentence does not match the grammar.
In order to get a better understanding of the Earley algorithm, the sentence "I saw the saw" is exemplary parsed. As mentioned in Section 2.1, a language cannot be covered by a context-free grammar because the number of terminals is immense. Commonly, a PoS tagger would be used on the sentence, and the results are the non-terminals. However, for this example, the sentence is not tagged, and the terminals are covered with rules in the grammar. With the following grammar, the sentence is parsed:

```
S   —> VP  NP
NP  —> ART N
VP  —> N   V
N   —> I
V   —> saw
ART —> the
N   —> saw
```

Five charts are needed to parse this sentence. The sentence has four tokens, and the number of charts is token size +1. At the beginning of the algorithm, the charts are created accordingly to Figure 2.1.

Figure 2.2 displays the parsing of the first chart of the Earley algorithm. At the start, the given grammar is searched for any possible rules with a starting literal on the left-hand side. In this case, the starting literal is the sentence terminal "S". The rule gets written in the chart as an Earley item. The marker is added at the first position of the right-hand side. In this example, it is represented with a star. The index of the chart, in which the item is added, is stored in the brackets behind the rule. With a starting rule added to the first chart, the algorithm starts parsing

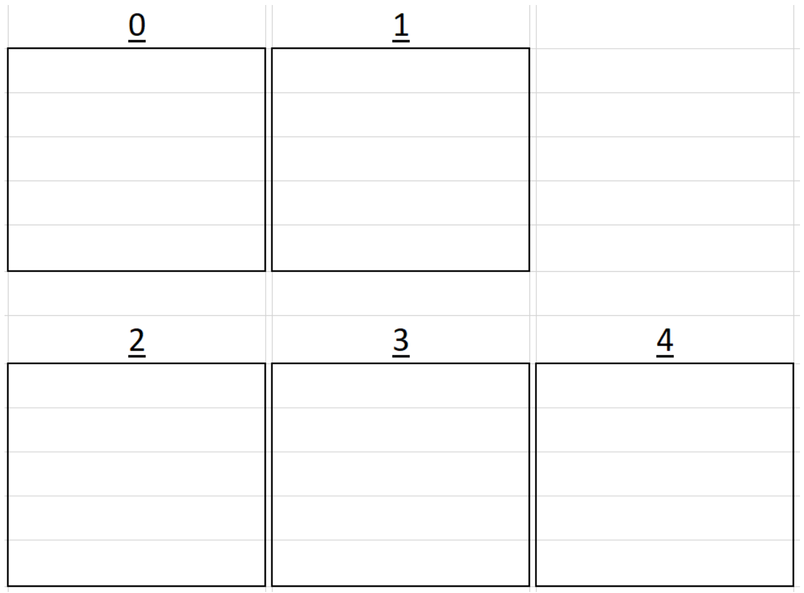


Figure 2.1: Empty Earley table

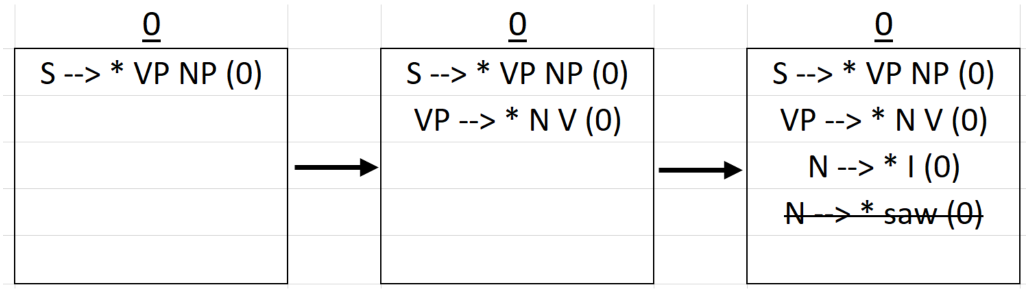


Figure 2.2: First step of the Earley algorithm

the first chart. The parser iterates over all items in the chart. At first, it verifies if the item is already completed. Because this is the starting rule, the item is not completed yet, and the current token is examined if it is a terminal. In this case, it is a non-terminal. Consequently, the predictor is used to expand the chart. The predictor searches the grammar for each rule containing the non-terminal "VP" on the left-hand side. Every found rule is added as an Earley item to the chart. Once the predictor has finished, the iteration process continues.
The newly added item is parsed accordingly to the starting rule. It is not yet completed, nor is the current token a terminal. The predictor is called to find new rules in the grammar. Now two rules match the conditions and are both added to the chart.
The new item is not finished, but the current token is a terminal. Hence, the scanner is called. The scanner examines if the terminal matches the current token of the sentence. The current token is "I" and matches the terminal. The marker is now moved behind the token, and the modified item is added to the following chart. The second item is deleted from the list because it is not used.

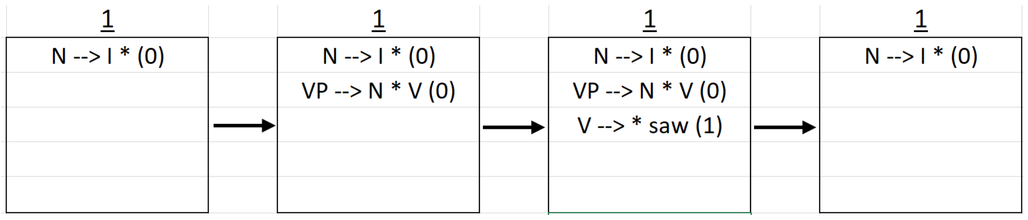
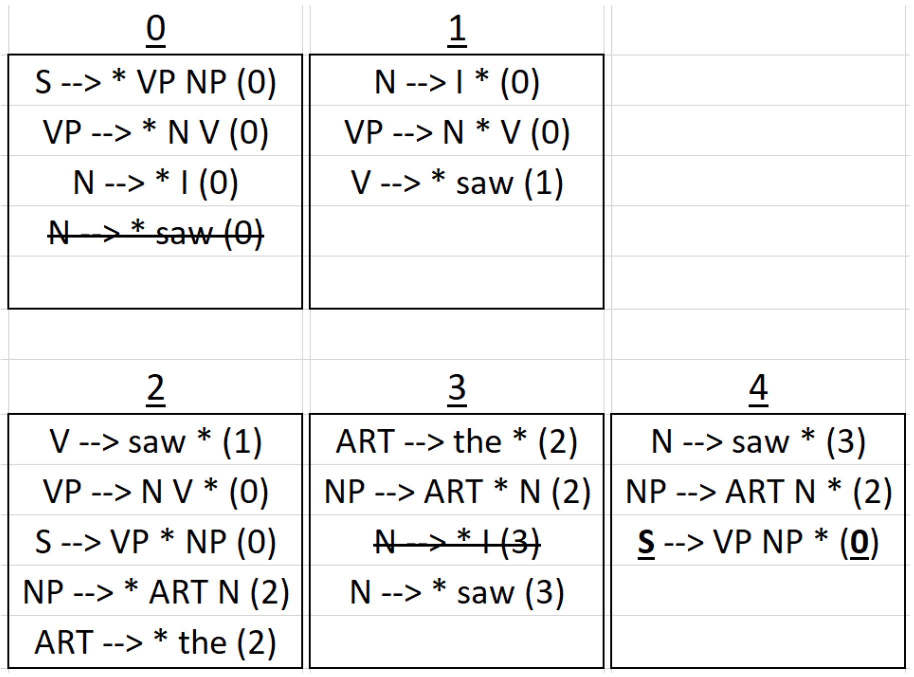


Figure 2.3: Second step of the Earley algorithm

Figure 2.3 illustrates the parsing process of the second chart. The item from the first chart is added at the top of the new chart. The added chart index stays the same, and the marker is moved to the next token. Now the iteration process starts over. First, the item is validated if it is completed, and in this case, the item is finished. Therefore, the completer is called to add new Earley items to the chart. The completer inspects the chart with the index of the completed item. In this case, the completer identifies every item in the first chart, whose marker is in front of the token "N". The chart index of the rules is untouched. After the completer added all rules, the iteration process continues. One rule was added to the chart and is now parsed. The algorithm works accordingly to the first chart. At first, the predictor is used to find rules matching the "VP", followed by finding rules for "V". At last, the scanner is called to identify if the terminal symbol matches the current token of the sentence. Both match and the item get added to the following chart with the marker moved.



| 0 | 1 | |
|---|---|---|
| S --> * VP NP (0) | N --> I * (0) | |
| VP --> * N V (0) | VP --> N * V (0) | |
| N --> * I (0) | V --> * saw (1) | |
| ~~N --> * saw (0)~~ | | |

| 2 | 3 | 4 |
|---|---|---|
| V --> saw * (1) | ART --> the * (2) | N --> saw * (3) |
| VP --> N V * (0) | NP --> ART * N (2) | NP --> ART N * (2) |
| S --> VP * NP (0) | ~~N --> * I (3)~~ | **S** --> VP NP * (**0**) |
| NP --> * ART N (2) | N --> * saw (3) | |
| ART --> * the (2) | | |

Figure 2.4: Completed Earley table

Figure 2.4 presents the completely parsed Earley table. Each chart and each item are parsed accordingly to the predictor, scanner, and completer. The parsing process reached the last chart, and it contains a finished item with a sentence annotation on the left-hand side. The item was also added to the first chart. That means the parsing process has finished successfully, and the parser was able to identify the sentence as part of the language.
The recognition process itself is straightforward and identifies if the sentence matches the grammar as well. The tricky part with the Earley parser is in creating a parsing tree for the parsed sentence. The charts have to be iterated backward to extract the parsing tree from the Earley table. The parsing tree is built from top to bottom, and it starts with the completed sentence item, which is the root of the parsing tree. All terminals and non-terminals on the right-hand side are added as the children of the root. The algorithm starts with the right child and searches the current chart for the finished item with the child on the left-hand side. The right-hand side of the item is set as children of the current root. If the right-hand side is a terminal, the algorithm sets it as the child of the current root and marks the current root as finished. The algorithm then walks back up the parsing tree until the next unfinished adjacent non-terminal is found. Once all children of the root sentence annotation are finished, the parsing tree can be returned.

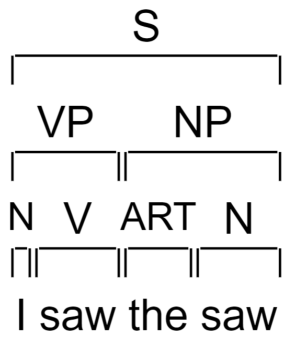


Figure 2.5: Parsing tree for the Earley example

Figure 2.5 visualizes the parsing tree of the example sentence with the given grammar. The brackets above the tokens represent the range of the literal. At the bottom, the sentence is written, and the PoS tags are added at first. Then the verb and noun phrases are built, followed by the sentence phrase.
Additionally, the Earley parser can be extended with probabilities. Each rule in the grammar is provided with a grammar; otherwise, the default is 100%. At the creation of the parsing tree, the algorithm can compute the probability of the found sentence. The algorithm multiplies the current probability with the probability of each completed rule in the parsing tree resulting in the total probability for the found sentence, starting with the probability of the sentence rule itself.

### 2.2.3 Bottom-Up Parsing

The second technique is bottom-up parsing, which creates the parsing tree from the bottom to the top. The parser starts with the bottom tags and tries to match them. The parser stops once it matches a rule covering all given tokens, and the parser identifies the rules in postfix order. Compared to top-down parsing, bottom-up parsing has more potential to identify possible parsing trees. The results of a bottom-up parser can be different parsing trees, whereas top-down parsing only returns one parsing tree. However, the parsing tree of a top-down approach is always correct, and this tree is part of the language. If a bottom-up approach finds a parsing tree, this tree might be cover all tokens but must not be completely valid.
The most known bottom-up algorithm is the CYK-algorithm. In the following section, the preconditions and workflow will be explained.

#### CYK-Algorithm

J. Cocke, D.H. Younger, and T. Kasami individually founded variants of the CYK-algorithm [10]. By now, the algorithm is known as the Cocke-Younger-Kasami algorithm, short CYK. The workflow of the parser is relatively straightforward. As a result, humans can understand the decision-making in every step, and it is possible to troubleshoot the reason for a parsing error. The algorithm creates a matrix for the input sentence and starts matching the content of the cells with one another. The matching process follows a strict and straightforward procedure, which a human can retract. Due to that, a flawed matrix can be troubleshot by a human using the same grammar.
In order to start the CYK algorithm, the given grammar must be a context-free grammar and converted into the Chomsky normal form (CNF) [11]. A grammar is defined as a CNF if every rule has the form A $\rightarrow$ a or A $\rightarrow$ BC, where a is terminal, and B and C are non-terminals. By using context-free grammar, it can be transformed easily into CNF grammar. In CNF, $\epsilon$-rules are prohibited and removed during the conversion. For that reason, the CYK-parser is more stable and does not have to interact with this kind of rule.
The algorithm consists of two different phases. The first phase is the recognition phase. The algorithm checks if the given terminal tokens match the requirements of the grammar. In the context of natural language processing, a terminal token is a word of the input sentence. During the recognition process, each word of the sentence is validated if it matches the language. Due to the endless number of rules needed to achieve this process, the recognition phase can be simplified by using a PoS tagger to label the tokens of the sentence. A PoS tagger is designed for labeling the tokens in a sentence and considering the formation of the tokens because the label of a token might be depended on the adjacent tokens. Covering every possibility with parsable grammar results in a complex and enormous syntax, and instead, the PoS tagger is used. This tagging process results in a recognition table, listing the words of the sentence and their label.
In the second phase, the created table is used to construct all possible derivations of the sentence with the given input grammar. At the end of the second phase, a matrix is returned containing all found matches of the tokens. Unlike the Earley al-

gorithm, the CYK algorithm might return several possible parsing trees. By adding possibilities to the rules of the grammar, the parsing trees can be ranked by their final likelihood.
To better explain the second phase of the parser, a sentence will be exemplary parsed, and in every step, the matching process will be highlighted. During this example, the recognition process is done by grammar. This process underscores the workflow of the algorithm in every detail. The following grammar is used to parse the sentence by the CYK:

```
S    —> NP   VP
VP   —> V    NP
S    —> V    NP
NP   —> NP   NP
V    —> fish
VP   —> fish
NP   —> people
NP   —> fish
NP   —> tanks
```

The grammar can be improved by adding probabilities to it. At the end of the example, the same sentence is parsed using a grammar with probabilities to highlight the impact.
The input sentence is "fish people fish tanks". This sentence is written into a table, where each token (word of the sentence) has its cell. Figure 2.6 shows the initial table. In the figure, a pyramid is highlighted, used as a guideline for the algorithm, and the next figures will explain the exact usage.

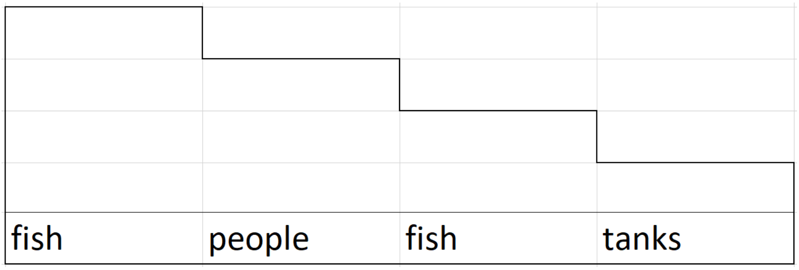


Figure 2.6: Initial CYK table with sentence tokens

The first step is the recognition phase. By using the grammar, each terminal token is labeled with a non-terminal token. The new token is written in the row above the token. The terminal tokens are never used again in all the following steps, and only their non-terminal labels are used to create the parsing tree. The recognition phase is crucial for the algorithm. For example, if the PoS tagger commits a mistake, the parsing process might return a wrong parsing tree or may not even finish the parsing process at all. Once every token is labeled, the table looks like Figure 2.7.

When all terminals are matched with non-terminals, the recognition phase is finished. Now the second phase of the algorithm starts with the created table. The algorithm tries to match as many cells as possible during each iteration, starting at the bottom row with the newly added terminals. The algorithm always parses

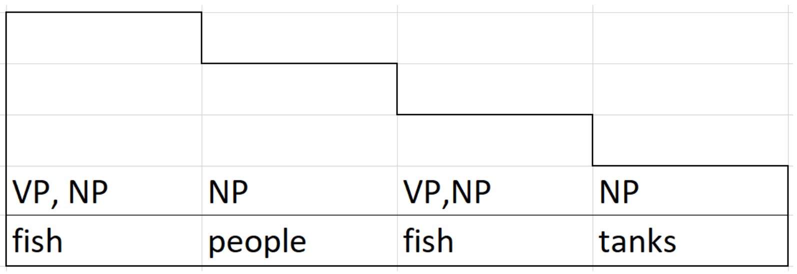


Figure 2.7: CYK table with terminal tokens

from left to right. During the first iteration, the algorithm works slightly differently compared to other iterations.

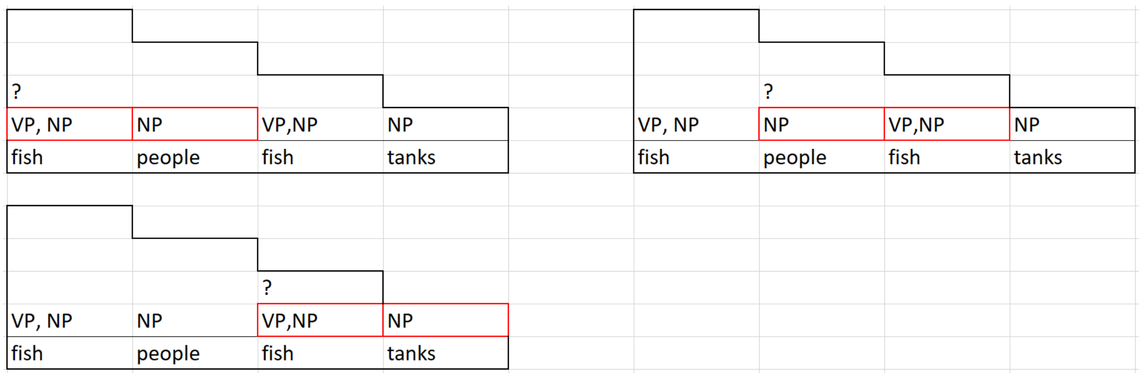
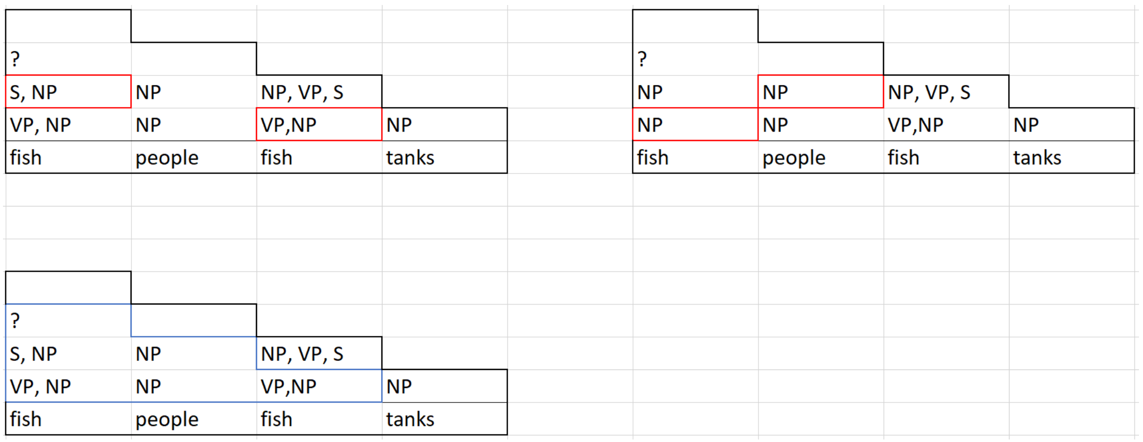


Figure 2.8: Starting second phase of CYK

In Figure 2.8, the start of the second phase is presented. In order to find an applicable rule for the cells in the second row, the algorithm matches the cell directly beneath the question mark with the adjacent cell on its right. The left part of an applicable rule is written in the current cell if a match is found. It is possible during the iterations that more than one result is found. In this case, every result is written in the current cell. In Figure 2.8, this process is shown with the red boxes functioning as pointers. The matching process continues in the second row until there is no right neighbor for the cell beneath the question mark. Because of this behavior, the resulting table is in the shape of a pyramid. In each subsequent iteration, the rows will be one cell shorter than their predecessors. Once the second row is finished, the algorithm jumps back to the beginning of the third row.

In Figure 2.9 the actual matching process for all further iterations is displayed, and the process is highlighted with the red pointers on the cells. The process can be described with a pyramid. At this point, the current cell is the top level of the pyramid. For each row going done, the pyramid is extended with another step. In Figure 2.9 this is highlighted with the blue pyramid in the parsing table. The matching algorithm starts with the cell directly beneath the current cell. It tries to match the cell with the bottom step of the blue pyramid. If a positive match is found, the result is stored in the current cell. Afterward, the algorithm moves the pointer beneath the current cell one cell down and the other pointer one step up on



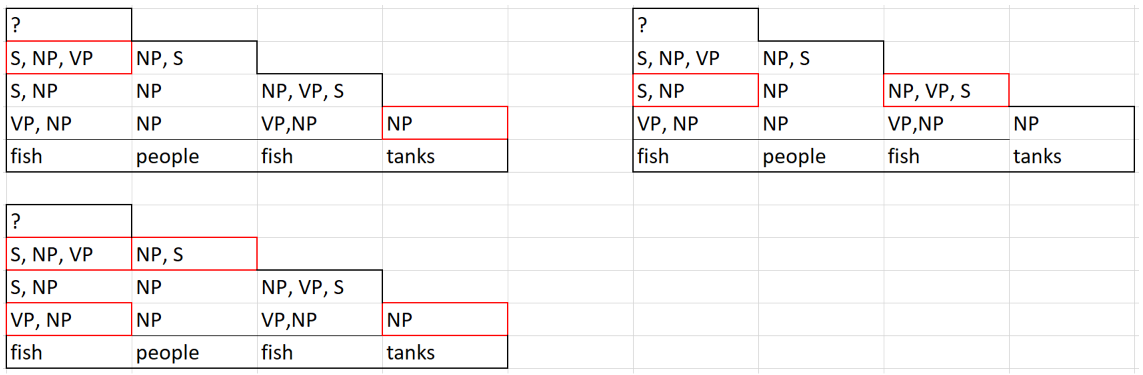
| | | | |
|---|---|---|---|
| | | | |
| ? | | | |
| S, NP | NP | NP, VP, S | |
| VP, NP | NP | VP,NP | NP |
| fish | people | fish | tanks |

| | | | |
|---|---|---|---|
| | | | |
| ? | | | |
| NP | NP | NP, VP, S | |
| NP | NP | VP,NP | NP |
| fish | people | fish | tanks |

| | | | |
|---|---|---|---|
| | | | |
| ? | | | |
| S, NP | NP | NP, VP, S | |
| VP, NP | NP | VP,NP | NP |
| fish | people | fish | tanks |

Figure 2.9: Starting second phase of CYK

the pyramid. The content of the two new cells is matched, and the results are added to the current cell. Once the pointer beneath the current cell is at the terminal row, the algorithm stops and moves a cell to the right in the current row and starts the matching process again. When all cells in the row are parsed, the algorithm moves one row up and iterates through this row as well.



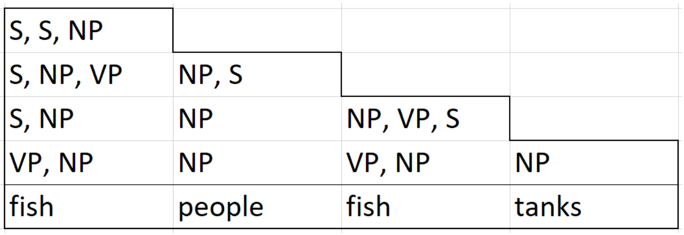
| | | | |
|---|---|---|---|
| ? | | | |
| S, NP, VP | NP, S | | |
| S, NP | NP | NP, VP, S | |
| VP, NP | NP | VP,NP | NP |
| fish | people | fish | tanks |

| | | | |
|---|---|---|---|
| ? | | | |
| S, NP, VP | NP, S | | |
| S, NP | NP | NP, VP, S | |
| VP, NP | NP | VP,NP | NP |
| fish | people | fish | tanks |

| | | | |
|---|---|---|---|
| ? | | | |
| S, NP, VP | NP, S | | |
| S, NP | NP | NP, VP, S | |
| VP, NP | NP | VP,NP | NP |
| fish | people | fish | tanks |

Figure 2.10: Final matching phase of CYK

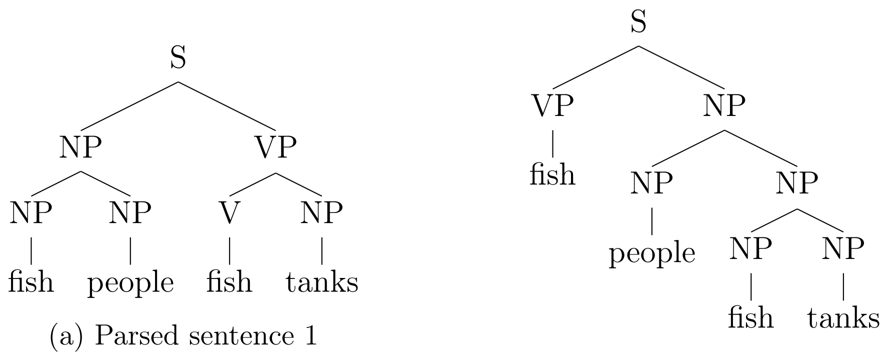
In Figure 2.10, the final matching process of the parsing table is presented. The matching algorithm moves accordingly to the other rows through the table. It starts at the cell beneath the top-level cell and repositions the pointers after each cell is finished. Once the algorithm is finished, the content of the top-level cell is evaluated.



| | | | |
|---|---|---|---|
| S, S, NP | | | |
| S, NP, VP | NP, S | | |
| S, NP | NP | NP, VP, S | |
| VP, NP | NP | VP, NP | NP |
| fish | people | fish | tanks |

Figure 2.11: Finished parsing table of CYK

In Figure 2.11, the finished parsing table is visualized. In this example, the parsing process is finished when the top-level cell contains an "S" token. This token represents a sentence token and means the input sentence is identified as a sentence for the given grammar. By using this sentence token, a parsing tree can be constructed by recursively retrieving all tokens leading to this sentence token. If there is more than one sentence token, the parser returns every found parsing tree to the user.



(a) Parsed sentence 1

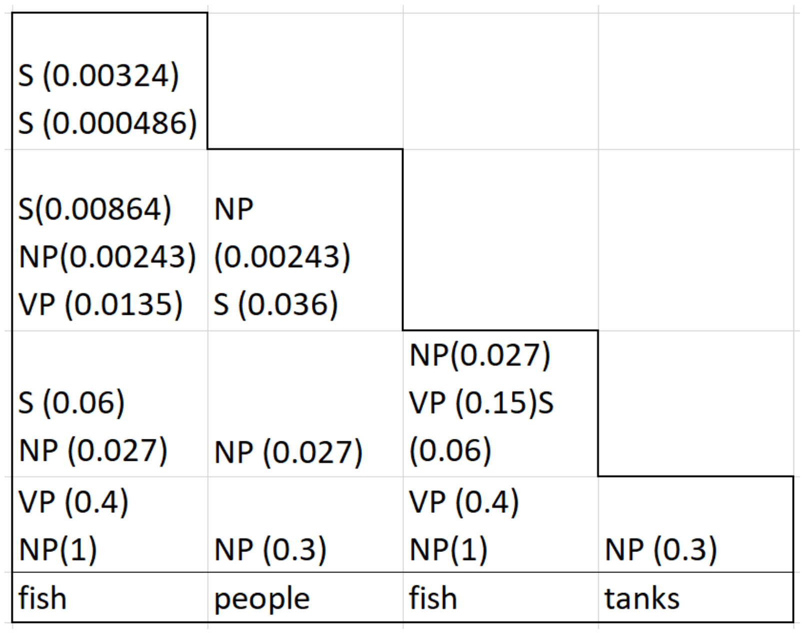
(b) Parsed sentence 2

Figure 2.12: Parsed sentences for the CYK example

Figure 2.12 visualizes both parsing trees of the CYK algorithm. The parsing tree can be extracted by stepping down the matrix and identifying the matches leading to the literals. Alternatively, the algorithm can store the two literals in the resulting literals to simplify the tree extraction. Even though both are parsing trees for the example sentence, the parsing tree 2.12a is a semantically and syntactically correct parsing tree. The parsing tree 2.12b lacks of semantics and is not valid and accepted. The algorithm can be improved by adding probabilities to the rules to enable a parser to decide the quality of a parsing tree. During each iteration, the probabilities of the tokens are multiplied. The example grammar can be expanded with probabilities to clarify the process.

```
S    —> NP  VP   0.8
VP   —> V   NP   0.5
S    —> V   NP   0.2
NP   —> NP  NP   0.3
V    —> fish     1
VP   —> fish     0.4
NP   —> people   0.3
NP   —> fish     0.3
NP   —> tanks    0.3
```

In case a matching rule is found during the process, the probability of the new token is computed by multiplying the probability of the two matching tokens with the probability of the applicable rule. The algorithm can even be configured to keep only the token with the highest probability in case of multiple matches for a cell. However, this might lead to the loss of some possible parsing trees.



| | | | |
|---|---|---|---|
| S (0.00324) S (0.000486) | | | |
| S(0.00864) NP(0.00243) VP (0.0135) | NP (0.00243) S (0.036) | | |
| S (0.06) NP (0.027) | NP (0.027) | NP(0.027) VP (0.15)S (0.06) | |
| VP (0.4) NP(1) | NP (0.3) | VP (0.4) NP(1) | NP (0.3) |
| fish | people | fish | tanks |

Figure 2.13: Parsing table of CYK with probabilities

Figure 2.13 shows the same parsing table as Figure 2.11 but is parsed with a grammar using probabilities. At the top level are two different sentence annotations. The algorithm can return the two parsing trees ordered by their probabilities, starting with the more likely.
In Figure 2.12 both possible parsing trees are presented. Using the probabilities of Figure 2.13, the parsing tree 2.12a is the sentence annotation with the higher probability (0.00324). A human can tell which is correct by comparing both parsing trees. An automatic parsing algorithm must rely on probabilities, or otherwise, both trees are equal.

# 3 Related Work

Parsing Expression Grammars (PEGs) are used by top-down parsers with backtracking. Backtracking is the ability to iterate backward if an error is encountered. The algorithm tries to find the start of the error and corrects itself. However, PEGs struggle with syntax errors. Most of the time, they cannot recover from the errors and stop the parsing process. Due to poorly error recovery systems, PEGs are unsuited to use in Integrated Development Environments (IDEs). Programs written in an IDE are most of the time syntactically incorrect and are changed dynamically. The authors of the paper "Syntax Error Recovery in Parsing Expression Grammars" [12] propose a conventional extension for PEGs. They add labels to the PEG, which are used to recover from errors and return a meaningful error message to the user. For example, a word of the sentence is missing. In the rule of the PEG, the word is annotated with a label. The parser marks the error point in the sentence and returns it according to the error message. The message can differ from the type of the word (e.g., "a semicolon is missing", "the ')' is missing"). After the error message is returned, the recovering algorithm is called. The authors wrote several mathematical formulas describing how far the parsing algorithm must move forward to continue parsing without interruption of the same mistake. The formulas differ from the missing token and the occurring error. They tested their algorithm on 180 syntactically invalid Lua programs. The grammar was annotated manually with 75 different labels. The recovery strategy applied for 57% of the sentences perfectly. All errors were found in 37% of the sentences, but too much information was lost in the recovering process. Only 9% of the sentences failed. The Earley parser can be extended with some of the features. Labeling the grammar with error messages helps the algorithm to return valuable messages. For example, the tokens of the STTS can be labeled. The parsers return what kind of token is missing in the sentence. Enabling the Earley parser to backtrack if the parser stops unexpectedly can be helpful. The usage of a recovering system with this scale is questionable. The Earley parser can identify the number of tokens needed for a label using its grammar. The algorithm can jump ahead in the sentence to where the next label should start. If the sentence is not parsable, the algorithm stops and starts over with the following sentence.
The paper "Error recovery in parsing expression grammars through labeled failures and its implementation based on a parsing machine" [13] from Sérgio Queiroz de Medeiros and Fabio Mascarenhas presents a similar approach to the previous paper. They removed the backtracking of the PEG and changed it to the farthest failure approach. If the parser finds an error in the input, the algorithm will not backtrack to the start of the sequence but goes to the farthest point of the input sequence to report the error. With this approach, the parser can log the exact error position and even return a list with possible tokens that can be used to resolve the error. The

grammar stays unchanged, and the user gets feedback about the mistakes. Next to this approach, the authors use the grammar extension of the previous paper. The automatically generated list might be too long or inaccurate. The creator of grammar can annotate the grammar with error messages. If the input fails at an annotated message, the error message is returned. If there is no annotated message, the parser will automatically generate the list of possible tokens. The recovering process depends on the missing input. If only a terminal is missing, the parser returns an error message and acts like the terminal has been parsed. If the input is a non-terminal, the algorithm starts again after the length of the terminal is skipped. The authors presented three complete error recovering systems for PEGs, which is not needed in this paper due to the different workflow of the Earley parser. However, extending the Earley parser with the farthest failure approach might be better than adding backtracking. The Earley parser finds every possible rule in each step. Therefore backtracking might be in vain. The automatic annotation of the errors can be used in the parser and the recovering process depending on the input type. Both features will be added to the Earley parser.
In the paper "Towards Automatic Error Recovery in Parsing Expression Grammars" [14] Sérgio Queiroz de Medeiros and Fabio Mascarenhas present an automatic system to annotate a grammar for error recovering. The authors published the previous paper [12] as well. They refined their previous approach by implementing an automatic procedure to label the literals of the grammar. The system labels every literal on the right-hand side of a grammar except the first one. If the token does not match the first literal, the wrong rule was selected, and another rule should be found. With their automatic procedure, they achieve similar results compared to an already labeled grammar. They tested their recovering error system in the Titan Parser. The recovering system achieves 88% excellent recoveries, 5% good, 5% poor, and 1% failed. The Earley grammar can be labeled with an automatic process as well. The non-terminals of the grammar can be labeled with their STTS group. For example, if a token "VVFIN" is missing, the error message could be "A verb is missing here". This reporting process can be done with every non-terminal literal in the grammar, and the user gets feedback on what token was expected at this point. An automatic error message for a terminal is challenging. An option is to return the expected token sequence and the missing tokens. For example, the non-terminal "NP" (noun phrase) consists of the sequence "N V" (noun and verb). If the noun phrase cannot be parsed, the parser returns that a noun and verb are missing at this point. The problem with this method is with several possible tokens sequences for a non-terminal. Either the most likely sequence or every sequence is returned. The parser skips to the end of the sequence and then continues parsing the sentence. Otherwise, the parser skips the entire sentence and starts directly with the following sentence.
The paper "Repairing Syntax Errors in LR Parsers" [15] of Corchuelo et al. presents an LR parser with the ability to repair syntax errors in the input. An LR parser is a bottom-up parser using a context-free grammar. The parser is also known as LR(k)-parser because it parses from left to right and can look at k tokens ahead. The k tokens are described as lookahead. The parser does not need backtracking or guessing the label of a token using this feature. Usually, the parser's lookahead

is predetermined to 1. The parser processes the input from left to right by shifting through the tokens and reducing them if a rule fits the stack. The variant Corchuelo et al. present can change the input if there are no reducing operations possible. The parser tries to insert, delete or switch tokens at the error position. This procedure should enable the parser to find a syntactically correct sentence and finish the parsing. There is no need for a user to step in and help the parsing process with this method. If one of the operations is successful, the parser can then return the changes to the user as an error log. To the time of the paper, this method was able to compete with other state-of-the-art methods for error detecting and error recovering. This error repairing mechanism can be implemented in the CYK algorithm as well. Initially, the method will be modified to fit the CYK parser. If the parser cannot parse a sentence completely, the parser will start and delete, swap, and insert tokens into the sentence. After each change to the sentence, the CYK algorithm is called again if the input can be parsed now. The tagset of the STTS is used to insert the new tokens. At each position, every possible token will be inserted and verified if the sentence is syntactically correct. If a correction method is successful, the operation can be returned as an error log to the user. It might be possible to use this algorithm while the CYK algorithm is parsing. This process has to be investigated once the algorithm is functioning on at the bottom level.
Top-down and bottom-up parser have their advantages and disadvantages. Kiran et al. compare in their paper "A Comparative Study on Parsing in Natural Language Processing" [16] LL- and LR-parsing. LR-parsing has been introduced with the previous paper. LL-parsing is a top-down approach with a similar workflow as LR-parsing. The parser uses context-free grammar and can look at k tokens ahead (called lookahead). The tokens of the sentence are added to a stack. At the top of the stack is the starting token of the sentence. The tokens are then compared with the grammar. If they match, the token is removed from the stack, and the next token is examined. If the stack is empty after the parsing process, the sentence is part of the grammar. During their research, the authors compare the LL- and LR-parser on their techniques and results. They conclude that both are highly capable of parsing correct sentences. However, the bottom-up approach is slightly better because of the range of applications and the better functionality. This approach is similar to the Earley and CYK parser.
These are the most common parser in their domain. However, the CYK parser can find different parsing trees and more matches in a sentence. The Earley algorithm is more straightforward and, therefore, easier to apply error recovery systems to it. A mix of Earley and CYK algorithms is implemented to combine both parsing techniques. The CYK-algorithm parses the sentence as far as possible. When the algorithm stops, all possible parsing trees fragments are extracted from the matrix. Afterward, they are given input to the Earley parser to find possible parsing trees and use the error recovering processes.

# 4 Domain

The efficiency and functionality of the parsing variants can not be presented with a proper testing domain. Usually, a domain is split into a training and validation set. The algorithm uses the training set to understand the specifics of the domain. With the validation set, the algorithm's performance is evaluated by using unknown parts of the domain. A good domain has several syntactically correct and incorrect sentences to improve the techniques and evaluate them. The correct ones constitute the training set. The incorrect sentences are used to evaluate and illustrate the effectiveness of the error correction. Covering an entire domain like literature or medicine, the resulting grammar would be immense and complex to design. At some point, the grammar can be incomprehensible, and it is unclear which structures are already covered. The complexity of the grammar might lead to falsified results of the error recovering mechanisms. Therefore, a small and clear proof-of-concept (POC) domain is used to evaluate the effectiveness of the algorithms. The domain covers 36 different German sentences with moderate complexity. Of these 36 sentences, 15 are the training set and used to create the grammar for the domain. The other 21 sentences are syntactically incorrect with at least one mistake. Either a word is missing, has to be deleted, or needs to be inserted in the sentence. Several other sentences contain syntactically unknown structures to the grammar. The parser should be able to identify them and add found rules to the grammar.

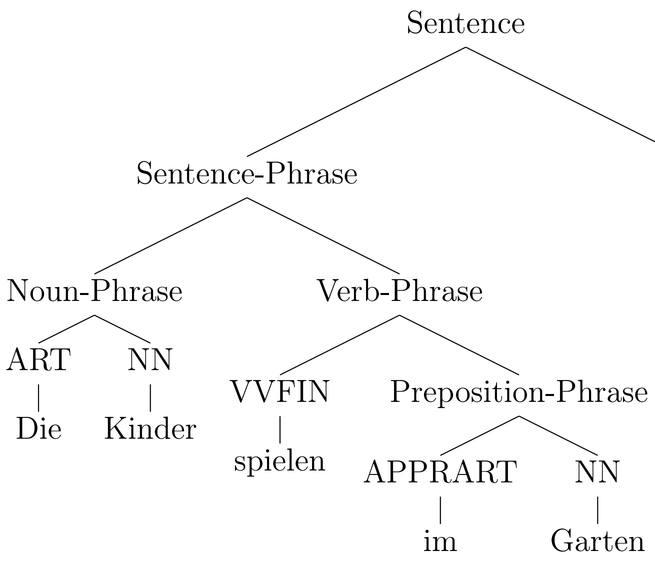
**Correct sentences**:
Zu jedem Zeitpunkt fährt der Zug auf Schienen.
Die Kinder spielen im Garten.
Die Hunde bellen, weil sie nicht erzogen sind.
Im Vergleich zum Nachbarn ist der Rasen ungepflegter.
Bei der Kirche findet eine grosse Versammlung statt.
Menschen leben ungesünder auf Grund von fehlender Bewegung.
Hin und wieder verstoßen Menschen beim Autofahren gegen Gesetze.
Wenn Kinder Süßigkeiten essen, können sie nicht mehr gut einschlafen.
Es lassen sich keine Hinweise auf eine schwerwiegende Krankheit finden.
Mit dem Austauschen eines Bauteils wird das Problem gelöst.
Software ist aktualisiert und die Maschine ist eingestellt.
Der Kunde bemängelt, dass die Fahrkupplung schwergängig ist.
Die Kontrollleuchte leuchtet nach dem Anziehen der Bremse nicht mehr auf.
Wir essen Semmeln und Brot.
Alte Teile sind entfernt und neue Bauteile sind eingebaut.

By using these sentences, a grammar is created to cover the domain. Parsing algorithms cannot judge the semantic meaning of a sentence. The grammar only covers the syntactical side of a sentence and cannot verify its meaning. The sentences cover a wide variety of syntactical structures and complexity in the German language. The following grammar is designed to parse the domain.

```
S = (SP | SP , SC | SC , (SP | VP) | SP KON (SP | VP)) .
                                        [Sentence]
SP = (NP | PP | ADVP | PPER) VP | NP VVFIN | NN VP
                                        [Sentence Phrase]
SC = KOUS ((NP | NN) SCVP |PPER (VP | SCVP))
                                        [Subordinate Clause]
SCVP = NN VVFIN | ADJD VAFIN            [SC Verb Phrase]
ADVP = PTKNEG ADV | ADV KON ADV         [Adverb Phrase]
PP = APPR (NN | NP) | APPRART NN | APPR? PIAT (NN | NP)
                                        [Preposition Phrase]
NP = ART (ADJA | PIAT)? NN | ADJA NN | NN KON NN | NP NP
                                        [Noun Phrase]
VP = VVFIN ((NP | NN) (PP | PTKVZ)? | PP+ | ADJD PP* | PRF | PPER |
     PP ADVP PTKVZ) | PTKNEG? VVPP VAFIN | VAFIN NP? (ADJD | VVPP) |
     VMFIN ( PP VP | PPER ADVP ADJD VVINF) | ADJA VVPP | VP PP VVINF
     | (VVFIN | VP) PTKNEG PTKVZ        [Verb Phrase]

```

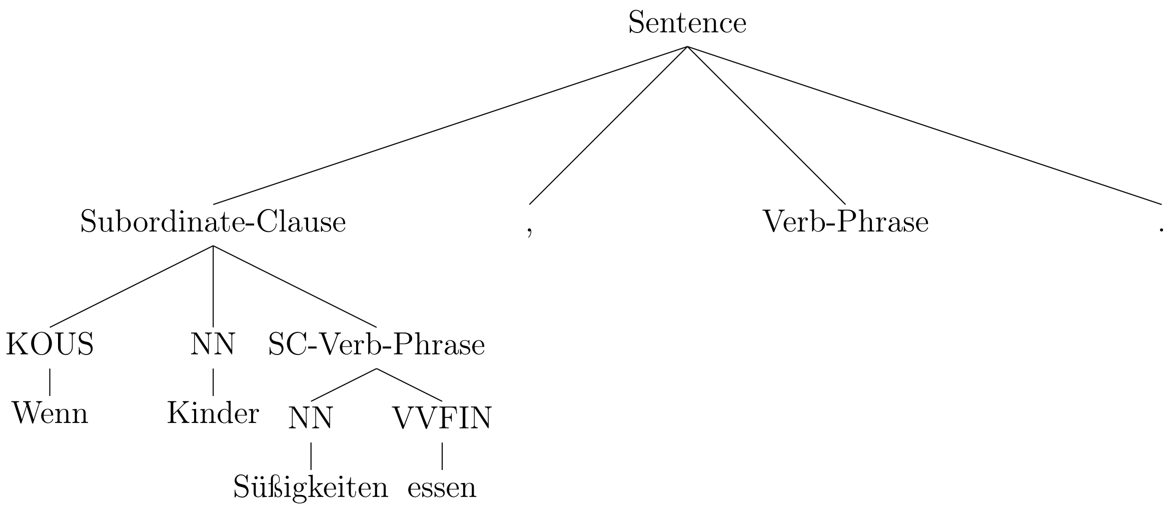
The grammar consists of different phrases to classify a given input sentence. During the parsing process, the given tokens are combined with annotations derived from grammar rules, and each line constitutes a rule. At the start of each line, the symbol of the rule is set. For example, in the first line, the annotation is called "S". After the symbol declaration, an equal sign is used to determine the combinations, which are accepted for this annotation. The operators mentioned in Subsection 2.2.1 can be used to design and simplify the combinations. The usage of these operators leads to a shorter and more compact grammar. After the combinations, the probability of an annotation can be added. If the symbol of a rule is not clear enough, a name can be added at the end of each line. However, the name has to be in square brackets.
In order to create the grammar for the POC domain, the sentences are labeled by a PoS tagger with the STTS tagset [7]. The tags are used in the combinations to create the phrases. Generally speaking, each simple German sentence shares the same structure. The sentence starts with a subject, followed by a verb, and ends with an object. The subject and object are mostly noun phrases. In the grammar, the rules featuring the noun phrases are in line 7. These rules cover phrases like "eine schwerwiegende Krankheit" or "Semmeln und Brot". Typically, the verb of a sentence is the critical feature in parsing sentences. Usually, each sentence is parsed depending on the verb. In this grammar, the verb and its right-hand side are stored in a verb phrase. In a simple sentence, the verb and the object are covered by a

verb phrase. As a result of the significance of the verb, the rules of the verb phrase cover most scenarios in the entire grammar. Once the noun phrases are built, the verb and noun phrase of the object are combined into a verb phrase. Afterward, a sentence phrase is built with the noun phrase of the subject and verb phrase. The rules of the sentence phrase are displayed in line 2. Different scenarios depend on the complexity of the sentence to build a sentence phrase. Once a sentence phrase is found, the input can be parsed as a sentence using the sentence phrase and an ending period. However, the syntax of a given sentence can be more complex. Next to the noun phrase and verb phrase, preposition phrases cover structures like "in dem Garten" (line 6), and adverb phrases like "Hin und wieder" (line 5) are possible combinations in a sentence. The consisting phrases are expanded with the phrases. Also, subordinate clauses are a standard structure in the German language. The syntax of a subordinate clause differs from a regular sentence phrase. Commonly, the clause starts with conjunction and has the structure of subject, object, verb. A specific subordinate clause verb phrase is added to the grammar (line 4), covering the object and verb combination. As a result of this, the position of the object and verb is crucial and significant. In the subordinate clause verb phrase, the object is first followed by a verb and vice versa by the typical verb phrase. The leading conjunction, subject, and subordinate clause verb phrase are combined to the subordinate clause (line 3). A sentence has to contain a sentence phrase and a possible subordinate clause separated by a comma. The rules of the sentence annotation are extended to cover the possible sub-clauses. Similar to a sentence phrase, preposition phrases, and adverb phrases are possible in a subordinate clause.
Each correct sentence of the domain can be parsed, and a parsing tree is returned. For example, the sentence "Die Kinder Spielen im Garten." has simple syntax (subject-verb-object). The associated parsing tree underlines the assumption.

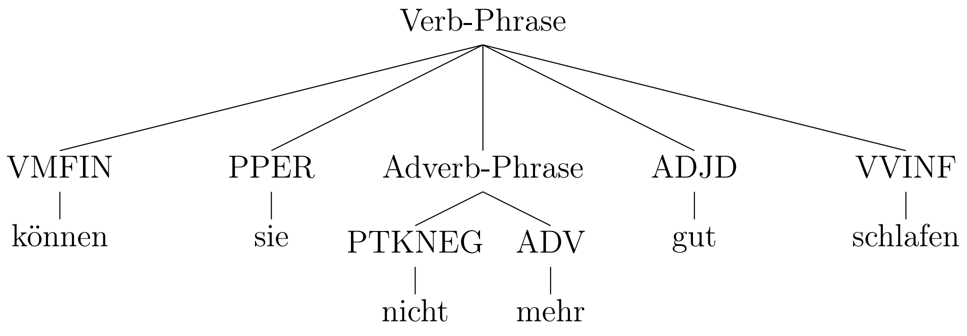


At first, the parsing process combines the verb and object to a verb phrase. Next, the sentence phrase is created using the subject and verb phrase. The sentence is identified as part of the language because the sentence phrase and the period are matched to a sentence annotation covering every token.

If the syntax of a sentence is complex, the parsing tree is enormous and has more children. For example, the parsing tree of the sentence "Wenn Kinder Süßigkeiten essen, können sie nicht mehr gut einschlafen.":



(a) Parsing tree of the sentence



(b) Verb phrase of the tree

The size of the parsing tree increases with the complexity of the sentence. The first part of the sentence is the subordinate clause, identified by the "KOUS" token at the start and the verb syntax. The second part of the sentence is the main clause, which contains only one verb phrase. The subordinate clause, the comma, and the verb phrase build a sentence phrase. The input is identified as a sentence by combining the sentence phrase and the period.
The other parsing trees of the correct sentences are added to the appendix in the Appendix A (parsed by CYK) and B (parsed by Earley).
Next to the correct sentences, the domain covers several sentences with at least one mistake. Either a word is missing, switched, or has to be deleted. Next to the incorrect syntactical sentences, the domain contains several sentences, which are syntactically correct but are not parsable with the current grammar. By analyzing the structure, the parser should suggest rules with which the sentence is parseable.

**Incorrect sentences**:
*Missing word*:
Das fährt auf der Autobahn.
Das Teil ist.
Kinder essen die nicht auf.
Autos gefahren.
Rote Äpfel besser.
Die Kinder schreien weil sie ängstlich sind.

*Interchanged words*:
Die meisten Menschen bekommen nicht mit Nachrichten.
Die Untersuchung zeigte Überraschungen keine.

*Delete word*:
Schrauben und und Muttern verrosten.
Drehzahl Sensor meldet Fehler.
Schrauben sind sind abgebrochen.
Autos fahren, auf der Autobahn.
Äpfel rote schmecken besser.

*Mixed mistakes*:
Die Kinder tanzen schreien weil sie ängstlich sind.
Kinder die schreien weil sie ängstlich sind.
Kinder die schreien, weil sie ängstlich tanzen sind.

*Not covered sentences*:
Unsere Wand ist rosa.
Der Mann sollte weghören.
Wer kennt den Namen?
Meine Figur schlägt Deinen.
Das Essen schmeckt nicht gut.

In the first couple sentences, different words and, therefore, STTS-tags are missing. The mistakes range from a missing noun to a missing comma. After adding the missing words to the sentence, they can be parsed by grammar. In the subsequent sentences, a word is out of place. The word has to be placed in the correct spot to return a parsable sentence. The subsequent sentences are used to demonstrate the ability to extend the existing grammar. The sentences are syntactically correct, however unknown to the grammar. Therefore, the sequences result from unknown STTS-tags or unknown syntactical structures. The last section covers the sentences used to evaluate the rule suggestion algorithm. Similar to the previous paragraph, the sentences are syntactically correct. Nevertheless, there are unknown STTS-tags or structures to the grammar and need to be detected.
In the next paragraph, the problematic locations are highlighted and corrected.

**Corrected sentences:**
Das Auto fährt auf der Autobahn.
Das Teil ist defekt.
Kinder essen die Mahlzeiten nicht auf.
Autos werden gefahren.

Die Kinder schreien , weil sie ängstlich sind.

Die meisten Menschen bekommen Nachrichten nicht mit ~~nicht mit Nachrichten~~.
Die Untersuchung zeigte keine Überraschungen ~~Überraschungen keine~~ Überraschungen keine.

Schrauben und ~~und~~ Muttern verrosten.
~~Drehzahl~~ Sensor meldet Fehler.
Schrauben ~~sind~~ sind abgebrochen.
Autos fahren ~~,~~ auf der Autobahn.
Äpfel ~~rote~~ schmecken besser.

Die Kinder ~~tanzen~~ schreien , weil sie ängstlich sind.
Die Kinder ~~Kinder die~~ schreien , weil sie ängstlich sind.
Die Kinder ~~Kinder die~~ schreien, weil sie ängstlich ~~tanzen~~ sind.

Unsere Wand ist rosa.
Der Mann sollte weghören.
Wer kennt den Namen?
Das Essen schmeckt nicht gut.
Meine Figur schlägt Deine.

# 5 Method and Implementation

The project is implemented in Kotlin and uses Maven to add external libraries to the project structure. The fundamental structure of the CYK and Earley algorithm is implemented in Kotlin as well. As an input for the parsing process, a single or several sentences are possible. The parser uses the UIMA framework [17] to return the parsing trees. Therefore, a CAS instance is created with the given sentences as the covering text. The created parsing trees are added to the CAS as annotations. A CAS instance is created for each tree if several possible parsing trees are found for a sentence. There is also the opportunity to use an existing CAS instance and parse the covered text. The created CAS can be exported to a location chosen by the user. The ARIES RF-tagger [18] is used as a PoS-tagger for the parser.
In this chapter, three modified versions of the parser are presented. At first, the CYK algorithm is extended by a brute-force error recovering algorithm. The subsequent modification is based on the Earley algorithm. The algorithm is extended to overcome errors in the sentence and return the error message to the user. The last algorithm is used to extend the parser's grammar, and as the fundamental parsing technique, the CYK algorithm is used. If the algorithm cannot parse the sentence correctly, all possible incomplete parsing trees are extracted. These parsing trees are compared with the grammar, and possible extensions are returned to the user.

## 5.1 Brute force error correction CYK

The brute force error correction is an extension of the CYK algorithm. Therefore, the primary methods of the CYK are still used, and the brute force algorithm only intercepts between the parsing algorithm and annotation of the CAS instance. If the parsing process returns an incomplete parsing tree, the error correction is called. In the field of error correction, the "edit-distance" [19] is a significant factor in solving the error. The edit-distance describes the number of errors in the sentence, precisely the number of operations needed to solve the errors. For example, a sentence with the edit-distance 1 needs one operation to return a correct sentence. An operation is either deleting a redundant token, moving a token in the sentence, or inserting a new token. If the sentence has an edit-distance of 2, there are two operations needed. The first operation has to be executed before the second operation. For instance, a token must be deleted before a new token can be inserted in the sentence.
The most significant advantage of the CYK algorithm is its powerfulness and ability to find several parsing trees. However, this is also one of its most significant disadvantages in the field of error correction. In theory, it is possible to evaluate the cell content at every position of the created matrix. On the one hand, the reasons for a successful match can be analyzed. On the other hand, it is possible to determine the reasons if there has been no match. However, evaluating every possible cell in the

matrix is too complex and needs enormous computing power. For example, if the input sentence has 10 tokens, the resulting matrix has 45 cells. These 45 cells lead to 210 possible matches between the cells (assumed one item in each cell). Troubleshooting for the errors in every possible match exceeds the profit. Furthermore, the complexity of troubleshooting the errors increases significantly with the cell's height in the matrix. The proposed algorithm is called brute force because only the lowest level of the matrix is changed. In theory, the algorithm can identify if a word is missing, switched, or redundant. In case entire word structures are missing, the algorithm will exceed its limits and fail.
However, the brute force algorithm can solve sentences up to the edit-distance 2. At first, all possible edit-distance 1 changes are executed. If these changes do not result in any possible parsing tree, the edit-distance 2 is executed.

## Edit-Distance 1

If the parsing algorithm does not find a possible parsing tree, the error correction algorithm is called. The algorithm uses the tokens generated by the PoS-tagger as a foundation for the correction process. In each edit-distance, the algorithm completes the same three steps. At first, the deleting process is initialized. Thereby, each PoS-token is deleted one at a time (except the period). After deleting, the new sentence is reconstructed and is tagged again. This step is significant for error correction. As mentioned in Section 2.1, the PoS-tagger uses Hidden-Markow-Models to anticipate the tag of a token. These models depend on the constellation of the tokens in the sentence. Once the sequence is altered, the algorithm has to tag the sentence and update the labels if necessary. After reconstructing and tagging the sentence, the parsing algorithm is executed with the new tokens. If a complete parsing tree is found, it will be stored in a list and marked as "altered". An error message is created for the user covering the executed operation as well as the redundant token. The user can remove the token and correct the mistake.
After the deletion process, the algorithm switches the tokens in the sentence. In order to cover a broad base, the tokens are not only switched with their direct neighbors. Each token is placed at every possible position. The only two positions, which are not possible, are the initial position and the position of the period in the sentence. For example, the first token is placed between the second and third token in the sentence. Afterward, between the third and fourth, until it reaches the period and the process is stopped. After each swap, the sentence is reconstructed and tagged. Accordingly to the deletion process, if a completed parsing tree is found, it gets stored in a list. The error message indicates the wrongly positioned token and the correct position of it in the sentence.
Finally, new tokens are added between the tokens of the sentence. In the Appendix C the entire tag set of the STTS is depicted. Inserting the STTS tags into the PoS-tag-sequence does not work because the additional word might change the tags of the other tokens. Therefore, instead of the tag, a word leading to this tag is inserted in the sentence. In the Appendix C, the last column is filled with examples for each tag. The corresponding words are used to insert them in the sentence. After adding the token, the sentence is reconstructed and tagged. Afterward, the sentence

is parsed, and in case a parsing tree is found, it is added to the list. In this case, the error message returns the location of the missing word and its tag. Because the parser cannot judge the semantic of the sentence, it is only possible to return the word type and not a specific word. At each possible position, all tags of the STTS tag set are added and parsed. Once all are completed, the position is changed until the period is reached.
In order to avoid parsing duplicates, a set is filled with every parsed sentence for the entire edit-distance 1. If the current sentence is not contained in the set, it is parsed and added. In order to calculate the maximum number of iterations in edit-distance 1, the algorithm does not find any duplicate sequence while parsing, and the sentence has $n$ tokens. The deletion process needs $n - 1$ iteration because the period is never deleted. The swapping process needs for each word $n - 2$ iterations because the initial sequence is blocked and the swapping with the period. Therefore, the entire swapping process needs $(n - 1)(n - 2)$ iterations. Most computing time needs the inserting process. At each position, 53 words are added. This process leads to $53(n-1)$ iteration for the inserting process. In sum the entire edit-distance 1 needs $f(n) = (n - 1) + (n - 1)(n - 2) + 53(n - 1) = n^2 + 51n - 52$ iterations. For example, a sentence with 10 tokens needs $f(10) = 558$ iteration to complete edit-distance 1.
To better illustrate how it works, a section of each operation is shown with an example. The example sentence is "Die meisten Menschen bekommen nicht mit Nachrichten." from the domain. When the deletion process is started, each token is deleted once, and then the sentence is parsed again.

```
meisten Menschen bekommen nicht mit Nachrichten.
Die Menschen bekommen nicht mit Nachrichten.
Die meisten bekommen nicht mit Nachrichten.
Die meisten Menschen nicht mit Nachrichten.
Die meisten Menschen bekommen mit Nachrichten.
Die meisten Menschen bekommen nicht Nachrichten.
Die meisten Menschen bekommen nicht mit.
```

The List 5.1 displays the parsed sentences during the deletion process. Only the period token is never deleted from the sentence because it is crucial to the syntax. In order to give a brief insight into the process, only one word is moved through the sentence. To find the correct sentence, the word "Nachrichten" is used for swapping.

```
Nachrichten Die meisten Menschen bekommen nicht mit.
Die Nachrichten meisten Menschen bekommen nicht mit.
Die meisten Nachrichten Menschen bekommen nicht mit.
Die meisten Menschen Nachrichten bekommen nicht mit.
Die meisten Menschen bekommen Nachrichten nicht mit. (correct sentence)
Die meisten Menschen bekommen nicht Nachrichten mit.
```

The process results in six new sentences because the original sequence is not parsed again, and the word is not swapped with the period. During this swapping, a possible correct sentence is found. The parsing tree is created and annotated with the error message "Please insert 'Nachricht' between 'bekommen' and 'nicht'!". The tree is added to a list to be later annotated to the CAS instance. Furthermore, the sequence shows the reason for re-tagging. By comparing the original sentence with the corrected sentence, the PoS-tag of the word "mit" changes:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Die | meisten | Menschen | bekommen | nicht | mit | Nachrichten | . |
| ART | PIAT | NN | VVFIN | PTKNEG | APPR | NN | $. |
| Die | meisten | Menschen | bekommen | Nachrichten | nicht | mit | . |
| ART | PIAT | NN | VVFIN | NN | PTKNEG | PTKVZ | $. |

The two different labels are highlighted. The parser will not identify the correct sentence if the words are switched in the sentence without re-tagging. Due to this behavior, it is necessary to re-tag every sentence after operating, despite the needed computing time.
Lastly, the STTS tags are added to the sentence, and the same behavior can be observed. It is not possible to only add the direct tags to the sentence and parse. The other tags might be influenced by the added one. Therefore, a word, which would result in the wanted tag, is added to the sentence. Afterward, the tokens are reconstructed into a sentence and newly parsed. The following sentences give a brief overview of possible added words:

| | |
|---|---|
| Die meisten schlaue Menschen bekommen nicht mit Nachrichten. | *ADJA* |
| Die meisten weil Menschen bekommen nicht mit Nachrichten. | *KOUS* |
| Die meisten nach Menschen bekommen nicht mit Nachrichten. | *APPR* |
| Die meisten schon Menschen bekommen nicht mit Nachrichten. | *ADV* |

By adding words instead of a tag, a problem is not getting the anticipated tag. There are tag combinations, which are unlucky or even impossible. Therefore, a set is filled with all parsed sentences. If adding a new word leads to a known combination, the word is skipped.
This example gives a brief overview, and only some sentences are presented. During the entire edit-distance 1, there are a total of 420 corrections computed and parsed. If no parsing tree is found during edit-distance 1, the algorithm starts process edit-distance 2.

### Edit-distance 2

The operations done in edit-distance 2 are identical to edit-distance 1. The tokens are deleted, swapped, or added at the lowest level. In contrast to the first process, the edit-distance 2 computes up to two changes in the sentence. There are two possibilities where the edit-distance 2 could be called. On the one hand, it can be initialized directly after a failing edit-distance 1 operation. The sentence is already changed according to the edit-distance 1 and is reconstructed. On the other

hand, the edit-distance 2 can be called after all level 1 operations are computed and unsuccessful. This approach should be preferred because if the edit-distance 1 is successful, the level 2 computations are redundant. By waiting for all operations to finish in the first level, the algorithm might need more time to compute the changes again. However, the duration of finding the edit-distance 1 errors sinks rapidly.
In this work, the second variant is preferred. All operations of the edit-distance 1 are finished and unsuccessful. Then the edit-distance 2 is called. However, this is optionally adjustable by the user. The maximum edit-distance can be configured at the initializing of the parser.
In order to achieve edit-distance 2, the algorithm repeats the operations for edit-distance 1. In contrast to the original, the sentence is now no longer tagged after the first change. The token sequence is reconstructed to a sentence, and the method is called recursively with the altered sentence. Once the second operation is finished as well, the sentence is reconstructed and re-tagged. Afterward, the parser uses the new tokens to parse the sentence and return possible parsing trees. If a complete parsing tree is found, it is added to a list and annotated with a message. Once all edit-distance 2 operations are finished for an altered sentence, the list of completed parsing trees is returned to the recursive call. Then the parsing trees are annotated with the edit-distance 1 change to provide the best possible feedback for the user. The annotations are sorted hierarchically, starting with the level 1 change. The edit-distance 2 completes the same 3 operations tasks as edit-distance 1. After deleting, swapping, and adding all possible tokens, the algorithm returns the parsing trees. Once all altered edit-distance 1 sentences are parsed in edit-distance 2 as well, each found parsing tree and the according to error log is returned to the user in a separated CAS instance.
By adding the edit-distance 2, the number of parsed sentences rises accordingly. Even though the algorithm is programmed not to allow duplicate sentences to be parsed again, the function set up above can also be used here. It describes the number of iterations in edit-distance 1 or 2. However, the number of iterations is computed by multiplying the result of edit-distance 1 with edit-distance 2. This results in the following formula, whereas $n$ is the number of tokens in a sentence, $g(n) = (n-1)f(n-1) + (n-1)(n-2)f(n) + 53(n-1)f(n+1)$. The part of the function that affects the most is adding the tokens. For example, a sentence with 10 tokens has 345078 iterations. The edit-distance 2 needs approximately 600% more capacity than the computing edit-distance 1. The number of iterations also has a massive impact on the runtime of the algorithm.
A few sentences from the edit-distance 1 example are changed to edit-distance 2 to illustrate the difference between the two edit-distance. For example, the sentence "Die meisten Menschen bekommen nicht mit Nachrichten." is altered to "Die Menschen bekommen nicht mit Nachrichten." to show the edit-distance 2.

```
Menschen bekommen nicht mit Nachrichten.
Die bekommen nicht mit Nachrichten.
Die Menschen nicht mit Nachrichten.
Die Menschen bekommen mit Nachrichten.
Die Menschen bekommen nicht Nachrichten.
Die Menschen bekommen nicht mit.
```

During the deletion process, the tokens are removed, similar to edit-distance 1. The last sentence of the list might even be labeled as a correct sentence.

```
Nachrichten Die Menschen bekommen nicht mit.
Die Nachrichten Menschen bekommen nicht mit.
Die Nachrichten Menschen bekommen nicht mit.
Die Menschen Nachrichten bekommen nicht mit.
Die Menschen bekommen Nachrichten nicht mit. (correct sentence)
Die Menschen bekommen nicht Nachrichten mit.
```

The same sentence structure as in edit-distance 2 is label correct. In this example, the removed word is an optional modifier for the word "Menschen". By removing it, the sentence's syntax does not change dramatically and can still be fixed by replacing the word "Nachrichten". Later, the found parsing tree is annotated with the error messages to remove the word "meisten" and replace the word "Nachrichten" in the sentence.

| | |
|---|---|
| Die schlaue Menschen bekommen nicht mit Nachrichten . | *ADJA* |
| Die weil Menschen bekommen nicht mit Nachrichten . | *KOUS* |
| Die nach Menschen bekommen nicht mit Nachrichten . | *APPR* |
| Die schon Menschen bekommen nicht mit Nachrichten . | *ADV* |

The inserting process needs the most resources of all three operations. In this example, there is no possible parsing tree found by adding words to the sentence. Due to the enormous number of parsed sentences, the possibility of finding a parsing tree is quite good. However, there is also much room for misinterpretation. The parser might find correct sentences based on its grammar but come into conflict with the natural language. These can be avoided by fine-tuning the grammar.

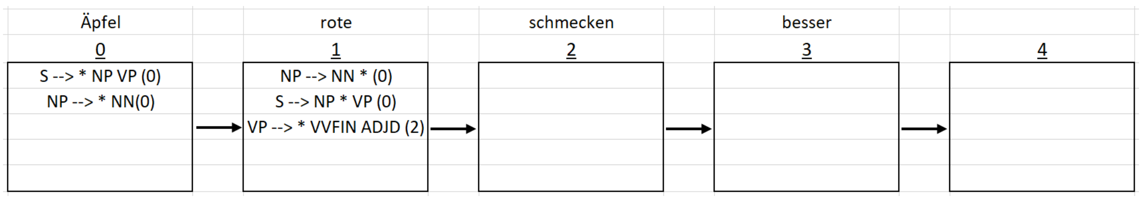
## 5.2 Error recovery in the Earley parser

The challenging feature of a top-down parser is the inability to parse an incorrect sentence entirely. If the parser encounters an unknown tag or cannot match the current tag with the rules, the parser stops and cannot finish parsing the remaining tokens. The Earley parser creates a chart with all states for each word in the sentence. If a rule is completed in a chart, it is added to the adjacent chart. The marker is moved accordingly. If a chart does not return any completed rule, the parser stops and returns the incomplete chart list for the sentence. It is possible to use the same brute force approach of the CYK algorithm. If the parser fails to create a parsing tree, the tokens of the sentence are changed in every possible way, and the parsing process is repeated. However, the advantage of a top-down parser is the ability to troubleshoot the position of failure. Nevertheless, this position does not have to correspond to the actual source of the error. The parser merely does not get any further here. Instead of altering the tokens in every position, the changes can only be applied to the problematic position. This ability reduces the number of changes dramatically. An error recovery system can be implemented to focus on precisely this behavior. The system is designed to alter the chart list as well as the tokens of the sentence. If the recovery is successful, the parser can resume the parsing process. The error recovery system can delete a token from the sentence and the related chart to add a new token to the sentence and expand the chart list or change the tokens' order. Like the brute force approach in the CYK algorithm, the system used the edit-distance to determine the degree of change. First, the algorithm determines all edit-distance 1 changes and verifies if the parser completes the process. If no parsing tree can be obtained, the system determines all edit-distance 2 changes and returns possible parsing trees. The given edit distance is the limit for changing the Earley table during the entire error recovering process. After each adaption, the counter of the edit-distance is increased until it is equal to the limit. This configuration enables the user to influence the behavior and effort of the algorithm.

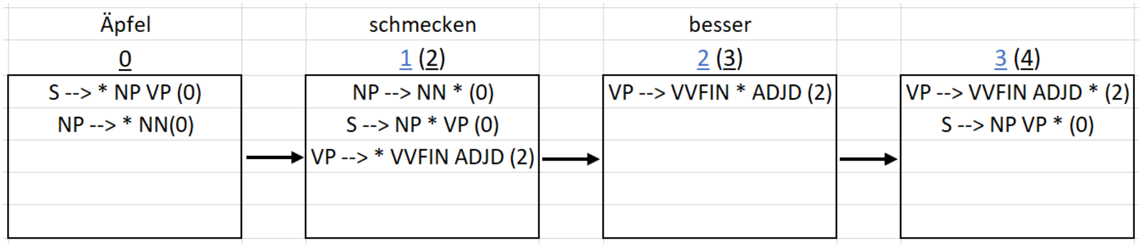
## Edit-distance 1

The parser implementation is similar to the basic Earley parser. The only difference is that the upper limit for the edit-distance has to be defined during the initializing. Once a sentence is given as input, an Earley table is created accordingly to its tokens. Then, the table is parsed using the standard Earley algorithm. The error recovery system is called if the table does not return a valid and complete parsing tree. The recovery system uses the same methods as in the brute force CYK algorithm. The incomplete Earley table is given to the recovery system, and then the algorithm troubleshoots the problem. The only reason why a table is not complete is that a chart failed to match the sentence token with a rule in this state list. For example, the sentence token is an 'NN'. The related state list must contain any kind of rule similar to "NP = *NN". Then the marker is moved, and the changed rule is added to the following chart. If no rule matches the token sequence, the following chart cannot be filled, and the algorithm fails to parse the sentence. Therefore to find the corrupted chart in the Earley table, the corrector iterates backward from the last chart to the first chart. Once the algorithm identifies a chart with any content, this chart is responsible for the incomplete table, and the index is stored. Afterward, the algorithm executes the three correction scenarios. Each scenario alters the token sequence as well as the chart list and to avoid redundancies. The altered token sequence is stored in a global setting. The scenario is only executed if the changed token sequence is not contained in the set and added to it.
The first scenario is to delete the corrupted chart and sentence token. The parser stopped because the token did not match any of the rules in the state list. Therefore, it is possible that the token should not be in the sentence. However, the content of the corrupted chart is not wrong because it depends on the token prior to the unidentified token. Therefore, the content is duplicated in the adjacent chart to enable the restart of the parsing process. Once the content is duplicated, the algorithm deletes the corrupted chart and the according sentence token. Because the token sequence has been altered, the newly formed sentence must be re-tagged to ensure the tags are unchanged. The already parsed tags are then compared to the newly tagged. If any tag is different, the Earley table is clear from its entire content, and the parsing

process is restarted from scratch. However, if the two token sequences are identical, the former corrupted chart continues the parsing process.



(a) Initial Earley table



(b) Earley table after deleting a token

Figure 5.1: Error recovering system by deleting token

Figure 5.1, illustrates a example of the scenario. For the example, the sentence "Äpfel rote schmecken besser" is parsed with the grammar:

```
S  = NP VP
NP = NN
VP = VVFIN ADJD
```

The initial parsing returns an incomplete Earley table to the user. The parser identifies the noun phrase in the sentence and determines the components for the needed verb phrase. However, the following sentence token is 'ADJD', which is not part of the verb phrase. Therefore, the parsing algorithm stops at the chart 1, and the error recovery system is called. The initial parsing effort is presented in Figure 5.1a. After identifying the corrupted chart, the index of the chart is returned. The algorithm then deletes the token related to the corrupted chart from the sentence. The size of the chart has to be decreased because there is one token less in the sentence now. The new sentence is re-tagged. In this case, the tokens are identical before and after changing the sentence. Therefore, the parser resumes the parsing process at chart 1. However, now the word 'schmecken' is related to this chart. The algorithm returns a complete and valid parsing tree to the user as well as the following error message:

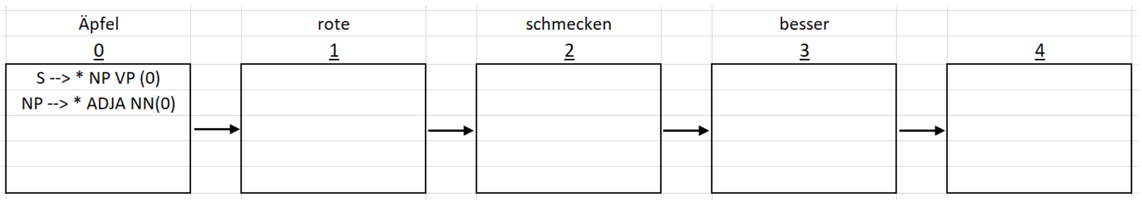
```
[ERROR]: Äpfel rote schmecken besser
Please delete 2. word "rote" from the sentence!
```

The second scenario is changing the positions of the tokens in the sequence. Tokens might not be in the correct order, and therefore, the sentence is not parsable. The index of the corrupt chart is used as a starting point to identify the missing tag. The state list of the chart is iterated to determine the missing tag, and the current literal of each rule is analyzed. If the marker is in front of a terminal symbol, the symbol

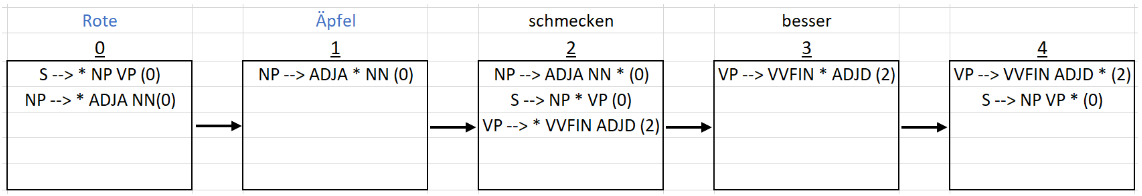
is added to a predefined set and stored. Once all rules are evaluated and analyzed, the set contains all possible terminals needed to continue parsing. The set is then used to determine if a sentence token, which has not been parsed, matches one of the terminals and could be replaced. Therefore, the content of the set is iterated through, and the sentence tokens after the corruption are used as a comparison. For each terminal in the sentence, each token of the sliced sequence is compared. If both match, the token is a possible solution to the incomplete parsing table. The token is removed from its current position and added prior to the unidentified sentence token. Then the altered token sequence is reconstructed and re-tagged. If the already parsed tags are not identical to the re-tagged, the Earley table is clear of its content, and the parsing process is executed from the start. Otherwise, the parsing process is continued with the corrupted chart, which should be parsable with the altered token sequence.
If this process does not return any parsing trees, the possibility remains that the token itself is in the wrong place. The token might be an optional rule literal, and the parser can parse the sentence without the word. Therefore, the token related to the corrupted chart is removed from its current position and added in every position in the sentence. The sentence is then re-build and re-tagged. If the altered tokens are identical to the parsed tokens, the parsing process is continued from the position where the token has been inserted. All charts after this position are clear of their content.
For example the sentence "Äpfel rote schmecken besser" is parsed. The sentence is parsed with two different grammars. At first, the following grammar is used:

```
S = NP VP
NP = ADJA NN
VP = VVFIN ADJD
```

By parsing the sentence with this grammar, the parser fails at the first chart in the list. The Earley algorithm tries to identify a noun phrase, which starts with a word of the type 'ADJA'. However, the first word in the sentence is of the type 'NN'. Therefore, the error recovery system is called with the incomplete table. The Earley table is illustrated in Figure 5.2a. The recovery system iterates through all tags after the first word to fulfill the requirement of the rule. After identifying the word 'rote' as 'ADJA', the algorithm removes it from its original position and inserts it in front of the first word of the sentence. The sentence is re-tagged, and the Earley parser is called again to parse the sentence. The parser can find a complete and valid Earley table with the new sentence, which is presented in Figure 5.2b.



(a) Initial Earley table



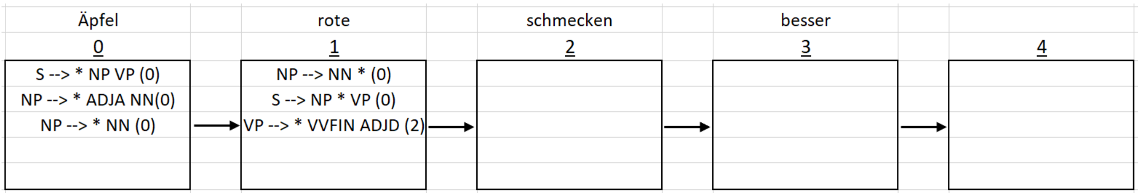
(b) Earley table after switching the token

Figure 5.2: Error recovering system by switching token

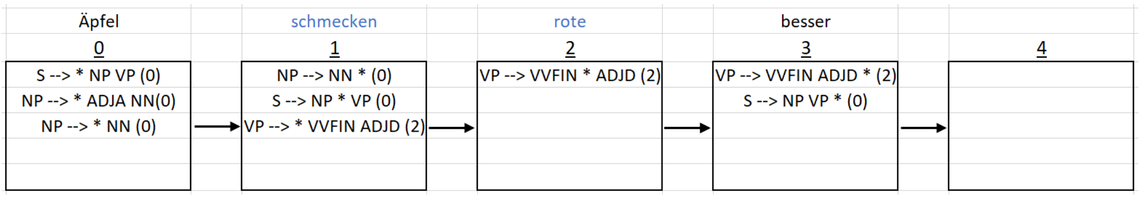
To show the second switching process, the grammar is adjusted and the rule "NP = NN" is added to it, resulting in the grammar:

```
S  = NP VP
NP = ADJA NN
NP = NN
VP = VVFIN ADJD
```

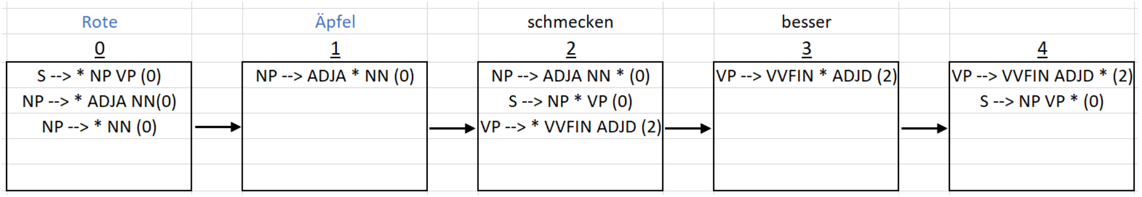
The initial Earley table is displayed in Figure 5.3a using this grammar. The table cannot be parsed entirely. However, the parsing process stops at the chart 1. The algorithm tries to identify the needed verb phrases and fails at it.



(a) Initial Earley table



(b) Earley table after switching the word 'schmecken'



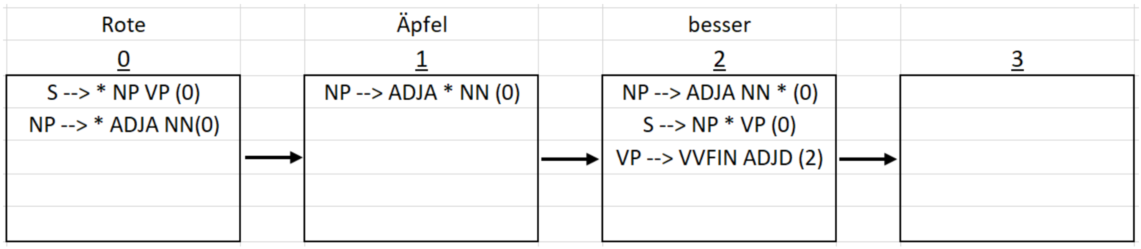
(c) Earley table after switching the word 'rote'

Figure 5.3: Error recovering system by switching token

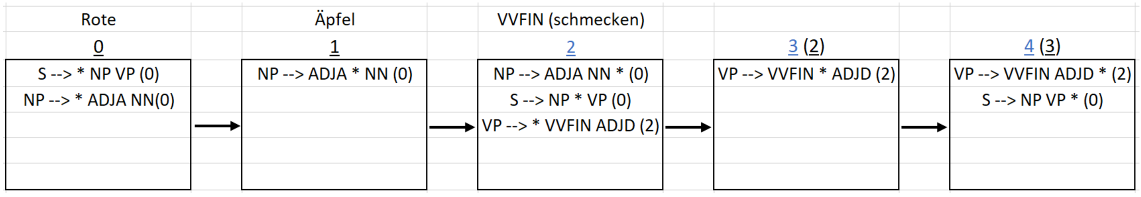
With the initial Earley table, the error recovery system is called. At first, the system iterates through all words after the corrupted position. The word 'schmecken' does meet the rule's requirements, and it is removed from its original position and inserted before the word 'rote', and the algorithm resumes parsing. However, the table contains a finished sentence rule. It does not cover the entire sentence and cannot be valid. The incomplete Earley table is presented in Figure 5.3b. The second procedure is triggered, and the word 'rote' is moved in the sentence. By inserting the word 'rote' in front of 'Äpfel', a valid and complete parsing table is returned, which is illustrated in Figure 5.3c.
For both exemplary scenarios, the identical parsing tree and the error message are returned to the user.

```
    [ERROR]: Äpfel rote schmecken besser
    Please insert a word 'rote in front of 'Äpfel'!
```

The last scenario is adding a new token to the token sequence to continue parsing. Now, the corrupted chart is used as a starting point. At first, the chart list of the Earley table is extended by one chart. Then the content of the state list from the corrupted chart is iterated, and if the current literal of a rule is a terminal, the marker is moved, and the changed rule is added to a different list. Once the iteration is complete, the list contains all rules with a terminal before the marker. Next, a set is created and filled with all skipped terminals to fill the new chart. At last, the token sequence has to be altered to meet the requirements of the list. Therefore, each terminal is added to the sentence at the problematic index. In order to re-tag the changed sequence, not the terminal is added to the sentence but the according word stored in the STTS tag set. Once the terminal is added to the sequence, the sentence is reconstructed and re-tagged. If the tags are identical for and after the re-tagging process, the parsing is continued at the corrupted. Therefore, the rules of the list related to the terminal are added to the following chart. However, if the tags are not identical, the Earley table is clear of its content, and the process is started from scratch.



(a) Initial Earley table



(b) Earley table after adding a token

Figure 5.4: Error recovering system by adding token

Figure 5.4, illustrates a example of the scenario. For example, the sentence "Rote Äpfel besser" is parsed with the grammar:
By parsing the initial sentence, the parser stops at the third chart. In order to continue parsing, the algorithm expects the current token to be a 'VVFIN' and not an 'ADJD'. Therefore, the algorithm stops and cannot find a verb phrase in the sentence. The initial parsing process is presented in Subfigure 5.4a. The incomplete Early table is submitted to the error recovery system. The system identifies that the second chart is corrupted and the reason for the incomplete table. The algorithm learns that the parser expected a token of the type 'VVFIN' by analyzing the chart's content. For that reason, the recovering system alters the token sequence and adds a new word between 'Äpfel' and 'besser', which has the type 'VVFIN'. Because the token size is increased, the chart list is modified to meet the requirements. The algorithm re-builds and re-tags the altered token sequence and ensures the parsed tokens are identical. The error recovery system continues parsing at the chart 2 because the tags did not change. The sentence is then parsed completely, and a valid parsing tree can be obtained. The Earley table of the altered token sequence is illustrated in Subfigure 5.4b. Once the parsing tree is constructed, the algorithm returns an error message to the user:

```
[ERROR]: Rote Äpfel besser
Please insert a word of the type 'VVFIN' between 'Äpfel' and '
    besser'!
```

## Edit-distance 2

The basic principle for edit-distance 2 is identical to the edit-distance 2 of the CYK parser. Once the operations for edit-distance 1 are finished and unsuccessful and the configured limit of the parser is not exceeded, the edit-distance 2 is computed and executed by the parser. Here, the operations are similar to the edit-distance 1, but the error recovery algorithm is called twice. The algorithm starts by deleting the corrupted chart and sentence token. Afterward, the parser is called again to parse the Earley table as far as possible. If the Earley table is not completed, the error recovery system is called recursively with the altered Earley table and token sequence. The algorithm applies the three scenarios to the changed input and returns all found parsing trees with their error message to the recursive call. Then the parsing trees are annotated with the error message regarding the deleted token. Once the deletion scenario is completed, the other two scenarios are called accordingly. All found parsing trees of the three scenarios are stored to be returned to the user. During the entire process, a global set is filled with the already parsed token sequences to avoid redundancies and duplicated error messages. The algorithm is only allowed to parse sentences, which are not contained in the set. The token sequence is directly added to the set and parsed afterward. Once all scenarios are parsed, the parsing trees are created and annotated. They are returned to the user with the error message.

## 5.3 Rule Suggestions Algorithm

A significant disadvantage of parsing is not knowing if the sentences are incorrect or not covered by the grammar. The last two modifications enable the parser to report possible errors to the user. However, sentences might be correct, and the used grammar is flawed. Therefore, the last modification supports a user to build proper grammar by analyzing the input sentence. The algorithm extracts all incomplete parsing trees from a parser. By comparing the extracted sequence with the grammar rules, the algorithm should suggest possible changes. In theory, both parsers can be used to extract possible parsing trees. Practically, the CYK algorithm is the better choice because of its workflow. A bottom-up approach can find significantly more parsing tree segments through the entire sentence. Once the basic Earley parser is stuck, the rest of the sentence cannot be parsed. Therefore, the CYK algorithm is the fundamental parsing technique for the rule suggestions.
In order to apply the suggestion process, an input sentence has to be parsed. The algorithm is implemented as an extension of the CYK algorithm and is hence called a suggestion parser. As a result, the algorithm can perform all methods implemented in the CYK. The suggestions process is designed to either find new possible rules or to create an entirely new grammar. If the suggestion parser is created with an existing grammar, the algorithm tries to pinpoint the best location for inserting the missing sequences.
The suggestion parser starts by parsing the input sentence. If the basic CYK parser does not find a complete parsing tree, the suggestion algorithm is called. The method is structured as a recursive call. At each iteration, the algorithm gets a sequence of tokens and the symbol of the rule. The algorithm tries to find applicable rules, which start with the symbol, and suggest extending the rules. In the beginning, the algorithm starts with the symbol "S", because the sentence annotation is missing. The token sequence is the incomplete parsing tree, which is extracted. In the first iteration, the algorithm collects all possible rules starting with a sentence symbol. The tokens to achieve the rule are compared to the token sequence extracted from the parsing tree. The algorithm matches each token of the tree sequence with the first token of the rule. If two tokens are equal, both tokens are removed from their sequence. If there are tokens prior to the matching token in the tree sequence, the tokens are extracted and stored in a different list for suggestions. If the first token of the rule sequence is removed, the algorithm starts over with the second token. However, the index of the tree sequence is not reset and continues from the deleted token. If the token of the sequence cannot be matched with the tree sequence, the token is skipped. The next token is then analyzed and matched. If a match is found, the remaining tokens of the tree sequence are extracted and stored, as are the tokens of the rule sequence. Once the algorithm matches all rule tokens, the actual suggestion method starts. Two lists are filled with the extracted tokens of the tree and rule sequence during the algorithm's matching. The two lists are called tsl (tree sequence list) and rsl (rule sequence list). Each entry can contain several tokens. The two lists are synchronized with each other. Therefore, if only tree tokens have been extracted, the equivalent entry in the rsl is empty.

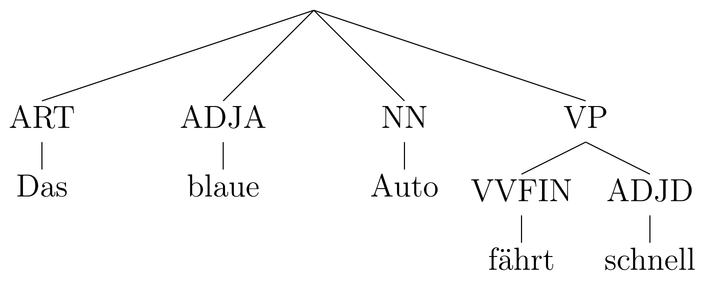
```
1.  S = PP VP NP
        APPR NP VP NP \$.
    rsl = {}
    tsl = {}

2.  S = PP VP NP
        APPR NP VP NP \$.
    rsl = {}
    tsl = {}

3.  S = PP VP NP
        APPR NP VP NP \$.
    rsl = {[PP]}
    tsl = {[APPR, NP]}

4.  S = ~~PP VP~~ NP
        ~~APPR NP VP~~ NP \$.
    rsl = {[PP]}
    tsl = {[APPR, NP]}

5.  S = ~~PP VP NP~~
        ~~APPR NP VP NP~~ $.
    rsl = {[PP]}
    tsl = {[APPR, NP]}

6.  S = ~~PP VP NP~~
    ~~APPR NP VP NP $.~~
    rsl = {[PP], [ ]}
    tsl = {[APPR, NP], [$.]}
```

The listing visualizes the theory in finding possible rules. In step 1 and 2, the literal 'PP' from the rule is compared to the first two of the sentence tokens. Usually, the rule token is compared to all sentence tokens before skipping it, but only the first two sentence tokens are analyzed to create a brief example. Step 3 skips the comparison of the first two sentence token with the second rule token 'VP', which would be in vain. However, the third sentence token matches the 'VP'. Both tokens are removed from their sequence. In front of the matching tokens, there are still several left in their sequence and get removed and added to the rsl and tsl. The entry of the rsl covers the token 'PP' and the related entry of the tsl "APPR NP". Step 4 compares the following two tokens, which match and are removed. In step 5, a literal from the token sequence is left, and the rule sequence is already finished. Therefore, the literal is added to the tsl, and the related entry in the rsl is empty.
If the rule is a possible candidate, the two lists are analyzed, and the entries are computed based on three different scenarios.
The first scenario is that the entry of the tsl is not empty, but the rsl is. The algorithm tries to shorten the token sequence by using the plus operator. Afterward, the shortened sequence is added to the rule at the position they are extracted. Therefore, they are enclosed in round brackets, and a question mark is added to the end of the sequence. For instance, the sequence is extracted between the second and third token of the rule. It is added between them. However, the sequence might extend the rules of the tokens, which surround them. Then, the recursive part of

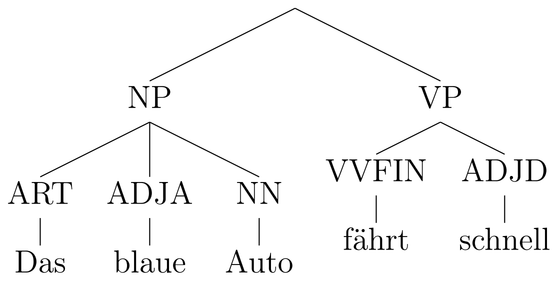
the algorithm is called. The algorithm verifies that rules cover the tokens, and they are not just STTS-tags. If rules exist, the method is called recursively with the symbol of the token and the not reduced sequence. If the algorithm returns any rule suggestions, the initial suggestion is removed due to redundancy.
Another scenario is that there is one entry in the rsl and several others in the tsl. The token sequence is reduced, and the shortened form with the token symbol of the rsl is used to create a rule. The next step is to verify if the rules of the token from the rsl can be extended. Therefore, the algorithm is recursively called, using the not reduced token sequence and the token symbol. In case the rules of the token can be extended, the initial rule suggestions are deleted.
The last scenario is that several entries are in the rsl and must be matched with the sequence in the tsl. The algorithm analyzed every possible combination between the tokens of the rsl and tsl. The key feature is that one token of the rsl needs at least two tokens of the tsl to form a proper rule. Otherwise, the algorithm would not be able to parse the created rule. The token sequence of the tsl is split for each token of the rsl using this restriction. The resulting sublist is then matched with the token of the rsl. Now, all possible combinations of the token sequence are matched with the resulting token. The procedure of adding a combination to a token is equivalent to the second scenario of the algorithm. The algorithm first adds a rule containing the combination and the token, and afterward, the algorithm is called recursively to find possible better suggestions. Once a token and all combinations are done, the next sublist is created considering the restrictions. The algorithm is finished once all tokens are matched with their related sublist.
There are two possible reasons for the disqualification of a rule. Either the entry of the rsl is not empty, and the entry of the tsl is, or the entries in the tsl are not enough to cover all rsl. Each token in the rsl needs at least two tokens of the tsl. Therefore, the rule declines if the number of tokens is minor $n * 2$ when the rsl has $n$ tokens in the entry.
In order to illustrate the theory for rule suggestions, several example sentences are parsed covering every scenario. For this example, the grammar presented in Section 4 is not used due to its complexity. The operation of the algorithm can be shown with a simpler example. During this example, the following grammar is used:

```
S = NP VP
S = NP VP NP
S = NP KON VP
NP = ART NN
VP = VVFIN ADJD
```

The grammar is minimalist but covers most of the scenarios. There have to be slight adjustments to present all successfully. To show scenario 1 and 2, the sentence "Das blaue Auto fährt schnell" is parsed. The parser will not find any parsing tree and will return the following incomplete tree:



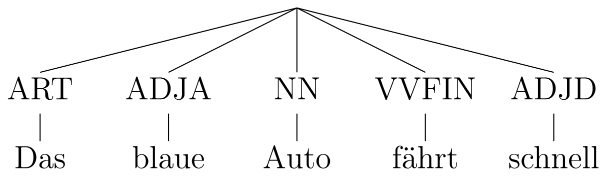
The top-level children of the incomplete tree are used as the token sequence, resulting in "ART ADJA NN VP". Now, the algorithm searches for all rules with the starting symbol "S". To show the rejection process, the rule "S = NP VP" is tested last. The first rule to compare is "S = NP VP NP". The symbol "NP" cannot be found in the entire token sequence. Therefore, the next symbol is verified. The algorithm finds the "VP" at the end of the token sequence. All tokens prior to the symbol are added to the tsl as well as the rsl, resulting in the entries "NP" and "(ART ADJA NN)" (the bracket symbol the tokens related to one entry). Then the algorithm tries to find possible tokens for the last "NP" in the rule sequence. However, there are no more tokens left in the token sequence. The rule gets declined, and it is not possible to expand. The following rule is "S = NP KON VP". The algorithm cannot match any tokens with "NP" or "KON" but with "VP". Therefore, the rsl has the entry "(NP KON)" and the tsl "(ART ADJA NN)". The algorithm declines the rule because the number of entries in the tsl is smaller than the entries of the rsl times 2.
With the two rules declining, only one rule is left. The algorithm cannot find "NP" in the token sequence, but "VP" is removed from the rule and token sequence. The rsl and tsl are created accordingly to the first rule. The rsl contains one element, and tsl contains three. Therefore, the rule might be extendable. At first, the rule is constructed without any further computations, resulting in "NP = ART ADJA NN". After the basic rule is stored, the scenario 2 is triggered, and the algorithm tries to find a more refined version. Hence, the algorithm recursively starts the method with the symbol set to "NP" and the token sequence "ART ADJA NN". There is only one rule in the grammar covering "NP", "NP = ART NN". The first token of the rule matches the sequence, and both are removed from their sequence. Then the algorithm tries to find "NN" in the token sequence. After successfully removing this, the tsl adds as an entry "ADJA", and the rsl adds an empty entry. The sequence is parsed entirely and without any complications. Afterward, the two lists are inspected. Because the rsl entry is empty and the tsl is not, the scenario 1 is triggered. The algorithm tries to reduce the entry of the tsl, which is not possible. Then the algorithm expands the rule with the entry, resulting in "NP = ART ADJA? NN". Because the two adjacent tokens to the added sequence are STTS-tags and not non-terminals, the algorithm is not called recursively again to analyze the rules further. To the user, the suggestion "NP = ART ADJA? NN" is returned. The initial suggestion of scenario 2 is removed because the recursive call found a better solution. If the user changes the rule of the grammar, the parser can find the following valid parsing tree:



In order to illustrate the last scenario, the grammar is changed. All rules are removed except the first one, resulting in the grammar:

```
S = NP VP
```

The parser now knows that a sentence needs an "NP" and "VP" to be accepted. However, no rules define the structure of "NP" and "VP". example presents the ability to create an entirely new grammar.
The algorithm starts by parsing the sentence "Das blaue Auto fährt schnell". Because of the grammar, the parser is not able to match any tokens with the rules. This behavior results in the parsing tree:

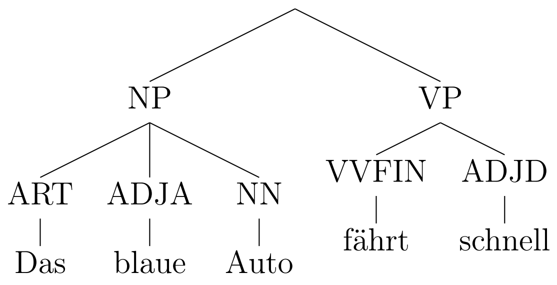


The suggestions algorithm is called to identify possible new rules. At first its starts with the rules containing the sentence symbol. There is only the rule "S = NP VP". The algorithm tries to match the rule's tokens with the token sequence of the tree. This matching might be unsuccessful. Therefore, all tokens of the rule are added to an entry of the rsl "(NP VP)", and the entire token sequence is added to one entry in the tsl "(ART ADJA NN VVFIN ADJD)". Afterward, scenario 3 is triggered. Starting with the first token in the rsl, the suggester analyzed which tokens might be related to "NP". Because each token in the rsl needs at least two tokens to form a valid rule, the last two tokens of the token sequence cannot be related to "NP" and are reserved for "VP". The token sequence "ART ADJA NN" could belong to "NP".
Because the parser cannot understand the semantic of a sentence, the algorithm iterates over the sequence and builds every possible rule. At first, only the first two tokens are used "ART ADJA". Then the algorithm triggers scenario 2 with "NP" as the symbol and "ART ADJA" as the token sequence. The suggester tries to extend possible rules of the grammar starting with "NP". Because there are no such rules, the basic rule "NP = ART ADJA" is added as a suggestion. Now the algorithm adds "NN" to the sequence, resulting in "ART ADJA NN". The same procedure is triggered. It might also be possible that the first tokens of the given sequence do not relate to "NP". Therefore, the first tokens are removed, and the procedure is executed with the sequence "ADJA NN". There are no other possible connections left. The suggester returns to the second token of the rsl's entry and tries to find rules for "VP". Due to the restrictions of the rules, the token sequence "NN VVFIN

ADJD" is used for "VP", leaving two tokens for "NP". Then the same procedure is performed. The algorithm starts adding the tokens to a new sequence, and once all are added, it removes the first tokens. The suggester returns the following rules to the user to expand the grammar:

```
    NP = ART ADJA
    NP = ART ADJA NN
    NP = ADJA NN
    VP = NN VVFIN
    VP = NN VVFIN ADJD
    VP = VVFIN ADJD
```

If the user adds the second and last rule of the suggestion to the initial grammar, the parser can construct a valid parsing tree for the sentence.



If scenario 2 is triggered, the algorithm uses the tokens of the rsl and tsl and executes every possible combination for scenario 2 to avoid redundancies.
The suggestion parser can shorten the token sequence. Assuming the sequence of the example for scenario 1 is "ART ADJDA ADJA NN", and the grammar 5.3 is used, the unknown sequence is "AJDA ADJA". Without using abbreviations, the suggested rule is "ART (ADJA ADJA)? NN". Nevertheless, using a reduction procedure, the algorithm reduced the unknown sequence to "ADJA+". The sequence could be added to the rule, resulting in "ART (ADJA+)? NN". However, the algorithm can improve the rule even further by using the "*"-operator. Because there is only one token in the brackets with a "+"-operator, the sequence can be changed to "ART ADJA* NN". The algorithm suggests the user change the rule to "NP = ART ADJA* NN".
By using the three scenarios, a user can either improve a grammar or create a new grammar.

# 6 Evaluation

The three modifications are tested and evaluated based on different metrics. Most notably is the precision of the implementation. The golden-standard-based evaluation is commonly used to calculate the precision. Therefore, each incorrect sentence is corrected by hand. These improvements are used as a golden-standard to verify whether the algorithms reach the same result. Furthermore, the recall is calculated as well using the golden-standards. Next to the mathematical evaluation, the duration of the process is measured as well. Since the grammar does not change during the evaluation, the duration is static and equal during all parsing processes. Next to the statistical evaluations methods, the semantic of the error corrections are evaluated as well. If a suggestion differs from the golden-standard, but is semantically correct, it is accepted as well.
A MacBook Air 2017 with an Intel Core i5 and 8GB RAM is used for the evaluation.

## 6.1 Brute force error correction CYK

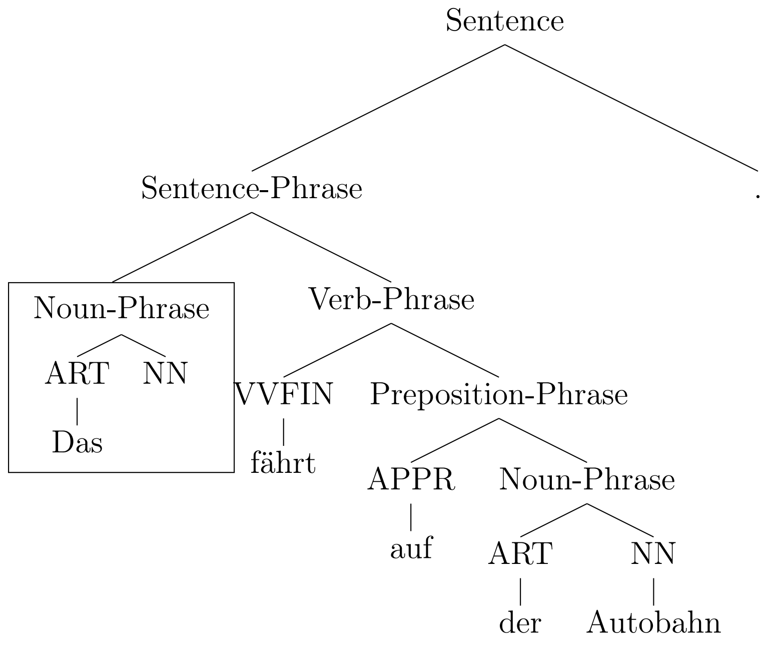
At first, the brute force algorithm is evaluated. By using the PoC domain presented in Section 4, the general functionality of the parser is illustrated. The domain covers 16 syntactically incorrect sentences. Most of them can be corrected with edit-distance 1, only 3 of the sentences need an edit-distance 2 change to be corrected. The golden-standards for all of the incorrect sentences are shown in the Box 4. During the parsing process, each sentence is parsed on its own. The measured duration reflects the time needed to import the grammar, parse the sentence once, complete the error correction, create all parsing trees, annotate the error message and create a CAS instance for each tree. The error messages of the parsing trees are then returned to the user via the console. During the evaluation, each sentence is analyzed by its error messages and the related parsing trees. The 16 sentences of edit-distance 1 can be grouped by deleting, adding, or swapping tokens. The other 3 sentences of edit-distance 2 have mistakes from two groups. At first, the sentences of edit-distance 1 are presented and evaluated grouped by their error type.

## Adding tokens

At first, the sentences are evaluated, which are missing a word. The parser is only able to identify the possible type of the missing token. It is not possible to tell the exact missing word. The suggestion is valid if the sentence is semantically correct. The first sentence is "Das fährt auf der Autobahn.". There is a noun missing between 'Das' and 'fährt', and the parser returns the following error messages:

```
[ERROR]: Das fährt auf der Autobahn.
Please insert a word of the type 'NN' between 'Das' and 'fährt'!
```

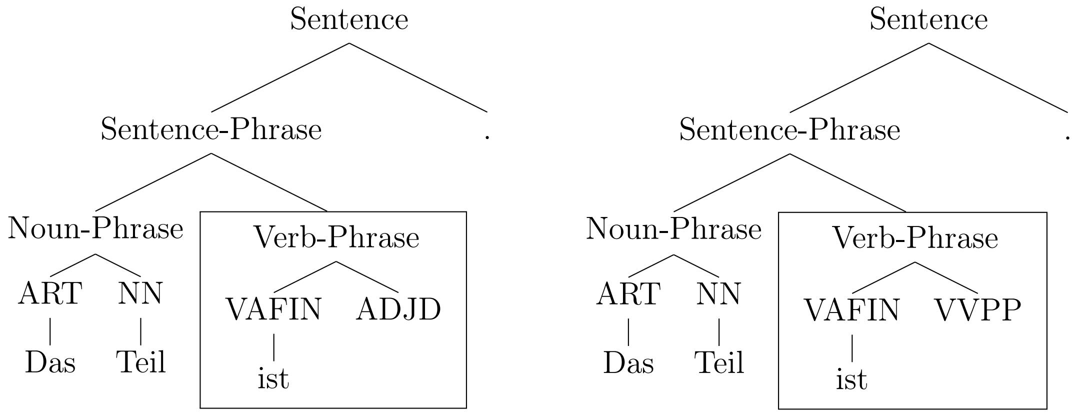
The algorithm identified only one possible error. The sentence lacks a noun between the word 'Das' and 'fährt'. Therefore, the algorithm corrected the error without any problems. The corrected parsing tree is returned to the user.



In the parsing tree, the changes are highlighted with the box. By adding a noun to the sentence, the parser can create a noun phrase and match it with the verb phrase to a sentence phrase. The sentence is then complete and parsable. The error correction took 4244ms to find the parsing tree.
The next sentence is "Das Teil ist.". After parsing and correcting the sentence, the algorithm returns the following error messages:

```
[ERROR]: Das Teil ist.
Please insert a word of the type 'ADJD' between 'ist' and '.'!

[ERROR]: Das Teil ist.
Please insert a word of the type 'VVPP' between 'ist' and '.'!
```

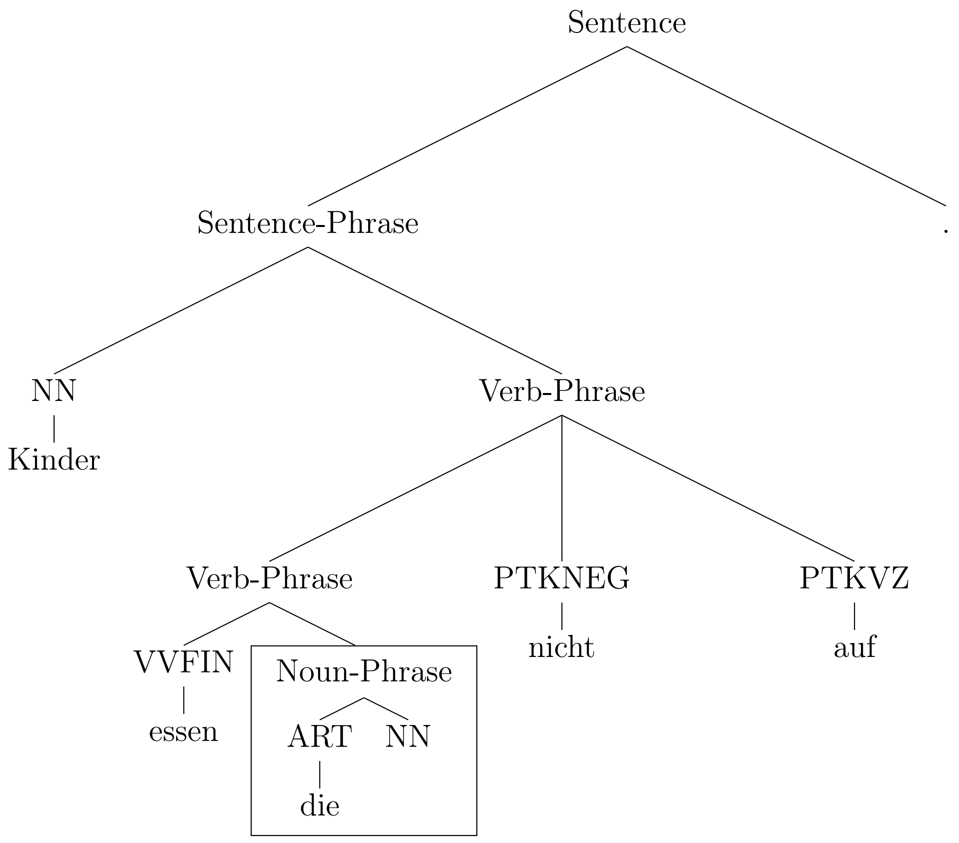
The parser finds two possible improvements for the sentence. An adjective or a verb must be added to the sentence between the words 'ist' and '.'. More precisely, the algorithm suggests that a participial verb or a predicative adjective should be inserted in the sentence. Both suggestions are semantically correct. However, the golden-standard suggests an adjective like 'defekt'. Therefore, the algorithm can identify and correct the error in the sentence. It is even able to give other suggestions as well, which are semantically valid. The sentence could be changed to "Das Teil is ausgeschaltet" by using a verb. Both Trees are similar except for the suggested new word. The algorithm needs 4420ms to find both parsing trees.

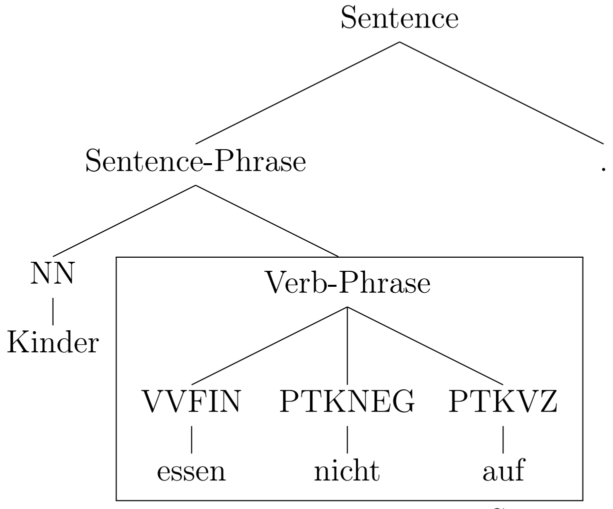


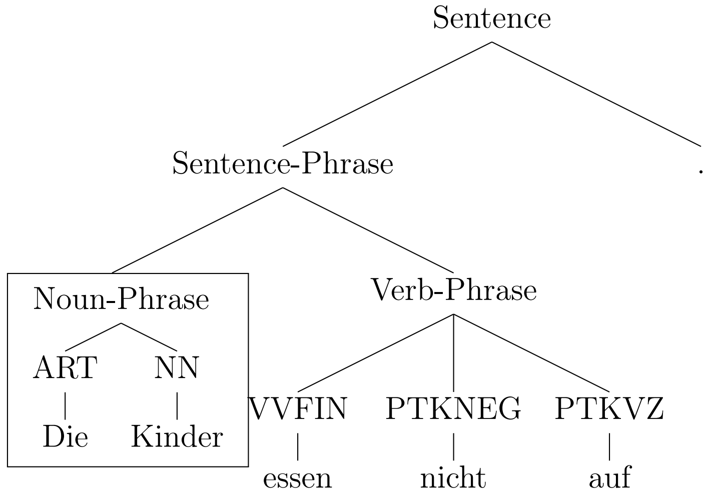
Afterwards, the sentence "Die Kinder essen die nicht auf" is parsed. The desired improvement is highlighting a missing noun between the words 'die' and 'nicht'. For example, the word 'Mahlzeiten' could be inserted. The parser returned the following error messages:

```
[ERROR]: Kinder essen die nicht auf.
Please insert a word of the type 'NN' between 'die' and 'nicht'!

[ERROR]: Kinder essen die nicht auf.
Please delete 3. word 'die' from the sentence!

[ERROR]: Kinder essen die nicht auf.
Please insert the word 'die'' before 'Kinder'!
```

The first error message reflects the anticipated message. The user should insert a noun between 'die' and 'nicht'. However, the parser returned two other suggestions as well. The following message suggests the user should remove the word 'die' from the sentence. The deleting would lead to a semantically correct sentence. The word 'die' is the only indicator that a noun is missing. The sentence's meaning is that the children are not finishing their meal without clarifying what they are eating by removing the article. The last suggestion is to move the article before the word 'Kinder'. This sentence is also semantically correct. The meaning of the sentence is similar to the second suggestion. Translating it to English, the sentence's meaning is "These children are not finishing their meal". The meal does not get specified. The algorithm can find the expected suggestion as well as return two more. The parser needed 4502ms to return the error messages to the user. With the different suggestions, the parsing trees differ.


Sentence
Sentence-Phrase
.
NN
Kinder
Verb-Phrase
Verb-Phrase
PTKNEG
nicht
PTKVZ
auf
VVFIN
essen
Noun-Phrase
ART
die
NN

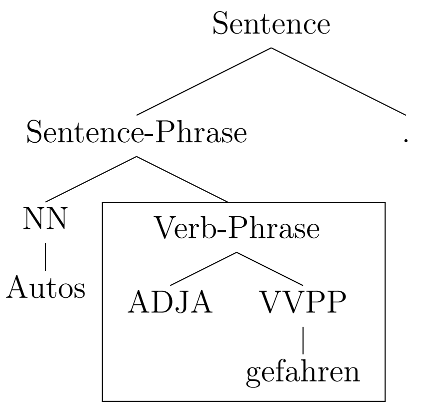

Sentence
Sentence-Phrase
.
NN
Kinder
Verb-Phrase
VVFIN
essen
PTKNEG
nicht
PTKVZ
auf

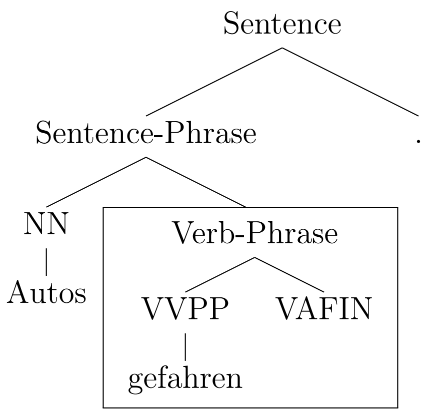

Sentence
Sentence-Phrase
.
Noun-Phrase
ART
Die
NN
Kinder
Verb-Phrase
VVFIN
essen
PTKNEG
nicht
PTKVZ
auf

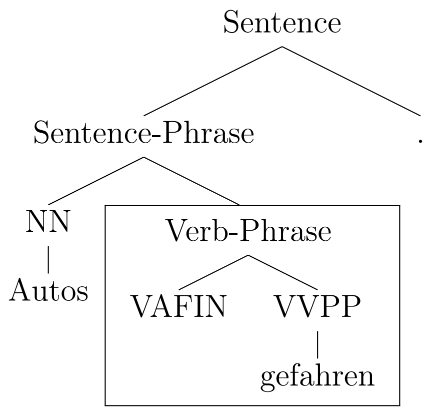
Thereafter, the sentence "Autos gefahren." is analyzed. The golden-standard suggestion is the insertion of a verb between the words 'Autos' and 'gefahren'. The parser returned the error messages:

```
[ERROR]: Autos gefahren.
Please insert a word of the type 'ADJA' between 'Autos' and '
    gefahren'!

[ERROR]: Autos gefahren.
Please insert a word of the type 'VAFIN' between 'gefahren' and
    '.'!

[ERROR]: Autos gefahren.
Please insert a word of the type 'VAFIN' between 'Autos' and '
    gefahren'!
```

The first messages suggest adding a word of the type 'ADJA' between the two words. Even though the correct position is determined, the suggestion does not lead to a semantically correct sentence. There is still an auxiliary verb missing in the sentence. In a more complex sentence structure, the two words may be part of a verb phrase. However, the second part of the verb phrase with the auxiliary verb is still missing. The wrong suggestion might be erasable by restructuring the rules for the verb phrase. The second suggestion does not provide a correct semantic as well. The algorithm identifies, which word type must be inserted in the sentence, but at the wrong position. This wrong behavior results probably from the same problem as in the first suggestion. The rules for the verb phrases might cover many different possibilities and could be restructured for better results. The third suggestion is the anticipated message similar to the golden-standard. The user must insert an auxiliary verb between 'Auto' and 'gefahren' like 'werden'. Then the sentence is grammatically correct. The algorithm needed 4257ms to parse, correct, and build the three parsing trees.



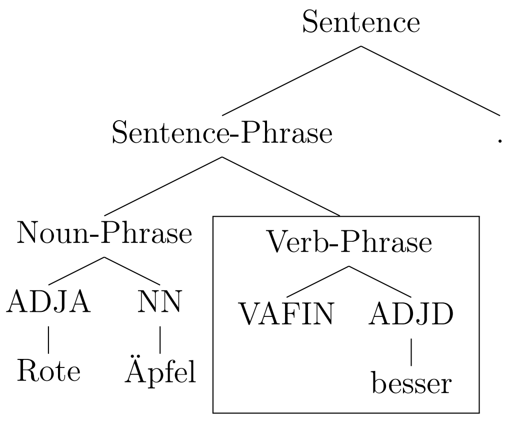


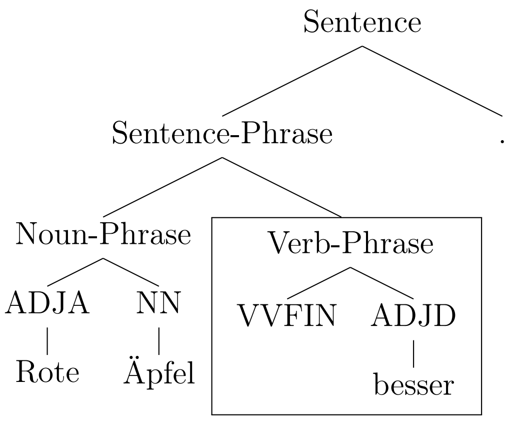


Next, the sentence "Rote Äpfel besser." is evaluated. The ideal suggestion is the insertion of a verb between the words 'Äpfel' and 'besser'. The parser returned the error messages:

```
    [ERROR]: Rote Äpfel besser.
    Please insert a word of the type 'VVFIN' between 'Äpfel' and '
       besser'!

    [ERROR]: Rote Äpfel besser.
    Please insert a word of the type 'VAFIN' between 'Äpfel' and '
       besser'!
    linking verb
```

The first suggestion is to insert a finite main verb in the sentence between 'Äpfel' and 'besser'. These are the standard suggestions. The user should add a word like 'schmecken' to the sentence to be semantically correct. The second suggestion is to insert a finite main verb or linking verb into the same position. In this case, both verb types are possible solutions resulting in a semantically correct sentence. It is possible to insert a verb like 'schmecken' or 'sind', and the sentence is valid. Therefore, both options can be suggested to the user. The correction process needed 4404ms to find all possible suggestions and returns the following parsing trees:

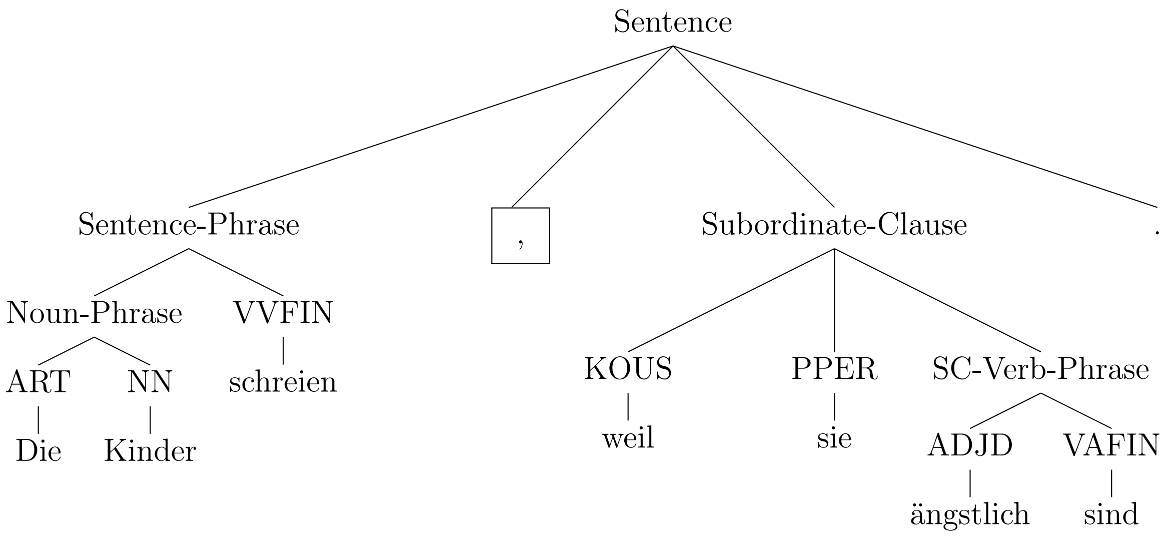




The last sentence is "Die Kinder schreien weil sie ängstlich sind.". In the sentence, the comma is missing, which is probably one of the most common errors. The subordinated clause is not prefaced with a comma; therefore, the parser cannot recognize it. In the golden-standard version, a comma is added between 'schreien' and 'weil'. The parser returns the following error messages:

```
[ERROR]: Die Kinder schreien weil sie ängstlich sind.
Please insert a word of the type ',' between 'schreien' and 'weil'!
```

Only the golden-standard suggestion is returned because the syntax of the main clause and the subordinated clause is too specific to suggest any other suggestion. The user only gets the error message to insert a comma. The entire process took 4975ms.

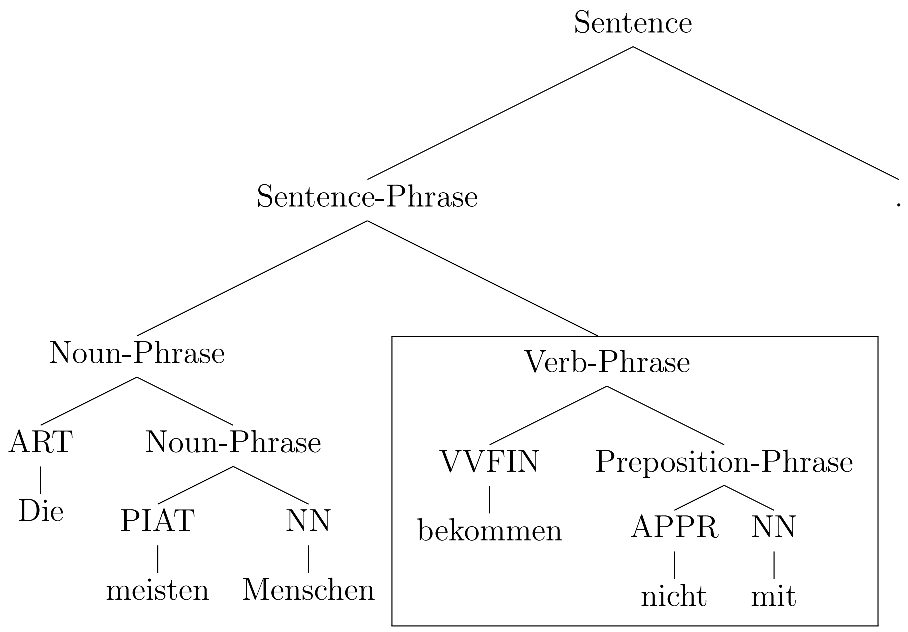


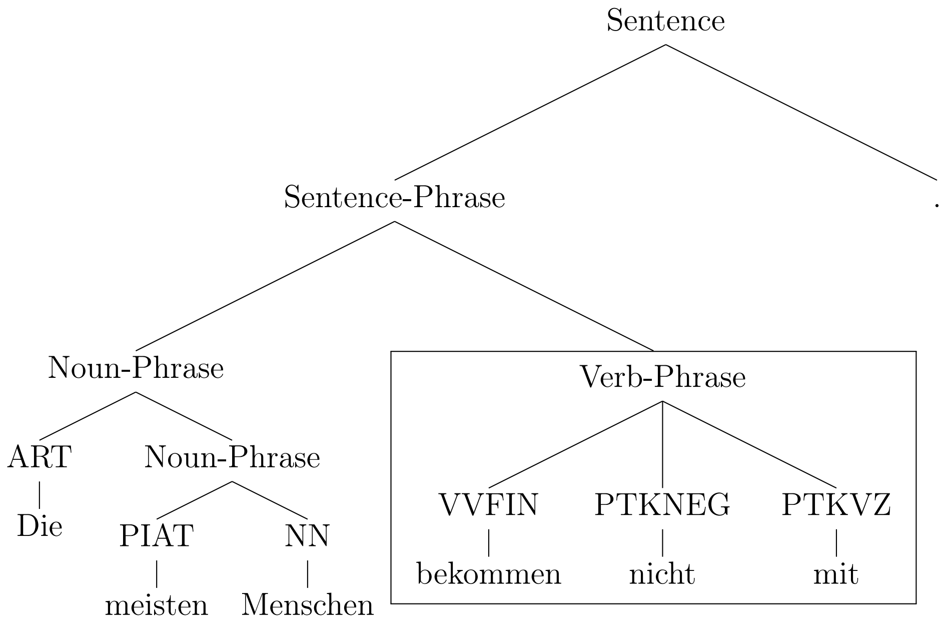
## Swapping tokens

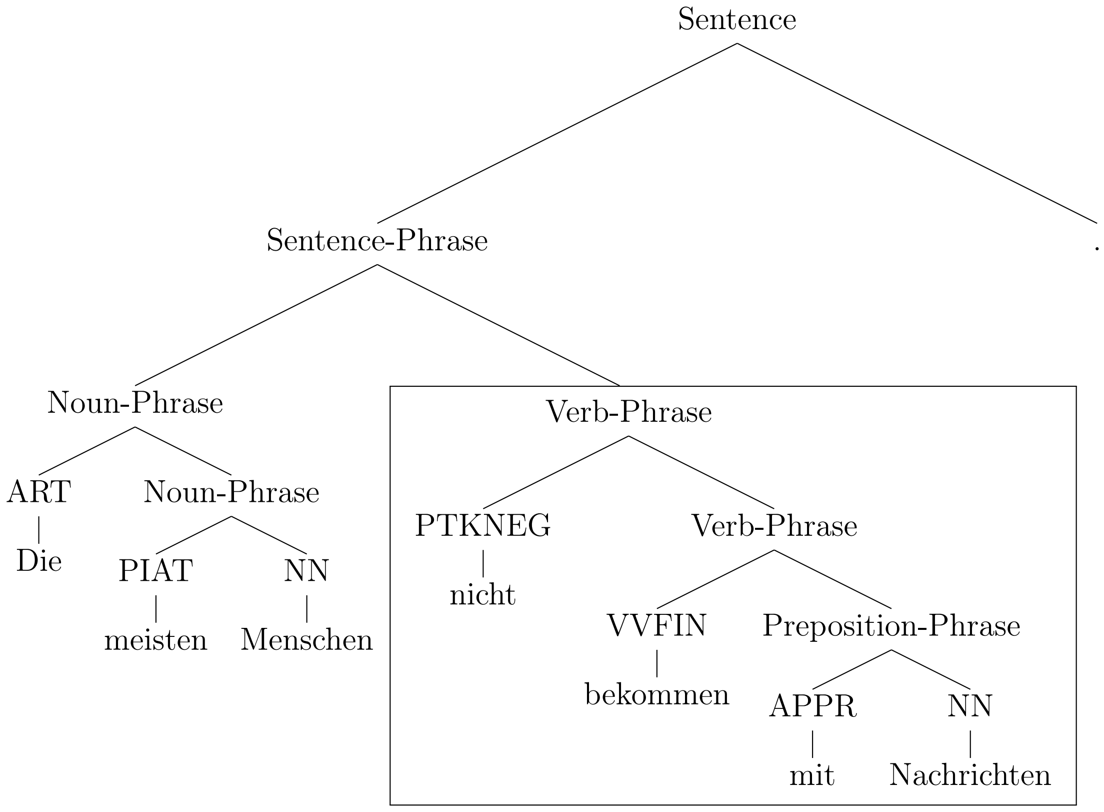
During the swapping, the tokens are placed in every possible position in the sentence. The following two sentences can be corrected by swapping a word. This change would be the ideal suggestion and the golden-standard.
Starting with the sentence "Die meisten Menschen bekommen nicht mit Nachrichten.", the word 'Nachrichten' has to be placed between the words 'bekommen' and 'nicht', resulting in the golden-standard change. The algorithm returns the error messages:

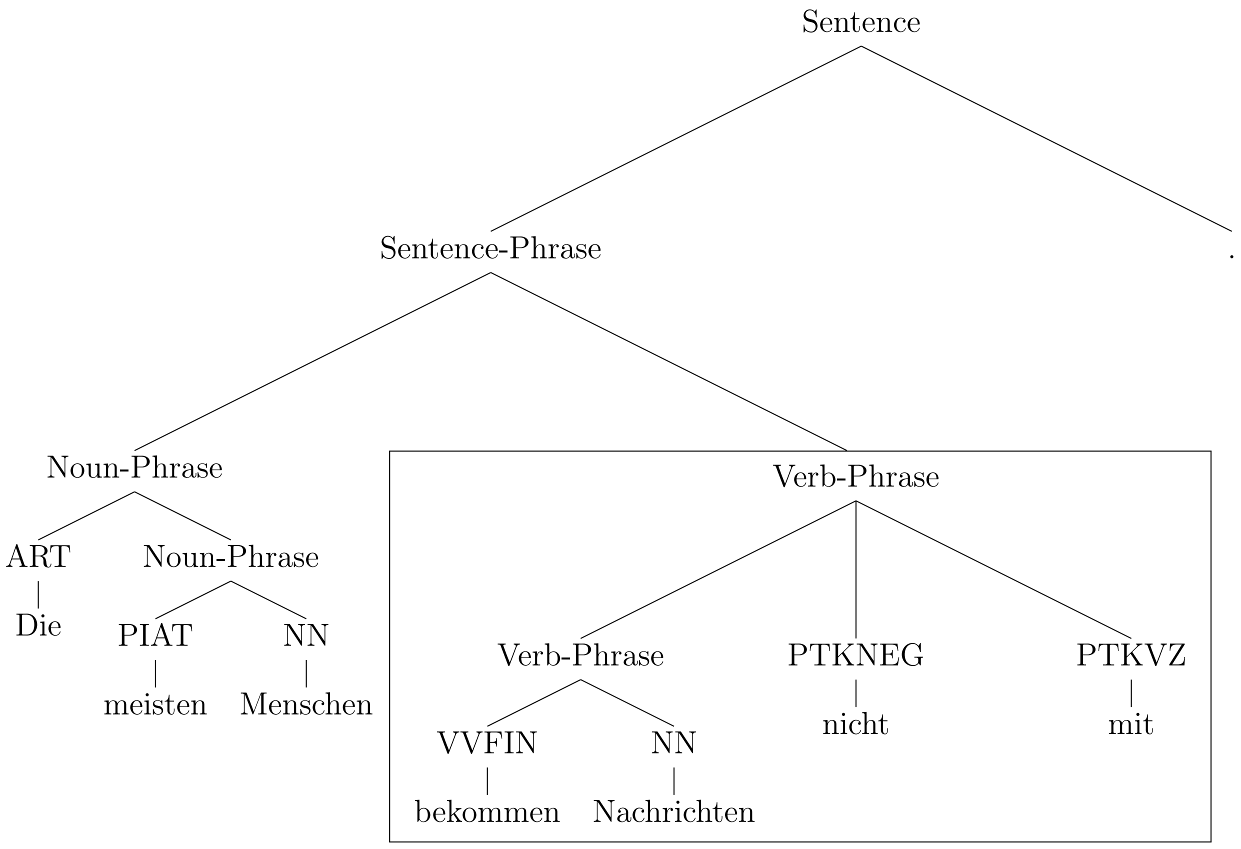
```
[ERROR]: Die meisten Menschen bekommen nicht mit Nachrichten.
Please delete 5. word 'nicht' from the sentence!

[ERROR]: Die meisten Menschen bekommen nicht mit Nachrichten.
Please delete 7. word 'Nachrichten' from the sentence!

[ERROR]: Die meisten Menschen bekommen nicht mit Nachrichten.
Please insert the word 'bekommen' between 'nicht' and 'mit'!

[ERROR]: Die meisten Menschen bekommen nicht mit Nachrichten.
Please insert the word 'Nachrichten' between 'bekommen' and 'nicht
   '!
```

The first suggestion implies deleting the word 'nicht' from the sentence. The words 'mit' and 'Nachrichten' are then combined to a preposition phrase. The sentence is then parsable and returns a sentence annotation. However, the semantic of the sentence is incorrect. The underlying problem is that the parser is not able to understand the message of the sentence. In this case, the preposition phrase is not possible because the meaning is incorrect. Deleting the word 'nicht' does not provide a valid suggestion for the sentence.
The next suggestion is to remove the word 'Nachrichten' from the sentence. A similar problem occurs here because by analyzing the labels of the words, the sentence is correct. However, the semantics and meaning depend on the words. By using the word 'bekommen', there has to be a noun that is related to it. Therefore, the suggestion is wrong as well.
The next suggestion is to replace the word 'bekommen'. This change does not lead to a correct sentence. The problem here might be the number of inaccurate rules. By refining the grammar, it might be possible to erase this false suggestion.
The last error log is equal to the golden-standard. The word 'Nachrichten' is placed between 'bekommen' and 'nicht'. The sentence is then semantically correct.
To parse, correct, and create all the parsing trees the algorithm needed 5146ms. The parsing tree of all four suggestions are illustrated in the next figure:




Sentence
Sentence-Phrase
.
Noun-Phrase
Verb-Phrase
ART
Die
Noun-Phrase
PIAT
meisten
NN
Menschen
VVFIN
bekommen
PTKNEG
nicht
PTKVZ
mit


Sentence
Sentence-Phrase
.
Noun-Phrase
Verb-Phrase
ART
Die
Noun-Phrase
PIAT
meisten
NN
Menschen
PTKNEG
nicht
Verb-Phrase
VVFIN
bekommen
Preposition-Phrase
APPR
mit
NN
Nachrichten

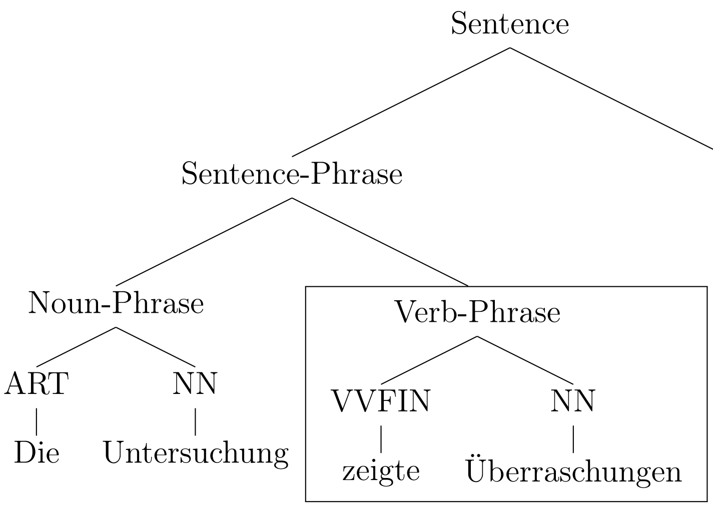


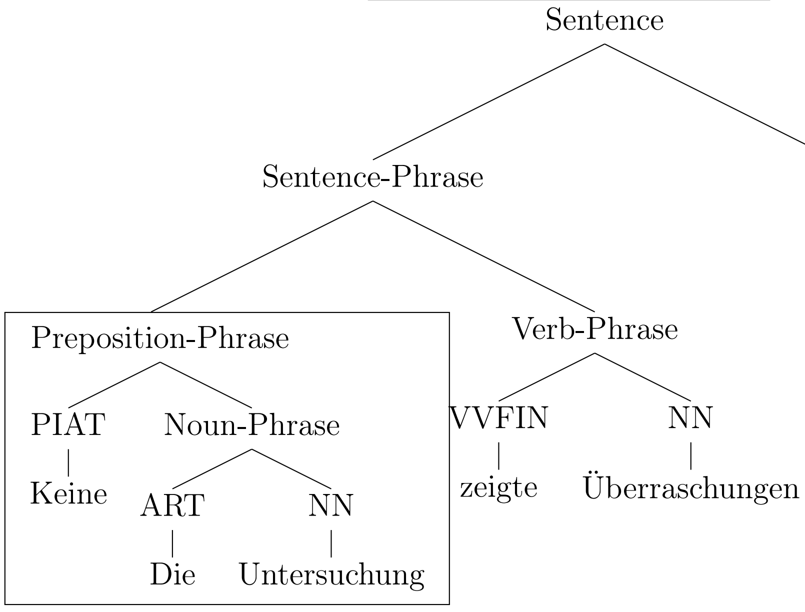
The other incorrect sentence is "Die Untersuchung zeigte Überraschungen keine.". The golden-standard approach is to switch the words 'Überraschungen' and 'keine'. By parsing the sentence, the algorithm returned these error messages:

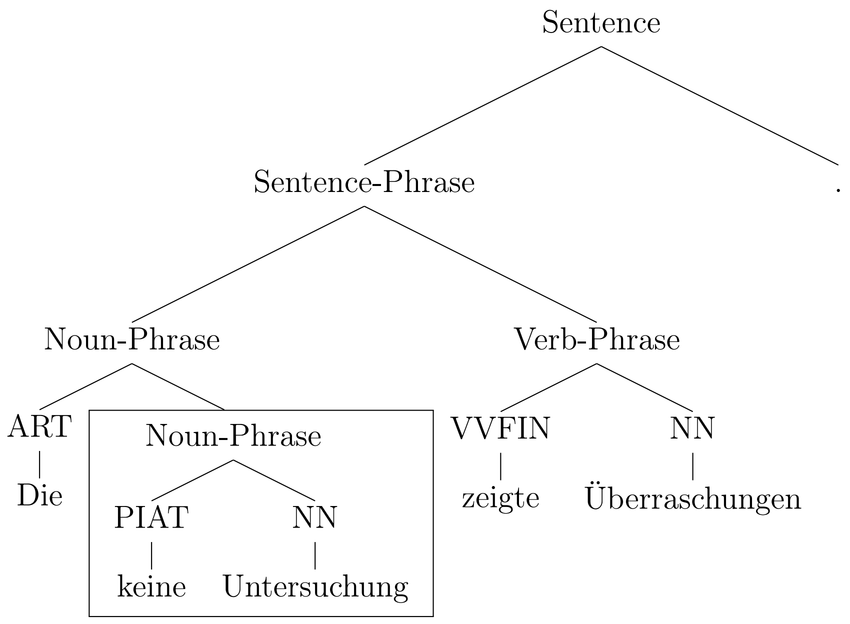
```
[ERROR]: Die Untersuchung zeigte Überraschungen keine.
Please delete 5. word 'keine' from the sentence!

[ERROR]: Die Untersuchung zeigte Überraschungen keine.
Please insert the word 'keine' before 'Die'!

[ERROR]: Die Untersuchung zeigte Überraschungen keine.
Please insert the word 'keine' between 'Die' and 'Untersuchung'!

[ERROR]: Die Untersuchung zeigte Überraschungen keine.
Please insert the word 'keine' between 'zeigte' and 'Überraschungen
    '!

[ERROR]: Die Untersuchung zeigte Überraschungen keine.
Please insert a word of the type 'NN' between 'keine' and '.'!
```

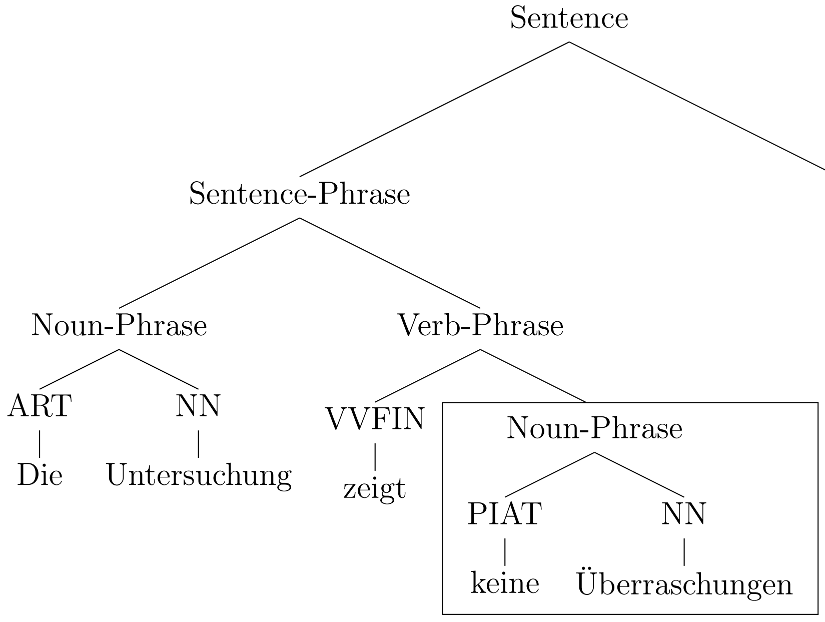
The parsing process found five different suggestions to fix the sentence. The first one is to remove the word 'keine'. Even though the meaning of the sentence is changed, it is semantically correct and valid. The word would negate the sentence; therefore, the sentence is valid without it.
The next two suggestions result from the same problem. By switching the word to the two positions, the wrong rules are achieved. To possibly prevent this behavior, the rules of the grammar could be refined more.

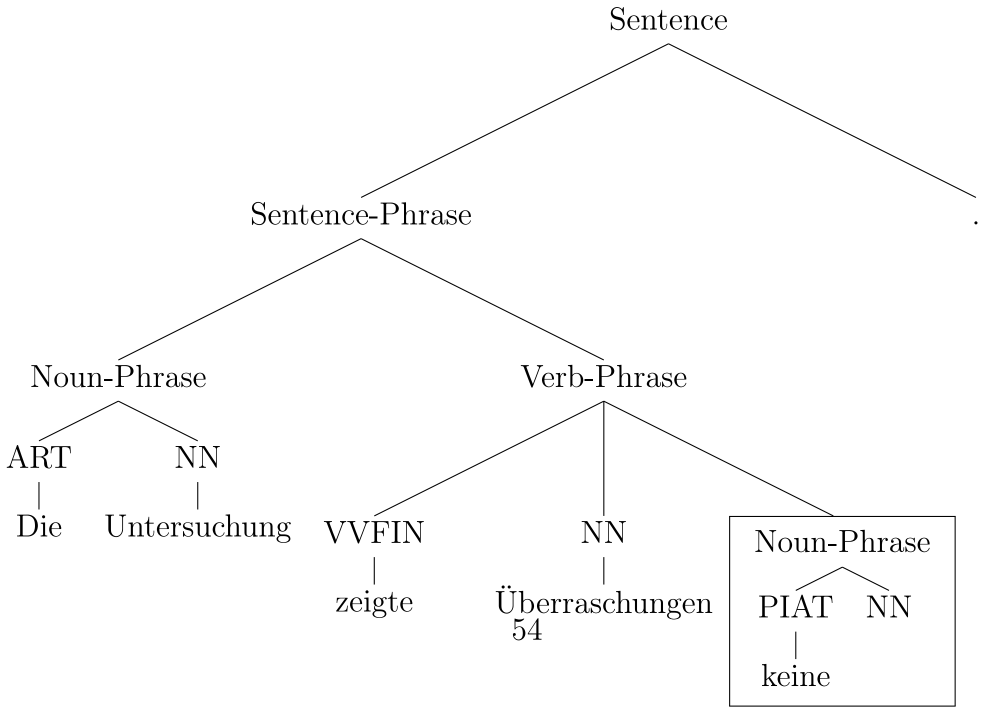
The next suggestion is equivalent to the golden-standard approach. The word is swapped in the correct place and returns a valid sentence.
By implementing the last suggestion, the sentence would be semantically incorrect. The suggestion occurs due to wrongly used rules of the grammar. That may result from the same problem as in the second and third suggestions.
The correction algorithm took 4954ms to parse and create all five parsing trees. To better understand the error messages, the created parsing trees are presented in the figure:






Sentence
Sentence-Phrase
.
Noun-Phrase
Verb-Phrase
ART
Die
Noun-Phrase
PIAT
keine
NN
Untersuchung
VVFIN
zeigte
NN
Überraschungen


Sentence
Sentence-Phrase
.
Noun-Phrase
ART
Die
NN
Untersuchung
Verb-Phrase
VVFIN
zeigt
Noun-Phrase
PIAT
keine
NN
Überraschungen

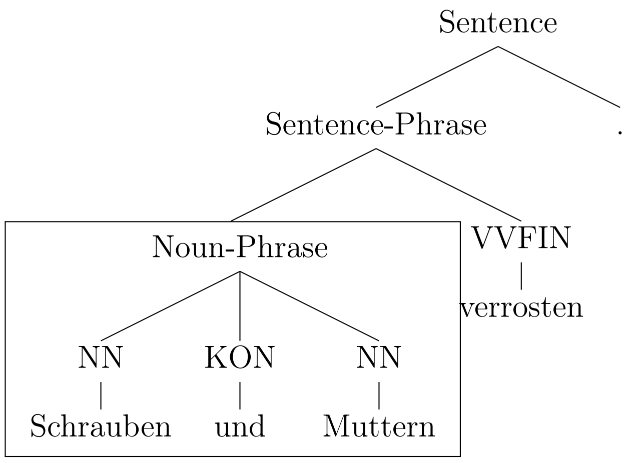

Sentence
Sentence-Phrase
.
Noun-Phrase
ART
Die
NN
Untersuchung
Verb-Phrase
VVFIN
zeigte
NN
Überraschungen
Noun-Phrase
PIAT
keine
NN

## Deleting tokens

The last operation is to delete a redundant token. In each sentence, a word has to be deleted to achieve the golden-standard. The first sentence is "Schrauben und und Muttern verrosten.". The word 'und' is duplicated, and therefore one is redundant and should be removed. The parser returned the error log:

```
[ERROR]: Schrauben und und Muttern verrosten.
Please delete 2. word "und" from the sentence!
```

The only suggestion of the parser is to remove the first 'und' and remove the duplicated word. There are no further suggestions to create any other parsing tree. In order to parse only the minimum amount of time, the parser is configured only to parse PoS-token sequences, which are unknown to it. The algorithm suggests only removing one of the 'und'. If the other 'und' is removed as well, the token sequence is identical after tagging. Even if the words' meanings are different, the parser only decides based on the tags. The algorithm found the golden-standard, and it took 4536ms. The corresponding parsing tree looks like this:

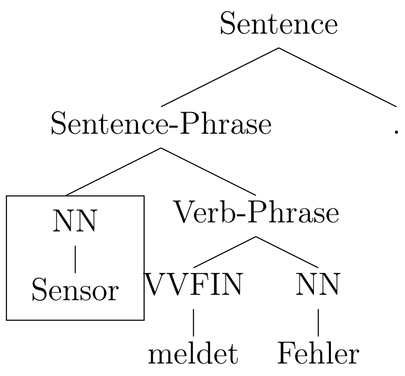


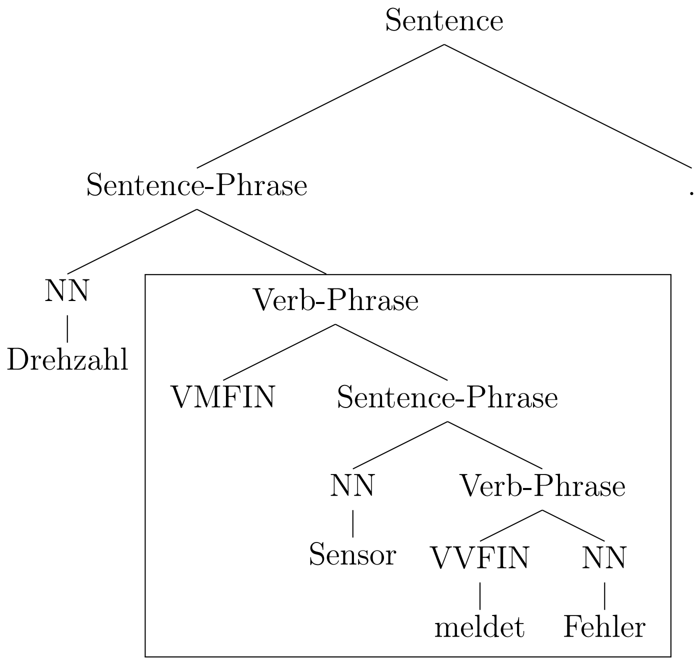
Next, the sentence "Drehzahl Sensor meldet Fehler." is parsed and corrected. The word "Drehzahl" has to be removed from the sentences to achieve the golden-standard. The parser returned the following error messages:

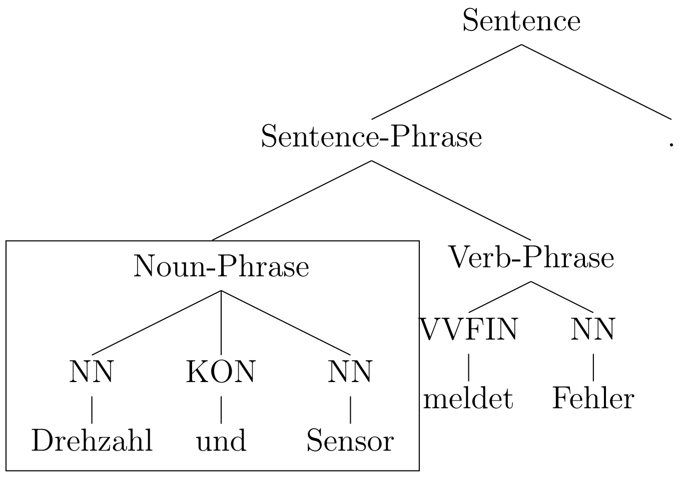
```
[ERROR]: Drehzahl Sensor meldet Fehler.
Please delete 1. word 'Drehzahl' from the sentence!

[ERROR]: Drehzahl Sensor meldet Fehler.
Please insert a word of the type 'VMFIN' between 'Drehzahl' and '
    Sensor'!

[ERROR]: Drehzahl Sensor meldet Fehler.
Please insert a word of the type 'KON' between 'Drehzahl' and '
    Sensor'!
```

The parser finds several parsing trees with the correction algorithm. The first returned error message is the excepted golden-standard improvement. The user has to remove the first word 'Drehzahl' from the sentence. There is no error message to remove the second noun because the token sequences are identical.

The subsequent suggestion results from a flaw in the grammar conversion. The algorithm is designed to minimize the grammar rules and replace sequences with already known sequences to avoid redundancies. In this sentence, the rule "VMFIN PP VP" is used for the suggestion. The grammar covers the rule "SP = PP VP" as well. The algorithm has removed all redundancies resulting in most likely the rule "VMFIN SP". Then the rule is triggered by adding a 'VMFIN' into the sentence. Nevertheless, the rule is not valid and incorrect.
The last suggestion is to insert a word of 'KON' into the sentence between the two nouns. Possible words are 'und' or 'oder', for example. Syntactically these improvements are correct. However, the semantic meaning is doubtful. The verb is singular, and therefore it is not correct when two nouns are joined with a conjunction. Nevertheless, with a different verb, the suggestion would be correct.
In order to compute all possible parsing trees, the algorithm took 4442ms and all of them are presented in the figure beneath:



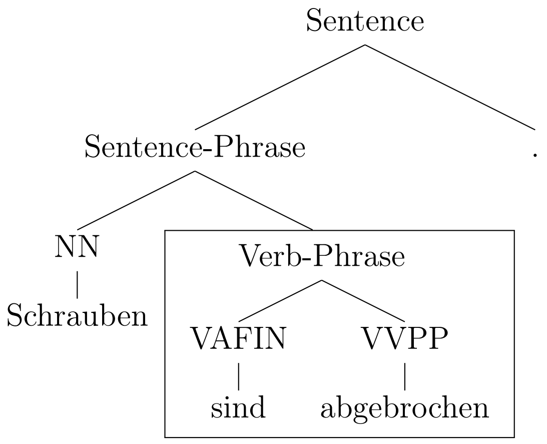




In the next sentence "Schrauben sind sind abgebrochen.", the auxiliary verb is double and needs to be deleted to achieve golden-standard. The parser determined the error log:

```
[ERROR]: Schrauben sind sind abgebrochen.
Please delete 2. word "sind" from the sentence!
```

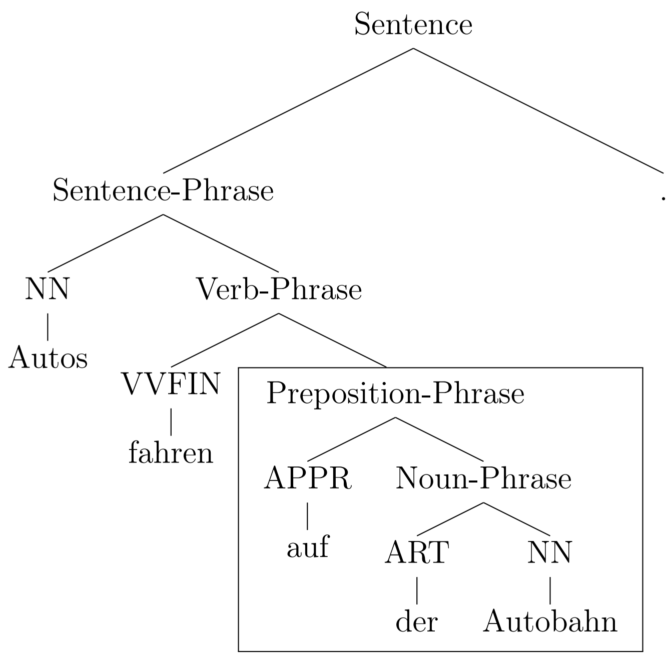
With this simple syntax and complexity, the parser only returns the golden-standard suggestion. Technically, both words could be removed and returned as an error message. The restrictions of the parser prevent this operation. The computation duration is 4310ms, and the created parsing tree is illustrated in the figure:



The penultimate sentence is similar to the last of the adding corrections. In the sentence "Autos fahren, auf der Autobahn.", a comma is redundant. Comma errors are frequent and should be detected. The parsing process resulted in this error message:

```
[ERROR]: Autos fahren , auf der Autobahn.
Please delete 3. word "," from the sentence!
```

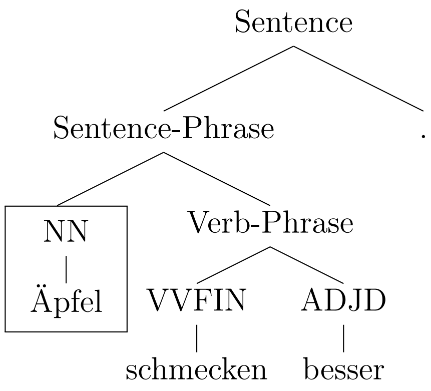
The parser only finds the golden-standard. Due to the different syntax of a subordinated clause, the parser cannot suggest any other operations. By using edit-distance 1, neither part of the sentence could be converted to the subordinated and main clauses. Therefore, only the golden-standard version is returned. During the computations, the algorithm needed 4308ms. The perfectly created parsing tree is illustrated beneath:

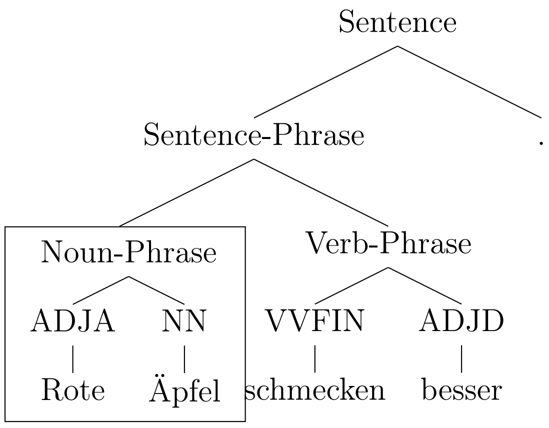


The last sentence is "Äpfel rote schmecken besser.". In this sentence the adjective 'rote' should be removed from the sentence to achieve a correct sentence. The parser returned the following error log:

```
    [ERROR]: Äpfel rote schmecken besser.
    Please delete 2. word "rote" from the sentence!

    [ERROR]: Äpfel rote schmecken besser.
    Please insert the word 'Äpfel' between 'rote' and 'schmecken'!
```

The error correction returns two suggestions. The first is the anticipated golden-standard correction. The user should remove the interfering adjective 'rote'. However, with the second suggestion, the user should place the adjective in front of the word 'Äpfel'. This suggestion leads to a semantically correct sentence as well. The adjective would specify a noun, and the sentence's meaning is only slightly changed. Therefore, both suggestions are valid and can be returned to the user. The computation process needed 4524ms, and both parsing trees are presented.



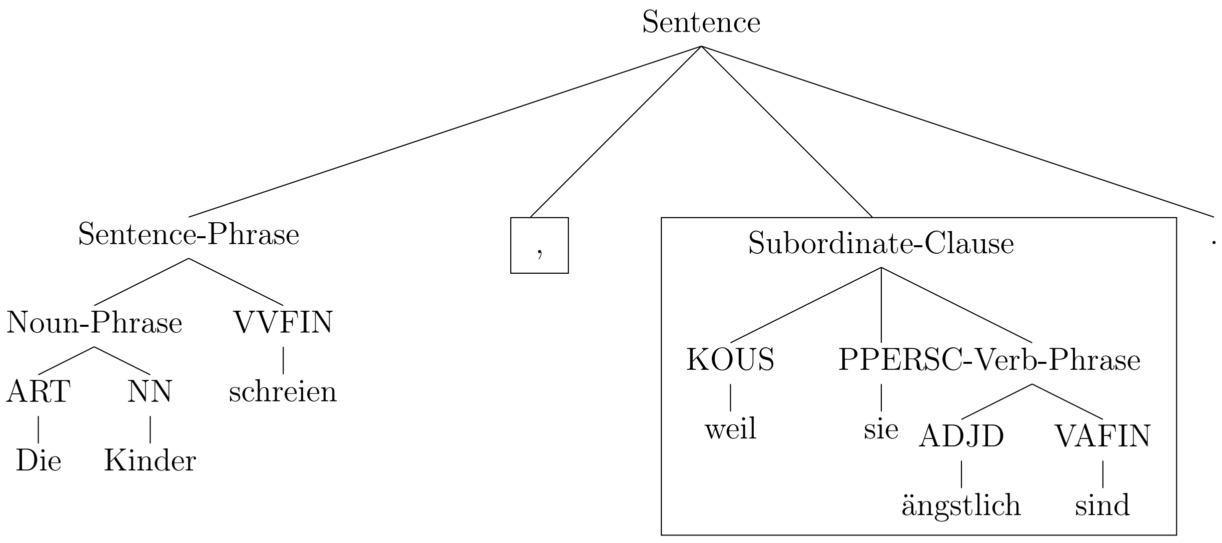


## Combined errors

In order to evaluate the behavior with edit-distance 2, three sentences with every two errors are parsed and should be corrected. The three sentences cover all possible combinations between the operations.
The first sentence is "Die Kinder tanzen schreien weil sie ängstlich sind.". The word 'tanzen' has to be removed to achieve a semantically correct sentence, and a comma needs to be added between the words 'schreien' and 'weil'. The sentence contains a primary and subordinate clause, which makes the detection for the parser probably easier. The parser returns this error log:

```
[ERROR]: Die Kinder tanzen schreien weil sie ängstlich sind.
Please delete 3. word "tanzen" from the sentence!
[ERROR]: Die Kinder tanzen schreien weil sie ängstlich sind.
Please insert a word of the type '\$,' between 'schreien' and 'weil
    '!
```

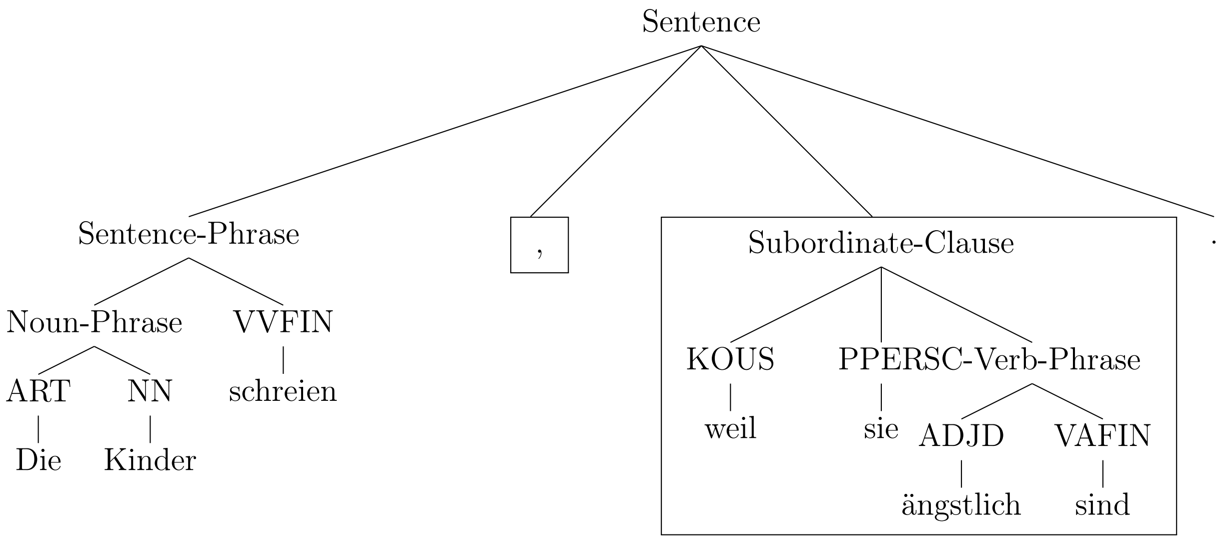
The parser only returns one suggestion containing two corrections. The first word 'tanzen' should be removed, and a comma should be added to the sentence. Therefore, the suggestion is identical to the golden-standard. Of course, it is possible to add the comma first and afterward remove the redundant verb. Due to the restriction of the parser, this suggestion is not returned. With the unique syntax of a subordinate clause, the parser can return only the anticipated suggestion. However, compiling the edit-distance 2 needs 414506ms to complete all operations. The created parsing tree is displayed underneath.



The next sentence is a combination of swapping and adding a token. In the sentence "Kinder die schreien weil sie ängstlich sind.", the word 'die' has to be moved before the word 'Kinder'. Next to this, a comma has to be added between 'schreien' and 'weil'. After parsing, the error message looks like this:

```
    [ERROR]: Kinder die schreien weil sie ängstlich sind.
    Please insert the word 'die'' before 'Kinder'!
    [ERROR]: Kinder die schreien weil sie ängstlich sind.
    Please insert a word of the type '\$,' between 'schreien' and 'weil
        '!
```

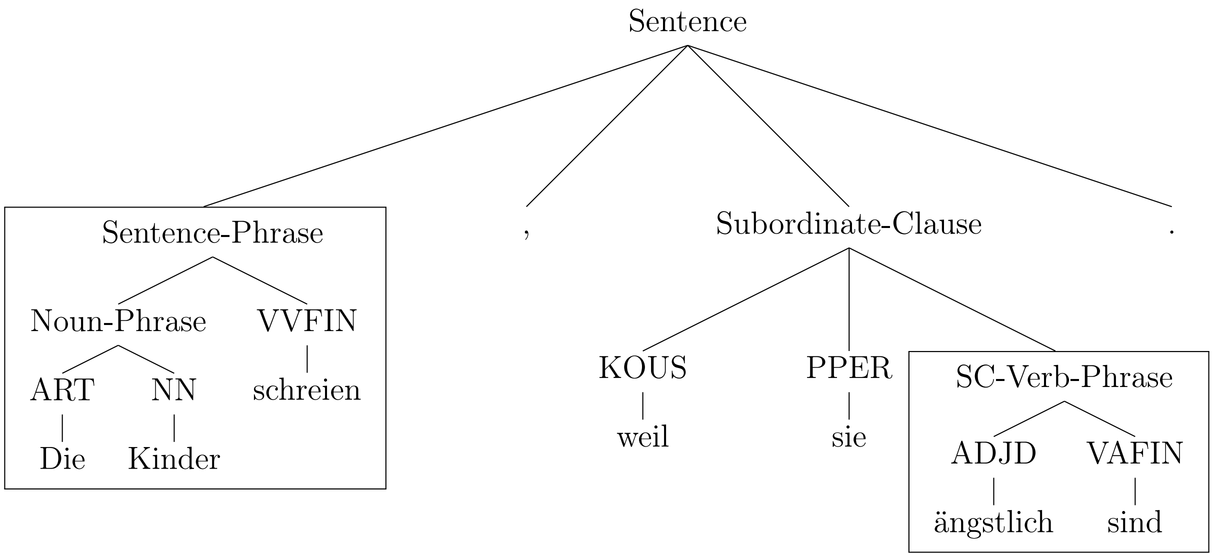
Again the parser only finds the golden-standard suggestion for the sentence. The word 'die' is switched, and then the comma is inserted in the sentence. The order of the correction is irrelevant. The parser needs 536461ms to find the perfect suggestion for the sentence.



The last sentence covers the combination of swapping and removing a token. In the sentence "Kinder die schreien, weil sie ängstlich tanzen sind.", the word 'die' has to be swapped with the word 'Kinder' and the word 'tanzen' should be removed. The parser returns the following error message:

```
[ERROR]: Kinder die schreien , weil sie ängstlich tanzen sind.
Please delete 8. word "tanzen" from the sentence!
[ERROR]: Kinder die schreien , weil sie ängstlich tanzen sind.
Please insert the word 'Kinder' between 'die' and 'schreien'!
```

The parser can find the golden-standard suggestion and returns only it to the user. The user can then remove the interfering word and swap the first two words of the sentence. The unique syntax of the subordinate clause helps the parser to find only the correct suggestion. However, it took the parser 787212ms to complete all possible operations.



## Summary

To summarize the evaluation of the brute force approach, 16 sentences are used to test and evaluate the workflow of the algorithm. In total, the parser returned 32 error messages to the user for the 16 sentences. Of these 21 suggestions, correct the sentence in a semantic and syntactic correct way. In every case, the parser can find and return the golden-standard version of the sentence. The parser achieves a precision [20] of $0,656$ and a recall [20] of 1. However, technically the parser is not the decisive factor. This evaluation proves that the parser can find and report the correct mistakes in every single sentence. Therefore, the workflow and error correction method is working as planned. The incorrect suggestions are also proving the operating modes of the parser. The problem with parsing is the limited intelligence of the algorithm. The significant factor is the intelligence of the used grammar. The parser is not able to decide if a suggestion is better than another. Therefore, the precision and recall of the algorithm refer more to the used grammar than the parser's capabilities.

The dependence on the grammar structure is apparent when a subordinate clause is included in the sentence. There are only a few sentences in the PoC domain covering subordinate clauses. Therefore, the rules are pretty thin and unique. The algorithm does not have much room to make different suggestions by troubleshooting a sentence with a subordinate clause. All three sentences from edit-distance 2 contain subordinate clauses, and even after all possible operations, the algorithm only returns the golden-standard version. All incorrect suggestions of the parser can be traced to a probably overloaded rule set in the grammar. By refining and fine-tuning the rules, the error margin of the parsing algorithm might be lower. The limiting factor is not the parser's technique, but it is the grammar.
In average the parser needed 4450ms to finish the edit-distance 1 operations and 579393ms for the edit-distance 2. The evaluation illustrates the amount of computing power needed to perform all edit-distance 2 operations. Therefore, an operating mode is available, limiting the parser to only use edit-distance 1 at all times.

## 6.2 Error recovery in the Earley parser

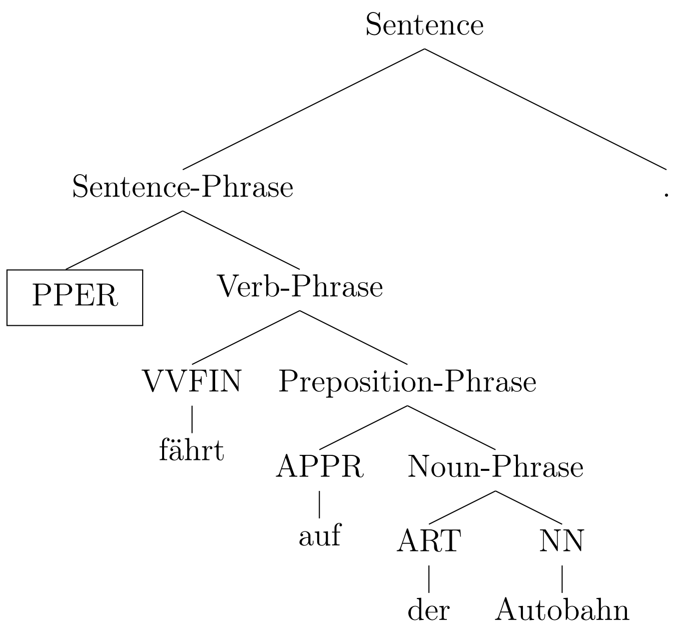
For the Earley algorithm, the same sentences are used to evaluate the behavior and performance of the error recovery system, and the PoC grammar is used as well. At first, the 13 edit-distance 1 sentences are parsed and corrected, and afterward, the 3 error-distance 2 sentences are analyzed. The golden-standard is used as the evaluation method. For the incorrect sentences, the golden standard can be found in box 4. The parser is evaluated on finding the error, creating the correct error message, and returning it to the user with the related parsing tree. The structure of the evaluation is identical to the CYK. At first, inserting is evaluated, followed by swapping, and lastly, the deletion of redundant words. Then the three edit-distance 2 sentences are evaluated.

### Adding tokens

The first sentences are missing a word somewhere in the sentence. Identical to the CYK parser, the Earley algorithm can only identify the type of the word, not an actual word. Nevertheless, the error messages and parsing trees are evaluated based on their syntax and semantic. Only if both are correct, the suggestion is valid.
The first sentences is "Das fährt auf der Autobahn.". The ideal correction is that a noun is missing between 'Das' and 'fährt'. However, the parser returned the following error log:

```
[ERROR]: Das fährt auf der Autobahn.
Please insert a word of the type 'PPER' before 'fährt'!
[ERROR]: Das fährt auf der Autobahn.
Please delete 1. word 'Das' from the sentence!
```

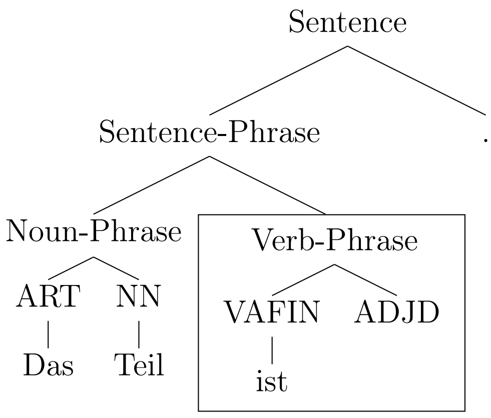
The error recovery system did not identify the corrupted chart correctly. The problem with this sentence is that the tags are different without a noun. Usually, if a noun follows, 'Das' would be tagged as an 'ART'. However, in this sentence, the noun is missing and should be added, and the word is labeled as 'PDS'. Because this tag is unknown to the parser, the algorithm fails at the first Earley chart. By trying to correct the error, the algorithm suggests that the word 'Die' should be removed from the sentences, and a word of the type 'PPER' should be inserted at its place. This would result in a sentence like 'Er fährt auf der Autobahn.'. This correction returns a semantically and syntactically valid parsing tree to the user. However, the sentence's meaning is slightly altered, and the parser returns an edit-distance 2 fix for an edit-distance 1 sentence. Nevertheless, a valid parsing tree is obtained by implementing these suggestions, and the sentence is corrected successfully. This operation needed 4490ms to parse through edit-distance 1 and 2. The following tree is returned from the program

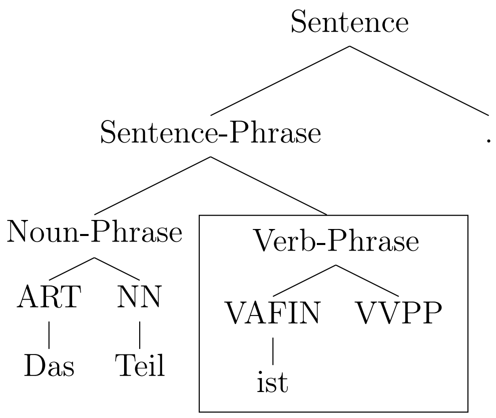


Next, the sentence 'Das Teil ist.' is parsed and corrected by the algorithm. The golden-standard is to insert an adjective like 'defekt' at the end of the sentence. The parser returns the following error log:

```
[ERROR]: Das Teil ist.
Please insert a word of the type 'VVPP' between 'ist' and '.'!

[ERROR]: Das Teil ist.
Please insert a word of the type 'ADJD' between 'ist' and '.'!
```

The error correction algorithm identifies possible corrections to the sentence to return a valid parsing tree. The first suggestion is to add a participial main verb at the end of the sentence. For example, the verb 'abgebrochen' could be added. This improvement differs from the golden-standard but returns a valid and correct parsing tree for the sentence. Therefore, the suggestion is correct. The second improvement from the error recovery system is to add a word of the type 'ADJD' to the sentence. This suggestion is identical to the golden-standard and the anticipated suggestion. The parser returned two valid and correct suggestions to the user and needed 4178ms to complete the parsing process. The two parsing trees are illustrated in the following figure:

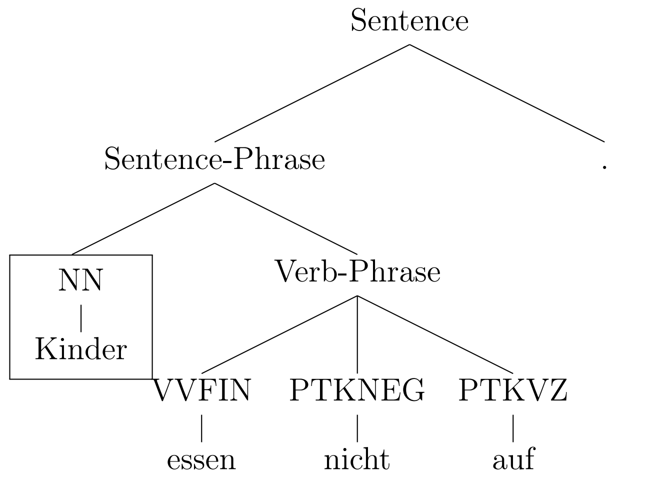


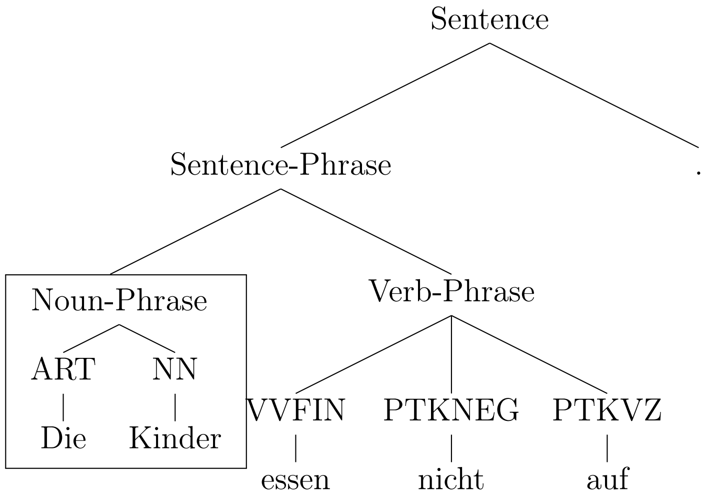


The subsequent sentence "Kinder essen die nicht auf." is parsed. The best suggestion is to insert a noun between the 'die' and 'nicht'. The parser returns the following error log:

```
[ERROR]: Kinder essen die nicht auf.
Please delete 3. word 'die' from the sentence!

[ERROR]: Kinder essen die nicht auf.
Please insert the word 'die' before 'Kinder'!
```

The error recovery system identifies that only deleting the word 'die' can return a valid and correct parsing tree. Nevertheless, the suggestion slightly changes the sentence's meaning, but the sentence is semantically and syntactically correct. Therefore, the suggestion is correct, and the parsing tree is accepted. The next improvement suggests inserting the word 'die' at the beginning of the sentence. That results in a similar semantically and syntactically sentence as the first suggestions. It is also a valid and complete parsing tree. The parser cannot find any other suggestions because of the same reason the first sentence fails. The word 'die' is not assigned the label 'ART', resulting in the parsing failing to find other suggestions. The parser is not analyzing if a word could be added after the word 'die' because the parsing process fails before that. The error recovery system returns two valid parsing trees in 4829ms but fails at finding the golden-standard.

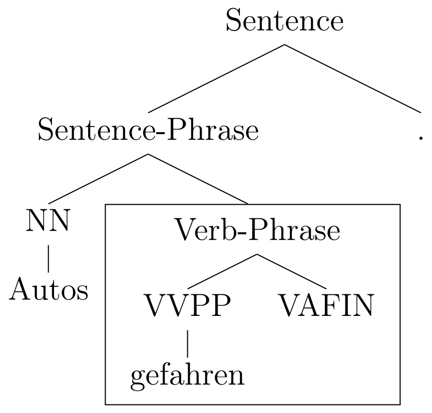




The next sentence is "Autos gefahren.". The ideal suggestion is to insert a verb between the words 'Auto' and 'gefahren'. The parser returns the following error message:

```
[ERROR]: Autos gefahren.
Please insert a word of the type 'VAFIN' between 'gefahren' and
    '.'!
```

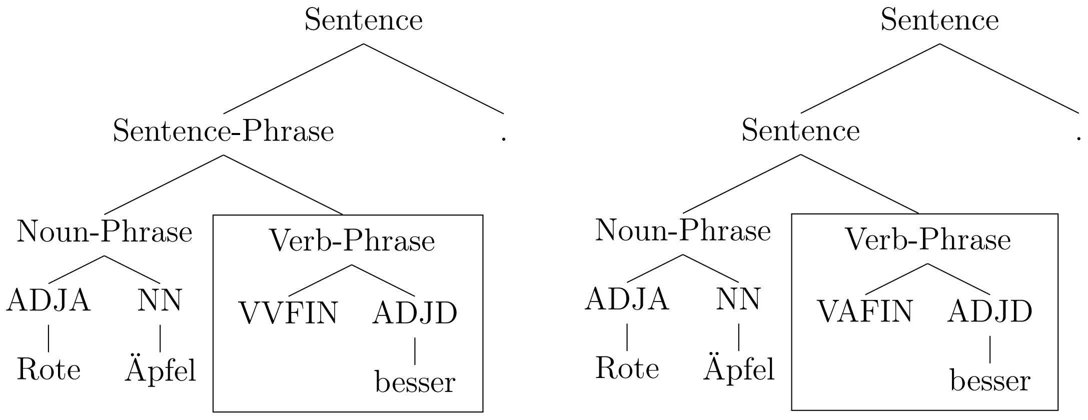
The parser returns only one error message to insert a finite main or linking verb between 'gefahren' and '.'. This suggestion is semantically and syntactically incorrect. During the parsing process, the algorithm tries to complete a sentence phrase. The only applicable rule for this sentence is "SP = NN VP". However, the algorithm searches for all verb phrase rules once the noun is parsed and finds "VP = VVPP VAFIN" rule. The first literal can be parsed and to finish the rule and the sentence phrase, a word of the type 'VAFIN' has to be inserted. Nevertheless, this suggestion is incorrect, but the parser troubleshoots the error at the wrong position. The parsing process needs 4427ms to finish and returns the following incorrect parsing tree:



The penultimate sentence, "Rote Äpfel besser." is parsed with the Earley algorithm. To achieve the golden-standard, a verb has to be added between the words 'Äpfel' and 'besser' to achieve the golden-standard. The error recovery system returned the following error messages:

```
[ERROR]: Rote Äpfel besser.
Please insert a word of the type 'VAFIN' between 'Äpfel' and '
    besser'!

[ERROR]: Rote Äpfel besser.
Please insert a word of the type 'VVFIN' between 'Äpfel' and '
    besser'!
```

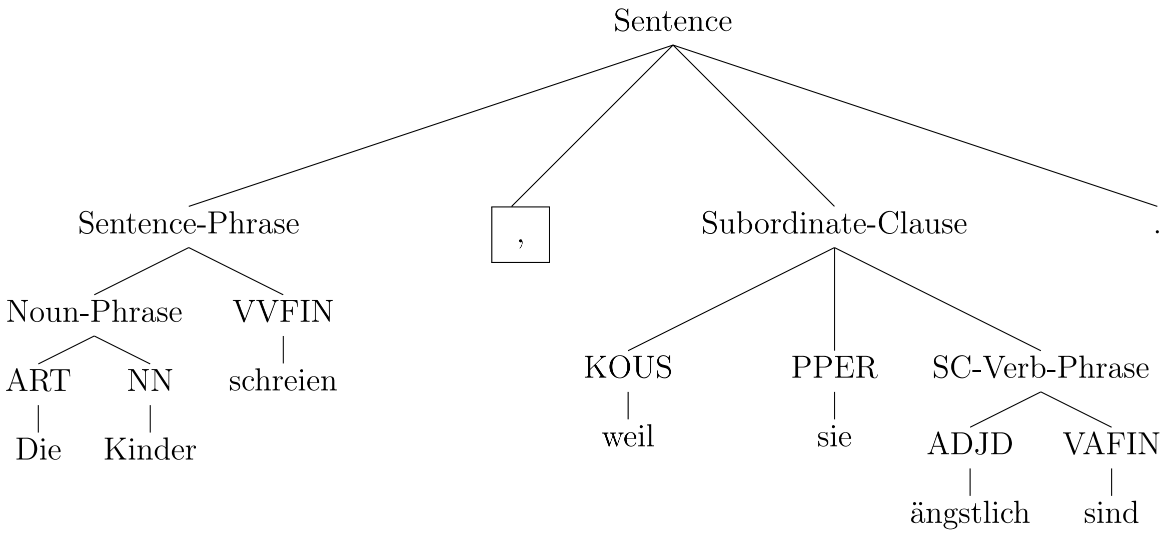
The algorithm returns two error messages and possible improvements to the user. At first, a finite main verb is added to the sentence. That is the ideal suggestion. The sentences could be changed to "Rote Äpfel schmecken besser". The improvement is semantically and syntactically correct and valid. The second suggestion is that a finite main and linking verb can be added to the sentence. Inserting a linking verb would also return a valid and correct parsing tree. Therefore, the suggestion is accepted as well. Both improvements suggest the user adds a verb to the sentence between 'Äpfel' and 'besser', so both achieve the golden-standard. The parsing and correction process needed 4094ms, and both parsing trees are presented in the following figure.



The last sentence covers the most common mistake in any language, a missing comma. In order to achieve the golden-standard in the sentence "Die Kinder schreien weil sie ängstlich sind.", a comma has to be added between the words 'schreien' and 'weil'. When parsed with the error recovery system, the parser returns the following error log:

```
[ERROR]: Die Kinder schreien weil sie ängstlich sind.
Please insert a word of the type '\$,' between 'schreien' and 'weil
    '!
```

The algorithm only returns the ideal suggestion. The iconic syntax of subordinated clause simplifies the detection for the algorithm. In the PoC grammar, words of the type 'KOUS' only occur when used in a subordinate clause. Therefore, the parser stops at the word 'weil' because the introductory comma is missing. The parser only finds one possible suggestion resulting from the unique syntax. However, the improvement is syntactically and semantically correct and needs 4459ms to be returned to the user.



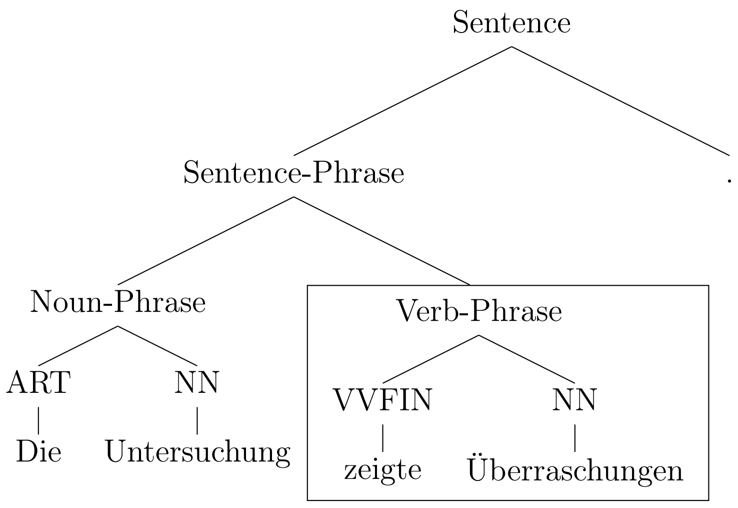
## Swapping tokens

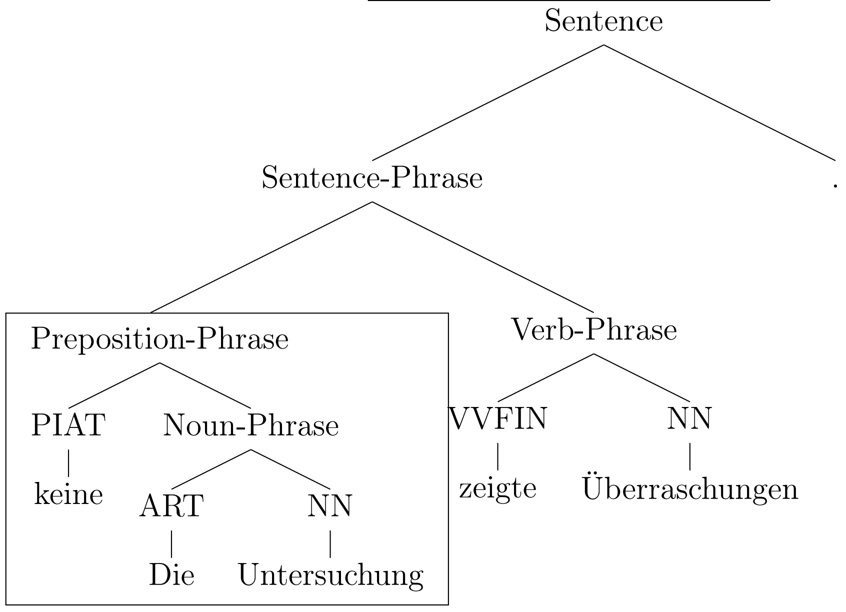
In the subsequent two sentences, a word is not at its correct position in the sentence. The parser should identify the word and rearrange it correctly. That would be the golden-standard for each sentence.
At first, the sentence "Die meisten Menschen bekommen nicht mit Nachrichten." is parsed by the parser. Ideally, the parser would recommend moving the word 'Nachrichten' between the words 'bekommen' and 'nicht'. However, the parser does not return any error message at all and cannot correct the sentence. When analyzing the Earley table returned from the parser, the algorithm can identify the noun phrase at the start of the sentence. However, by identifying the verb phrase in the sentence, the parser fails. The Earley algorithm tries to parse as far as possible and uses the "VP = VVFIN PTKNEG PTKVZ" rule. The first two literals are found in the sentence with 'bekommen' and 'nicht'. Nevertheless, due to the misplaced noun, the word 'mit' is labeled as an 'APPR' and not as 'PTKVZ'. Therefore, the parser stops at the word 'mit' and tries to troubleshoot the problem. Because the parser stops

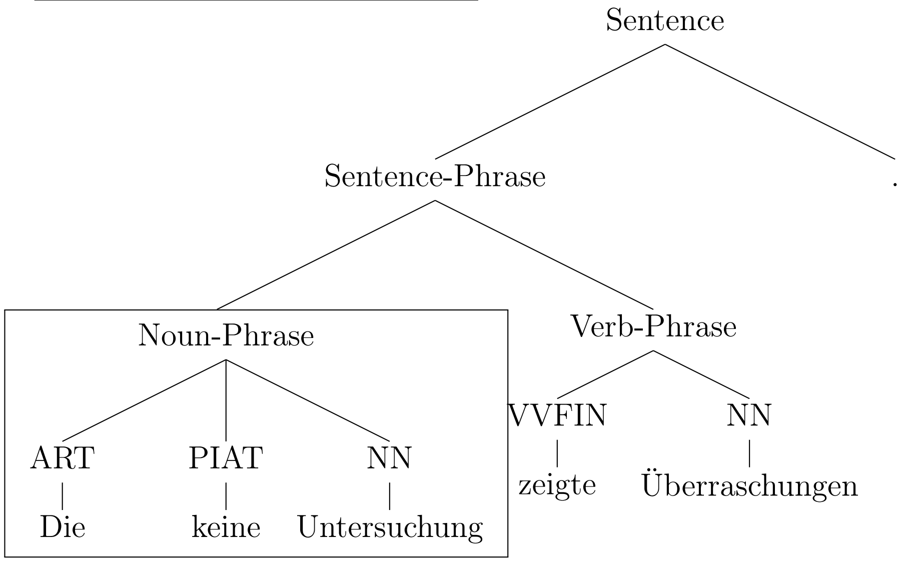
at the wrong word, it cannot find any suitable suggestion to overcome this problem. After 4346ms, the error recovery system is finished and does not return any possible suggestion to the user.
In the second sentence, "Die Untersuchung zeigte Überraschungen keine.", the word 'keine' is misplaced and needs to be inserted in front of the word 'Überraschungen'. After parsing, the error recovery system returns the following error log:

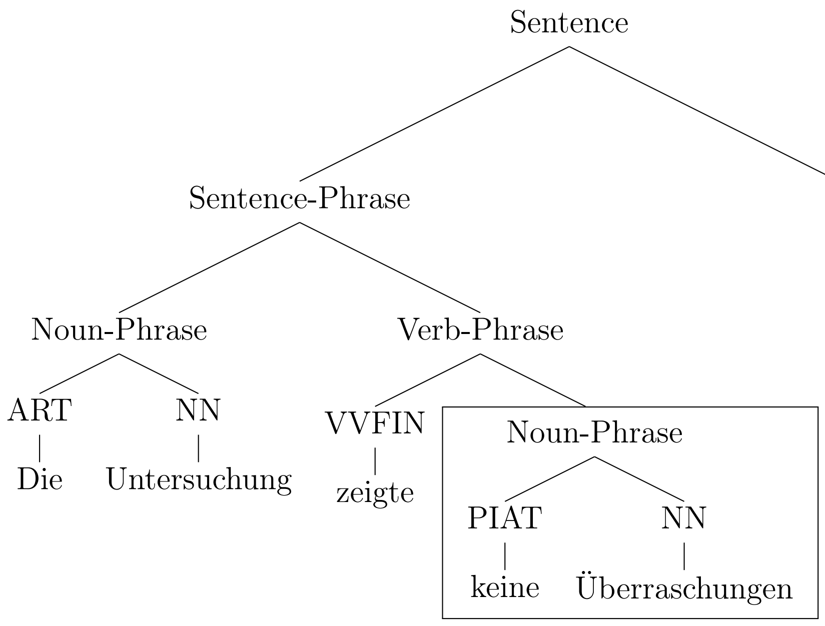
```
[ERROR]: Die Untersuchung zeigte Überraschungen keine.
Please delete 5. word 'keine' from the sentence!

[ERROR]: Die Untersuchung zeigte Überraschungen keine.
Please insert the word 'keine' before 'Die'!

[ERROR]: Die Untersuchung zeigte Überraschungen keine.
Please insert the word 'keine' between 'Die' and 'Untersuchung'!

[ERROR]: Die Untersuchung zeigte Überraschungen keine.
Please insert the word 'keine' between 'zeigte' and 'Überraschungen
   '!

[ERROR]: Die Untersuchung zeigte Überraschungen keine.
Please insert a word of the type 'NN' between 'keine' and '.'!
```

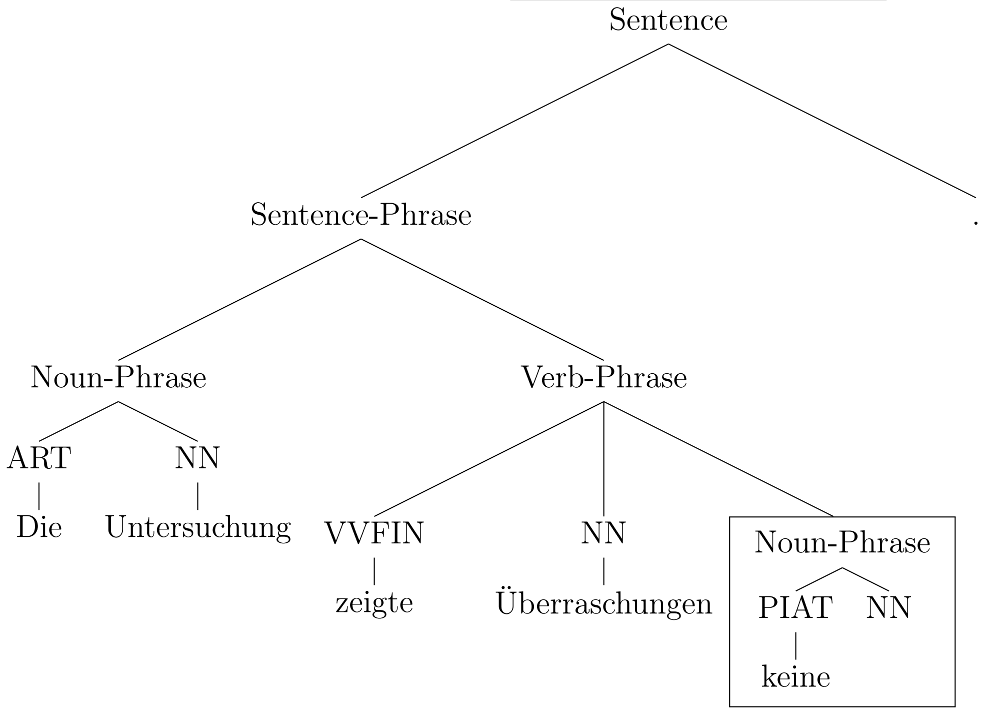
The algorithm returns five different error messages to the user. At first, the system suggests that the user removes the redundant word 'keine' at the end of the sentence. This improvement returns a valid and correct parsing tree. However, the meaning of the sentence is slightly altered.
Alternatively, the error recovery system recommends replacing the word 'keine' at the beginning of the sentence. The first three words are formed into a preposition phrase instead of a noun phrase. This improvement is semantically incorrect and is not accepted.
The following error message suggestion to move the word 'keine' between the words 'Die' and 'Untersuchung'. The first three words are formed into a noun phrase using this improvement, and thereby, the sentence is parsable. However, this suggestion is semantically incorrect as well, and the resulting parsing tree is invalid.
The subsequent suggestion is identical to the golden-standard. The word 'keine' should be moved between the words 'zeigte' and 'Untersuchung'. That leads to a syntactically and semantically correct sentence, and the resulting parsing tree is also valid.
The last suggestion is to insert a noun after the word 'keine'. The syntax of this improvement is questionable. Inaccurate rules probably trigger this suggestion in grammar. By refining the grammar and its rule, it might be possible to avoid this behavior. However, the semantic is incorrect, and therefore the parsing tree is invalid.
The error recovery system needed 5021ms to return these four parsing trees:


Sentence
Sentence-Phrase
.
Noun-Phrase
ART
NN
Die
Untersuchung
Verb-Phrase
VVFIN
NN
zeigte
Überraschungen


Sentence
Sentence-Phrase
.
Preposition-Phrase
PIAT
keine
Noun-Phrase
ART
NN
Die
Untersuchung
Verb-Phrase
VVFIN
NN
zeigte
Überraschungen


Sentence
Sentence-Phrase
.
Noun-Phrase
ART
PIAT
NN
Die
keine
Untersuchung
Verb-Phrase
VVFIN
NN
zeigte
Überraschungen



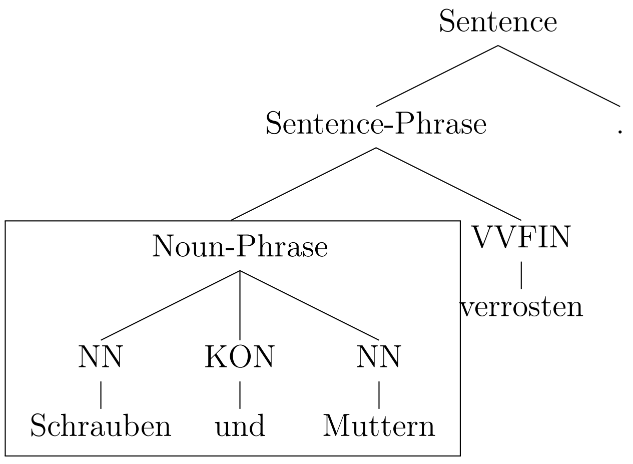


## Deleting tokens

In the last sentences, a word is redundant and must be removed to result in a complete parsing tree. Therefore, the golden-standard for each sentence is to remove the duplicate or unnecessary word.
The first sentence is "Schrauben und und Muttern verrosten.". One of the two 'und' has to be deleted to create a valid parsing tree. The error recovery system returns this error log:

```
[ERROR]: Schrauben und und Muttern verrosten.
Please delete 3. word 'und' from the sentence!
```

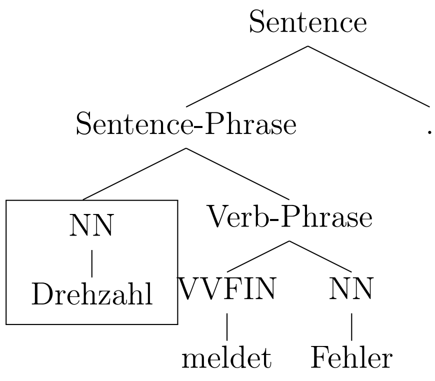
The algorithm finds only one improvement, and it is identical to the golden standard. The error recovery system reports to the user that the third word of the sentence has to be deleted, which is equivalent to the golden-standard. Nevertheless, the first 'und' could also be removed from the sentence since both are identical. The first 'und' was already parsed, and the algorithm stopped at the second word. Therefore, the error recovery system only suggests deleting the second word. For the parsing process, the algorithm needed 4319ms and returned the following parsing tree.

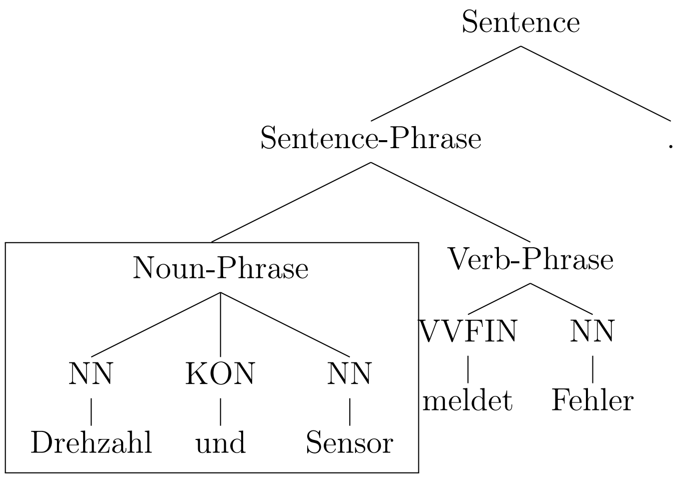


In the next sentence, "Drehzahl Sensor meldet Fehler.", one of the first nouns is redundant and should be removed. The error recovery system suggests these improvements to the user:

```
[ERROR]: Drehzahl Sensor meldet Fehler.
Please delete 2. word 'Sensor' from the sentence!

[ERROR]: Drehzahl Sensor meldet Fehler.
Please insert a word of the type 'KON' between 'Drehzahl' and '
    Sensor'!
```

The first suggestion is identical to the expected golden-standard improvement. The word 'Sensor' should be removed from the sentence to acquire a correct and valid one. Here is the same situation as in the prior sentence. Both nouns can be deleted from the sentence to return a valid parsing tree.
The other improvement suggests adding a word of the type 'KON' between the nouns and using a conjunction to link the two words results in a new noun phrase. However, the semantics of the sentence is slightly off. The verb in the sentence is in the singular form, but by adding the word, the verb has to be in the plural. Syntactically, the sentence is correct and valid, but the semantics are questionable. The error recovery system needed 4347ms to parse, correct, create and annotate the parsing trees in a CAS file.

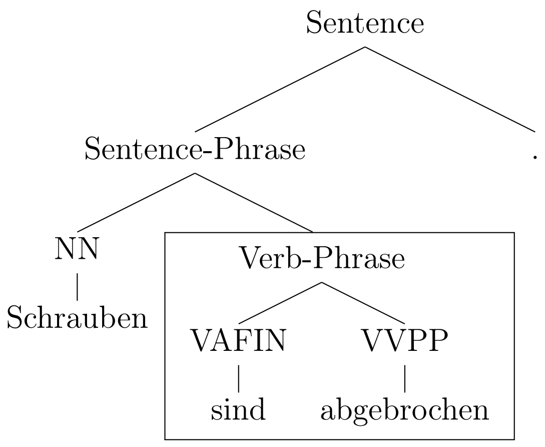




The next sentence is "Schrauben sind sind abgebrochen.". The word 'sind' is duplicated and needs to be removed to return a parsable sentence. This change is the golden-standard for the case. The algorithm returns the following error log:

```
[ERROR]: Schrauben sind sind abgebrochen.
Please delete 3. word 'sind' from the sentence!
```

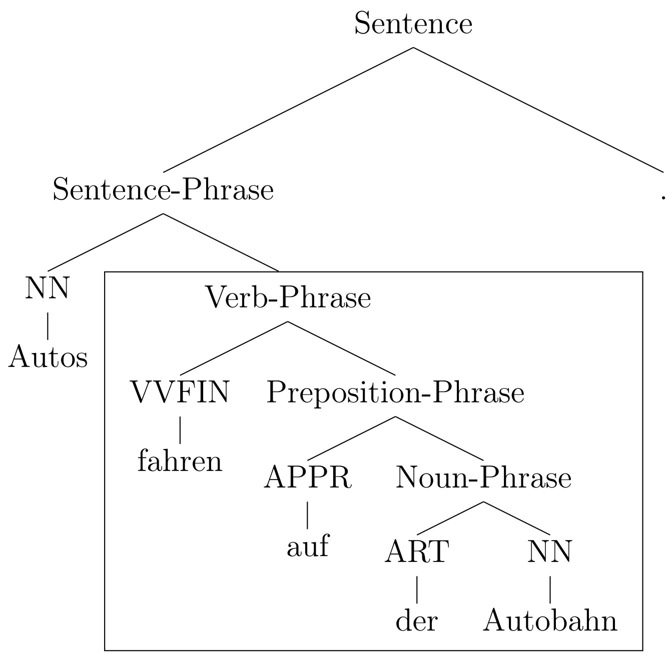
The error recovery system suggests only one improvement to the user: the second 'sind' should be removed from the sentence. The golden-standard is reached using this suggestion. At this moment, it is irrelevant if the first or second 'sind' is removed. The syntax of the sentence is straightforward and is easier to correct. The parser needed 4409ms to find the golden-standard parsing tree.



In the penultimate sentence, "Autos fahren, auf der Autobahn.", the comma is redundant and should be removed. That would also match the golden-standard. The parser returns the following error log:

```
[ERROR]: Autos fahren, auf der Autobahn.
Please delete 3. word ',' from the sentence!
```

The error recovery system returns only the golden-standard suggestion. The advantage of a redundant comma is the unique syntax coming along with it. Usually, a comma indicates either a starting or ending subordinate clause. These have special syntax, and in this sentence, the algorithm cannot identify one. Therefore, the only option is to remove the comma. The returned parsing tree is valid and correct, and it took the algorithm 4349ms to find it.



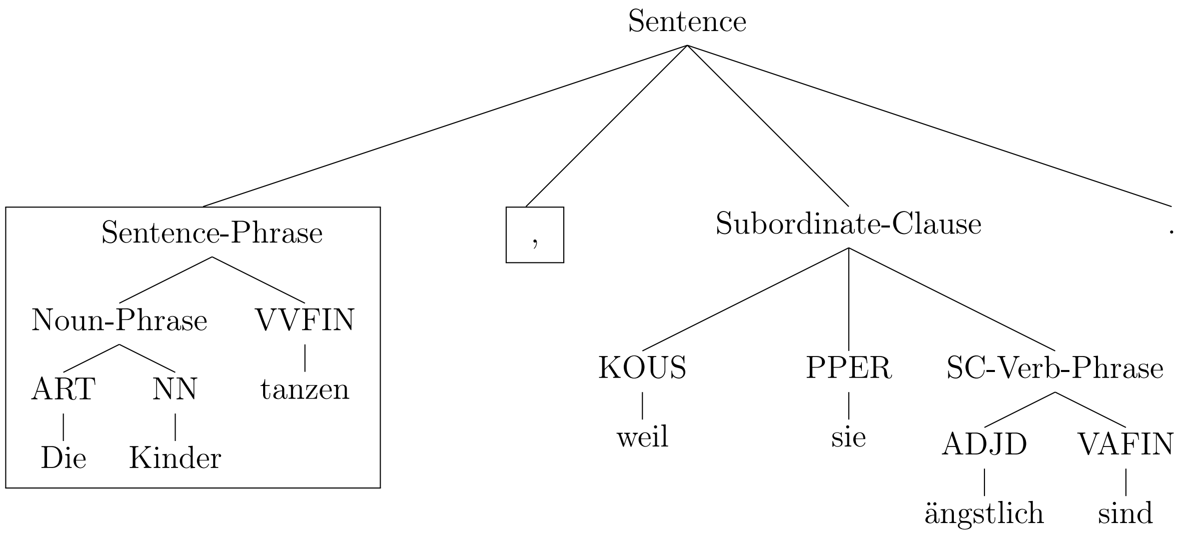
The last sentence is "Äpfel rote schmecken besser.". Ideally, the word 'rote' should be removed from the sentence to get the golden-standard. However, the error recovery system cannot identify a possible improvement to create a parsable sentence. The parser identifies the first word as 'NN' and uses "SP = NN VP" to build a sentence phrase. By identifying the verb phrase, the rule "VP = ADJA VVPP" is used. However, in the context of this sentence, this rule cannot be used. Nevertheless, this leads to the parser stopping at the wrong chart. Therefore, the error recovery system cannot find a possible solution to the unfinished Earley table. The actual corrupted chart is parsed and not analyzed by the system. Nevertheless, the parser needs 4491ms to analyze the incorrect Earley table.

## Combined errors

The last three sentences cover the edit-distance 2 errors. These are a combination of adding, deleting, repositioning tokens in the sentence. Every possible combination between the three scenarios is covered.
In the first sentence "Die Kinder tanzen schreien weil sie ängstlich sind.", the deletion and adding scenarios are combined. In order to get a correct sentence, the word 'tanzen' has to be removed. A comma has to be added between 'schreien' and 'weil'. That would be the golden-standard. The error recovering system returns the following error log:

```
[ERROR]: Die Kinder tanzen schreien weil sie ängstlich sind.
Please delete 4. word 'schreien' from the sentence!
[ERROR]: Die Kinder tanzen schreien weil sie ängstlich sind.
Please insert a word of the type '\$,' between 'tanzen' and 'weil'!
```

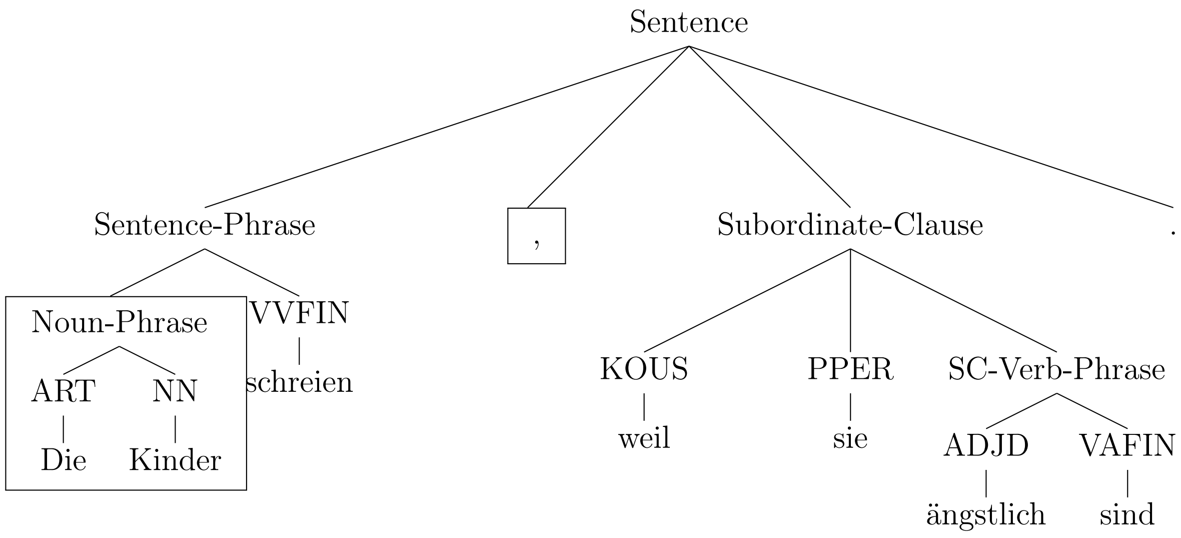
The algorithm suggests only one improvement with an edit-distance 2. At first, the user should remove the word 'schreien' from the sentence. It finds the correct word, which should be removed. However, the first verb could also be deleted, but the parser only judges the tags of the words. The semantic of the sentence is slightly changed but still correct. After deleting the word, the sentence is parsed again, and the error recovery system identifies the correct position where the comma should be inserted. Like the CYK brute force algorithm, a subordinate phrase simplifies the correction process for the error recovery system. The algorithm only returns the golden-standard and a valid and correct parsing tree as well. Overall, the parser needs 4518ms to finish all edit-distance 2 suggestions.



In the next sentence "Kinder die schreien weil sie ängstlich sind.", swapping and inserting a token is covered. Ideally, the error recovery system suggests replacing the word 'die' in front of the word 'Kind' and adding a comma between 'schreien' and 'weil'. That is the returned error log:

```
[ERROR]: Kinder die schreien weil sie ängstlich sind.
Please insert a word of the type '\$,' between 'schreien' and 'weil
    '!
[ERROR]: Kinder die schreien weil sie ängstlich sind.
Please insert the word 'die' before 'Kinder'!
```

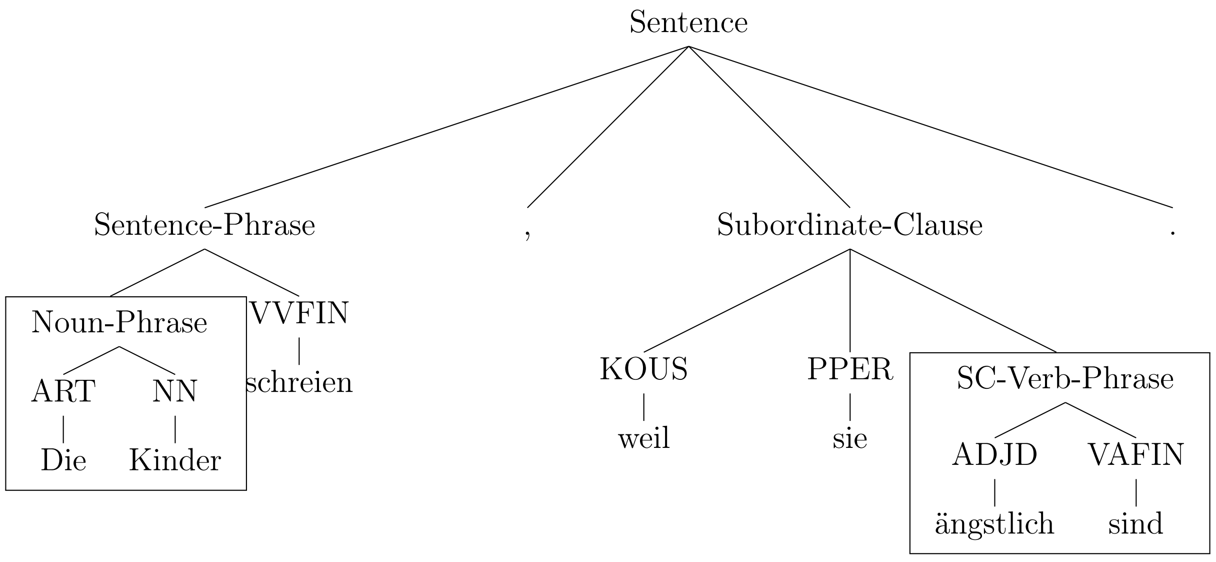
The returned error messages from the parser cover the golden-standard perfectly. The user should switch the two tokens at the start and add a comma between 'schreien' and 'weil'. Similar to the first sentence, the unique syntax of a subordinate clause simplifies the troubleshooting for the error recovery system. Therefore, only one suggestion with edit-distance 2 is returned to the user. The entire error recovery process took 4494ms.



The last sentences "Kinder die schreien, weil sie ängstlich tanzen sind." covers the swapping and deletion of tokens. To achieve the golden-standard, the words 'die' should be replaced in front of the word 'Kinder', and 'tanzen' should be removed. The algorithm returned the following error log:

```
[ERROR]: Kinder die schreien , weil sie ängstlich tanzen sind .
Please delete 8. word 'tanzen' from the sentence!
[ERROR]: Kinder die schreien , weil sie ängstlich tanzen sind .
Please insert the word 'die' before 'Kinder'!
```

The error recovery system only returns one improvement with edit-distance 2. The algorithm identifies the correct suggestions and reaches the golden-standard. To achieve a correct and valid sentence, the user should remove the redundant word and switch the first two words. The error recovery system needs 4658ms to return the user's error log and related parsing tree.



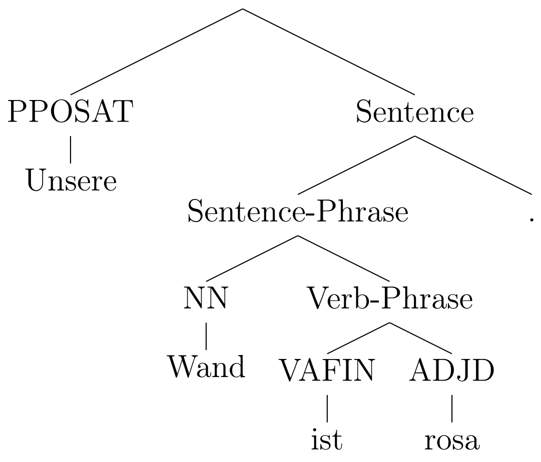
### Summary

To summarize the evaluation of the error recovery system for the Earley parser, 16 sentences are analyzed and evaluated. In total, the system returns 22 suggestions to the user. Of these suggestions, 17 improvements return a valid and correct parsing tree, resulting in a precision score of $0,772$ and a recall score of 1. On average, the algorithm needed 4732ms to read the grammar, parse the sentence, call the error recovery system, and create the parsing trees. There is no time difference between the edit-distance 1 and 2.
However, the error recovery system is not able to return a suggestion in every sentence. In two cases, the parser processed the actual corrupted chart and got stuck at a different chart. That makes it impossible for the system to identify possible solutions to continue parsing. The algorithm has significant deficits regarding replacing tokens in the sentence. These deficits probably result from the parsing algorithm and at which chart the algorithm stopped. The parser did not stop at the wanted chart in both sentences but continued parsing till another breaking point. In the first sentence, the error recovery system then does not find any improvement. However, in the second sentence, the recovery system returns 5 possible solutions. The first is to remove the corrupted token, and the remaining suggestions result from placing the token in every possible position. This procedure returns a correct suggestion, but three incorrect as well. If the replacing algorithm is removed, the error recovery system will identify 18 improvements for the 15 sentences with a precision score of $0,889$. At least in the PoC domain, the error recovery system would score better without the replacing algorithm.
Nevertheless, the precision score of the top-down approach is better than the bottom-up approach. Generally, the top-down approach can troubleshoot the corrupted chart and analyze why the algorithm stopped. The bottom-up approach uses brute-force to find possible improvements and thereby triggers incorrect rules in the grammar. Due to the brute-force approach, the algorithm can find a suggestion for each sentence, unlike the error recovery system. A significant difference is presented in the duration needed to parse the edit-distance 2. In the top-down approach, there is no difference between edit-distance 1 and 2. However, the bottom-up approach needs just seconds for the edit-distance 1 and minutes for the edit-distance 2.

## 6.3 Rule suggestion

At last, the rule suggestion is evaluated. For this process, the last five sentences of the PoC domain are used. At first, the sentence is parsed by the CYK algorithm. If the algorithm does not find a complete parsing tree, the rule suggestion algorithm is called. The suggester tries to find as many applicable rules as possible. The parser is configured to break an annotation into its children if the annotation name cannot be parsed. For example, if a sentence annotation is found in an incomplete tree, the resulting rules are not helpful because a sentence annotation must not be paired with any other annotation. Therefore, it is split into the children, which are used for the suggestion algorithm. When the parser is initialized, the names of these annotations are committed. Typically, only the sentence annotation is broken into its children.

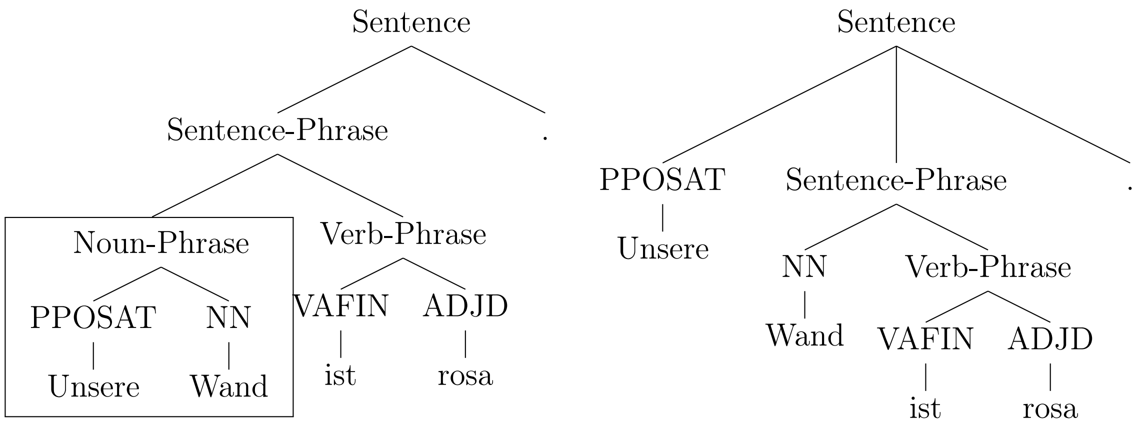
However, in one sentence, another annotation is added to avoid a wrong conclusion.
The first sentence is "Unsere Wand ist rosa.". In this sentence, the word 'Unsere' leads to a unknown PoS token and must be added to the PoC grammar. After parsing the algorithm, the parser returned the following parsing tree:



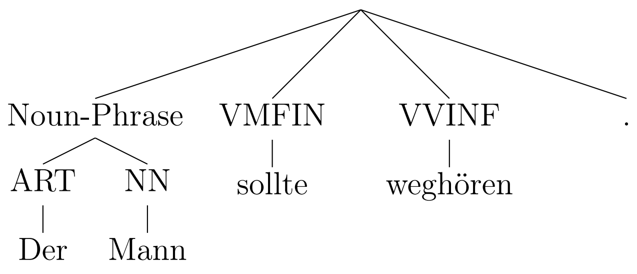
The parser identified a sentence annotation, which is missing the first token. This behavior is expected with this sentence. The algorithm iterates through all possible fitting rules to return the suggestions starting with the sentence rules. These are the returned suggestions:

```
SP = PPOSAT? NN VP [1]
NP = PPOSAT NN [1]
PP = PPOSAT NN [1]
ADVP = PPOSAT NN [1]
```

The most straightforward choice is to add the token the the 'SP'-annotation. The parser finds several possible rules which lead to a sentence phrase. The first is to expand the existing rule "SP = NN VP" with the token and a question mark. The next suggestions result from the same rule in the grammar ("SP = (NP|PP|ADVP) VP"). The tokens 'VAFIN' and 'ADJD' build the 'VP'-annotation in this sentence. Therefore, the algorithm matches the tokens 'PPOSAT' and 'NN' to result in 'NP', 'PP' or 'ADVP'. The first two rules in the list result in a correct and fully parsed tree. A fully parsed tree is returned when the grammar is expanded with the other two. However, this expansion returns in wrong semantic meaning. The number behind the rule indicates how often it is found during the suggestion process.
The algorithm needed 4418ms to calculate all possible rule suggestions for the sentence.



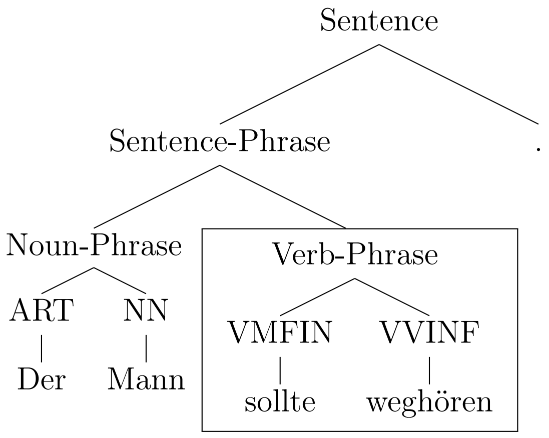
The next sentence is "Der Mann sollte weghören.". Here, the word 'sollte' is an unknown verb type to the PoC grammar, and the rules for the verb phrases must be adjusted. The parsing tree confirms the same assumption.



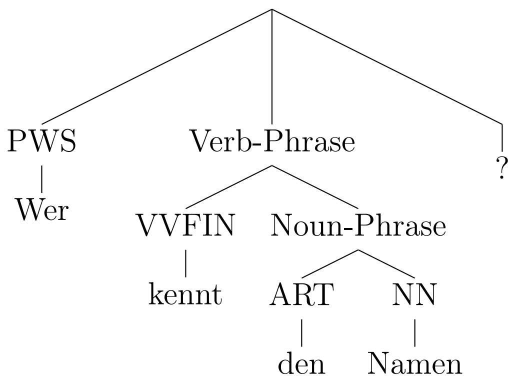
After analyzing and iterating through the grammar, the suggester returns only one suggestion to the user:

```
VP = VMFIN VVINF [1]
```

There is only one possible rule, which can be extended. Due to a missing comma, the first four tokens have to form a sentence phrase to return a correct sentence. However, the rule has to start with an 'NP'-annotation. Only two rules meet this requirement "SP = NP VP" and "SP = NP VVFIN". The rules are used to identify a possible candidate. At first, the 'NP' is removed from both rules as the sequence does have an 'NP' at the start. The two remaining tokens of the sequence 'VMFIN' and 'VVINF' must form either a 'VP' or 'VVFIN'. The second rule can be removed because 'VVFIN' is a PoS-tag, which rules cannot create. Therefore, only the first rule might be possible to expand, and the two tokens 'VMFIN' and 'VVINF' have to form a verb-phrase. By adding the suggestions, the parser returns a valid and complete parsing tree to the user.
The suggestion process took 4293ms to process the incomplete parsing tree and return the result.



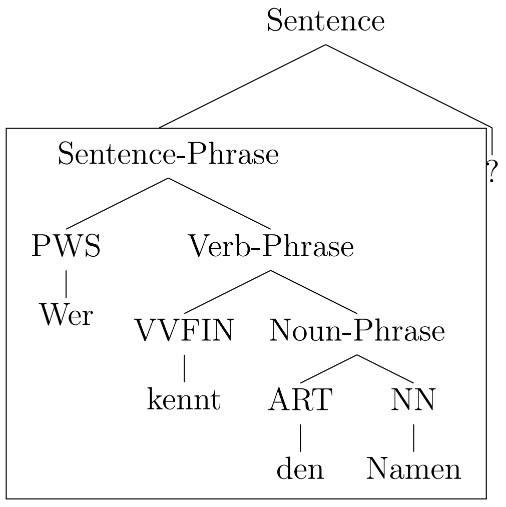
The next sentence is question and therefore not a 'normal' sentence syntax. After parsing the sentence "Wer kennt den Namen?", the algorithm returns the following parsing tree:



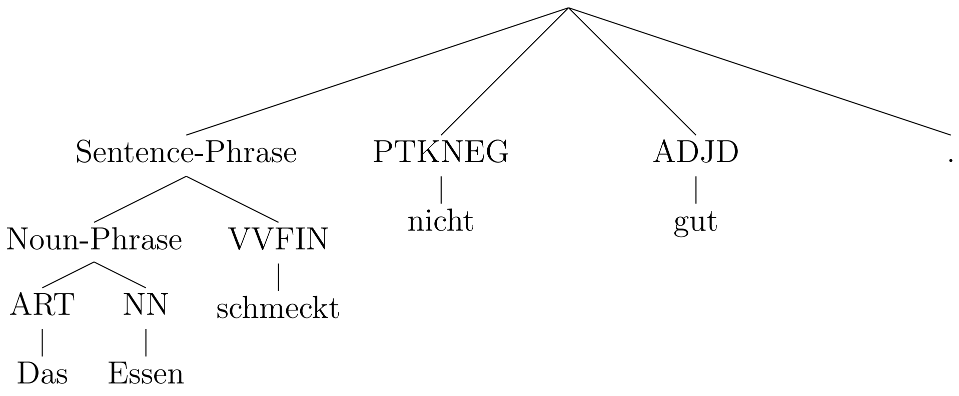
Using the STTS tag set, the PoS tagger handles all punctuation similar to the period except the comma. Therefore, the question mark at the end of the sentence is parsed as a period. The suggester returns only one possible rule expansion to the user:

```
SP = PWS VP [1]
```

The parser does not recognize the token 'PWS'. However, to form a proper sentence, a sentence-phrase is needed. The suggester does not find any rule to expand. Therefore the algorithm suggests adding an entirely new rule part to the sentence-phrase. It is debatable whether it would make more sense to make a new category for questions. However, the proposal returns a valid and complete parsing tree. The algorithm needed 4372ms to return the suggestion.



The next sentence is "Das Essen schmeckt nicht gut.". The combination of the words 'nicht' and 'gut' is unkown to the grammar and results in an incomplete parsing tree. By parsing the sentence, the following tree is returned:

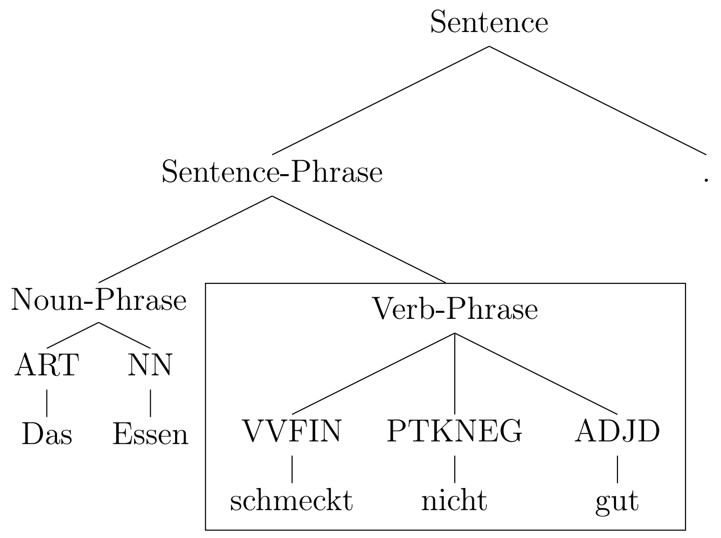


Ideally, the parser identifies the words and combines to either a own phrase or to a verb-phrase with the verb. The suggestor returns the following suggestions to the user:

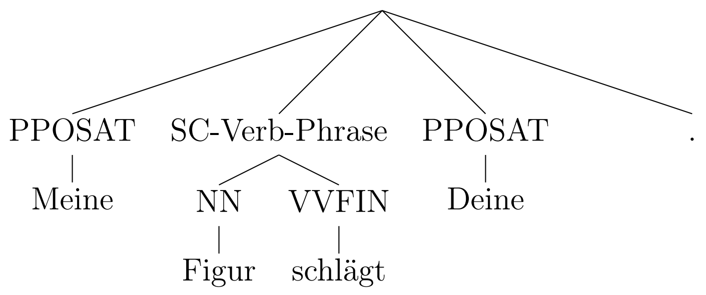
```
NP = PTKNEG ADJD [1]
PP = PTKNEG ADJD [1]
VP = VVFIN PTKNEG? ADJD [1]
SP = NP VVFIN (PTKNEG ADJD)? [1]
PP = NP VVFIN [1]
VP = PTKNEG ADJD [1]
ADVP = NP VVFIN [1]
```

The first suggestion is to combine the two words into a noun-phrase. In this context, the suggestion is incorrect. Neither of the two words meets the requirements to be labeled as a noun-phrase. The next suggestion is to form a preposition-phrase with the two words. Technically, this is a possible rule to use, which returns a complete parsing tree to the user. Nevertheless, neither of the words justify a preposition-phrase, because they are not labeled as a preposition. The subsequent suggestion is a perfect suggestion of this case. The existing rule of the verb phrase is

expanded with the tag 'PTKNEG'. This improvement consumes the least amount of space in the grammar and returns a valid and complete parsing tree. The next suggestion is to expand a sentence-phrase rule to cover the words "PTKNEG ADJD". That results in a correct and valid parsing tree. However, the more elegant suggestion is to extend the verb phrase. Nevertheless, the improvement is syntactically correct. Sadly, all the following suggestions do not provide a good or even correct improvement for the user. Neither would return a valid or complete parsing tree. However, the suggester can find the two anticipated extensions.
It took the parser 4287ms to return all suggestions. The following figure illustrates the two correct parsing trees.



As mentioned above, for this sentence another annotation is added to the list of disallowed list. When the sentence "Meine Figur schlägt Deine." is parsed, the algorithm returns this parsing tree:



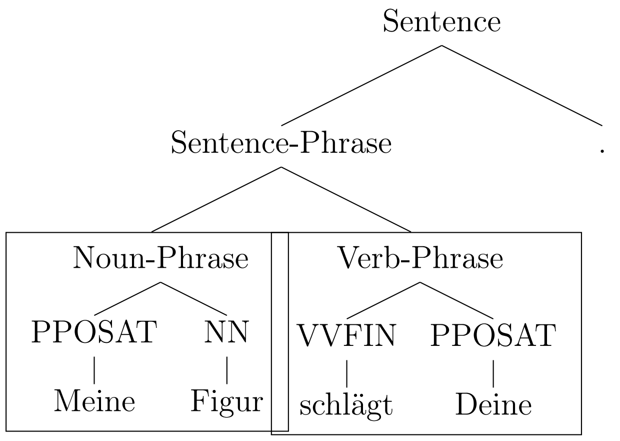
When the subordinate-clause-verb-phrase is not broken into its children, the algorithm returns only one suggestion:

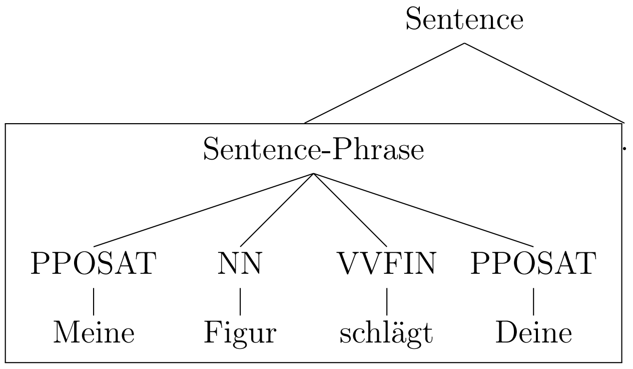
```
SP = PPOSAT SCVP PPOSAT [1]
```

Even though this improvement might return a complete parsing tree, the problem is that the tree is not valid. In this sentence, a subordinate-clause-verb-phrase cannot be combined. However, the algorithm does not know when this phrase is valid and combines it. Therefore, this phrase is added to the disallowed annotations, and the parsing algorithm is restarted. Then the suggester returns the following suggestions to the user:

```
SP = PPOSAT? NN VVFIN PPOSAT? [2]
NP = PPOSAT NN [1]
VP = VVFIN PPOSAT [1]
PP = PPOSAT NN [1]
ADVP = PPOSAT NN [1]
```

The first improvement is found twice during the suggestion algorithm. The algorithm suggests the user adds a rule extending the sentence-phrases. The extension returns a correct and valid parsing tree. The solution might not be the most elegant. Nevertheless, it is correct and flawless. The subsequent two suggestions combined return a perfect, valid and complete parsing tree. These would be the perfect combination. The first two tokens are combined into a noun-phrase, and the last two tokens are combined into a verb-phrase. Both phrases build a sentence-phrase, and therefore the sentence is completely parsed. With this suggestion, the general grammar structure of subject-verb-object is obtained because the verb and object build the verb-phrase similar to the remaining rules. The subsequent two suggestions might lead to a complete parsing tree, but the semantic is not valid. Combining a personal pronoun and a noun does not result in a preposition-phrase or adverb-phrase. The last two suggestions do not return a complete tree at all. These two are false and cannot be used to achieve a better parsing tree.
The parsing process needed 6329ms, and the two ideal parsing trees are illustrated in the following figure.





## Summary

To summarize the evaluation of the rule suggestion, the algorithm always found possible and valid suggestions to achieve the requirements of the sentence. However, several suggestions either lacked semantic correctness or did not lead to a complete parsing tree. In total, for the five sentences, 19 suggestions are returned to the user. From that, 9 suggestions are correct and valid. That results in a precision score of 0.474 and a recall score of 1.
However, the algorithm is configured to return as many suggestions as possible. It is not designed to identify if a rule leads to a semantically correct sentence or just a complete tree. If the precision score is computed based on the completion of the parsing tree, the score is 0.714 instead of 0.429. The quality of the suggestions also depends on the grammar. If the grammar is complex and intransparent, the algorithm has difficulty finding excellent and valid suggestions. If the PoC grammar is refined and declares rules more specific, the suggestions might be even better. On average, the suggestion algorithm needs about $4,773$ms to return the improvements. Nevertheless, the algorithm can improve its workflow and results. At this moment, the parser is just a suggestion algorithm without any high knowledge.

# 7 Conclusion and Outlook

With the rising globalization, people have to communicate with each other and overcome language barriers. This thesis presents two error recovery and report systems as well as a grammar-creating tool to support people with foreign languages
During this work, the bottom-up CYK parser and the top-down Earley parser are presented. Their essential technologies are explained and shown, as well as their advantages and disadvantages. Using parsing techniques, the problem of errors in the sentence always occurs. The algorithm can only parse a sentence if it is 100% covered by the grammar. Otherwise, the sentence is rejected, and no parsing tree is returned. That is why each of the parsing algorithms gets a customized error recovery system to underline their strength.
The CYK algorithm is provided with a brute-force approach to find possible errors. The system can delete, insert, and swap tokens from the sentence to recreate new sentences. If the algorithm finds a parsing tree with the altered sequence, an error message is returned to the user describing what needs to be changed. The approach is implemented to solve sentences with a maximum of two errors (edit-distance 2). The brute force algorithm reaches a precision score of $0,656$ and a recall score of 1 in the proof-of-concept domain. The approach can return a correct message for each sentence and even find unexpected improvements. Despite that, the algorithm also returns several incorrect suggestions, which can be tracked down to the used grammar. In average, the edit-distance 1 is analyzed in 4450ms and the edit-distance 2 needs 579393ms. The brute force algorithm has an enormous impact on the computation time by parsing hundred of thousand sentences.
The Earley parser is provided with a more elegant error recovery system. The parser computes the sentence as far as possible. Once the algorithm does not find a matching rule, it is canceled. The created Earley table is then given to the error recovery system to fix it. The system identifies the corrupted chart and uses the same operations as the CYK to resolve the problem. However, the operations are more guided since the algorithm pinpoints only one position. After each operation, the Earley table and sentence are parsed again, and possible parsing trees are returned to the user. This error recovery system has a precision score of $0,772$ and a recall score of 1 in the proof-of-concept domain. One reason for the low precision score is the swapping operation. If it is removed, the system reaches a precision score of $0,899$. Nevertheless, the error recovery system is not able to identify an improvement in every sentence. In two sentences, the approach fails to deliver a suggestion to the user. Due to the more direct approach, the error recovery system needs 4732ms to finish analyzing the edit-distance 1 and 2.
When comparing the error recovery systems of the two parsers, the system based on the Earley has a better precision score and needs less time to determine suggestions. Especially, sentences with edit-distance 2 are parsed in seconds and not minutes.

However, unlike the Earley algorithm, the CYK approach finds at least one possible solution to every sentence.
Perhaps designing a model combining the error recovery system of the Earley and the CYK might be even more powerful. For example, the CYK algorithm is used to parse the edit-distance 1, and edit-distance 2 is handled by the Earley. Another possible combination is to use the CYK only as a parsing technique. If the algorithm does not find a parsing tree, all incomplete parsing trees are extracted and given to the Earley parser to find possible solutions. However, the error recovery system must be able to skip terminals and non-terminals to return helpful suggestions. It might be promising to combine the two algorithms and design an error recovery system using the advantages of both systems. Nevertheless, the results of both algorithms showcase the ability of a parser to find problematic sequences in a sentence and return insightful feedback to the user on how to fix the error. The algorithm might not work perfectly, but they function as a reasonable basis for further improvements and present the ability to overcome the disadvantages linked with parsing.
Next to the error recovery systems, a rule suggestions algorithm is presented. The algorithm uses the CYK parser to find incomplete parsing trees for the sentence. The algorithm matches the incomplete parsing trees with the rules of the grammar. If rules have the potential, they are stored and later expanded to cover the desired sequence. The rule suggestion algorithm can expand rules using the "?"-operator as well as the "+"- and "*"-operator. The rules are stored in a set, and once all are processed, the set is returned to the user. The suggestion algorithm reaches a precision score of $0,429$ under normal circumstances. When only rules are counted, which lead to a complete sentence, the algorithm reaches a precision score of $0,714$. Nevertheless, the algorithm is programmed to suggest rules to the user. It is expected that the precision score is not the best and could be improved. One possible improvement to the suggestion algorithm is to equip the rules with necessary tags. For example, a noun phrase only covers rules containing "NN" or "NE" tags. All other rules are rejected. That might make the algorithm more intelligent and result in a better precision score. Parsing techniques cannot only be used to find and report errors in the sentence. This improvement showcases the ability to correct a grammar and return good suggestions to overcome unknown sequences.
Overall, the decisive factor for all three algorithms is grammar. A parser is only as intelligent as the used grammar. In order to achieve better results in every category, a possible way is to detail and adjust grammar. If unique structures are covered by one rule alone, the error recovery systems can detect a false syntax easier. Probably, the best way to improve the error recovery systems is to enhance the rule suggester. Building grammar is a complicated and complex job. Using an algorithm helping with that might lead to a better-shaped grammar. Furthermore, an algorithm can be implemented, which compresses grammar and deletes duplicate rules. Using both algorithms to create grammar can result in detailed and well-adjusted grammar for the used domain.
To summarize, the paper presents good and working algorithms, and they can answer and prove the questions asked in the introduction. However, these can still be improved and refined to achieve better results and help people. For this and future investigations, this work provides the necessary basics.

# List of Figures

# Bibliography

[1] “Statistisches bundesamt: Migration,” visited 2021-08-26. [Online]. Available: https://www.destatis.de/DE/Themen/Gesellschaft-Umwelt/Bevoelkerung/Migration-Integration/_inhalt.html

[2] “Anteil von deutschen und ausländern in verschiedenen berufsgruppen,” visited 2021-08-26. [Online]. Available: https://de.statista.com/statistik/daten/studie/167622/umfrage/auslaenderanteil-in-verschiedenen-berufsgruppen-in-deutschland/

[3] “Unesco language ranking,” visited 2021-08-26. [Online]. Available: https://synergyfocusblog.wordpress.com/2018/10/26/unesco-top-10-most-difficult-languages-in-the-world/

[4] “Deutsch in der welt,” visited 2021-11-28. [Online]. Available: https://www.deutschland.de/de/topic/wissen/muttersprache-deutsch-in-42-laendern-der-welt

[5] A. Voutilainen, “Part-of-speech tagging,” *The Oxford handbook of computational linguistics*, pp. 219–232, 2003.

[6] L. Bahl, P. Brown, P. De Souza, and R. Mercer, “Maximum mutual information estimation of hidden markov model parameters for speech recognition,” in *ICASSP’86. IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 11. IEEE, 1986, pp. 49–52.

[7] “Stts tagset tiger (stts),” visited 2021-08-30. [Online]. Available: https://www.ims.uni-stuttgart.de/forschung/ressourcen/lexika/germantagsets/

[8] D. Grune and C. J. Jacobs, “Parsing techniques (monographs in computer science),” 2006.

[9] J. Earley, “An efficient context-free parsing algorithm,” *Communications of the ACM*, vol. 13, no. 2, pp. 94–102, 1970.

[10] D. H. Younger, “Recognition and parsing of context-free languages in time n3,” *Information and control*, vol. 10, no. 2, pp. 189–208, 1967.

[11] N. Chomsky, “On certain formal properties of grammars,” *Information and control*, vol. 2, no. 2, pp. 137–167, 1959.

[12] S. Medeiros and F. Mascarenhas, “Syntax error recovery in parsing expression grammars,” in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, 2018, pp. 1195–1202.

[13] S. Q. de Medeiros and F. Mascarenhas, “Error recovery in parsing expression grammars through labeled failures and its implementation based on a parsing machine,” *Journal of Visual Languages & Computing*, vol. 49, pp. 17–28, 2018.

[14] ——, “Towards automatic error recovery in parsing expression grammars,” in *Proceedings of the XXII Brazilian Symposium on Programming Languages*, 2018, pp. 3–10.

[15] R. Corchuelo, J. A. Pérez, A. Ruiz, and M. Toro, “Repairing syntax errors in lr parsers,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 24, no. 6, pp. 698–710, 2002.

[16] S. Kiran, C. R. S. Sai, M. Pooja *et al.*, “A comparative study on parsing in natural language processing,” in *2019 3rd International Conference on Computing Methodologies and Communication (ICCMC)*. IEEE, 2019, pp. 785–788.

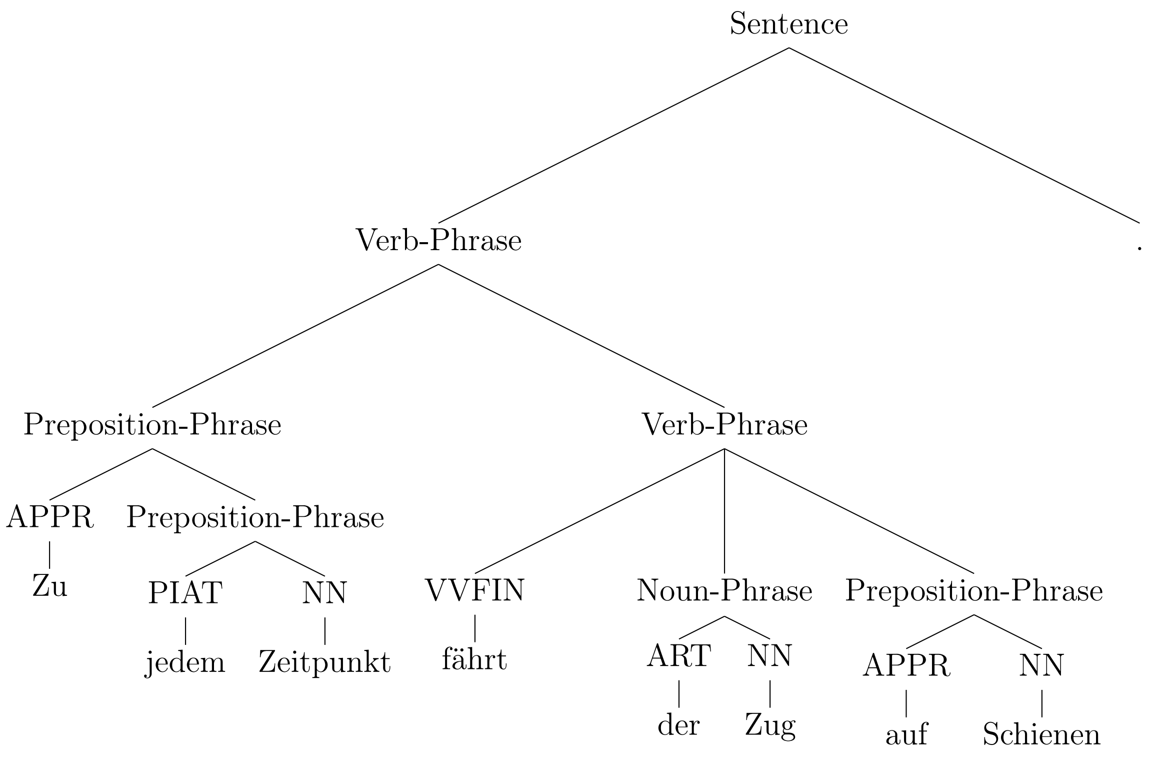
[17] “Repository for uima framework,” visited 2021-11-28. [Online]. Available: http://artifactory-ls6.informatik.uni-wuerzburg.de/artifactory/libs-release

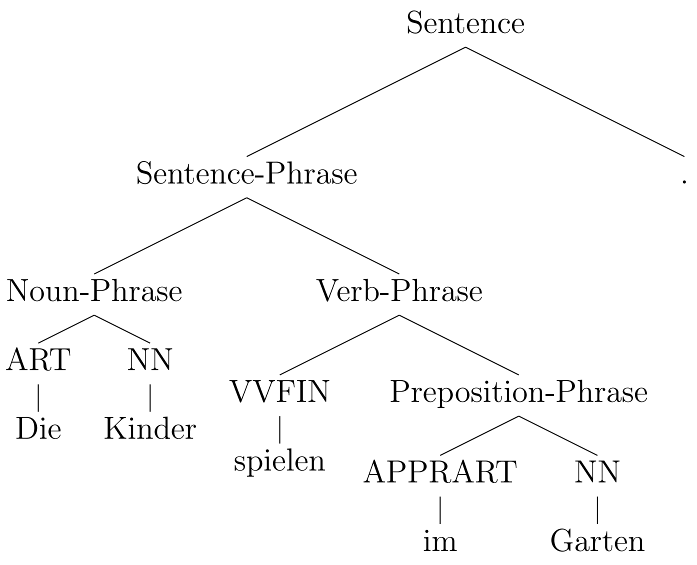
[18] “Aries rf-tagger,” visited 2021-11-28. [Online]. Available: https://www.cis.uni-muenchen.de/~schmid/tools/RFTagger/

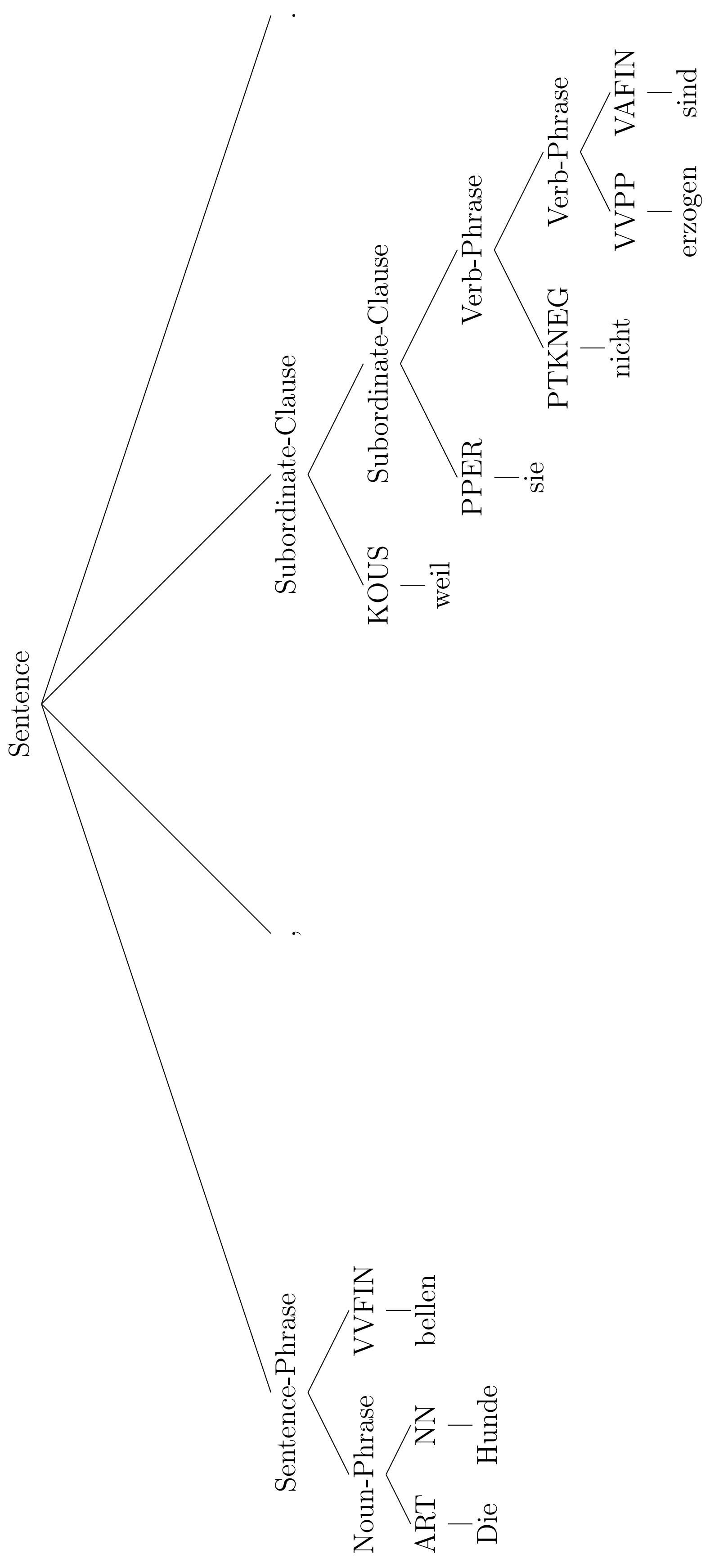
[19] G. Navarro, “A guided tour to approximate string matching,” *ACM computing surveys (CSUR)*, vol. 33, no. 1, pp. 31–88, 2001.

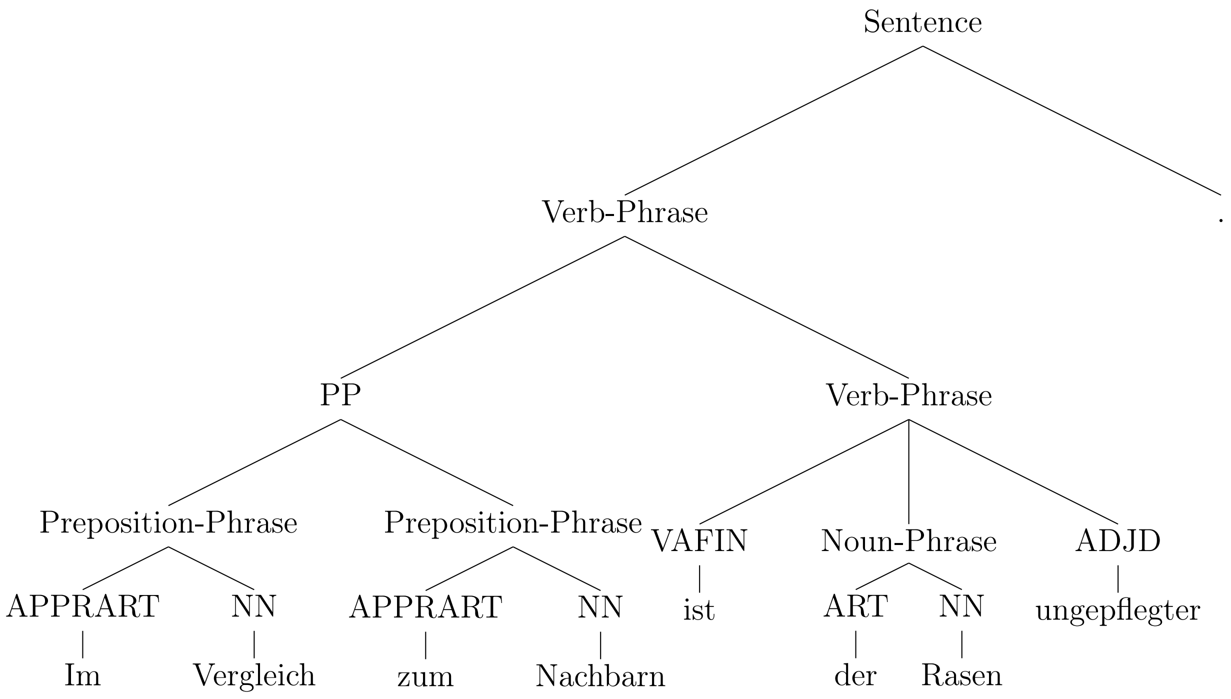
[20] D. M. Powers, “Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation,” *arXiv preprint arXiv:2010.16061*, 2020.

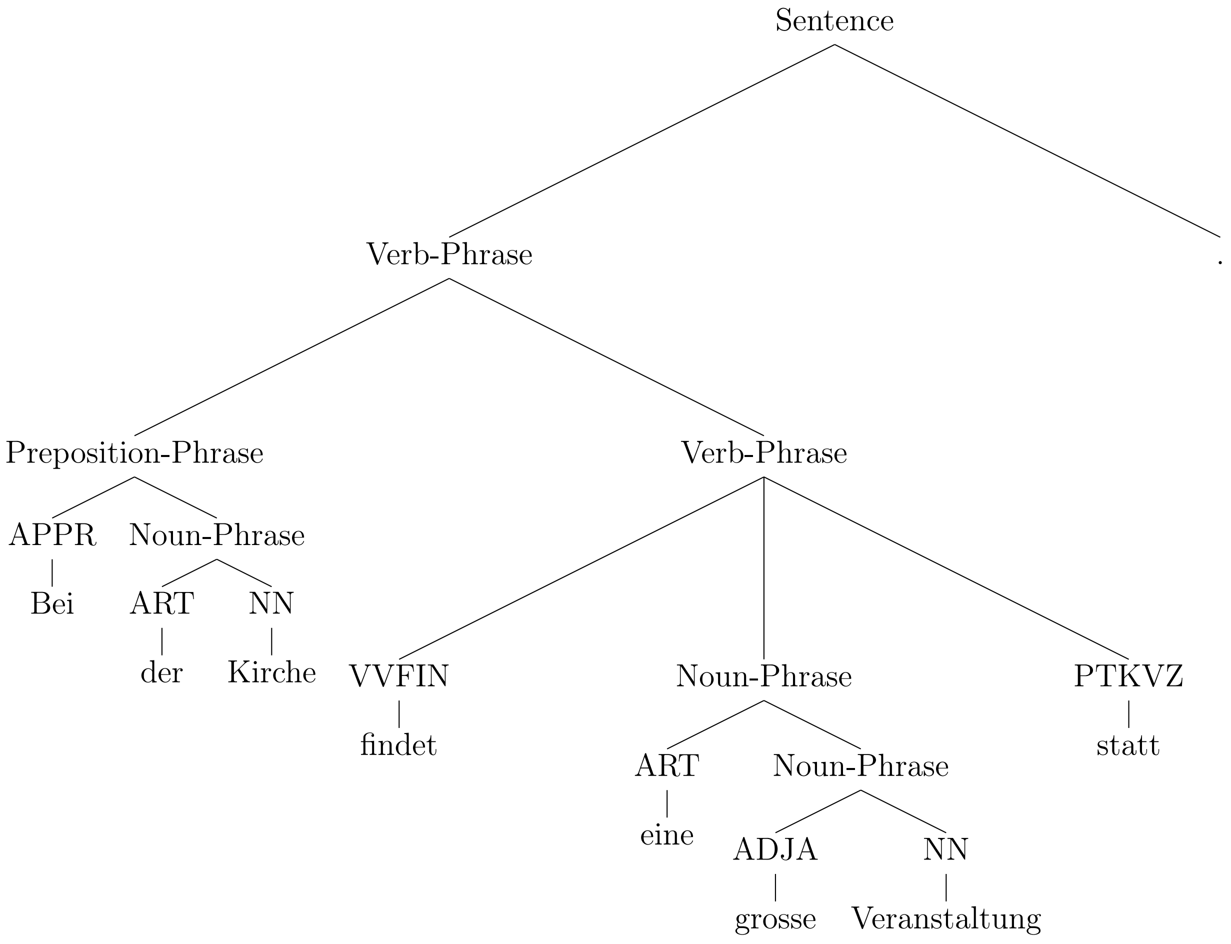
# A Parsing trees of PoC-domain using CYK

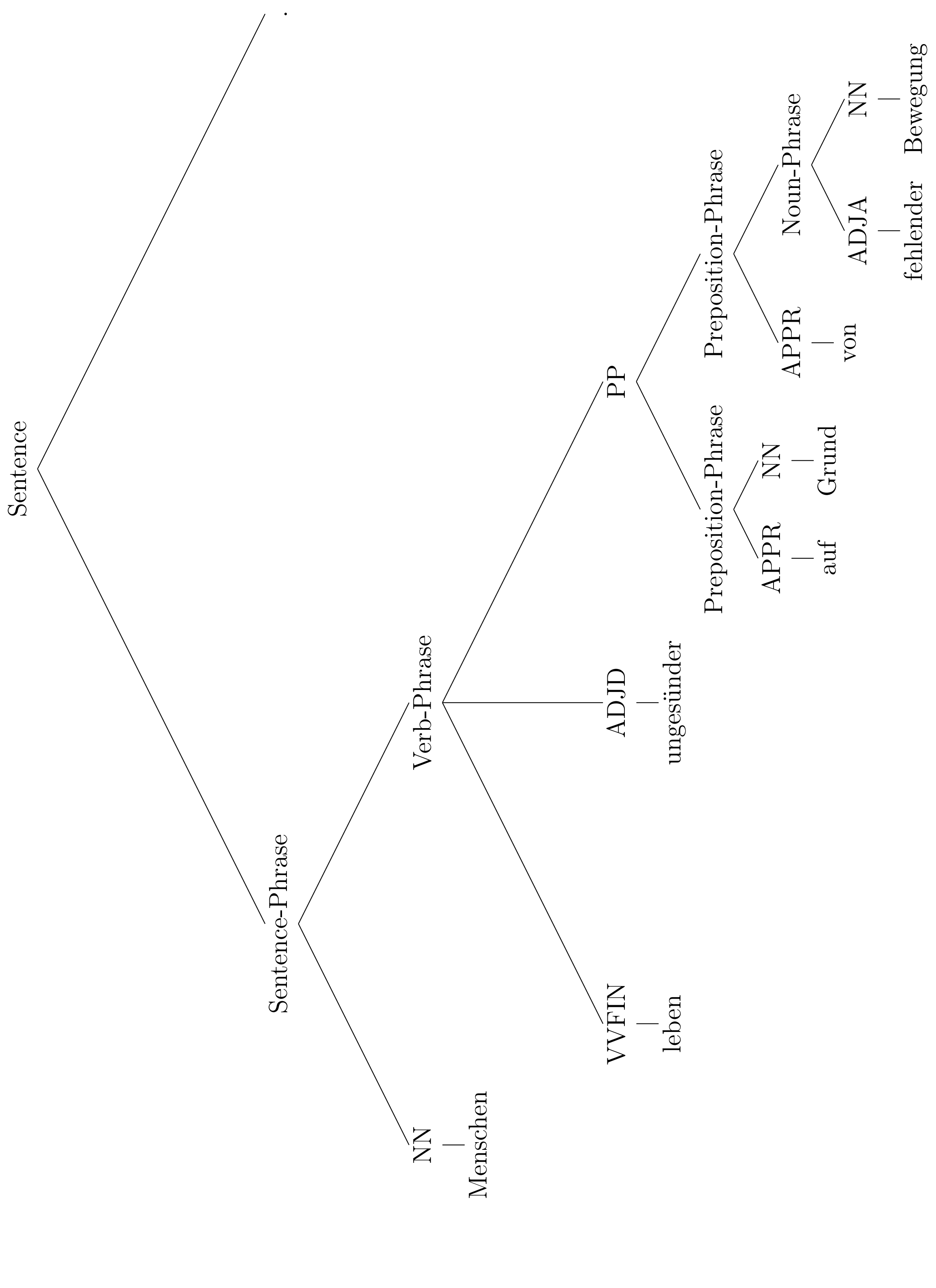


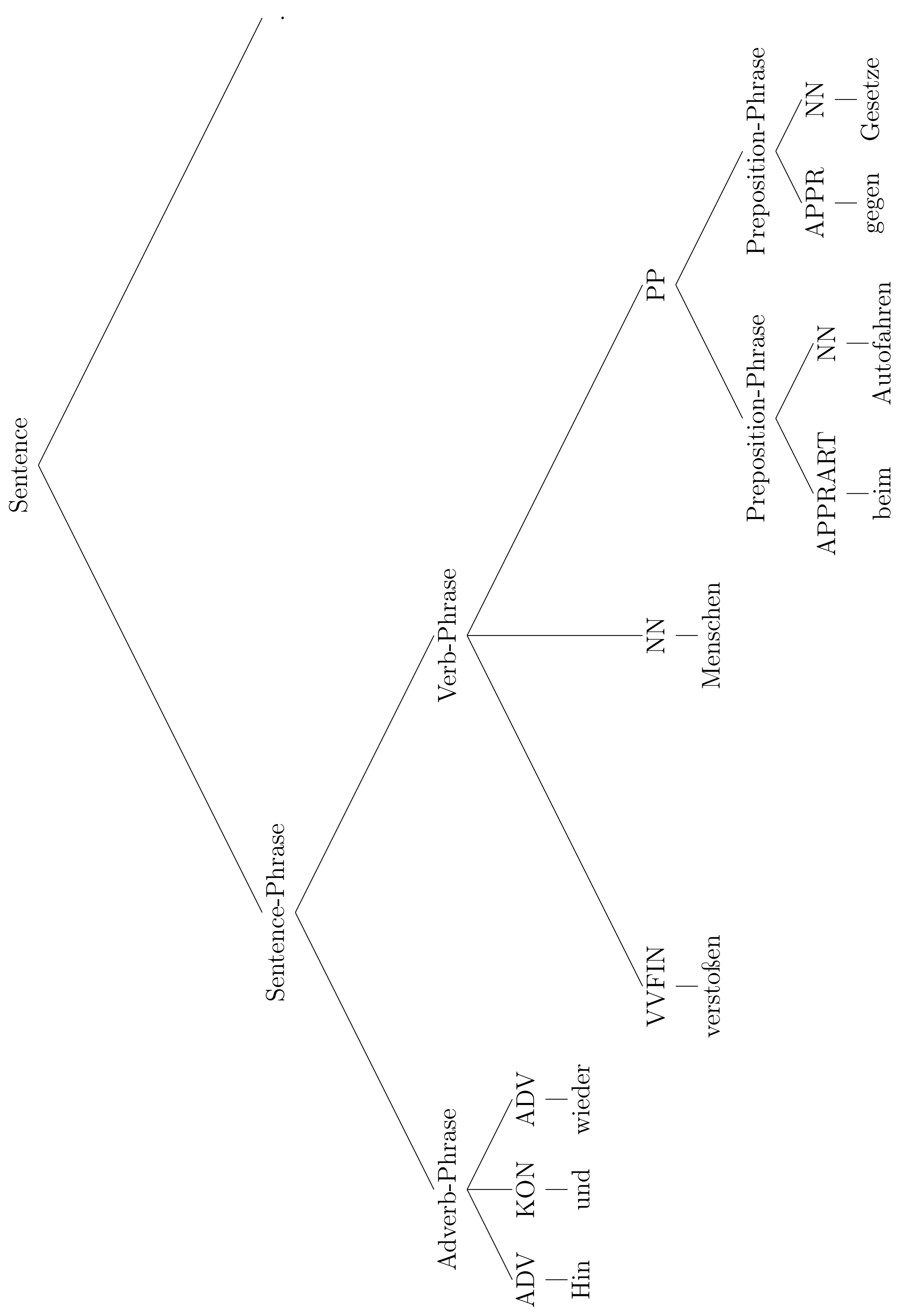


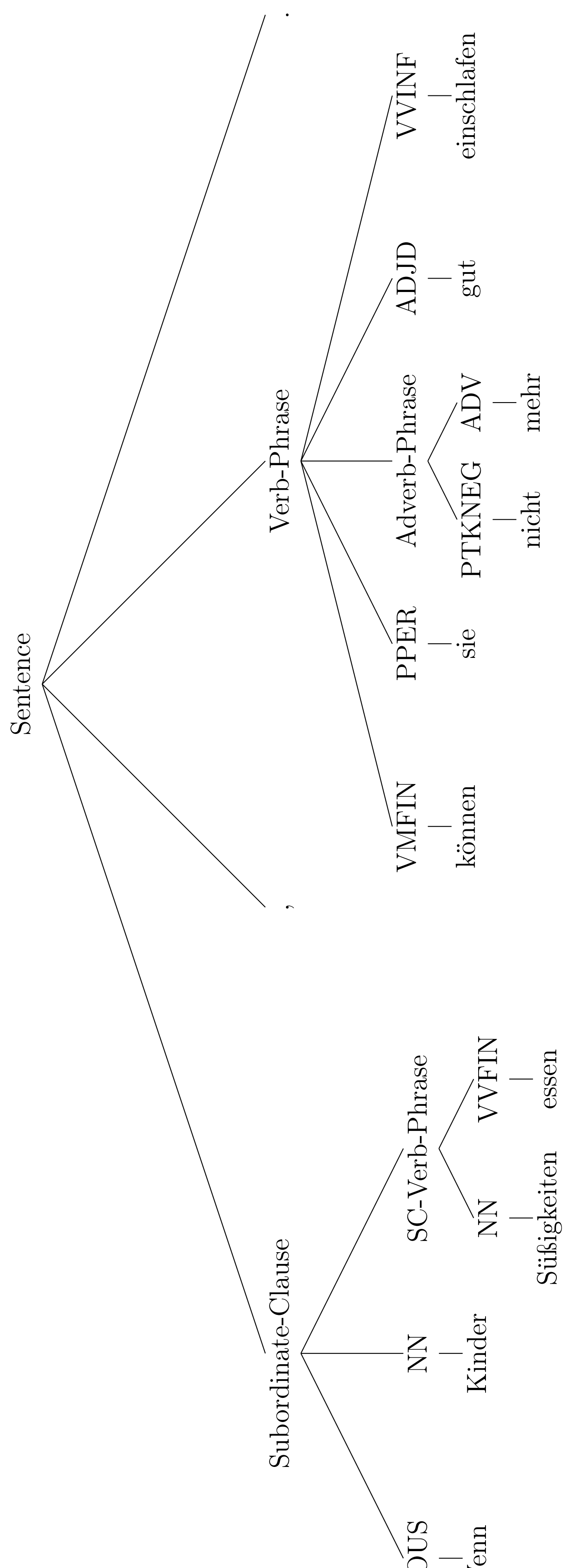

Sentence
Sentence-Phrase
Noun-Phrase
ART
Die
NN
Hunde
VVFIN
bellen
,
Subordinate-Clause
KOUS
weil
Subordinate-Clause
PPER
sie
Verb-Phrase
PTKNEG
nicht
Verb-Phrase
VVPP
erzogen
VAFIN
sind
.

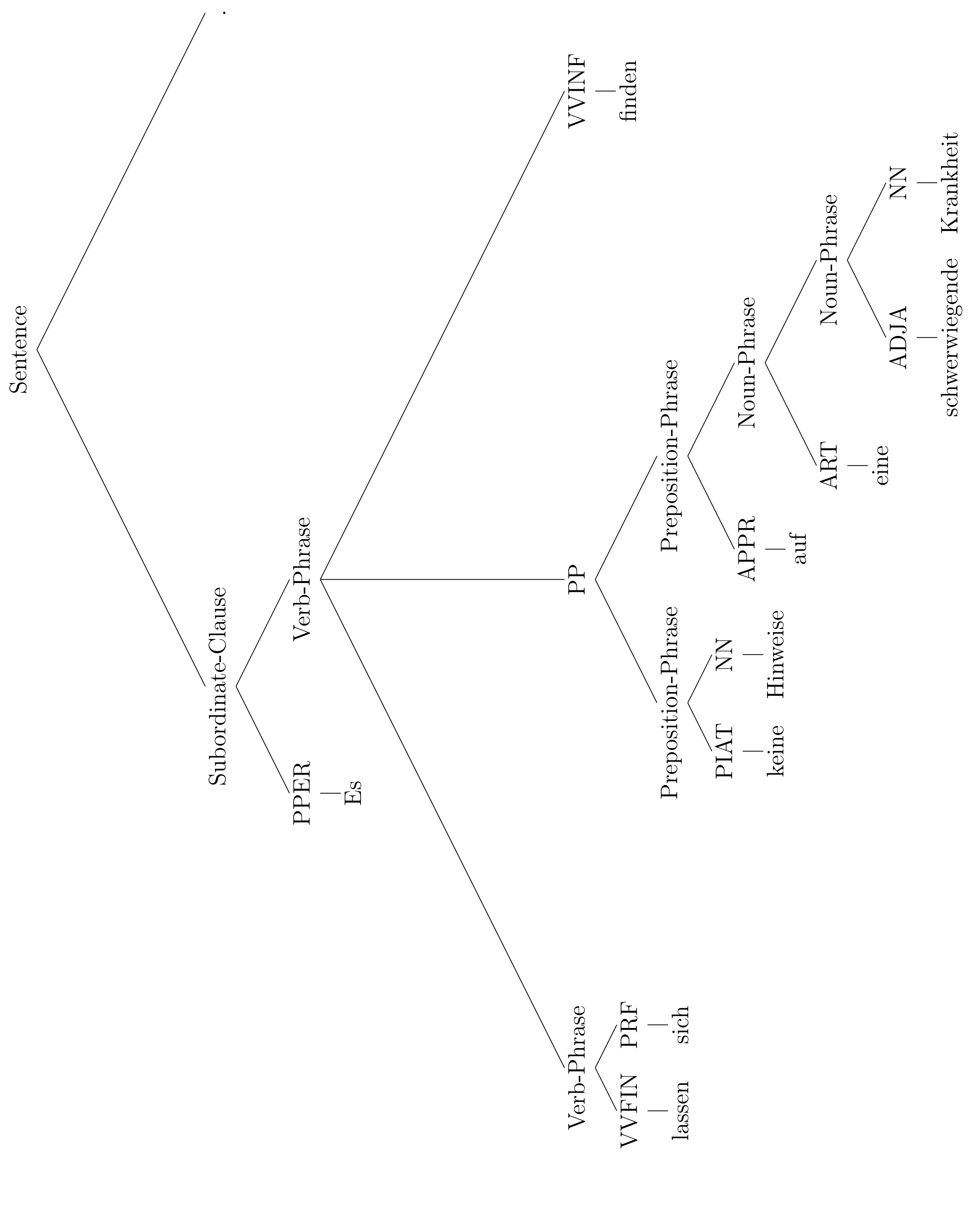

Sentence
Verb-Phrase
.
PP
Verb-Phrase
Preposition-Phrase
Preposition-Phrase
VAFIN
Noun-Phrase
ADJD
APPRART
NN
APPRART
NN
ist
ART
NN
ungepflegter
Im
Vergleich
zum
Nachbarn
der
Rasen

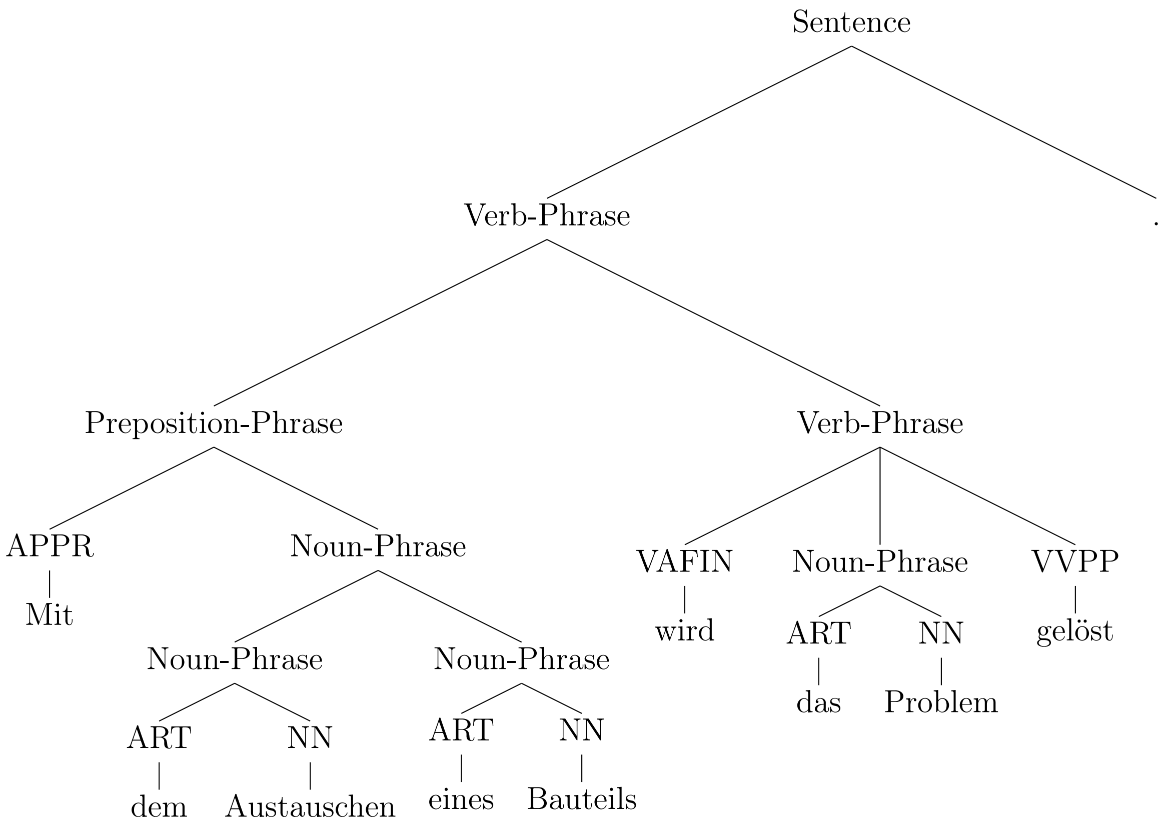

Sentence
Verb-Phrase
.
Preposition-Phrase
Verb-Phrase
APPR
Noun-Phrase
Bei
ART
NN
der
Kirche
VVFIN
Noun-Phrase
PTKVZ
findet
ART
Noun-Phrase
statt
eine
ADJA
NN
grosse
Veranstaltung

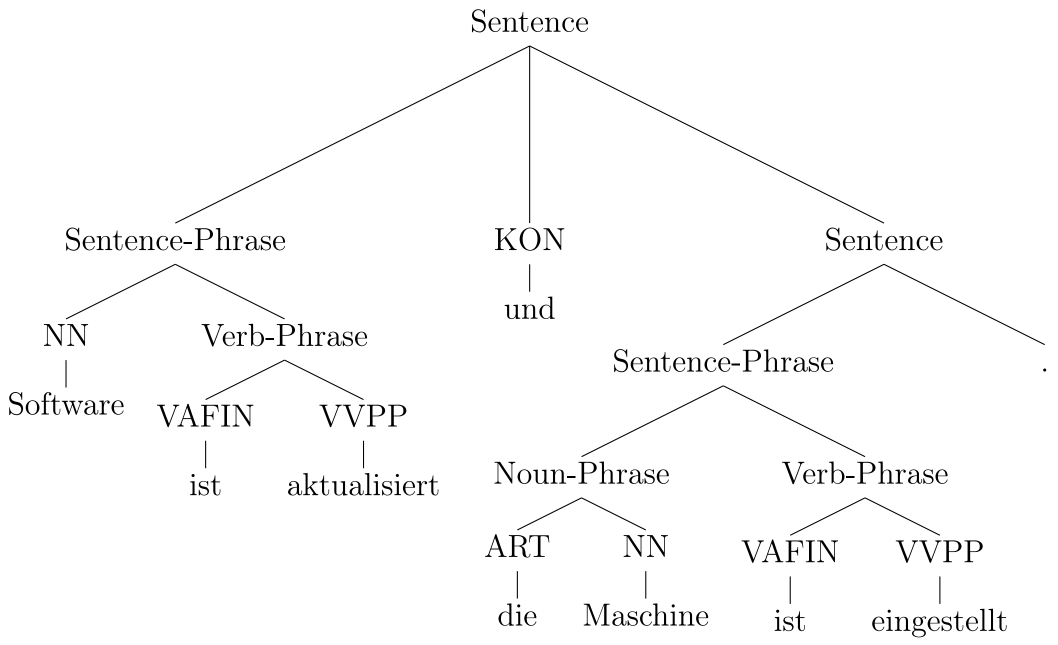

Sentence
Sentence-Phrase
.
NN
Menschen
Verb-Phrase
VVFIN
leben
ADJD
ungesünder
PP
Preposition-Phrase
APPR
auf
NN
Grund
Preposition-Phrase
APPR
von
Noun-Phrase
ADJA
fehlender
NN
Bewegung

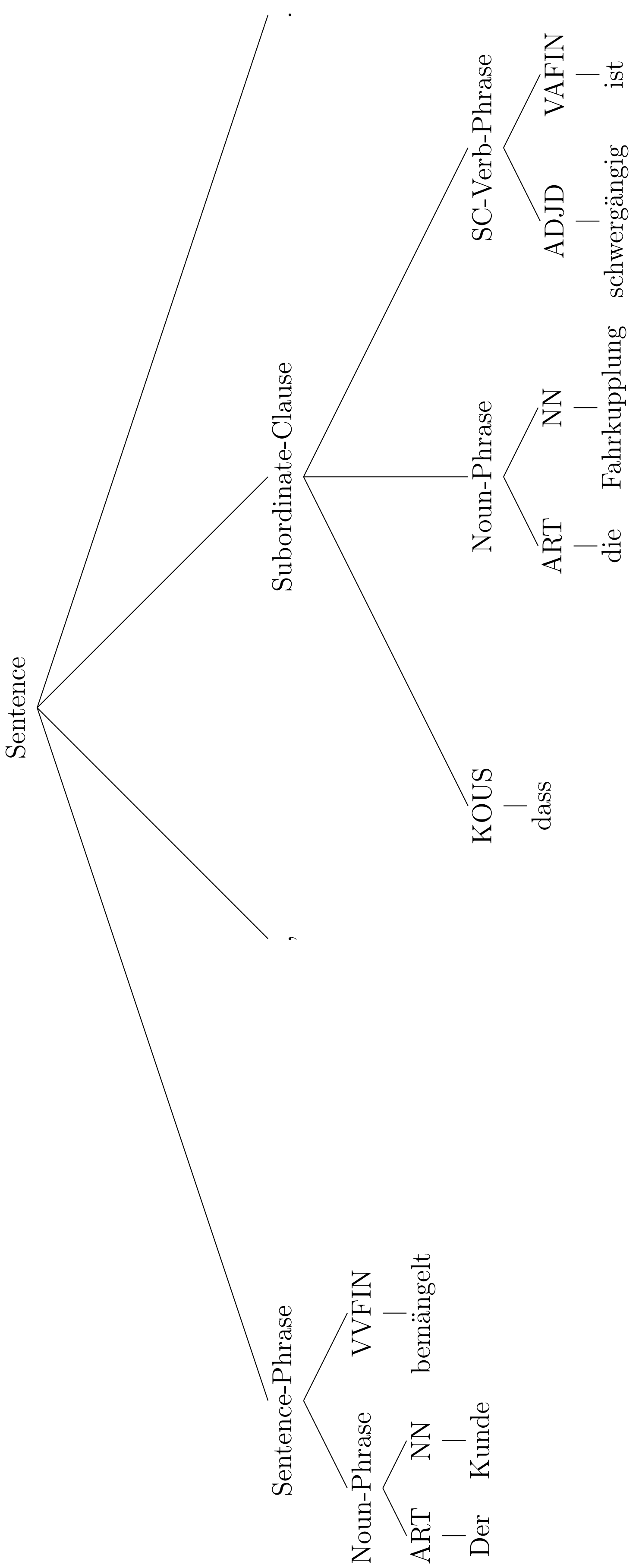

Sentence
.
Sentence-Phrase
Adverb-Phrase
ADV
Hin
KON
und
ADV
wieder
Verb-Phrase
VVFIN
verstoßen
NN
Menschen
PP
Preposition-Phrase
APPRART
beim
NN
Autofahren
Preposition-Phrase
APPR
gegen
NN
Gesetze

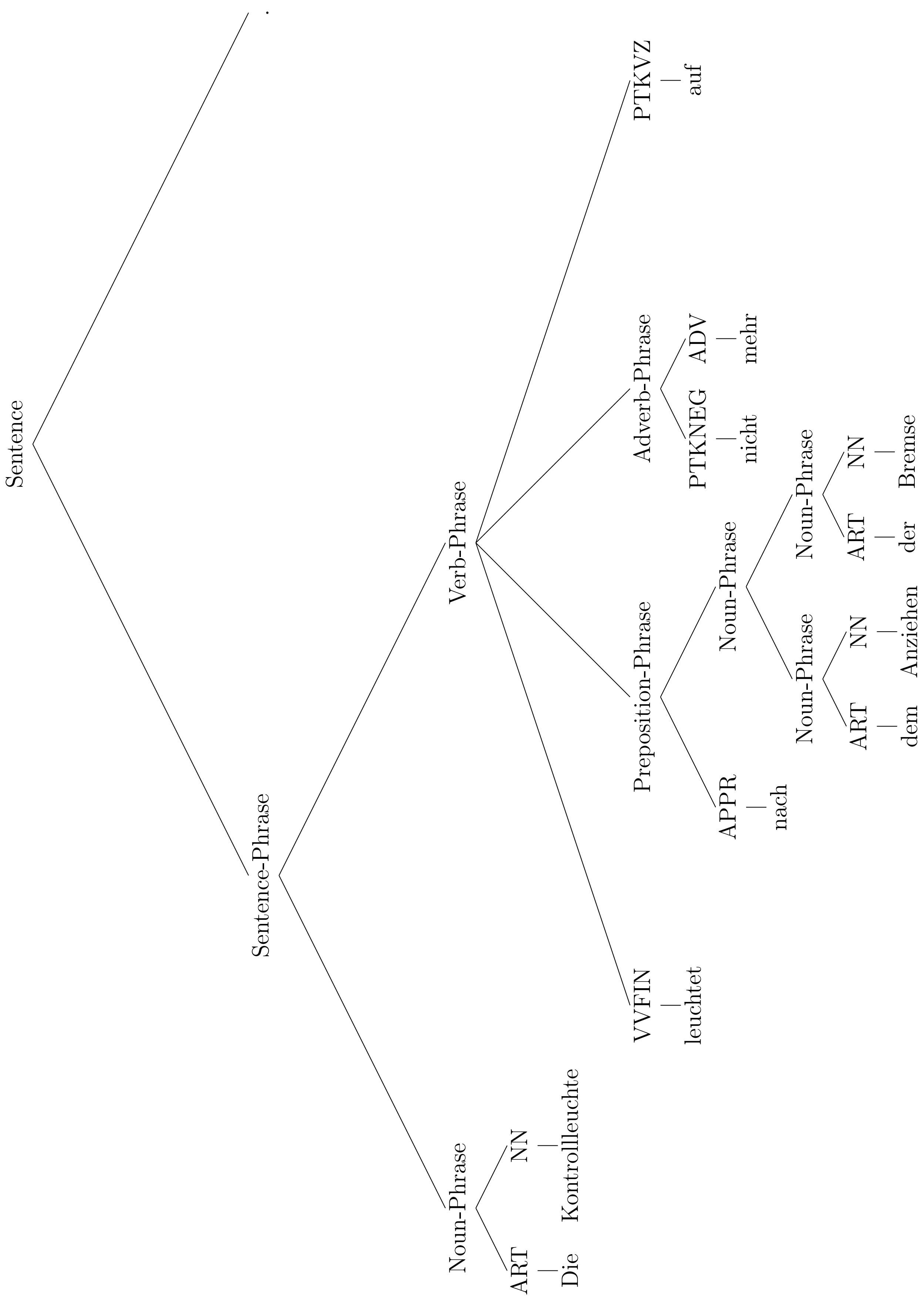

Sentence
Subordinate-Clause
KOUS
Wenn
NN
Kinder
SC-Verb-Phrase
NN
Süßigkeiten
VVFIN
essen
,
Verb-Phrase
VMFIN
können
PPER
sie
Adverb-Phrase
PTKNEG
nicht
ADV
mehr
ADJD
gut
VVINF
einschlafen
.

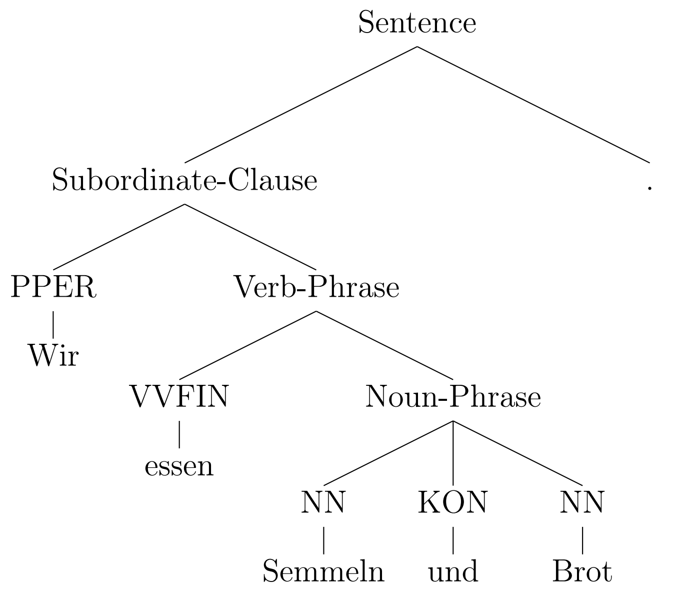

Sentence
Subordinate-Clause
.
PPER
Es
Verb-Phrase
Verb-Phrase
VVFIN
lassen
PRF
sich
PP
Preposition-Phrase
PIAT
keine
NN
Hinweise
Preposition-Phrase
APPR
auf
Noun-Phrase
ART
eine
Noun-Phrase
ADJA
schwerwiegende
NN
Krankheit
VVINF
finden

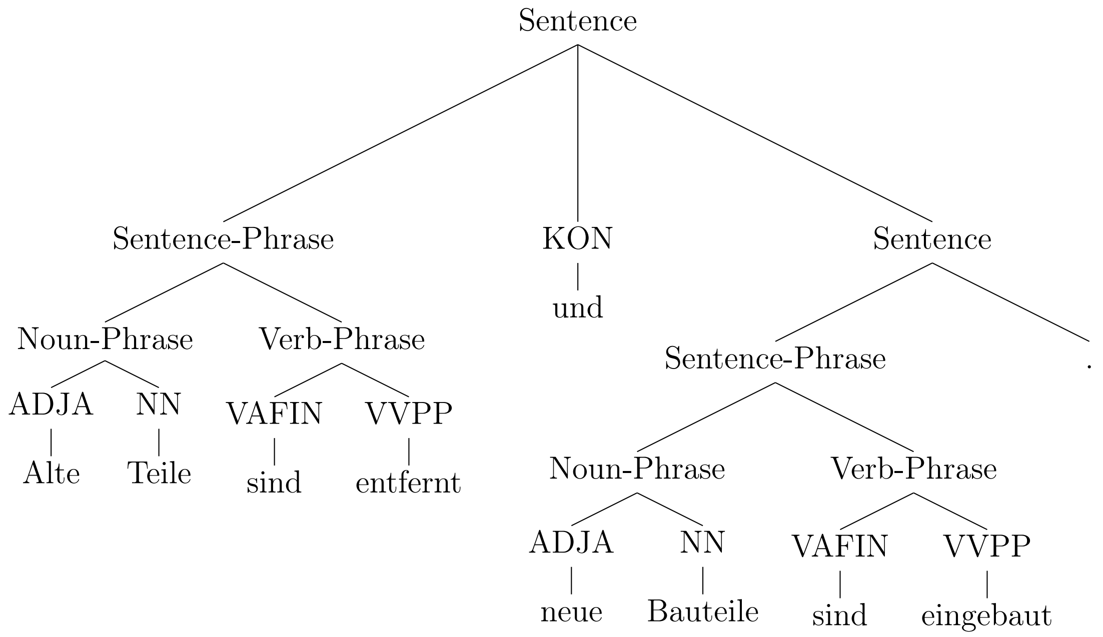

Sentence
Verb-Phrase
.
Preposition-Phrase
Verb-Phrase
APPR
Mit
Noun-Phrase
Noun-Phrase
ART
dem
NN
Austauschen
Noun-Phrase
ART
eines
NN
Bauteils
VAFIN
wird
Noun-Phrase
ART
das
NN
Problem
VVPP
gelöst


Sentence
Sentence-Phrase
NN
Software
Verb-Phrase
VAFIN
ist
VVPP
aktualisiert
KON
und
Sentence
Sentence-Phrase
Noun-Phrase
ART
die
NN
Maschine
Verb-Phrase
VAFIN
ist
VVPP
eingestellt
.

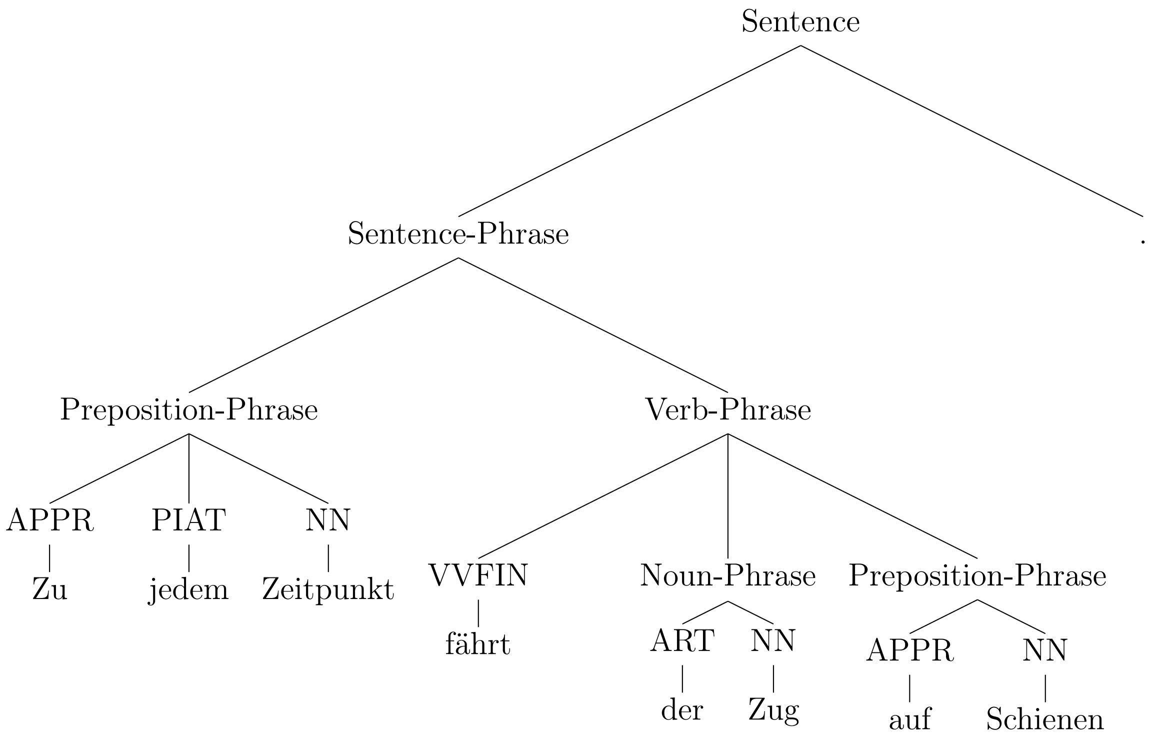

Sentence
Sentence-Phrase
Noun-Phrase
ART
Der
NN
Kunde
VVFIN
bemängelt
,
Subordinate-Clause
KOUS
dass
Noun-Phrase
ART
die
NN
Fahrkupplung
SC-Verb-Phrase
ADJD
schwergängig
VAFIN
ist
.

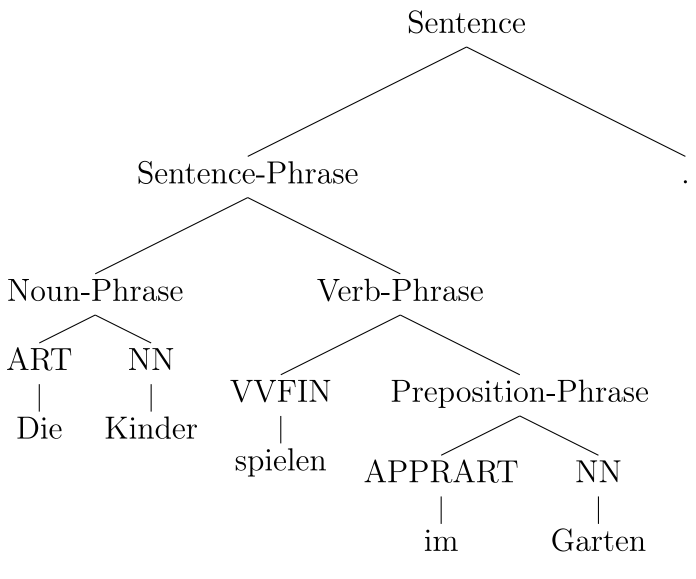

Sentence
Sentence-Phrase
.
Noun-Phrase
ART
Die
NN
Kontrollleuchte
Verb-Phrase
VVFIN
leuchtet
Preposition-Phrase
APPR
nach
Noun-Phrase
Noun-Phrase
ART
dem
NN
Anziehen
Noun-Phrase
ART
der
NN
Bremse
Adverb-Phrase
PTKNEG
nicht
ADV
mehr
PTKVZ
auf

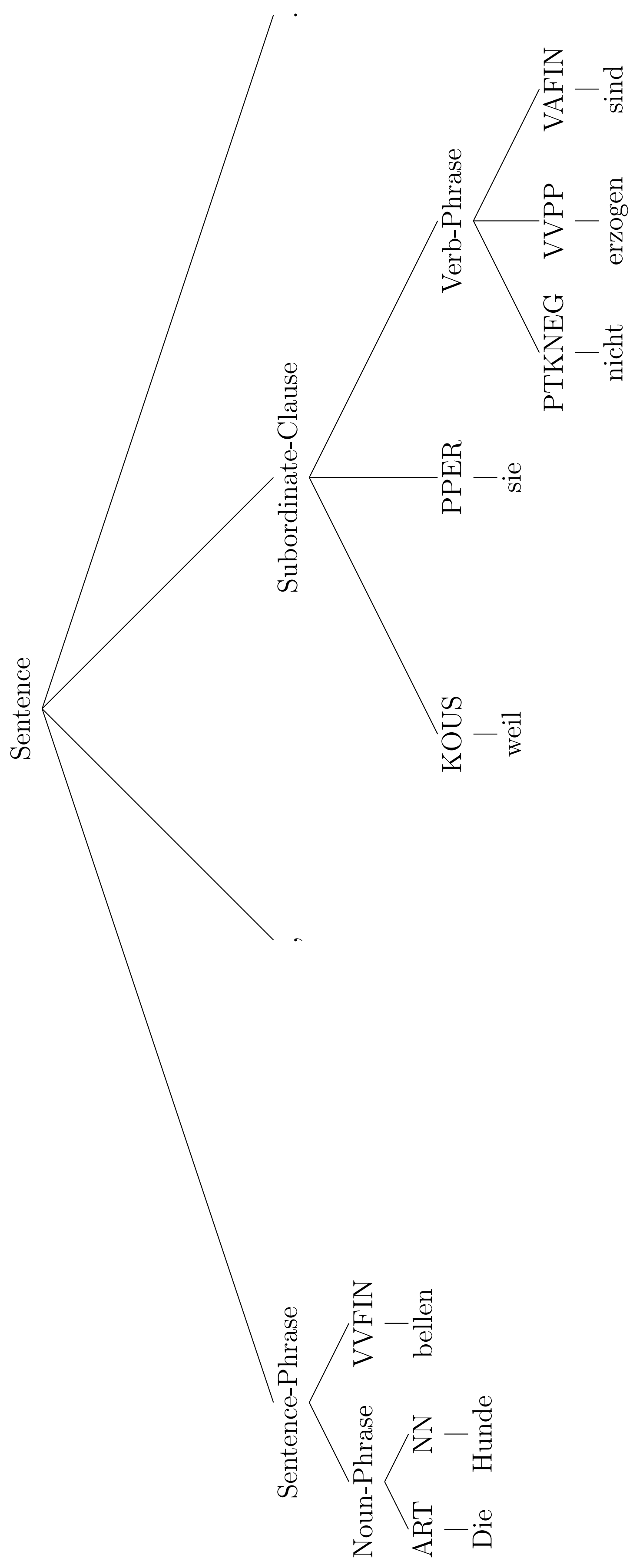

Sentence
Subordinate-Clause
.
PPER
Wir
Verb-Phrase
VVFIN
essen
Noun-Phrase
NN
Semmeln
KON
und
NN
Brot

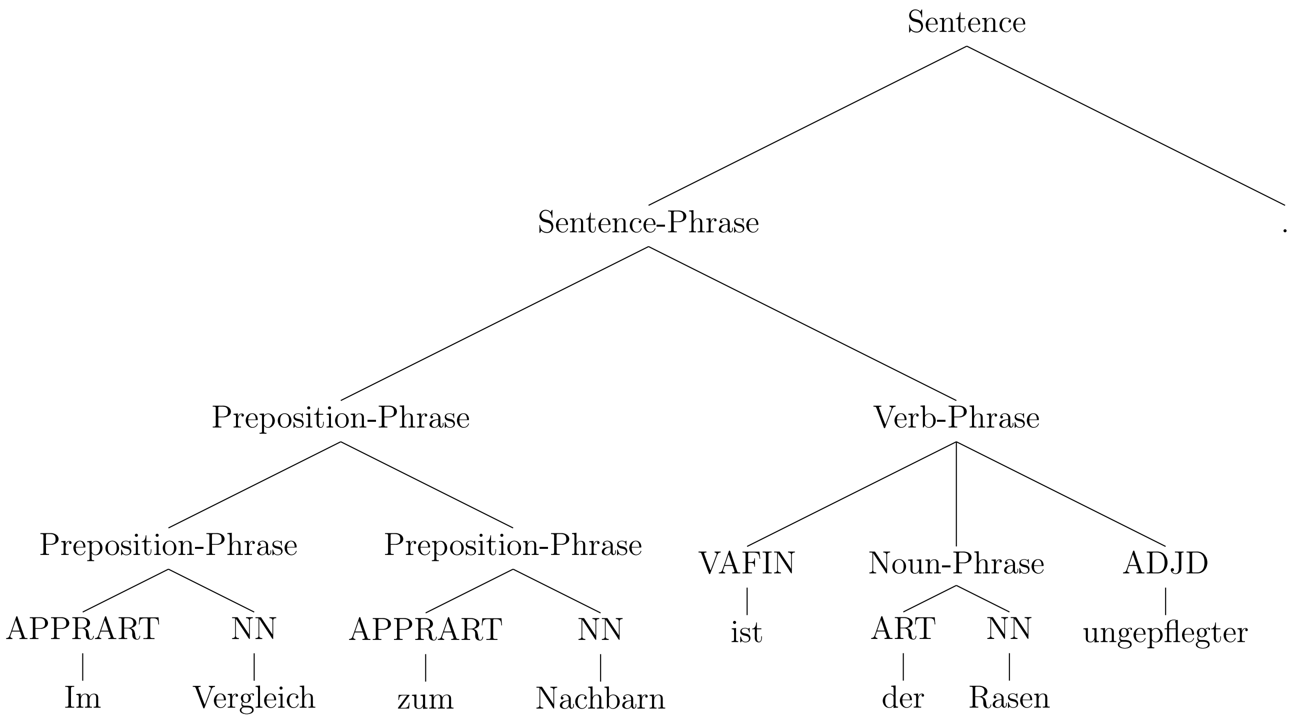

Sentence
Sentence-Phrase
KON
und
Sentence
Noun-Phrase
Verb-Phrase
ADJA
Alte
NN
Teile
VAFIN
sind
VVPP
entfernt
Sentence-Phrase
.
Noun-Phrase
Verb-Phrase
ADJA
neue
NN
Bauteile
VAFIN
sind
VVPP
eingebaut

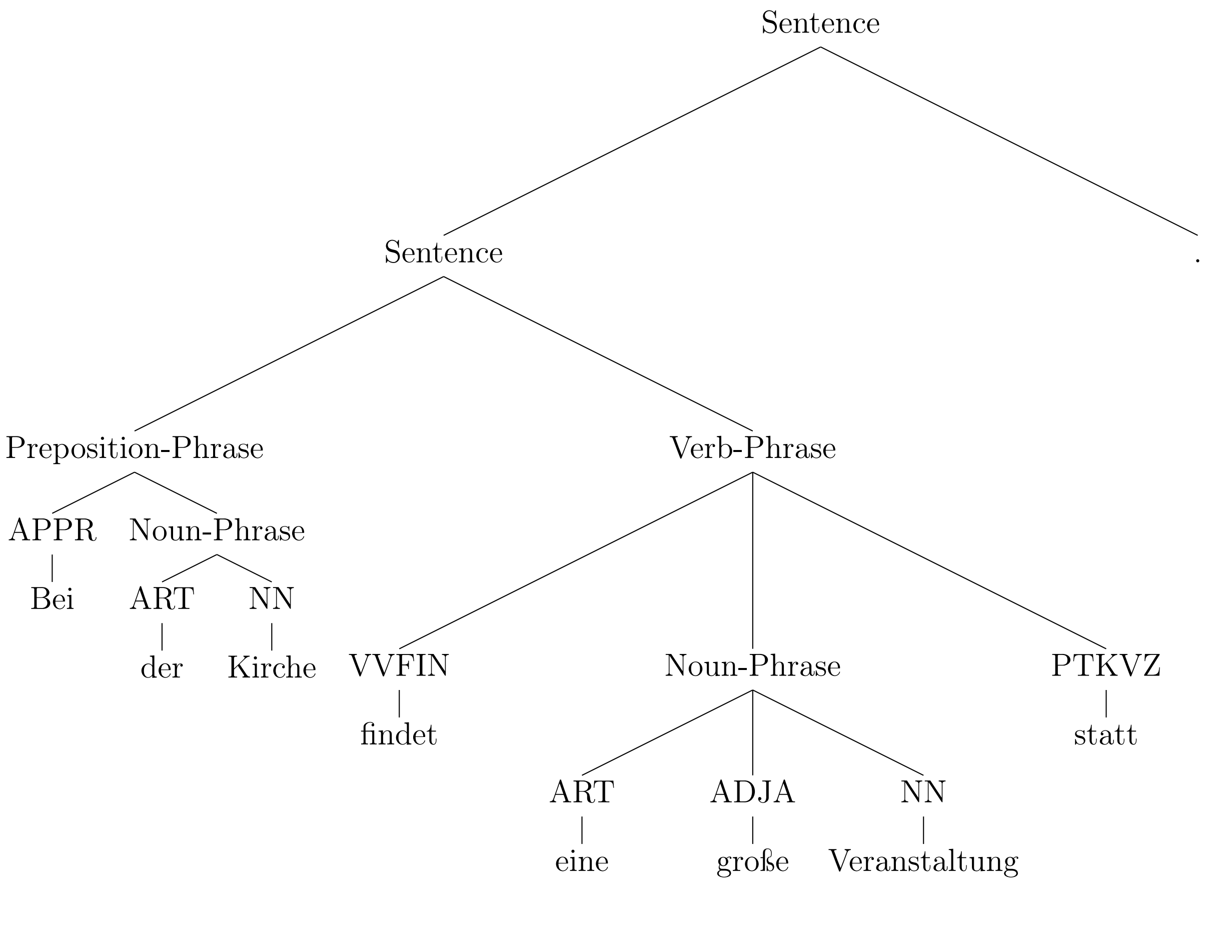
# B Parsing trees of PoC-domain using Earley

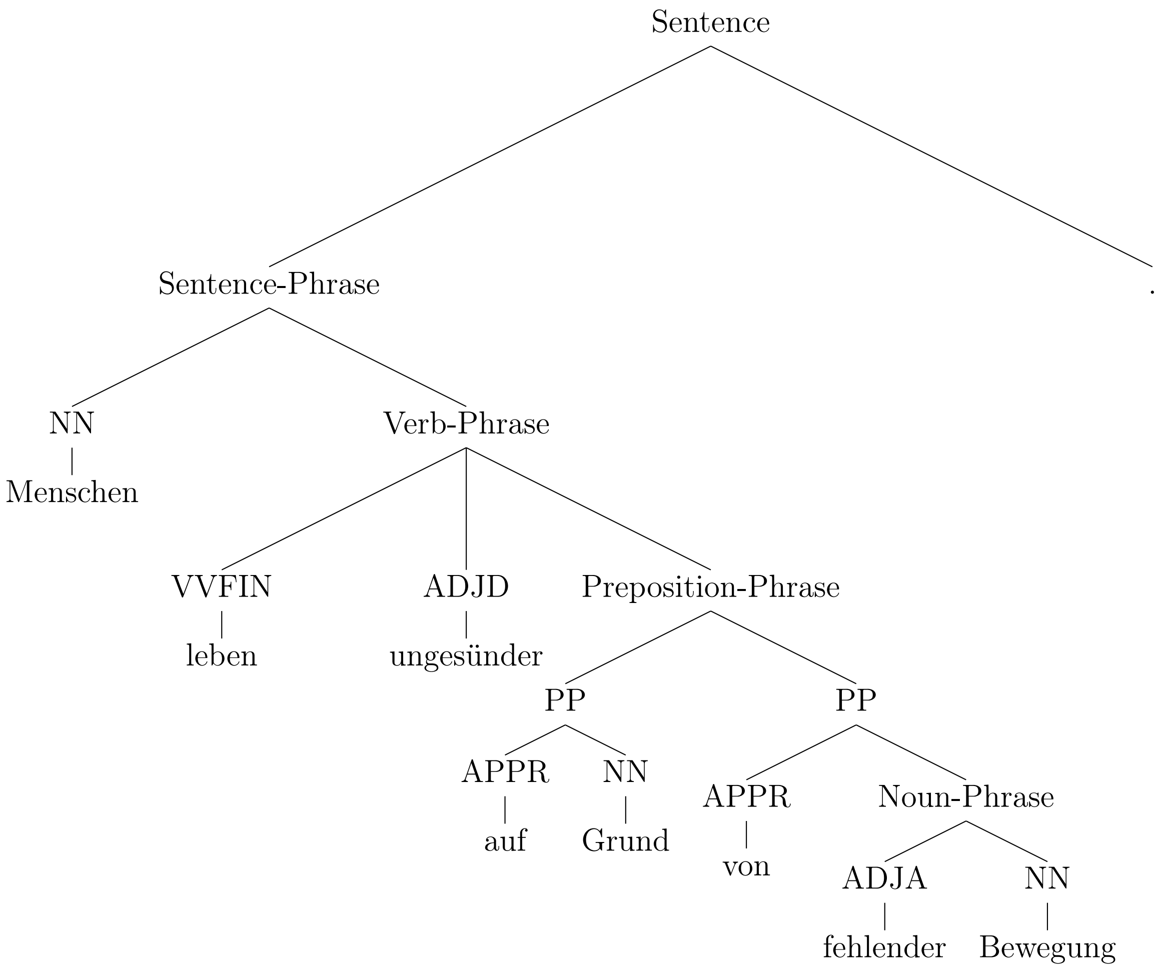


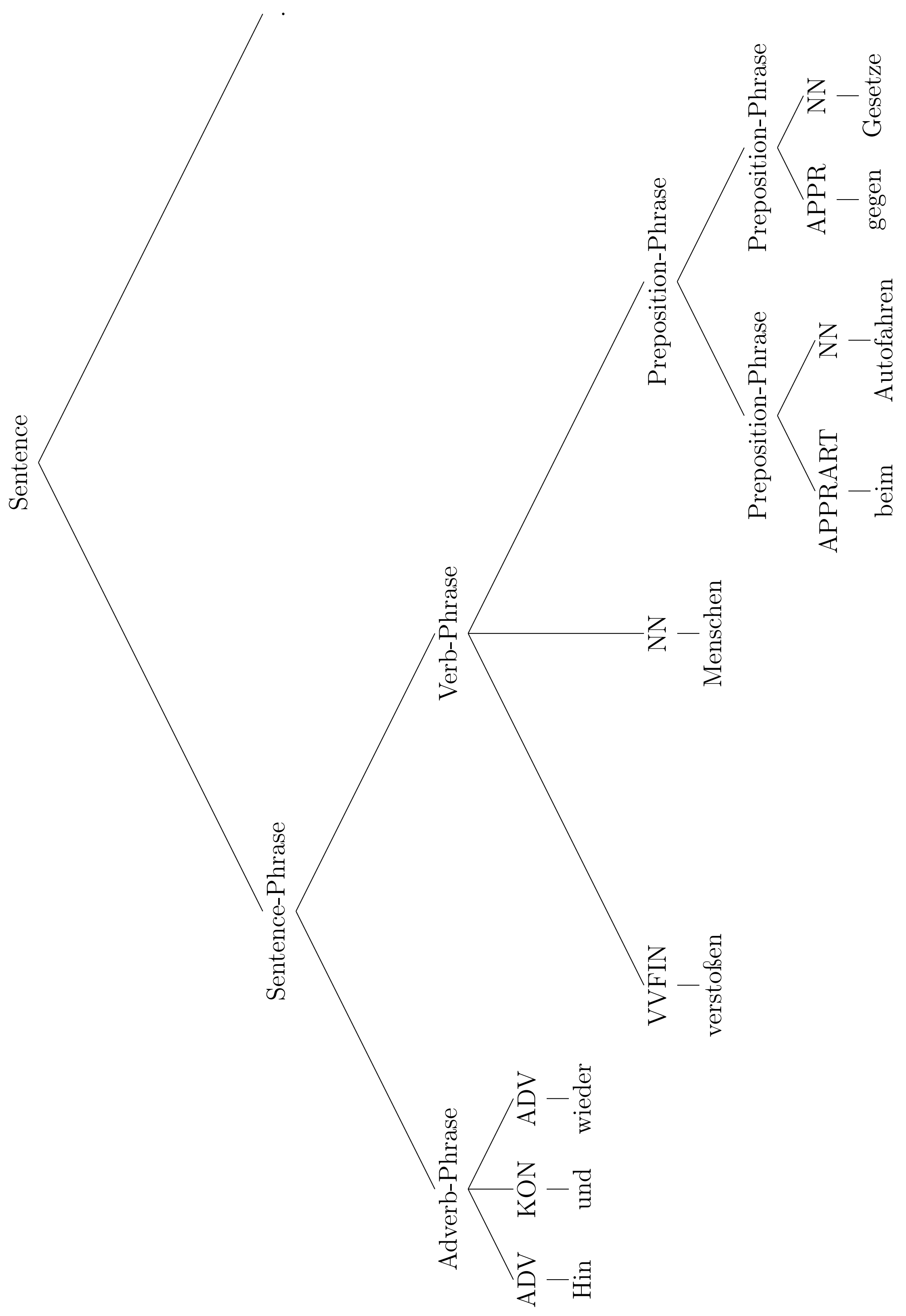


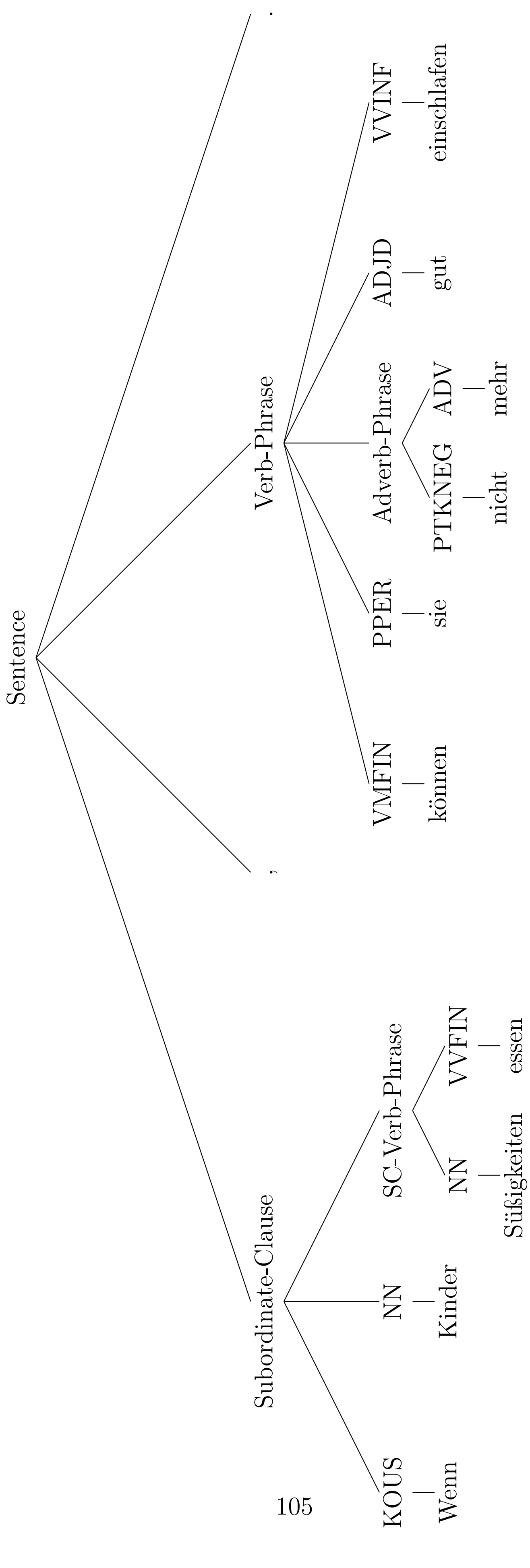

Sentence
Sentence-Phrase
Noun-Phrase
ART
Die
NN
Hunde
VVFIN
bellen
,
Subordinate-Clause
KOUS
weil
PPER
sie
Verb-Phrase
PTKNEG
nicht
VVPP
erzogen
VAFIN
sind
.

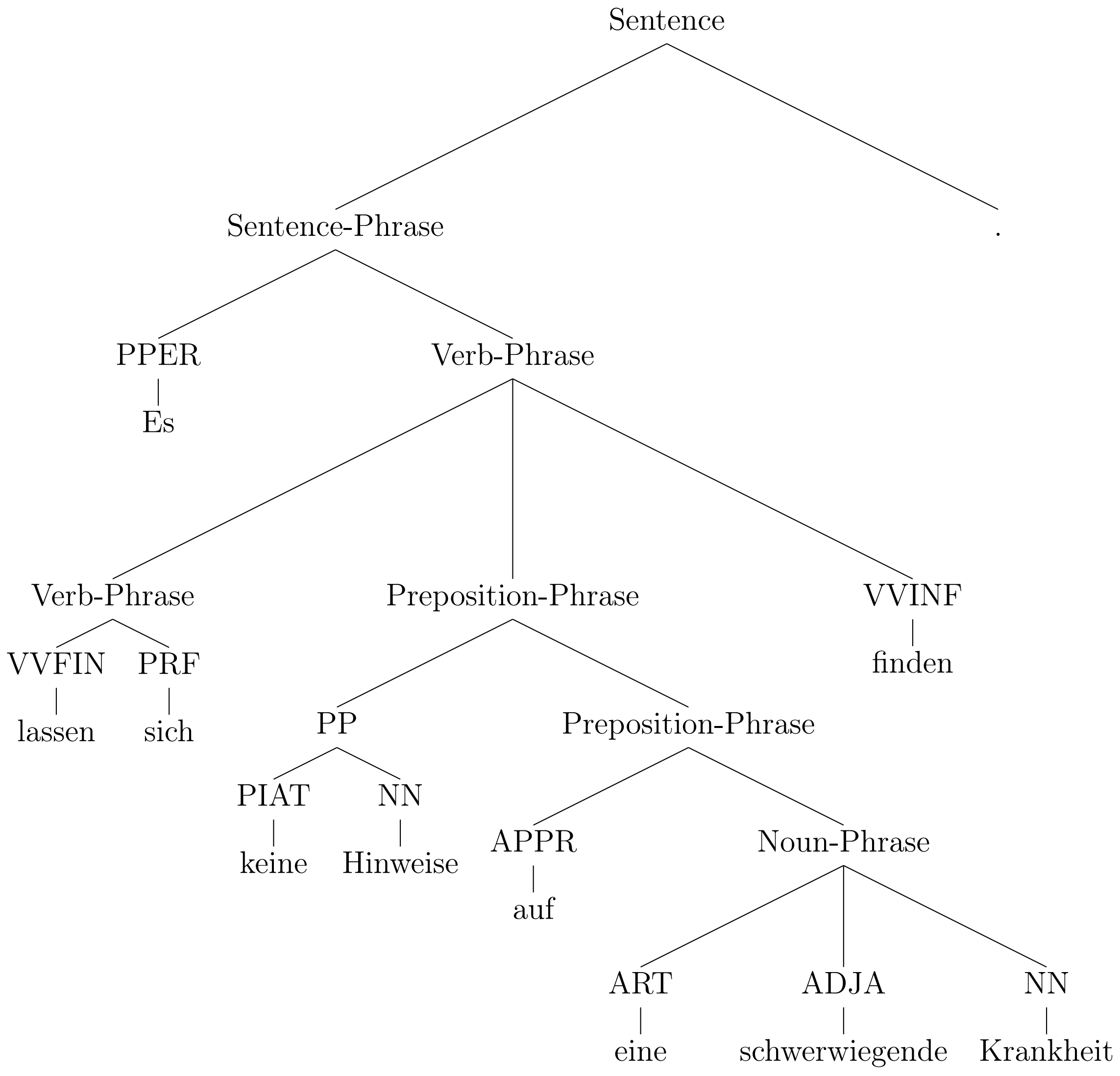

Sentence
Sentence-Phrase
.
Preposition-Phrase
Verb-Phrase
Preposition-Phrase
Preposition-Phrase
VAFIN
Noun-Phrase
ADJD
APPRART
NN
APPRART
NN
ist
ART
NN
ungepflegter
Im
Vergleich
zum
Nachbarn
der
Rasen

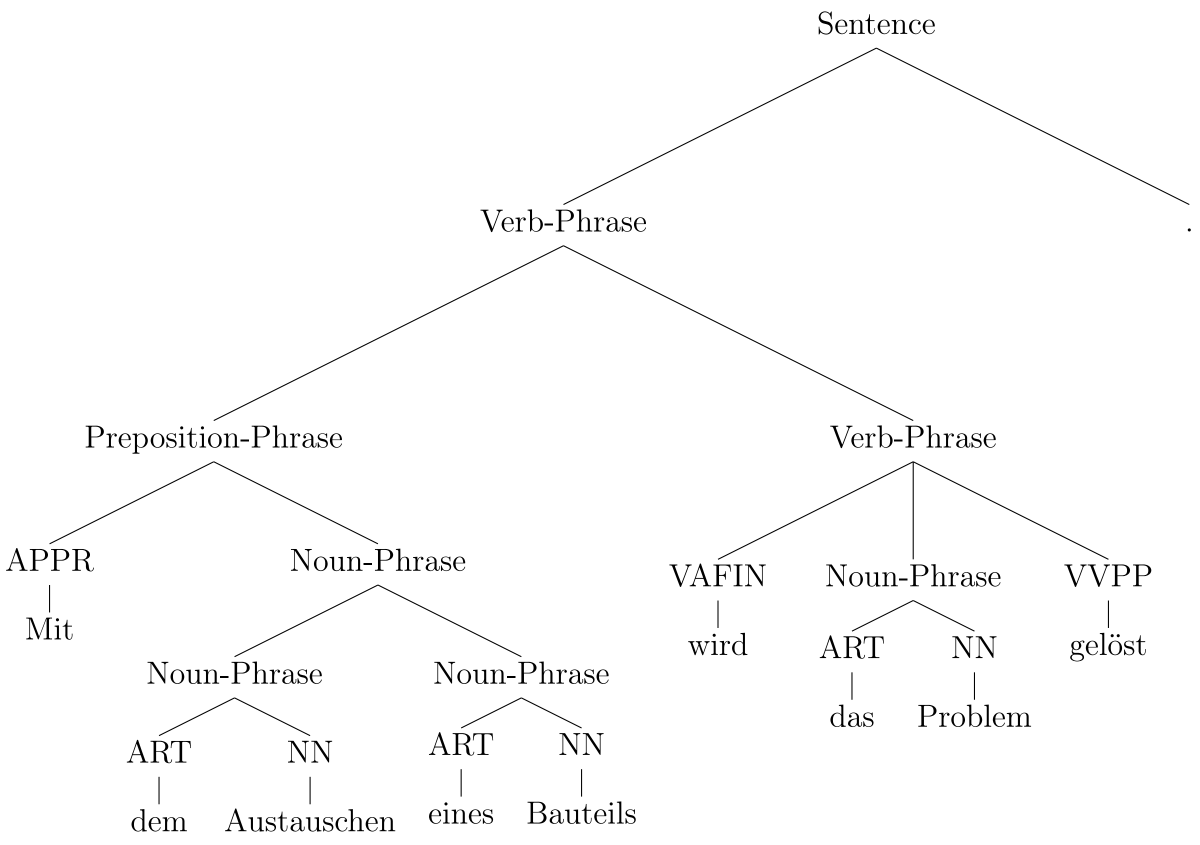

Sentence
Sentence
.
Preposition-Phrase
Verb-Phrase
APPR
Noun-Phrase
Bei
ART
NN
der
Kirche
VVFIN
Noun-Phrase
PTKVZ
findet
statt
ART
ADJA
NN
eine
große
Veranstaltung

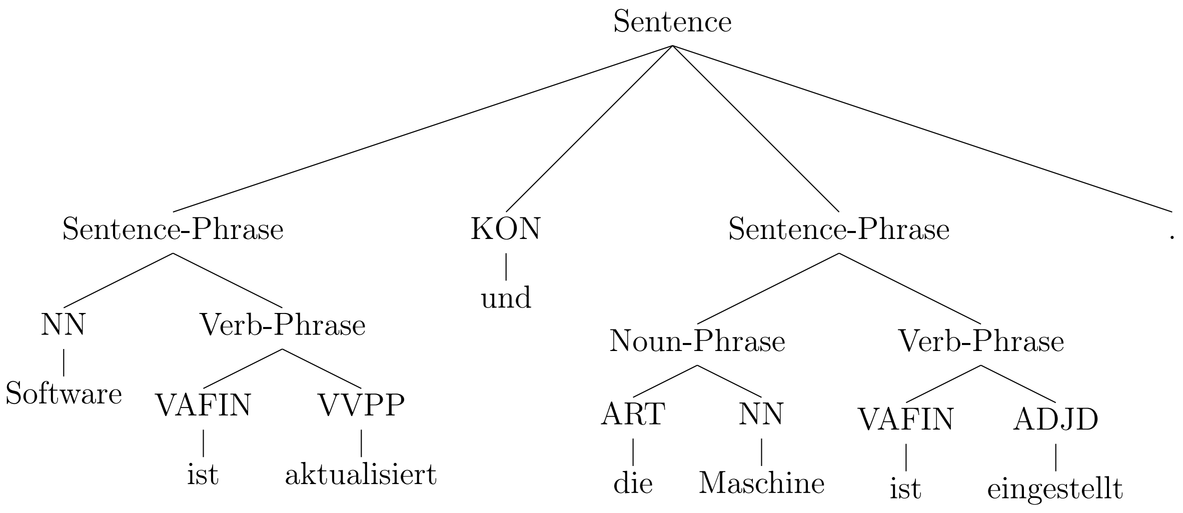

Sentence
Sentence-Phrase
.
NN
Menschen
Verb-Phrase
VVFIN
leben
ADJD
ungesünder
Preposition-Phrase
PP
APPR
auf
NN
Grund
PP
APPR
von
Noun-Phrase
ADJA
fehlender
NN
Bewegung

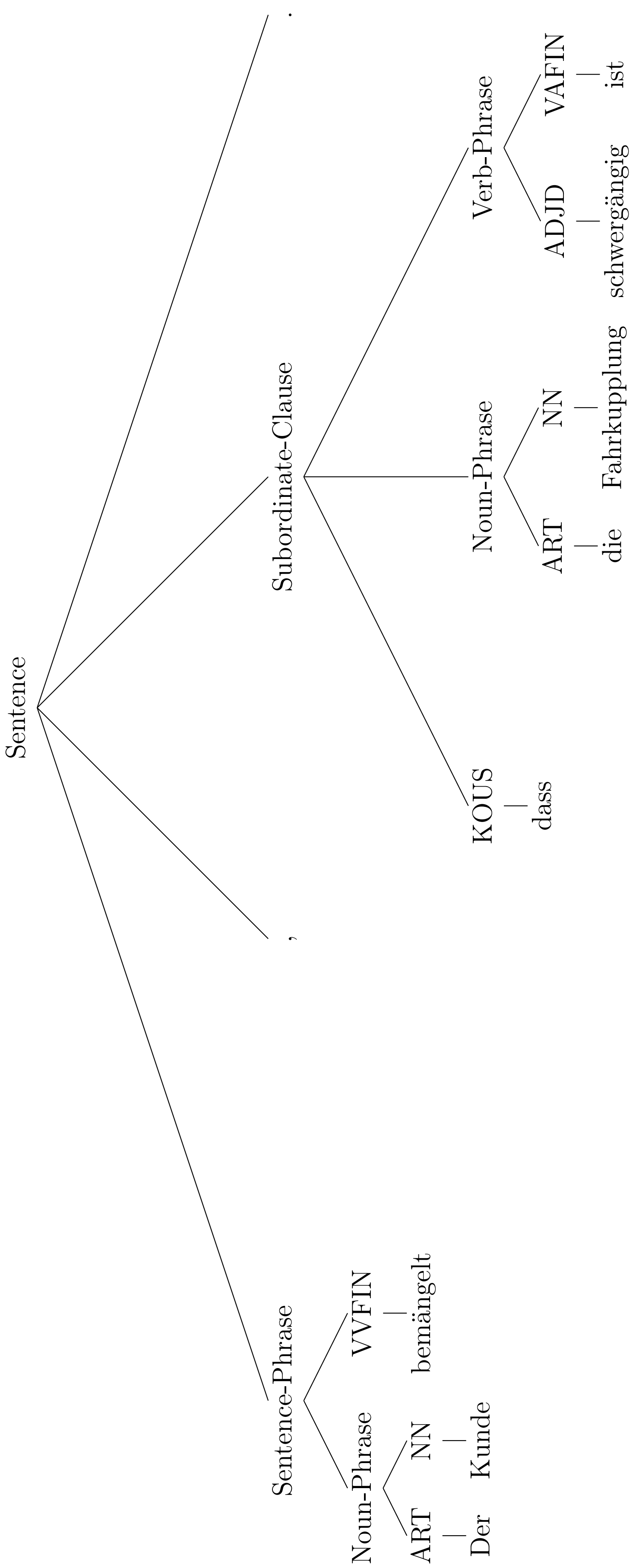

Sentence
Sentence-Phrase
.
Adverb-Phrase
ADV
Hin
KON
und
ADV
wieder
Verb-Phrase
VVFIN
verstoßen
NN
Menschen
Preposition-Phrase
Preposition-Phrase
APPRART
beim
NN
Autofahren
Preposition-Phrase
APPR
gegen
NN
Gesetze

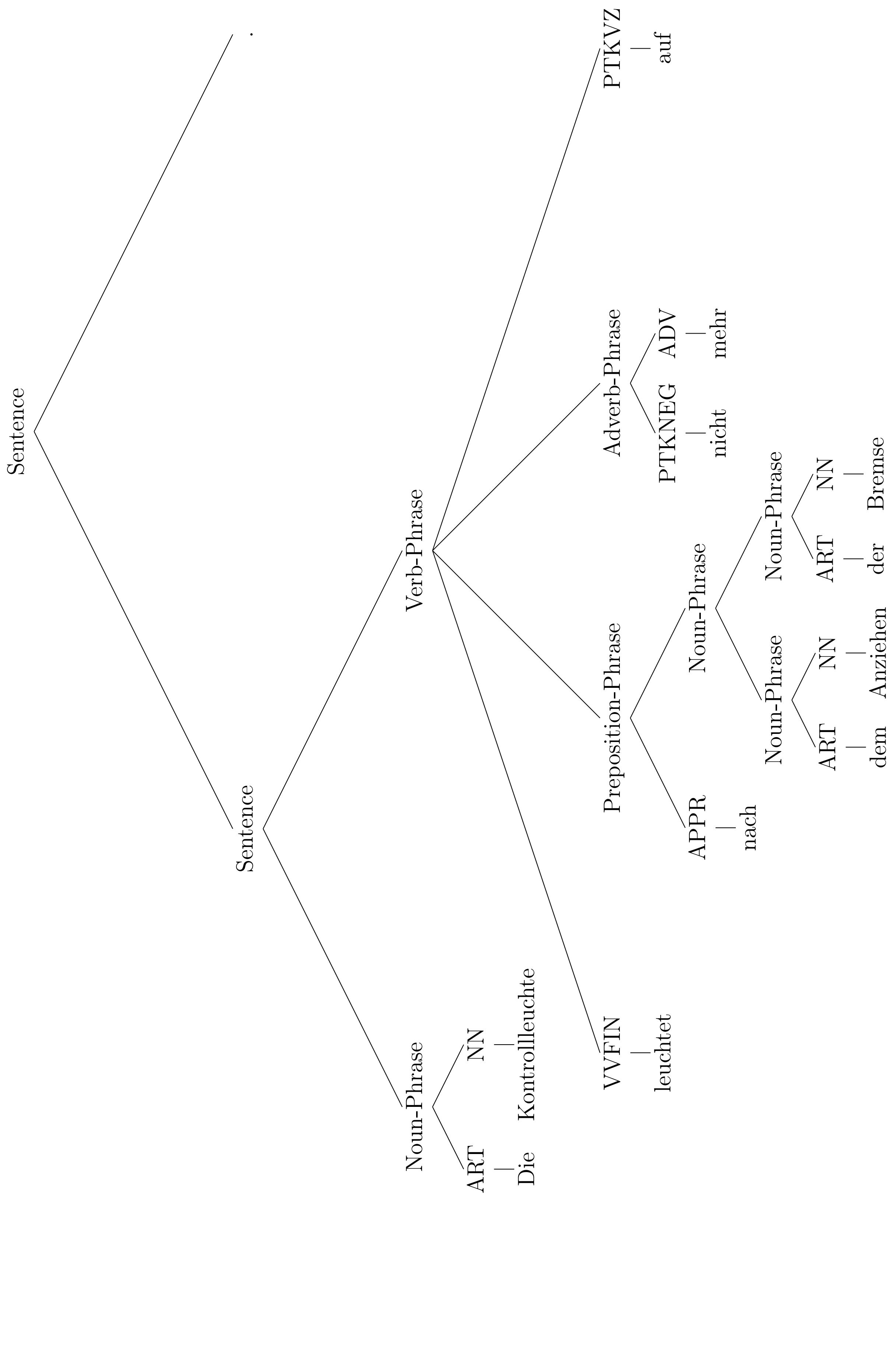

Sentence
Subordinate-Clause
,
Verb-Phrase
.
KOUS
Wenn
NN
Kinder
SC-Verb-Phrase
NN
Süßigkeiten
VVFIN
essen
VMFIN
können
PPER
sie
Adverb-Phrase
PTKNEG
nicht
ADV
mehr
ADJD
gut
VVINF
einschlafen

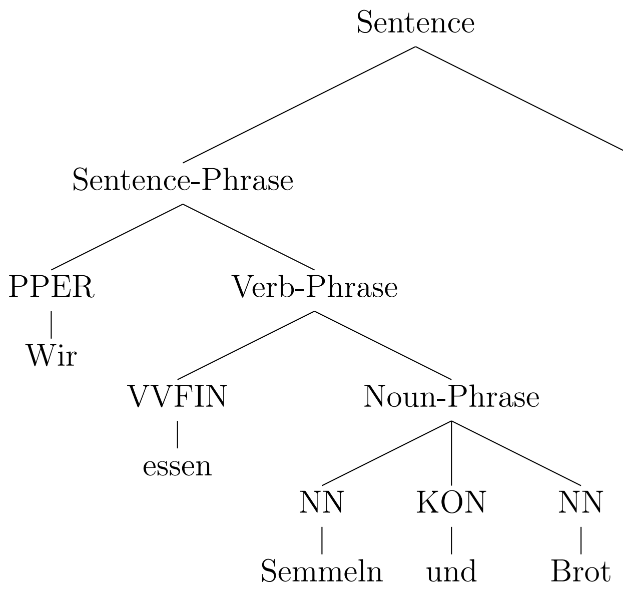

Sentence
Sentence-Phrase
.
PPER
Es
Verb-Phrase
Verb-Phrase
VVFIN
lassen
PRF
sich
Preposition-Phrase
PP
PIAT
keine
NN
Hinweise
Preposition-Phrase
APPR
auf
Noun-Phrase
ART
eine
ADJA
schwerwiegende
NN
Krankheit
VVINF
finden

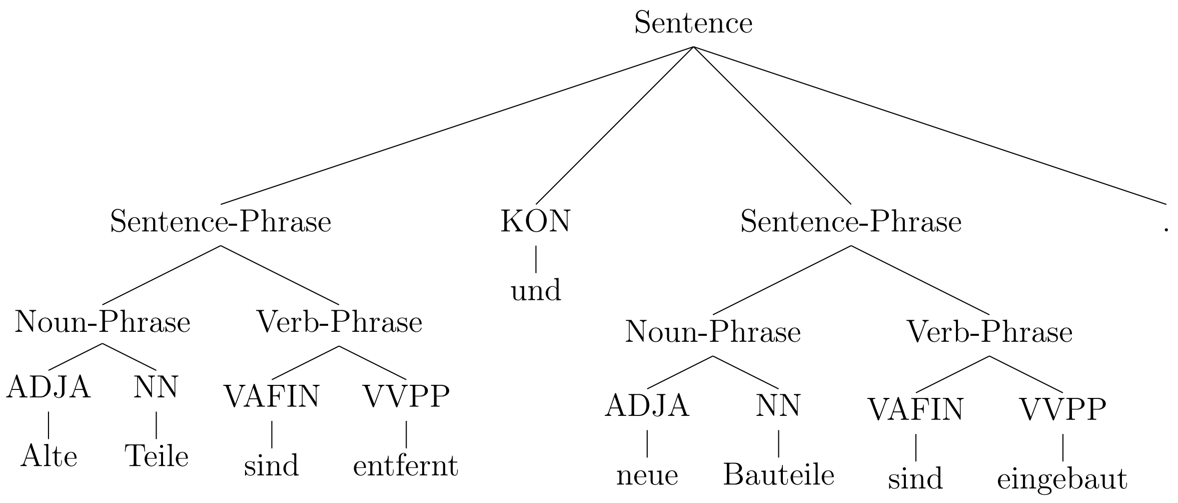

Sentence
Verb-Phrase
.
Preposition-Phrase
Verb-Phrase
APPR
Mit
Noun-Phrase
Noun-Phrase
ART
dem
NN
Austauschen
Noun-Phrase
ART
eines
NN
Bauteils
VAFIN
wird
Noun-Phrase
ART
das
NN
Problem
VVPP
gelöst


Sentence
Sentence-Phrase
KON
und
Sentence-Phrase
.
NN
Software
Verb-Phrase
VAFIN
ist
VVPP
aktualisiert
Noun-Phrase
ART
die
NN
Maschine
Verb-Phrase
VAFIN
ist
ADJD
eingestellt


Sentence
Sentence-Phrase
Noun-Phrase
ART
Der
NN
Kunde
VVFIN
bemängelt
,
Subordinate-Clause
KOUS
dass
Noun-Phrase
ART
die
NN
Fahrkupplung
Verb-Phrase
ADJD
schwergängig
VAFIN
ist
.


Sentence
Sentence
.
Noun-Phrase
ART
Die
NN
Kontrollleuchte
Verb-Phrase
VVFIN
leuchtet
Preposition-Phrase
APPR
nach
Noun-Phrase
Noun-Phrase
ART
dem
NN
Anziehen
Noun-Phrase
ART
der
NN
Bremse
Adverb-Phrase
PTKNEG
nicht
ADV
mehr
PTKVZ
auf


Sentence
Sentence-Phrase
.
PPER
Wir
Verb-Phrase
VVFIN
essen
Noun-Phrase
NN
Semmeln
KON
und
NN
Brot


Sentence
Sentence-Phrase
KON
und
Sentence-Phrase
.
Noun-Phrase
ADJA
Alte
NN
Teile
Verb-Phrase
VAFIN
sind
VVPP
entfernt
Noun-Phrase
ADJA
neue
NN
Bauteile
Verb-Phrase
VAFIN
sind
VVPP
eingebaut

# C Stuttgart-Tübingen-Tagset

| pos-tag | Beschreibung | Beispiel(e) |
|---|---|---|
| ADJA | attributives Adjektiv | *der schlaue/ADJA Mitarbeiter* |
| ADJD | adverbiales ODER | *er spricht schnell/ADJD* |
|  | prädikatives Adjektiv | *Sein Sprechen ist schnell/ADJD* |
| ADV | Adverb | *Bald/ADV schon/ADV kommt sie wohl/ADV* |
| APPR | Präposition; Zirkumposition links | *nach/APPR Berlin; ohne/APPR Hund* |
| APPRART | Präposition mit Artikel | *zum/APPRART Streichen; zur/APPRART Sache* |
| APPO | Postposition | *ihm zuliebe/APPO; der Sache wegen/APPO* |
| APZR | Zirkumposition rechts | *von mir aus/APZR* |
| ART | bestimmter ODER | *Der/ART Mann schenkt die/ART Rose* |
|  | unbestimmter Artikel | *einer/ART unerwarteten Frau* |
| CARD | Kardinalzahl | *zwei/CARD Männer im Jahre 1994/CARD* |
| FM | Fremdsprachliches Material | *Er sagte:" Hasta/FM luego/FM, amigos/FM ."* |
| ITJ | Interjektion | *Mhm/ITJ, ach/ITJ, tja/ITJ, dann halt nicht.* |
| KOUI | unterordnende Konjunktion mit (*zu*-)Infinitiv | *Sie kommt, um/KOUI zu arbeiten* |
|  |  | *Anstatt/KOUI anzufangen, geht sie wieder* |
| KOUS | unterordnende Konjunktion | *Emma wartet, weil/ob/solange/dass/KOUS sie stiehlt* |
| KON | nebenordnende Konjunktion und, oder, aber | *Sie und/oder/KON Emma kommen und/KON streichen* |
| KOKOM | Vergleichskonjunktion als, wie | *blauer als/KOKOM er; blau wie/KOKOM er* |
| NN | normales Nomen | *am Tage/NN dem Mann/NN den Schlaf/NN* |
| NE | Eigennamen | *die Emma/NE dem Hans/NE sein HSV/NE* |
| PDAT | attribuierendes Demonstrativpronomen | *Jene/PDAT Männer sprachen dieses/PDAT lockere Spanisch* |
| PDS | substituierendes Demonstrativpronomen | *Denen/PDS war dies/PDS nicht übelzunehmen* |
| PIAT | attribuierendes Indefinitpronomen | *Manche/PIAT Rose währt einige/PIAT Tage* |
| PIS | substituierendes Indefinitpronomen | *Manche/PIS verzeiht niemandem/PIS* |
| PPER | (nicht-reflexives) Personalpronomen | *Er/PPER schenkt sie/PPER ihr/PPER* |
| PPOSAT | attribuierendes Possessivpronomen | *Unsere/PPOSAT Wand ist rosa* |
| PPOSS | substituierendes Possessivpronomen | *Meiner/PPOSS schlägt deinen/PPOSS* |
| PRELS | substituierendes Relativpronomen | *die Mannschaft, der/PRELS du nacheiferst* |
| PRELAT | attribuierendes Relativpronomen | *die Mannschaft, deren/PRELAT Aura du verehrst* |
| PRF | Reflexivpronomen | *Erinnere dich/PRF, wie er sich/PRF ereiferte* |
| PWS | substituierendes Interrogativpronomen | *Wer/PWS hat wen/PWS gestohlen?* |
| PWAT | attribuierendes Interrogativpronomen | *Wessen/PWAT Hund wurde gestohlen?* |
| PWAV | adverbiales Interrogativpronomen | *Warum/PWAV schneidest du die Rose?* |
| PROAV | Pronominaladverb | *Deswegen/PROAV sprechen wir darüber/PROAV* |
| PTKZU | "zu" vor Infinitiv | *Ich versuche zu/PTKZU verschlafen* |
| PTKNEG | Negationspartikel *nicht* | *Nicht/PTKNEG schlecht, wie du nicht/PTKNEG hinsiehst* |
| PTKVZ | abgetrennter Verbzusatz/Verbpartikel | *Pass auf/PTKVZ und hör weg/PTKVZ!* |
| PTKANT | Antwortpartikel | *ja; nein; danke; bitte* |
| PTKA | Partikel "*am*" o. "*zu*" vor Adjektiv o. Adverb | *Zu/PTKA teure Rosen welken am/PTKA schnellsten* |
| TRUNC | abgetrenntes Kompositionserstglied | *Mallorca liegt zwischen An-/TRUNC und Abreise* |
| VVFIN | finites Vollverb | *Wir passen/VVFIN auf und hören/VVFIN* |
| VAFIN | finites Voll- oder Kopulaverb | *Sie ist/VAFIN blumig. Du hast/VAFIN weggehört.* |
| VMFIN | finites Modalverb | *Sie sollte/VMFIN passen* |
| VVINF | infinites Vollverb | *Wir wollen weghören/VVINF.* |
| VAINF | infinites Hilfsverb oder Kopulaverb | *Sie soll rot geworden sein/VAINF* |
| VMINF | infinites Modalverb | *Er hat nicht schlafen können/VMFIN.* |
| VVIMP | Vollverb im Imperativ | *Pass/VVIMP auf und hör/VVIMP weg!* |
| VAIMP | Kopulaverb im Imperativ | *Sei/VAIMP wach!* |
| VVPP | partizipiales Vollverb (Partizip II) | *Wir haben verschlafen/VVPP.* |
| VAPP | partizipiales Hilfs-/Kopulaverb (Partizip II) | *Das ist verdrängt worden/VAPP* |
| VMPP | partizipiales Modalverb (Partizip II) | *Sie hat spielen gedurft/VVPP* |
| VVIZU | Vollverb/Partikelverb im "zu"-Infinitiv | *Wir planen wegzuhören/VVIZU* |
| XY | Nichtwort, Sonderzeichen, Kürzel | *Es enthält viel D2XW3/XY* |
| $, | Komma | , |
| $( | sonstige satzinterne Interpunktion | *( )* u.a. |
| $. | satzbeendende Interpunktion | *. ? ! ;* |

Available on the website: https://www.linguistik.hu-berlin.de/de/institut/professuren/korpuslinguistik/mitarbeiter-innen/hagen/STTS_Tagset_Tiger