UNIKASSEL VERSITÄT | ELEKTROTECHNIK INFORMATIK

Using Shared Packet Parse Forests to compute all Minimal Corrections

Master Thesis

Submitted in partial fulfillment of the requirements

for the degree of Master of Science

Maurice Herwig

Department:	Theoretical Computer Science/ Formal Methods
Examiner:	Prof. Dr. Martin Lange Dr. Jens Kosiol
Supervisor:	Dr. Norbert Hundeshagen

Abstract

An error-correcting parser computes a sequence of deletions, insertions, and replacements (correction) to transform a word so that it is contained in a context-free language. However, often, it is useful to compute a set of corrections rather than a single correction. Therefore, in this work, we present a cubic algorithm that is based on the Earley Parser to compute the possible infinite set of corrections for a word and a context-free language. In addition, we present correction shared packed parse forests (CSPPF), an adapted data structure of shared packed parse forests, to store the infinite set of corrections computed by the algorithm. Furthermore, we introduce some filters to determine for different definitions of minimality the set of minimal corrections from all corrections.

Declaration

I confirm that the submitted thesis is original work and was written by me. Some text passages have been revised using AI, but this has not changed the content.
Appropriate credit has been given where reference has been made to the work of others.
The thesis was not submitted before, nor has it been published. The submitted electronic version of the thesis matches the printed version.

Kassel, August 2024

Maurice Herwig

Contents

1 Introduction **4**

2 Preliminaries **6**
2.1 Mathematical Foundations 6
2.2 Context-Free Languages 7
2.3 Earley Parser 8
2.4 Syntax Trees 9
2.5 SPPF: Shared Packed Parse Forest 13
2.6 Generalised Earley Parser 19
2.7 Disambiguation Filter 20
2.8 Corrections 22
2.9 Error-correcting Parsing 26

3 Indexless Corrections **29**
3.1 Corrections vs. Indexless Corrections 32
3.2 Order on Indexless Corrections 39

4 Computing all Corrections **45**
4.1 CSPPF: Correction Shared Packet Parse Forest 46
4.2 All-Correction Earley Parser 54
4.3 Correctness 61
4.4 Analysis 68

5 Filter on CSPPF **74**
5.1 Minimality Filters 74
5.1.1 A Definition of Minimality 75
5.2 No-Loop Filter 77
5.3 Simplification Filter 83
5.4 No-Super Correction Filter 88

6 Conclusion **93**

References **95**

A Algorithm of the Generalised Earley Parser **97**

B Earley Items of Example 2.7 **101**

C Earley Items of Example 2.11 **102**

D Earley Items of Example 4.3 **104**

1 Introduction

Every developer knows the problem: if the program code contains at least one syntax error, it cannot be processed by a compiler or interpreter. Modern IDEs[1] usually highlighting such syntax errors and offer suggestions for correcting them. But what are good correction suggestions in this case? Therefore, let us consider the following Python[2] code:

```
def fibonacci(n):
    if n > 2:
        return fibonacci(n - 1) + fibonacci(n - 2)
    else:
```

This code is obviously incorrect Python code because an indented block after the *else* statement is expected. The simplest correction in this case is to delete the *else* statement. However, this is probably not what a developer wanted to express semantically. Therefore, it would be desirable to receive suggestions for corrections, such as: inserting *return 1* or replace *else:* by *return 1*.

Another use case that illustrates the importance of good correction suggestions is teaching hypothesis formulation in the biology classroom [6]. The setting in this case is that all valid hypotheses are described by a context-free grammar. Students formulate a hypothesis and automatically receive feedback suggestions to correct their hypothesis if their hypothesis is syntactically incorrect. This leads the students into an *error-driven learning cycle*. It is obvious that good correction suggestions enhance the student's learning process.

From a more mathematical point of view, a correction is a sequence of insertions, replacements, and deletions and can be computed for a context-free language and a word in the parse process of the word. A parser such as the Earley Parser[4] or the CYK-algorithm [22] checks whether a word is contained in a given context-free language. If this is the case, the parser can compute a syntax tree for the word, which describes the syntactic structure of the word. Different syntax trees regarding a word and a grammar can be efficiently stored in an *shared packed parse forest* (SPPF) [14, 19] through nodes with the same subtree in different syntax trees are *shared*. Such a SPPF can be computed by generalized parsers [15, 16, 18, 17]. If the word is not in the language, then an *error correction parser*[1, 13] calculates a correction. For instances the correction with the minimum number of edits. However, only one correction is calculated.

But in the introductory examples, we were interested in a set of corrections that made sense with regarding to the use case. This set is also called the set set of *minimal corrections* and can be specified by some definition of minimality. For example: as the set of corrections consisting only of a maximum number of operations; as corrections resulting in target words with a length in a given range; as corrections for which there is no smaller correction by a given strict partial order on the set of all corrections; ... or a more complex definition as in the use case of teaching hypothesis formulation: A *correction* is contained in the set of *minimal corrections* if, for the correction itself and for all corrections resulting from a change in the order of operations, *no subsequence* of these corrections already leads into the target language [5].

But how can we compute the set of *all minimal corrections* for a context-free grammar and a word with respect to a given *definition of minimality* in an *efficient* way? This is the main aim of this thesis. Therefore, in this work we present an efficient algorithm to solve this problem. The algorithm can be divided into two independent steps:

1. Compute a *(correction) shared packet parse forest* that contains *all corrections*.
2. *Filtering* out all minimal corrections.

[1] `https://en.wikipedia.org/wiki/Integrated_development_environment`
[2] `https://www.python.org/`

For the first step of our algorithm, we adapt the generalized Earley Parser [15] by adding to the conventional Earley rules: *scanner*, *predictor* and *completer* a *rule* each for *insertion*, *replacement* and *deletion*. In addition, we extend the SPPF so that it contains the set of all corrections regarding a word and a context-free grammar by labeling the leaves of the SPPF from left to right with the operations of the correction. Thus, a data structure can be computed that contains *all corrections* in time $\mathcal{O}(n^3)$ regarding an word of length n.

A big advantage is that we can compute in an *efficient way* the set of all corrections. But we are interested in the set of minimal corrections, depending on the used definition of minimality. Therefore, we define for the second step of our algorithm different *filters* for different definitions of minimality. More complex definitions of minimality, as the mainly considered definition in this work, can be expressed by combinations of different filters. Which allows us to easily adapt the algorithm for different definitions of minimality by chaining the used filters. The concept of filtering an SPPF is based on the concept of *disambiguation filters* [7, 9, 20].

Related Work

A approach [5] of finding all minimal corrections is to enumerate all corrections and solve the word problem for the corrected word and for all resulting words by applying a subcorrection of the correction or a correction with changed operation positions. If the set of minimal corrections is finite, then the algorithm terminates at one point. But this is a very inefficient way of solving the problem of finding the set of all minimal corrections.

A more efficient way, but not yet proven, is to compute the set of minimal corrections iteratively by an extended version of the CYK-algorithm [3]. The basic idea is to store for each nonterminal in the CYK table a set of minimal corrections for the subword corresponding to that cell, and to compose the corrections by filling in new cells. Note: In the same way, we can also extend the Earley Parser by additionally storing a set of minimal corrections for each *Earley item* (see Sec. 4 of this work).

However, these approaches can only deal with a predefined definition of minimality. Therefore, to use another definition of minimality, the parser must be updated. This is a big difference from the algorithm presented in this work, where the minimality is defined by combinations of filters. This makes our algorithm more modular and allows more complex minimality definitions to be easily defined, for example, using a different minimality inside a *for*-loop than in the rest of the code. In addition, the approach presented in this work can also compute the set of minimal corrections if this is not finite due to the used definition of minimality.

The idea of first computing everything by storing it in an SPPF and then filtering out the required one by *combinations of filters* is not entirely new. This idea is also followed by [9] in the use case of disambiguation filters. All already defined filters [9, Sec. 4] for the realization of disambiguation rules can also be used to implement a minimality filter.

Structure of This Thesis

We consider in Section 2 the necessary preliminaries. In Section 3 we present with *indexless corrections* a formalization equivalent to corrections. This formalization of corrections allows us to define *correction shared packed parse forest* (CSPPF) in Secction 4.1. In addition, we present how we can extend the generalized Earley Parser to compute a CSPPF (Sec. 4.2) and show that the extended algorithm called *all-correction Earley Parser* computes a CSPPF that includes all indexless corrections (Sec. 4.3). To use different definitions of minimality, we present in Section 5 different filters on CSPPF and present how a combination of filters computes the set of all minimal corrections for the definition of minimality from [5].

2 Preliminaries

This chapter introduces the prerequisites needed for the following chapters. First, we introduce the necessary mathematical foundations (Sec. 2.1). Then we look at context-free languages (Sec. 2.2) and show how we can solve the word problem for this class of languages using the Earley Parser (Sec. 2.3). Syntax trees can be used to analyse the syntactic structure of a word according to a context-free grammar. Therefore we consider syntax trees in Section 2.4 and present with shared packed parse forests (Sec. 2.5) an efficient way to store a set of syntax trees. In addition, we show how we can compute a shared packed parse forest by a generalised version of the Earley Parser (Sec. 2.6). In Section 2.7 we consider disambiguation filters to filter out exactly one syntax tree from the set of syntax trees contained in a shared packed parse forest. At least we consider in Section 2.8 and Section 2.9 the case where a word is not contained in the given language and how we can compute a correction so that the input word can be corrected to a word in the language.

2.1 Mathematical Foundations

In this section, we will briefly introduce the basic mathematical principles required in this work. The main reference for the following section are the lecture slides of *Formale Sprachen und Logik* at the University of Kassel in 2019 [10] and the book *Mathematische Grundlagen der Informatik* [11].

Alphabet, Word and Language An *alphabet* Σ is a finite, non-empty set of alphabet symbols are also named *terminals*. A *word* w over some alphabet Σ is a finite sequence of letters $w = a_0 a_1 \dots a_{n-1}$, where the *empty word* is denoted as ε. For a word w, the length $|w|$ is defined as the number of symbols $|w| = |a_0 a_1 \dots a_{n-1}|$. The set of all words over some alphabet Σ is denoted by Σ^*. A *language* (L) is a subset of Σ^* ($L \subset \Sigma^*$). We call a language a *not empty* language if it contains at least one word $|L| > 1$.

Subwords A *subword* of a word is a sequence of consecutive characters occurring within that word. Formally, a subword of the word $w = a_0 a_1 \dots a_{n-1}$ is the sequence of consecutive symbols $a_i a_{i+1} \dots a_j$ where $i \leq j \leq n-1$. If $i = 0$, we call the subword a *prefix*, and if $j = n$, we call it a *suffix*. In this work, we write $w' \blacktriangleleft w$ if w' is a prefix of w.

Scattered Subwords A *scattered subword* (also known as a *subsequence*) is a sequence of letters that can be derived from a word by deleting some (or no) characters without changing the order of the remaining characters. A word w' is a scattered subword of the word $w = a_0 a_1 \dots a_{n-1}$, written as $w' \preceq w$, if it is of the form $a_{k_1} a_{k_2} \dots a_{k_m}$ with $0 \leq k_1 < k_2 < k_m \leq n$. If $w' \preceq w$ and $w' \neq w$ then we write $w' \prec w$. Every subword is also a scattered subword, but not every scattered subword is also a subword.

Example 2.1. Let $w = n * (n + n)$ be a word over the alphabet $\Sigma = \{n, *, +, (,)\}$, then $(n + n)$ is a subword that also is a suffix of w, and $n * n$ is a scattered subword of w.

Partially Ordered Set A *partial order* is a pair $(X, \preceq)$ of a set X and a *relation* $\preceq \subseteq X \times X$ that is *reflexive*, *antisymmetric*, and *transitive*. Therefore, it must satisfy the following conditions for all $x, y, z \in X$:

- **Reflexive** $x \preceq x$
- **Antisymmetric** if $x \preceq y$ and $x \preceq y$ then $x = y$
- **Transitive** if $x \preceq y$ and $y \preceq z$ then $x \preceq z$

Strict Partially Ordered Set A *strict partial order* is a pair $(X, \prec)$ of a set X and a relation $\prec \subseteq X \times X$ that is *irreflexive*, *asymmetric*, and *transitive*. Therefore, it must satisfy the following conditions for all $x, y, z \in X$:

- **Irreflexive** not $x \prec x$
- **Asymmetric** if $x \prec y$ then not $y \prec x$
- **Transitive** if $x \prec y$ and $y \prec z$ then $x \prec z$

Example 2.2. Let Σ be an alphabet, $\blacktriangleleft$ the prefix relation and $\prec$ the scattered subword relation, then the pair $(\Sigma^*, \blacktriangleleft)$ is an example of a partial order, and the pair $(\Sigma^*, \prec)$ is an example of a strict partial order.

2.2 Context-Free Languages

Context-free languages are widely used in computer science, especially in the modeling of programming languages, in compiler construction, in parsing algorithms, in natural language processing, etc., and as a formal language per se. The main reference for the following section are the lecture slides of *Formale Sprachen und Logik* at the University of Kassel in 2019 [10].

A *language* is contained in the class of *context-free languages* (CFL) if it is generated by a *context-free grammar* G, denoted as 4-tuple $G = (N, \Sigma, P, S)$, where N is a finite non-empty set called *nonterminals*, where Σ is an alphabet with $N \cap \Sigma = \emptyset$, where $a \in \Sigma$ is called a terminal, P is a set of *production rules* $P \subseteq N \times (N \cup \Sigma)^*$, and where $S \in N$ is the *start symbol*. As a convention, we number nonterminal symbols in uppercase $A, B, \ldots$, terminal symbols in lowercase $a, b, \ldots$, and a $(N \cup \Sigma)^*$-word, also called a word in *sentential form*, in lowercase from the Greek alphabet $\alpha, \beta, \ldots$.

Derivations A *one-step derivation* (written as $\Rightarrow_G$) is the derivation of a non-terminal by using a production rule of the grammar and is for a context-free grammar $G = (N, \Sigma, P, S)$ a relation $\Rightarrow_G \subseteq (N \cup \Sigma)^* \times (N \cup \Sigma)^*$ defined by $\alpha A \beta \Rightarrow_G \alpha\gamma\beta$ iff $(A, \gamma) \in P$. A *derivation* for a word w is a finite sequence of one-step derivations starting with the start symbol S and ending with the word w itself ($S \Rightarrow_G \gamma_1 \Rightarrow_G \gamma_2 \Rightarrow_G \gamma_3 \Rightarrow_G \ldots \Rightarrow_G w$, where $\gamma_i \in (N \cup \Sigma)^*$). For a derivation, we write in short: $S \Rightarrow_G^* w$, where $\Rightarrow_G^*$ is the reflexive and transitive closure of $\Rightarrow_G$. The *language* defined by a grammar is the set of all words for which there is a derivation in the grammar: $L(G) = \{w \mid S \Rightarrow_G^* w\}$.

Example 2.3. Let $G_{aexpr} = (\{E, T, F\}, \{+, -, *, /, (,), n\}, P, E)$ be a grammar with P as follows:

$$\begin{aligned} E &\to E + T \mid E - T \mid T \\ T &\to T * F \mid T/F \mid F \\ F &\to (E) \mid n \end{aligned}$$

This simple grammar is an example of a context-free grammar and defines the language $L_{aexpr} = L(G_{aexpr})$ of arithmetic expressions.

2.3 Earley Parser

The *word problem* for context-free languages is the decision problem to decide whether a word is contained in a context-free language or not. The word problem can be computed for a word of length n with the *Earley Parser* [4] in time $\mathcal{O}\left(n^3\right)$ [4].

The Earley Parser is a *parsing algorithm* that uses dynamic programming to parse words belonging to a context-free language. We introduce a slightly simplified version [2] of the Earley Parser, *without a lookahead* because the adaptations of the Earley Parser in this work negate the advantages of the lookahead.

For a word $w = a_0 a_1 \dots a_{n-1}$ and a context-free grammar $G = (N, \Sigma, P, S)$ the Earley Parser decides if w is contained in L or not. To do this, the Earley Parser uses states, called *Earley items*, which describe the current parsing progress. To advance the parsing process, the Earley parser uses the following 3 main operations:

- **Scanner:** Matches the next input symbol against the expected symbols.
- **Predictor:** Expands the possible production rules that might follow.
- **Completer:** Completes a production rule and links it back to previous states.

Earley Items and Sets

Definition 2.1. Let $G = (N, \Sigma, P, S)$ be a context-free grammar, $w \in \Sigma^*$ with $|w| = n$ and $A \to \gamma\alpha \in P$. Then, $[A \to \ \gamma \bullet \alpha, j]$ is called an *Earley item* with $A \in N$, $\bullet \notin (N \cup \Sigma)$, $\gamma, \alpha \in (N \cup \Sigma)^*$ and $0 \leq j \leq n$. If $\alpha = \varepsilon$, we call the item a *final Earley item.*

Each Earley item describes how much of a production rule has already been seen by the parser, where a final item indicates that the whole production rule is seen by the parser. Therefore, the right side of a production rule is separated by a bullet point into the part γ that is already seen and the part α of the rule that still needs to be seen by the parser. In addition, the number j indicates the starting position of the production rule in the word. For example, the Earley item $[E \to \ E + \bullet T, 3]$ indicates that the parser has seen $E+$ of the production rule $E \to E+T$, and the production rule started at position 3 of the parsed word.

Definition 2.2. Let $G = (N, \Sigma, P, S)$ be a context-free grammar, $w \in \Sigma*$ with $|w| = n$. Then, an *Earley set*, written as Q_i with $0 \leq i \leq n$, is a set of Earley items for w and G.

The Earley Parser initialized the first Earley set Q_0 with the *initial Earley item* $[S' \to \ \bullet S, 0]$. Where we add the production rule $S' \to \ S$ to our grammar and make S' to the *new start symbol* of the grammar G.

Earley rules Starting with the initial item, all further items are added to the Earley sets by successively applying the Earley rules defined in the following definition.

Definition 2.3. Let $G = (N, \Sigma, P, S)$ be a context-free grammar, $w \in \Sigma^*$ with $w = a_0 a_1 \dots a_{n-1}$ and $|w| = n$, let Q_i and Q_j be Earley sets with $0 \leq i \leq n$ and $0 \leq j \leq n$, and let $\alpha, \alpha', \gamma, \gamma \in (N \cup \Sigma)^*$, $b \in \Sigma$, $B \in N$. Then the Earley rules defined as follow:

Scanner (S): If $[A \to \alpha \bullet b\gamma', j]$ in Q_i and $a_i = b$, add $[A \to \alpha b \bullet \gamma', j]$ to Q_{i+1}.

Predictor (P): If $[A \to \alpha \bullet B\gamma', j]$ in Q_i, add $[B \to \alpha' \bullet, i]$ to Q_i for all production rules $B \to \alpha'$ in P.

Completer (C): If a final Earley item $[A \to \alpha \bullet, j]$ in Q_i, add $[B \to \alpha' A \bullet \gamma', k]$ to Q_i for all Earley items $[B \to \alpha' \bullet A\gamma', k]$ in Q_j.

The word w is generated by the grammar G if and only if $[S' \to S\bullet, 0]$ is contained in the Earley set $Q_{|w|}$ where S is the start symbol of the grammar. The algorithm of the Earley Parser is presented in Algorithm 1. Note: By using a lookahead as in the original Earley Parser [4] we can reduce the number of Earley items per Earley set by excluding Earley items.

Algorithm 1 EARLEY_PARSER

Require: A context-free grammar $G = (N, \Sigma, P, S)$ and a word $w = a_0, a_1, \ldots a_{n-1}$ of length n
Ensure: If $w \in L(G)$

```
1: Add S' → S to P.
2:
3: Q_i ← ∅ for all 0 ≤ i ≤ n
4: Add [S' → •S, 0] to Q_0
5:
6: for i = 0 to n do
7:
8:     while Q_i ≠ μ(Q_i) do          ▷ Where μ is the application of the rules P and C.
9:         Q_i ← μ(Q_i)
10:    end while
11:
12:    if Q_i = ∅ then
13:        return False
14:    end if
15:
16:    Q_{i+1} ← S(Q_i)               ▷ Scanner rule
17: end for
18:
19: return [S' → S•, 0] ∈ Q_n
```

Example 2.4. In Fig. 1, all Early sets have been calculated for the word $n*(n+n)$ and the grammar G_{aexpr} from Example 2.3. It is easy to verify that the Earley item $[S' \to S\bullet, 0]$ is contained in Q_7 and thus, the word $n*(n+n)$ is contained in the language L_{aexpr}.

2.4 Syntax Trees

To analyze the syntactic structure of a word according to a grammar, it is not sufficient to know whether the word is contained in the language of the grammar or not. Instead, we are interested in the structure of the derivations, especially which nonterminal symbol was derived/replaced by which production rule.

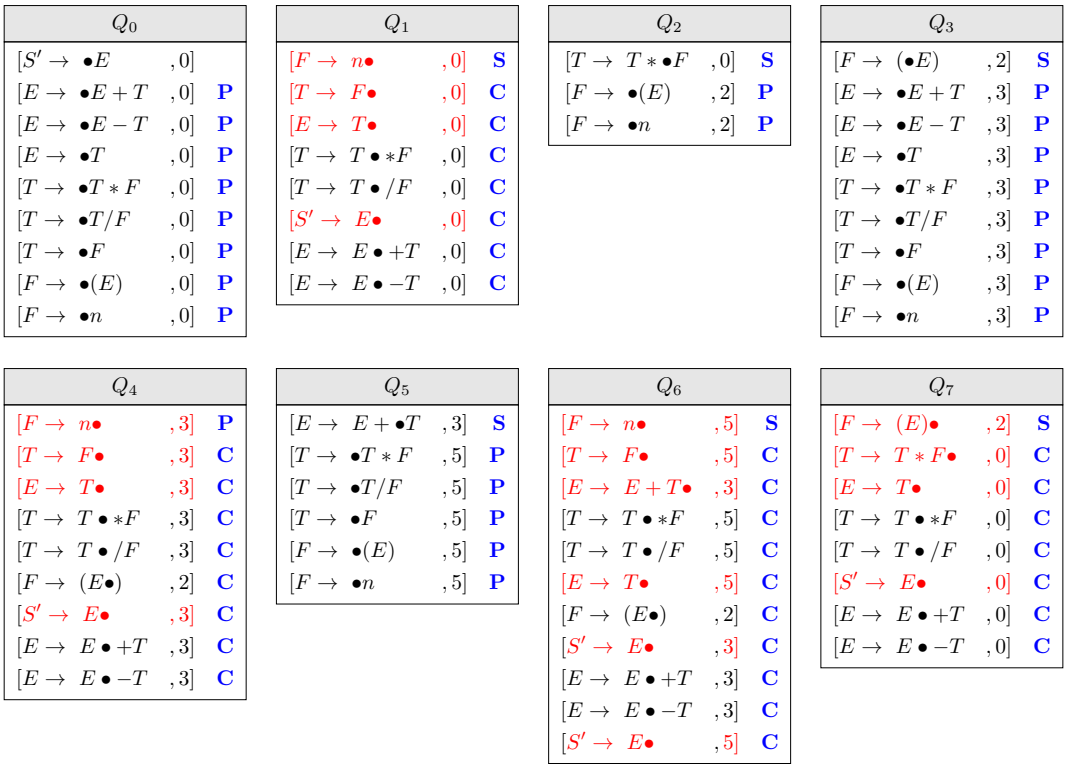


Figure 1: The Earley sets for the word $n * (n + n)$ and the context-free grammar G_{aexpr}. The red color of an Earley item indicates that this Earley item is a final item, and the blue letters next to the items indicate the rule by which the item is generated.

A *syntax tree* represents this syntactic structure of a word according to grammar as a tree. The idea is that each node represents a terminal or nonterminal of the derivation. The one-step derivation of a nonterminal symbol regarding a production rule is represented by the fact that the corresponding node has all symbols (terminals and nonterminals) as children, ordered from left to right by their occurrence on the right-hand side of the used production rule. Since terminal symbols cannot be derived further, these nodes form the *leaves* of the syntax tree and represent the derived word read from left to right.

Definition 2.4. (cf. [10]) Let $G = (N, \Sigma, P, S)$ be a context-free grammar, $w \in \Sigma^*$. A *syntax tree* for w and G is a $(N \cup \Sigma \cup \{\varepsilon\})$-node-labelled ordered tree with the following properties:

- The *root* node is labeled with the start symbol (S).
- The *leaves* of the tree are labeled with terminal symbols or the empty string ($\Sigma \cup \{\varepsilon\}$).
- For every inner node $A \in N$, if its children nodes are labeled $x_0 \dots x_n$, then $A \to x_0 \dots x_n \in P$.

A syntax tree for w regarding G is one whose leaf front reads from left to right, resulting in the word w (possibly with interspersed ε).

Ambiguity We say that a grammar is *ambiguous* if there is a word in the language of the grammar that has *more than one* different syntax trees, and *disambiguous* if for every word in the language *exactly one* syntax tree exits. Since different syntax trees for a word represent different derivations of the word.

Constructing Syntax Trees with the Earley Parser If we have already determined with the Earley Parser (Sec. 2.3) for some word w and some context-free grammar $G = (N, \Sigma, P, S)$ that the word is contained in the language of the grammar. Then the calculated Earley sets can be used to derive a syntax tree for the word. To do this, we use a recursive backward search. The search starts with the final Earley item ($[S' \to \bullet, 0]$) contained in the last Earley set $Q_{|w|}$ with the aim to reach the initial item ($[S' \to \bullet S, 0]$) in the first Earley set (Q_0) by a backward applying of the rules scanner, predictor, and completer [19]. The syntax tree is gradually built up by the items used in the search. Note that to compute all syntax trees for an ambiguous grammar, the recursion step must be performed for all possible backward searches, where each of them leads to a different syntax tree.

Example 2.5. The syntax tree for the word $n * (n + n)$ derived from the Earley items from Figure 1 for the disambiguous grammar G_{aexpr} from Example 2.3 is shown in Figure 2.

An example of an ambiguous grammar is the following, which describes all boolean expressions that use only the *and* ($\wedge$) and *or* ($\vee$) operators and is defined as $G_{bexpr} = (\{B\}, \{\wedge, \vee, \mathit{True}, \mathit{False}\}, P, B)$ with P as follows:

$$B \to \ B \wedge B \mid B \vee B \mid \mathit{True} \mid \mathit{False}$$

Figure 3 shows that for the boolean expression $\mathit{True} \vee \mathit{False} \wedge \mathit{False}$ there are two different syntax trees for the grammar G_{bexpr} exists and therefore the grammar is ambiguous. We can also see that, with a bottom-up evaluation of the subexpressions in the syntax trees, the left syntax tree evaluates to *True* and the right one evaluates to *False*. This shows the importance of syntax trees.

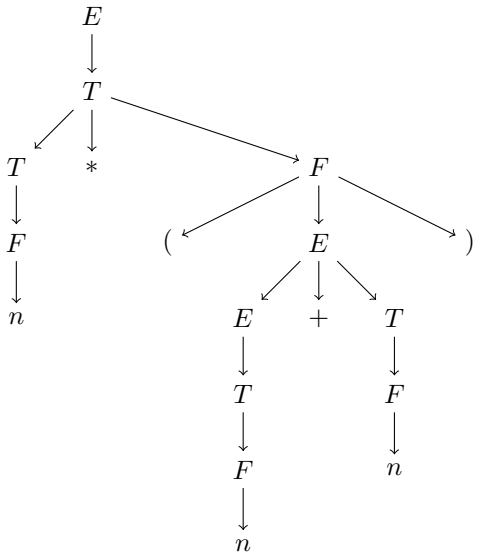


Figure 2: The syntax tree for the word $n * (n + n)$ and the context-free grammar G_{aexpr} from Example 2.3.

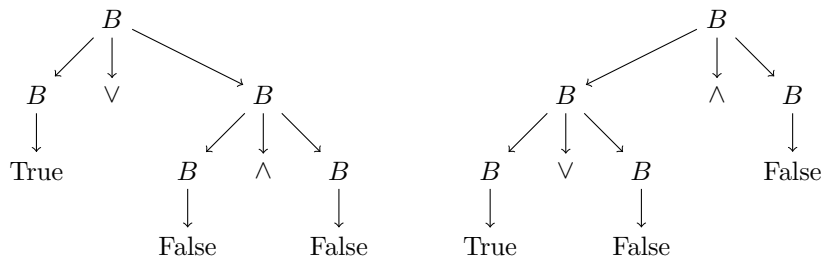


Figure 3: There are two different syntax trees for the boolean expression $True \vee False \wedge False$ and the grammar G_{bexpr} from Example 2.5.

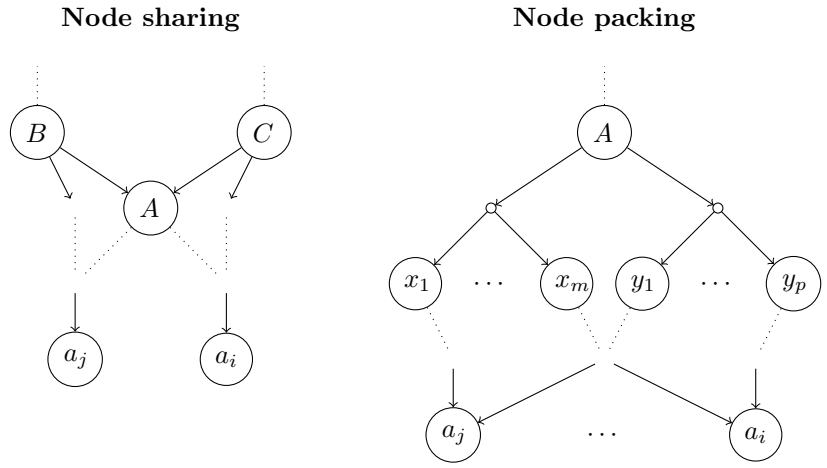


Figure 4: (cf. [18, Sec. 8]). The principle of sharing and packing nodes in a shared packed parse forest.

2.5 SPPF: Shared Packed Parse Forest

To reduce the space required to store all syntax trees for a word, and to store a possible infinite number of syntax trees, we need a special data structure that uses the structure of the syntax trees to store them efficiently. Such a data structure is a *shared packed parse forest* (SPPF)[14, 19] that represents all syntax trees in only one directed graph. The name shared packed parse forest can be explained, besides the fact that many trees form a forest, as follows:

- Nodes of different syntax trees that have the *same subtree* are *shared* (named *symbol* or *intermediate nodes*).
- Nodes that represent the *same subword* but *different one-step derivations* by different subtrees from the same nonterminal are *packed* (named *packed nodes*).

A graphical illustration of the idea of sharing and packing nodes is shown in Figure 4.

Definition 2.5. Let $G = (N, \Sigma, P, S)$ be a context-free grammar, $w \in \Sigma^*$ with $w = a_0 a_1 \dots a_{n-1}$ and $|w| = n$. A *shared packed parse forest* (SPPF) for w and G is a labelled directed graph (V, E), where V is the set of vertices and E is the set of edges. The set of vertices V of an SPPF is a subset of the 4 disjoint set of (possible) nodes: *leaf nodes*[3] (written as V_L), *symbol nodes* (written as V_S), *intermediate nodes* (written as V_i) and *packed*

[3]In most other CSPPF definitions, leaf nodes are also part of the symbol nodes and are not considered separately.

nodes[4] (written as V_p), defined as follows:

$$
\begin{aligned}
V \subseteq\ & (V_L \cup V_S \cup V_I \cup V_P) \\
V_L =\ & \{(a_i, i, i+1) \mid 0 \leq i < n-1\} \cup \{(\varepsilon, i, i) \mid 0 \leq i < n\} \\
V_S =\ & \{(A, i, j) \mid A \in N \text{ and } 0 \leq i \leq j < n\} \\
V_I =\ & \{(A \to \alpha \bullet \beta, i, j) \mid A \in N, \alpha, \beta \in (N \cup \Sigma)^+, \text{ such that } A \to \alpha\beta \in P \\
& \quad \text{and } 0 \leq i \leq j < n\} \\
V_P =\ & \{(A \to \alpha \bullet \beta, i, k, j) \mid A \in N, \alpha \in (N \cup \Sigma)^+, \beta \in (N \cup \Sigma)^*, \text{ such that} \\
& \quad A \to \alpha\beta \in P \text{ and } 0 \leq i \leq k \leq j < n\}
\end{aligned}
$$

We call the pair (i, j) contained in symbol nodes, packed nodes and intermediate nodes and the pair $(i, i+1)$ in the leaf nodes the *extent* of this nodes. The natural number k of packed nodes is called the *pivot*.

The set of edges E of an SPPF is a subset of the 6 disjoint sets of (possible) edges, where each of them describes edges from a particular node type to a particular node type. The different sets of edges are: edges from symbol nodes to packed nodes (written as E_{SP}); intermediate nodes to packed nodes (written as E_{IP}); packed nodes to symbol nodes (written as E_{PSR} and E_{PSL}); packed nodes to leaf nodes (written as E_{PLR}) and packed nodes to intermediate nodes (written as E_{PIL}).

$$
\begin{aligned}
E \subseteq\ & (E_{SP} \cup E_{IP} \cup E_{PSR} \cup E_{PLR} \cup E_{PSL} \cup E_{PLL} \cup E_{PIL}) \\
E_{SP} =\ & \{(u, v) \mid u = (A, i, j) \in V_S, v = (A \to \alpha \bullet \varepsilon, i, k, j) \in V_P \text{ such that } i \leq k \leq j\} \\
E_{IP} =\ & \{(u, v) \mid u = (A \to \alpha \bullet \beta, i, j) \in V_S, v = (A \to \alpha \bullet \beta, i, k, j) \in V_P \\
& \quad \text{such that } i \leq k \leq j\} \\
E_{PSR} =\ & \{(u, v) \mid u = (A \to \alpha \bullet \beta, i, k, j) \in V_P, v = (A', k, j) \in V_S, \gamma \in (N \cup \Sigma)^* \\
& \quad \text{such that } \alpha = \gamma A'\} \\
E_{PLR} =\ & \{(u, v) \mid u = (A \to \alpha \bullet \beta, i, k, j) \in V_P, v = (a'_i, i', i'+1) \in V_L, \gamma \in (N \cup \Sigma)^* \\
& \quad \text{such that } \alpha = \gamma a'_i, k = i' \text{ and } j = i'+1\} \\
E_{PSL} =\ & \{(u, v) \mid u = (A \to \alpha \bullet \beta, i, k, j) \in V_P, v = (A', i, k) \in V_S, \gamma \in (N \cup \Sigma)^* \\
& \quad \text{such that } \alpha = A'\gamma\} \\
E_{PIL} =\ & \{(u, v) \mid u = (A \to \alpha \bullet \beta, i, k, j) \in V_P, v = (A \to \alpha' \bullet \beta', i, k) \in V_I, \gamma \in (N \cup \Sigma)^* \\
& \quad \text{such that } \alpha = \alpha'\gamma \text{ and } \beta' = \gamma\beta\}
\end{aligned}
$$

After the formal definition of SPPFs, we will consider the meaning of the different types of nodes and their properties.

- **Leaf nodes** (V_L): For each letter a_i of the word $w = a_0 a_1 \dots a_{n-1}$ there is exactly one leaf node describing that this letter is derived. Therefore, unlike all other node types, leaf nodes have no children in an SPPF. Each leaf node has a set of packed nodes as predecessor.
- **Symbol nodes** (V_S): An symbol node (A, i, j) describes that we can derive from the nonterminal A the substring of the word w that starts at position i and ends at position j. This implies that the symbol node $(S, 0, n)$ is the *root node* of the SPPF for some grammar $G = (N, \Sigma, P, S)$ and a word $w \in \Sigma^*$ of length n. We can observe that the successors of symbol nodes are packed nodes, each of them representing an alternative to derive from the nonterminal A the substring of w corresponding to this node.

[4] In other definitions of SPPFs such as [19] or [14], there may be packed nodes with the same label that differ only in their predecessor and successors nodes. Therefore, in contrast to these definitions, we use the natural numbers i, j to uniquely identify the packed node based only on its labels.

- **Intermediate nodes** (V_I): An intermediate node $(A \to \alpha \bullet \beta, i, j)$ describes how we can derive from the sentantial form α the subword of w that starts at position i and ends at position j. In other words, an intermediate node represents how the part α of the production rule $A \to \alpha\beta$ can be derived. Like symbol nodes, intermediate nodes also have as successors a set of packed nodes, each of which represents an alternative to deriving the intermediate node.

- **Packed nodes** (V_P): As already mentioned each packed node describe an *alternative* to deriving a symbol or intermediate node. More detailed a packed node $(A \to \alpha \bullet \beta, i, k, j)$ with $\alpha = \alpha'\gamma$ ($\alpha \in (N \cup \Sigma)^*$, $\gamma \in (N \cup \Sigma)$) represents that the derivation of α is split into the left part α' and the right part γ. Where the derivation of the left part results in the subword of w that start at position i and ends at position k and the derivation of the right part results in the subword of w that start at position k and ends at position j. Both the left and right parts are represented by a successor node, where the left successors are determined by the edges in E_{PSL} and E_{PIL} and the right successors by the edges in E_{PSR} and E_{PLR}. We can observe that the left node is of type intermediate node or symbol node and the right node is of type symbol node or leaf node. Note that in the case $i = k$ the left successor node does not have to exist. Since the packed node is an alternative to derive the part α of the production rule $A \to \alpha\beta$, it follows that the intermediate node $(A \to \alpha \bullet \beta, i, j)$ is the predecessor of the packed node, described by the set of edges E_{IP}. Or in the case that $\beta = \varepsilon$ the symbol node (A, i, j) is the predecessor node, described by the set of edges E_{SP},. Therefore, each packed node has at most one successor and has at most one or two predecessors, where each packed node with the maximum number of predecessor and successors is uniquely described by its successors and its predecessor.

We can observe that symbol and intermediate nodes can only have nodes of type packed node as successors and packed nodes can only have nodes of type symbol, leaf or intermediate node as successors. As these are the only types of edges, this shows that any shared packed parse forest is a *bipartite graph*, where packed nodes represent one bipartition and all other node types represent the other bipartitions. Therefore, for a packed node with the predictor node x and the successor nodes u and v, we also say that the pair u, v, is a *child family* of the node x.

Visualisation In the visualizations of an SPPF, we draw symbol nodes as rectangles with rounded corners, intermediate nodes as rectangles, and packed nodes as circles. For the sake of clarity, we omit the labels of packed nodes in the visualization, since we have already discussed that these labels are easy to determine from the predecessor and successor nodes. In addition, we draw the edges of a packed node to its left child in red and to its right child in blue because, as most of the visualisations in this work contains packed nodes where the right node is not placed to the right of the left node for better readability.

Syntax trees contained in an SPPF All syntax trees contained in an SPPF can be computed by *unfolding* the share packed parse forest. This means that for each symbol node and intermediate node, only *one* of the *alternatives* (packed nodes) is selected. This results in a set of different induced subgraphs of the SPPF. Next, the cycles in the subgraphs must be removed to convert the subgraphs to trees by *unrolling* the *cycles*. Each number of unrolls of a cycle creates a separate tree. Note that trees can be created where the leaf front does not consist entirely of leaf nodes. These trees must be removed because they cannot be converted into a valid syntax tree for the corresponding word. For the sake of simplicity, and since the algorithm considered in Section 2.6 for creating an SPPF does not create such trees, we assume for the sake of simplicity that such trees are not included in the SPPF in the rest of this work. As a final step, the remaining trees must be converted to a

syntactically valid syntax tree by *removing* all *intermediate and packed nodes* and relabeling the symbol nodes. In the removing step, when a node is removed, its children are added to the removed node parent so that the order of the nodes is maintained. Relabelling the symbol nodes is simply removing the extent from the label.

Definition 2.6. The SPPF ψ contains a syntax tree τ (written as: $\tau \ \xi \ \psi$) if and only if:

- There is an induced subgraph τ' of the SPPF ψ that has exactly one successor for each intermediate and symbol node.
- τ' can be transform to τ as follow:
  - *Unwind* the *cycles* as many times as necessary.
  - *Remove* all *non-symbol nodes* which not a *edit node*. When a node is removed, its children are appended to the node parent without violating the order of the nodes.
  - *Relabel* the symbol nodes by removing the extent from the label.

Let $C(\psi)$ the set of all syntax trees contained in the SPPF ψ.

$$C(\psi) = \{\tau \mid \tau \ \xi \ \psi\}$$

Properties of SPPF Some properties of a shared packed parse forest are:

- If an SPPF has at least one cycle, then it stores an infinite number of trees. Because the cycle represents the repeated application of a cycle of the same production rule, so that each number of unrolls creates a separate tree.
- We say that a SPPF is *disambiguous* regarding a word and a grammar if every symbol and intermediate node has at most one successor. This means that only one alternative can be selected for each packed and intermediate node in order to derive it.
- If there is at least one symbol or intermediate node with more than one packed node children (alternatives), then we say the shared packed parse forest is *ambiguous* regarding a word and a grammar.

Example 2.6. The SPPF for our running example with the word $n * (n + n)$ and the grammar G_{aexpr} from Example 2.3 is shown in Figure 6. We can easily verify that the SPPF is ambiguous and contains only the syntax tree of Figure 2.

Another example of an SPPF for the boolean expression $\mathit{True} \vee \mathit{False} \wedge \mathit{False}$ and the grammar G_{bexpr} defined in Example 2.5 is shown in Figure 5. We can observe that the SPPF is ambiguous because for the symbol node $(B, 0, 5)$ there are two different alternatives of packed nodes that can be chosen. This implies that both syntax trees shown in Figure 3 are contained in that SPPF.

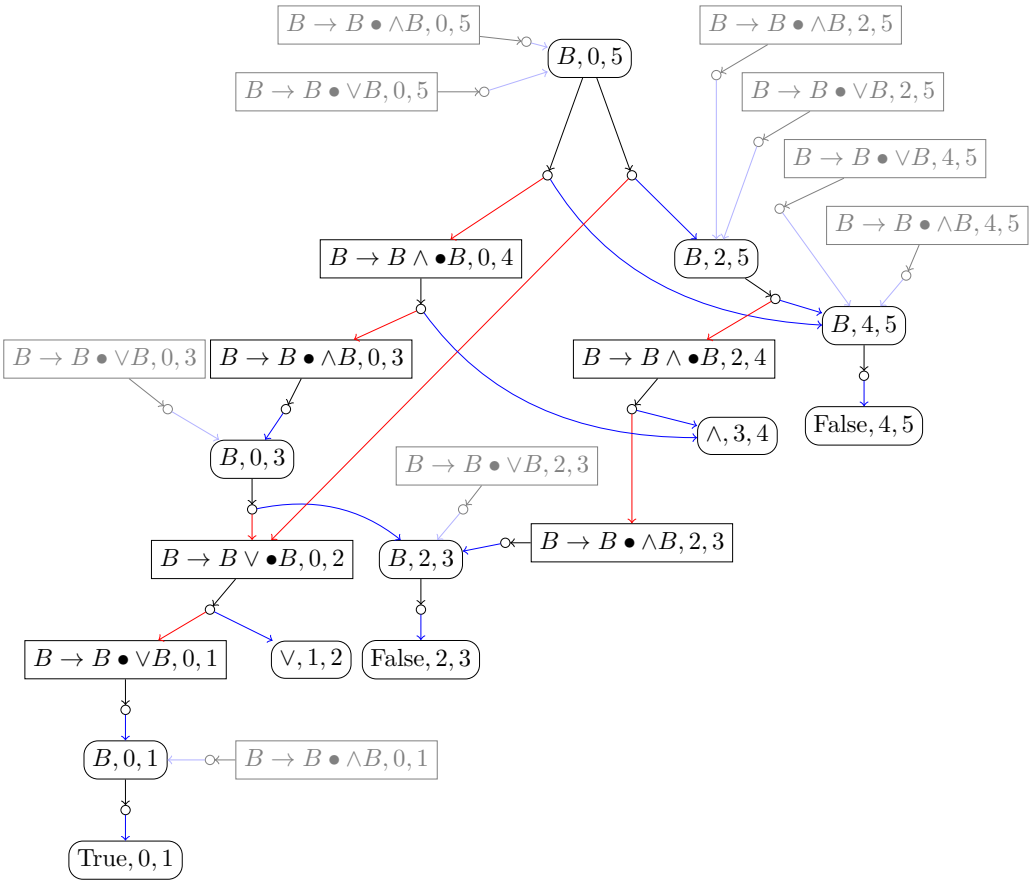


Figure 5: The shared packed parse forest for the boolean expression *True* $\vee$ *False* $\wedge$ *False* and the grammar G_{bexpr}. The gray nodes are created during the construction of the SPPF (Algorithm 12), but are not contained in any syntax tree for the boolean expression *True* $\vee$ *False* $\wedge$ *False* and the grammar G_{bexpr}.

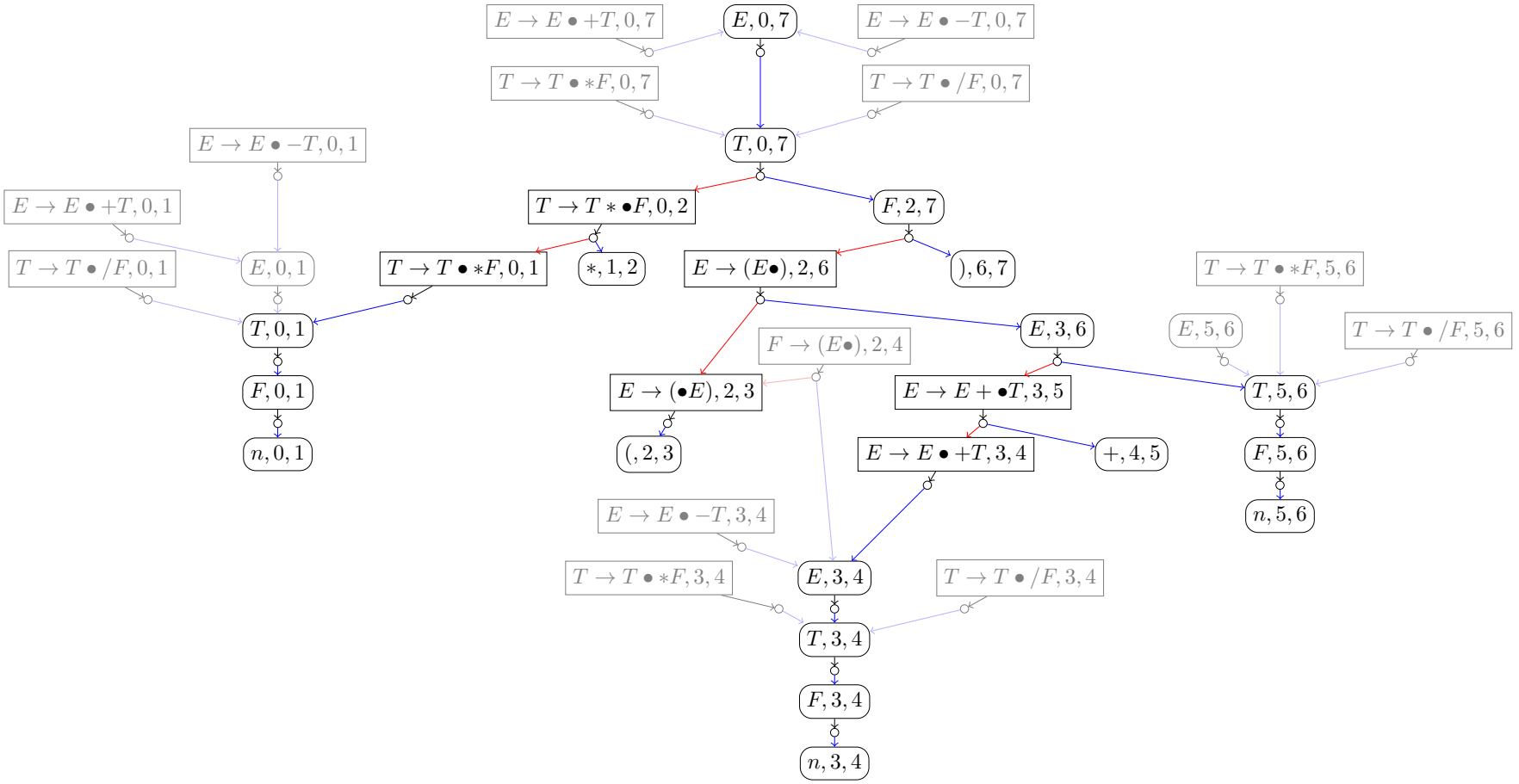


Figure 6: The shared packed parse forest for the word $n * (n + n)$ and the grammar G_{aexpr}. The gray nodes are created during the construction of the SPPF (Algorithm 12), but are not contained in any syntax tree for the word $n * (n + n)$ and the grammar G_{aexpr}.

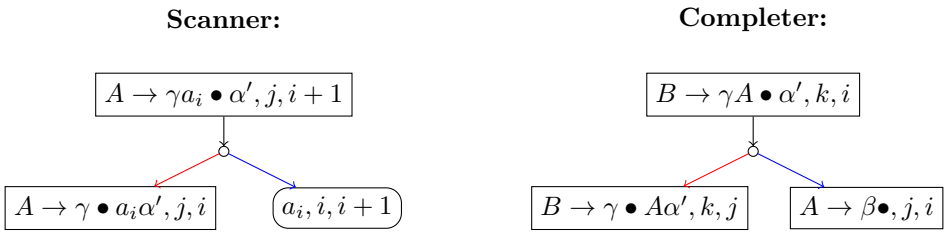


Figure 7: A graphical representation of the build-up process of the generalised Earley Parser for the Scanner rule (left) and Completer rule (right).

2.6 Generalised Earley Parser

To compute a shared packed parse forest for a context-free grammar and a word, a general version of the Earley Parser can be used [15]. We know from Section 2.3 that an Earley item is of the form $[A \to \ \gamma \bullet \alpha, j]$ (Definition 2.1). The existence of such an item in the Earley set Q_i describes that we can derive from the sentential form γ the subword that starts at position j and ends at position i, and that α remains to be seen by the parser. The only thing that is not clear for the Earley item is how we can derive γ. We can additionally observe that an intermediate node of the form $(A \to \gamma \bullet \alpha, j, i)$ describes the same as the Earley item above, with the additional information: in which different ways γ can be derived to the corresponding substring by the subtree below that intermediate node.

This implies the basic idea of the generalised Earley Parser to compute an SPPF by additionally storing for each Earley item the corresponding node in the SPPF. Therefore, we extend the Definition 2.1 of an Earley item by a node of the constructed SPPF.

Definition 2.7. Let $G = (N, \Sigma, P, S)$ be a context-free grammar, $w \in \Sigma^*$ with $|w| = n$ and $A \to \gamma\alpha \in P$. Then $[A \to \gamma \bullet \alpha, j, u]$ is called an *Earley item with node* with $A \in N$, $\gamma, \alpha \in (N \cup \Sigma)^*$, $\bullet \notin (N \cup \Sigma)$, $0 \leq j \leq n$ and u is a *symbol node*, *intermediate node* of the SPPF for G and w (Definition 2.5), or *null*.

In the introducing example, we assigned the Earley item ($[A \to \gamma \bullet \alpha, j]$) to an intermediate node. However, this is only possible if γ and α are not empty ($\gamma \neq \varepsilon$ and $\alpha \neq \varepsilon$). Because in the case that α is the empty word ($\alpha = \varepsilon$), we have to assign the Earley item to a symbol node. In the other case, if γ is the empty word ($\gamma = \varepsilon$), we assign the item to the dummy node *null* that doesn't store in the SPPF. We can do this because nothing can be derived from ε. Note: These are Earley items created by the predictor rule.

Building a SPPF The described assignment of Earley items to nodes in the SPPF implies that by extending Algorithm 1 to compute an SPPF by applying the *predictor* rule, the newly created Earley item will be associated with the dummy node *null*. For the *scanner* and the *completer* rule, Figure 7 introduces the process of building the SPPF by using one of these rules.

Applying the *scanner* rule to an Earley item assigned to the lower left intermediate node. We need to create a new leaf node for the scanned terminal a_i of the form $(a_i, i, i+1)$ if it doesn't already exist in the computed SPPF. If the node representing the new Earley item created by the scanner rule, the top node in our graphic, already exists, then add the other two nodes as further children family. If the node doesn't exist, create it and add the other two nodes as its first child family. Remember that adding a child family to a node means that we add a new packed node as a child of that node, and the packed node successor are the child family of this node.

The effect of applying the *completer* rule is that the nodes to which the existing Earley items are assigned together form a new child family of the node to which the newly created Earley item is assigned.

A detailed algorithm and some optimizations that are not presented here for the generalized Earley Parser version [15] can be found in the appendix of this work (App. A).

Example 2.7. In Example 2.6 we have already presented the shared packed parse forest for the word $n * (n + n)$ and the grammar G_{aexpr} (Fig 6), as for the boolean expression $\mathit{True} \vee \mathit{False} \wedge \mathit{False}$ and the grammar G_{bexpr} (Fig. 5). Both SPPFs can be computed with the generalized Earley Parser presented in this section. For the word $n * (n + n)$ and the grammar G_{aexpr} the computed Earley items with the associated node of the SPPF are outscored to the appendix (App. B, Fig. 22). For the second example, it is left to the reader to compute the Earley items as the assignments of the items to the nodes of the SPPF with the generalized Earley Parser.

Note: In Figure 22 there are no Earley items with the new start symbol S'. This is because we would rather not add nonterminals to a syntax tree that are not in the used grammar (see App. A).

2.7 Disambiguation Filter

We have already seen in Figure 3 that for the ambiguous grammar G_{bexpr} from Example 2.5 two syntax trees exists that represent the same boolean expression. One of this syntax trees evaluates to *True* and the other evaluates to *False* by a bottom-up evaluation of the subexpressions. So we have a contradiction in the evaluation of the boolean expression, and so we cannot, for example, use this boolean expression in an if-statement of an algorithm. Disambiguation filter can be used to select a certain syntax tree from such an ambiguous shared packed parse forest.

Disambiguation rules *Disambiguation rules* [7] define which of the possible syntax trees should be returned by the parser. There are several possible rules that cause exactly one syntax tree to be selected, such as a preference for the longest match, type-dependent rules, precedence relations between operators, etc.

Figure 8 shows in which phases of the parse process disambiguation rules can be implemented.

1. For a given context-free grammar, a disambiguation rule can be implemented by *transforming* the grammar. An example of this is the elimination of left recursion [12] in left recursive grammars.

2. Another way to implement disambiguation rules is to implement the rules into the parser (*parse generator*). The error-correcting parser from Section 2.9 is an example of this, where we adapt the Earley Parser to compute the correction with the minimum number of operations by saving only the minimum number of edit operations for each Earley item.

3. The third way is that we compute for the word a set of syntax trees, typically represented as a shared packed parse forest (Sec. 2.5), and *filter out* one syntax tree by the given disambiguation rules. For example, the evaluation order of operators can be determined by filtering out only those syntax trees from the SPPF in which operators with a higher precedence are above operators with a lower precedence (see Example 2.8).

For reasons of efficiency, disambiguation rules should be implemented as early as possible in the parse process.

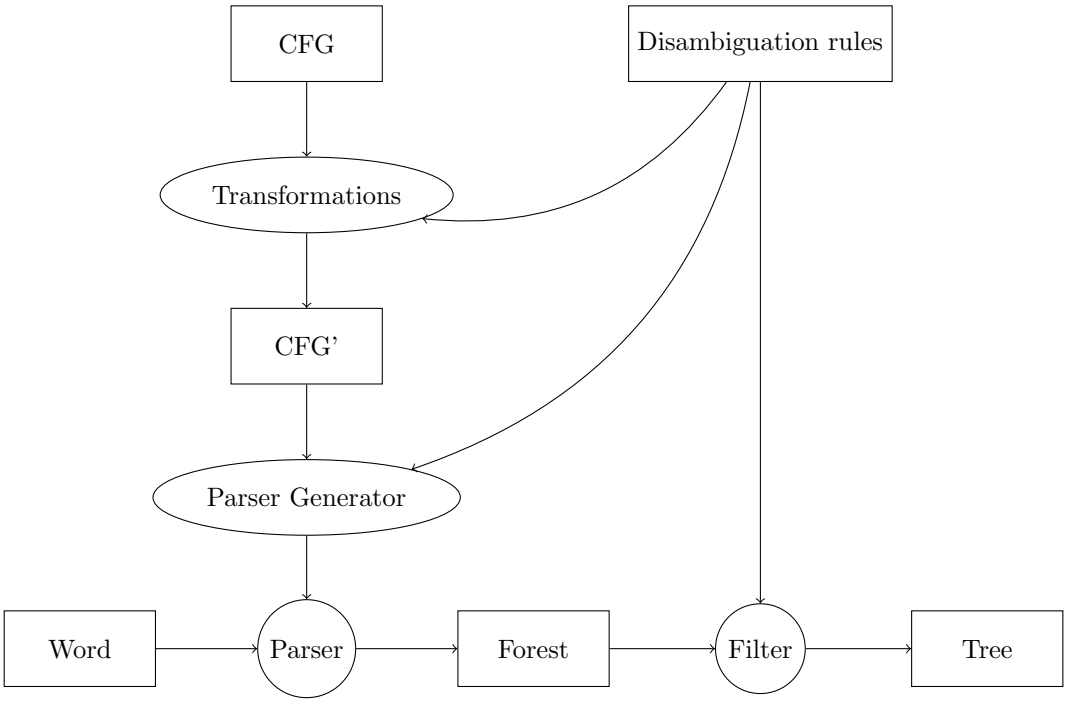


Figure 8: (cf. [7, Fig. 1]) Phases in parsing with ambiguous grammars.

Filter Although the first two methods (transformations and parser generator) are preferable for reasons of efficiency, many disambiguation rules are implemented by filters because most rules are easiest to define as filters, and often could not have been implemented earlier.

Definition 2.8. (cf. [7, Def. 4.1]) Let $G = (N, \Sigma, P, S)$ be a context-free grammar, $w \in \Sigma^*$, let Γ be the power set of all *syntax trees* corresponding to G and w. A *disambiguation filter* is a function f that maps a set of syntax trees X ($X \in \Gamma$) to a subset of X ($Y \subseteq X$).

$$f : \Gamma \to \Gamma,\ X \mapsto Y \text{ so that } X \supseteq Y$$

A filter f is *completely disambiguating* when $|Y| \leq 1$ for all $w \in \Sigma^*$ and the filter is f *completely* when $|Y| = 1$ for all $w \in L(G)$.

Therefore, a disambiguation rule for a context-free grammar can be realized by a function that is a complete filter. One of the most common ways to define such a filter function is by a relation over the set of syntax trees.

Definition 2.9. (cf. [7, Def. 4.12]) Let $G = (N, \Sigma, P, S)$ be a context-free grammar, $w \in \Sigma^*$, let S the set of all Syntax trees corresponding to G and w, let $\Gamma = \mathcal{P}(S)$ be the power set of all *syntax trees* and let $\prec \subseteq S \times S$ be a relation. Then the disambiguation filter $f^{\prec}$ generated by the relation $\prec$ is defined as follows:

$$f^{\prec}(\psi) = \{\tau\ \xi\ \psi \mid \neg\exists \tau'\ \xi\ \psi : \tau' \prec \tau\}$$

where ψ is a subset of the set of all syntax trees $\Gamma(\psi \in \Gamma)$.

Note for each reflexive and symmetric relation $\prec$, the filter $f^{\prec}$ rejects all syntax trees.

Therefore, the most useful relations are strict partial orders (see Sec. 2.1).

Example 2.8. An example of a filter is the one that rejects all syntax trees that represent a boolean expression that disregard the usual precedence of boolean operators (1. $\wedge$, 2. $\vee$). Next to possibly simpler definitions, the filter function $f^{\prec}$ can be generated by the following relation over all syntax trees S, where $|\tau|_D$ is the number of times that a syntax tree τ disregard the precedence of the operations and $<$ is the usual smaller relation on natural numbers.

$$\prec = \{(a, b) \in S \times S \mid |a|_D < |b|_D\}$$

Thus, after applying the filter $f^{\prec}$, only syntax trees that violate the sequence 0 times are retained. Let ψ be the set of all syntax trees defined by the shared packed parse forest of Figure 5, then $f^{\prec}(\psi)$ results in the set of syntax trees consisting only of the right one of Figure 3. This shows that this filter can be used to solve the introductory problem at the beginning of this section, in which we do not know how to evaluate the boolean expression due to several syntax trees.

2.8 Corrections

So far, we have already considered the case where a word is contained in a language. But what should a parser do if the word is not contained in the language? In this case, we can either reject the incorrect word (this solves the word problem as in the presented Earley Parser (Alg. 12)) or we can use a *correction* [1, 5] to correct the word by *deleting characters*, *inserting new letters* at any position or *replacing* a *character* by *another letter*, so that the resulting/*corrected word* is contained in the language.

Edit operations The operations of *replacing* (written as $b/_i c$), *inserting* (written as $b{\uparrow_i}$) or *deleting* (written as $b{\downarrow_i}$) a character on a word (where i is the position in the word) are called *edit operations*.

Definition 2.10. (cf. [5, Def. 3]) Let Σ be some alphabet and $\perp \notin \Sigma$, then an application of an *edit operation* (written as τ) is a mapping $\Sigma^* \cup \{\perp\} \to \Sigma^* \cup \{\perp\}$ such that for an word $w = a_0 a_1 \dots a_{n-1}$ τ is defined as follows:

$$\tau\left(a_0 \dots a_{n-1}\right) = \begin{cases} a_0 \dots a_{i-1} b a_i \dots a_{n-1}, & \text{if } \tau = b{\uparrow_i} \text{ and } 0 \leq i \leq n; \\ a_0 \dots a_{i-1} a_{i+1} \dots a_{n-1}, & \text{if } \tau = b{\downarrow_i} \text{ and } 0 \leq i < n \text{ and } a_i = b; \\ a_0 \dots a_{i-1} c a_{i+1} \dots a_{n-1}, & \text{if } \tau = b/_i c \text{ and } 0 \leq i < n \text{ and } a_i = b; \\ \perp, & \text{otherwise} \end{cases}$$

where $b, c \in \Sigma$ and $\perp$ is the undefined value that will be used if we try to perform an incorrect edit operation.

For example, the edit operation $b/_0 c$ on the word db is incorrect because there is a d at the first position of the word, but we want to replace a b with a c at the first position of the word.

Corrections A *correction* is the application of possibly more than one edit operation, and formalizes a more complex rewriting of a word than a single edit operation. The empty correction is denoted by ().

Definition 2.11. (cf. [5, Def. 4]) A *correction* (written as ρ) is a (possibly empty) sequence $\rho = (\tau_0, \tau_1, \ldots, \tau_m)$ of edit operations. Its application is defined as the application of all edit operations in order of sequence from left to right.

$$\rho(w) := \tau_m \left(\ldots \tau_1 \left(\tau_0(w) \right) \right)$$

We say that a correction p is *defined* regarding $w \in \Sigma^*$ iff $\rho(w) \neq \bot$.

Example 2.9. For the word $w = ++nn$ and the context-free language L_{aexpr} from Example 2.3 we can observe that $w \notin L_{aexpr}$. But with the following correction:

$$\rho = [n\uparrow_0, (\uparrow_2, +\downarrow_3, n\uparrow_3,)\uparrow_4, +\uparrow_5, n/_6(,)\uparrow_8]$$

which are applied to the word w as follows:

$$\begin{aligned}
n\uparrow_0 \; [++nn] &= n++nn \\
(\uparrow_2 \; [n++nn] &= n+(+nn \\
+\downarrow_3 \; [n+(+nn] &= n+(nn \\
n\uparrow_3 \; [n+(nn] &= n+(nnn \\
)\uparrow_4 \; [n+(nnn] &= n+(n)nn \\
+\uparrow_5 \; [n+(n)nn] &= n+(n)+nn \\
n/_6([n+(n)+nn] &= n+(n)+(n \\
)\uparrow_8 \; [n+(n)+(n] &= n+(n)+(n)
\end{aligned}$$

the word w can be corrected to the target word $\rho(w) = n+(n)+(n)$, which is contained in the language L_{aexpr}. Note: To avoid confusion with the alphabet symbols (and), we use square brackets instead of round brackets for the corrections and the function brackets in this example.

Similar Corrections Observe, that the correction $(b\downarrow_1, a\uparrow_3)$ has the same effect as the correction $(a\uparrow_4, b\downarrow_1)$ when both applied on a word for that both are defined.

Definition 2.12. (cf. [5]) Let be Σ be an alphabet and let $\rho = (\tau, \omega)$, $\rho' = (\omega', \tau')$ be pairs of edit operations. We call this pair *immediately similar* if and only if ρ' results from ρ by swapping the edit operations and adjusting the indices such that for all words $w \in \Sigma^*$ it applies: $\rho(w) = \rho'(w)$.

Table 1 contains all pairs of immediately similar pairs of edit operations.

Definition 2.13. (cf. [5, Def. 11 and Def. 12]) Let ρ be a correction, let (τ, ω) be successive pair of edit operation in ρ and let (ω', τ') be immediately similar pair to (τ, ω). Then the *one-step reordering* of ρ is defined as the replacement of (τ, ω) by (ω', τ') (denoted as: $(\tau, \omega) \rightsquigarrow (\omega', \tau')$). The *reordering of a correction* is defined as the reflexive, symmetric, and transitive closure of the one-step reordering.

Definition 2.14. (cf. [5]) A correction ρ *is similar* to a correction ρ' (written as $\rho \simeq \rho'$) of the *same length* if we can reorder ρ so that we get the correction ρ'. $[\rho]_\simeq$ is the equivalence class of all corrections similar to ρ.

$$[\rho]_\simeq = \{\rho' \mid \rho' \simeq \rho\}$$

	$j < i$	$j = i$	$j > i$
$(b{\uparrow_i}, c{\uparrow_j})$	$(c{\uparrow_j}, b{\uparrow_{i+1}})$		$(c{\uparrow_{j-1}}, b{\uparrow_i})$
$(b{\uparrow_i}, c{\downarrow_j})$	$(c{\downarrow_j}, b{\uparrow_{i-1}})$	$\emptyset$	$(c{\downarrow_{j-1}}, b{\uparrow_i})$
$(b{\uparrow_i}, c/_j d)$	$(c/_j d, b{\uparrow_i})$	$\emptyset$	$(c/_{j-1} d, b{\uparrow_i})$
$(b{\downarrow_i}, c{\uparrow_j})$	$(c{\uparrow_j}, b{\downarrow_{i+1}})$		$(c{\uparrow_{j+1}}, b{\downarrow_i})$
$(b{\downarrow_i}, c{\downarrow_j})$	$(c{\downarrow_j}, b{\downarrow_{i-1}})$	$(c{\downarrow_{j+1}}, b{\downarrow_i})$	
$(b{\downarrow_i}, c/_j d)$	$(c/_j d, b{\downarrow_i})$	$(c/_{i+1} d, b{\downarrow_i})$	$(c/_{j+1} d, b{\downarrow_i})$
$(b/_i c, c{\uparrow_j})$	$(c{\uparrow_j}, b/_{i+1} c)$		$(c{\uparrow_j}, b/_i c)$
$(b/_i c, c{\downarrow_j})$	$(c{\downarrow_j}, b/_{i-1} c)$	$\emptyset$	$(b/_i c, c{\downarrow_j})$
$(b/_i c, c/_j d)$	$(c/_j d, b/_i c)$	$\emptyset$	$(c/_j d, b/_i c)$

Table 1: Pairs of immediately similar pairs of edit operations.

τ \ ω	$b{\uparrow_i}$	$c{\uparrow_i}$	$c{\uparrow_{i+1}}$	$b{\downarrow_{i-1}}$	$b{\downarrow_i}$	$b{\downarrow_{i+1}}$	$b/_i d$
$b{\uparrow_i}$	$(b{\uparrow_i}, b{\uparrow_i})$	$(b{\uparrow_i}, c{\uparrow_i})$	$(b{\uparrow_i}, c{\uparrow_{i+1}})$	$(b{\uparrow_i}, b{\downarrow_{i-1}})$	$()$	$()$	$(d{\uparrow_i})$
$b{\downarrow_i}$	$()$	$(b/_i c)$	$(b{\downarrow_i}, c{\uparrow_{i+1}})$	$(b{\downarrow_i}, b{\downarrow_{i-1}})$	$(b{\downarrow_i}, b{\downarrow_i})$	$(b{\downarrow_i}, b{\downarrow_{i+1}})$	$(b{\downarrow_i}, b/_i d)$
$c/_i b$	$(c/_i b, b{\uparrow_i})$	$(b{\uparrow_{i+1}})$	$(b{\uparrow_i})$	$(c{\downarrow_i})$	$(c{\downarrow_i})$	$(c{\downarrow_i})$	$(c/_i d)$

Table 2: Applying the one-step simplification to all pairs (τ, ω), where only the cells with a blue background are a simplifications, all other cells are not.

Note that by the definition of similar corrections it applies for two similar corrections ρ, ρ' for all words $w \in \Sigma^*$ over some alphabet Σ that $\rho(w) = \rho'(w)$.

Simplified Corrections There are corrections that are unnecessarily complex; for example, the correction $(1{\uparrow_b}, 1{\downarrow_b})$ can be simplified to $()$ if the correction is defined. This leads us to introduce the *one-step simplification* operation. This is the simplification of a pair of edit operations $\rho = (\tau, \omega)$ to $\rho' = (\tau')$ or $\rho' = ()$ such that $\rho(w) = \rho'(w)$. All one-step simplifications are shown in Table 2 as cells with a blue background. For all other combinations (τ, ω) the one-step simplification does not lead to a simplification.

Definition 2.15. (cf. [5, Def. 7 and Def. 8]) Let ρ, ρ' be two corrections. The correction ρ ($\rho = (\tau_0, \ldots, \tau_{i-1}, \tau_i, \tau_{i+1}, \tau_{i+2}, \ldots, \tau_m)$) can be simplified to ρ' (written as $\rho \triangleright \rho'$) if they are a pair of edit operations (τ_i, τ_{i+1}) , with $0 \leq i \leq m-1$, in ρ which can be simplified according Table 2 to

- $()$ then ρ' has the form $(\tau_0, \ldots, \tau_{i-1}, \ldots, \tau_m)$, or
- (τ') then ρ' has the form $(\tau_0, \ldots, \tau_{i-1}, \tau', \tau_{i+2}, \ldots, \tau_m)$.

We called this a *one-step simplification*. A *simplification* is the reflexive and transitive closure of the one-step simplification.

Note that the order of the simplification steps has an effect on the resulting simplified correction. For example let $w = a$ and $\rho = (a\uparrow_0, a\downarrow_1, a\uparrow_1)$. Where we can simplify ρ by $(b\uparrow_i, b\downarrow_{i+1}) \triangleright ()$ to $\rho' = (a\uparrow_1)$ and we can also simplify ρ by $(b\downarrow_i, b\uparrow_i \triangleright ())$ to $\rho'' = (a\uparrow_0)$. We can observe that ρ' and ρ'' are both simplified and $\rho(w) = \rho'(w)$, but $\rho' \neq \rho''$.

In addition, Definition 2.15 omits the condition that a correction can only be simplified if the correction is also defined. We can do this because a result of [5, Cor. 10] is that if a correction is defined, then any simplification of the correction is also defined.

The definition of simplification and reordering allows us to bring any correction into a form with as few edit operations as possible. By repeatedly applying both operations, we then say that the corrections are *simplified*.

Definition 2.16. (cf. [5]) A correction ρ is *simplified* if we cannot further simplify the correction ρ and cannot further simplify any correction in $[\rho]_{\simeq}$.

As a representative of an equivalence class of *simplified* similar corrections, we choose a correction that is in *normalized* form, defined as follows.

Definition 2.17. (cf. [3]) A *simplified* correction ρ is called *normalised* if and only if

$$\rho = (b/_{i_0}c, \ldots, b/_{i_r}c, d\downarrow_{j_0}, \ldots, d\downarrow_{j_m}, e\uparrow_{h_0}, \ldots, e\uparrow_{h_k})$$

with $r, m, k \in \mathbb{N}$ so that $i_0 > \ldots > i_r$; $j_0 > \ldots > j_m$ and $h_0 \geq \ldots \geq h_k$.

From [3, Proposition 1] we know that for each correction ρ there is a normalised correction ρ' such that ρ can be simplified to ρ'' and $\rho' \in [\rho'']_{\simeq}$. Therefore in the rest of this work we only need to be consider corrections in normalised form.

It should be noted that both reordering and simplification only take into account the syntactic structure of the corrections. Thus, for a word w and two corrections ρ, ρ' in normalized form, with $\rho \notin [\rho']_{\simeq}$, it can be applied that: $\rho(w) = \rho'(w)$. For example, let $w = a$, $\rho = (a\uparrow_0)$ and $\rho' = (a\uparrow_1)$ then the application of both normalised corrections result in the word aa.

Example 2.10. We can observe that the correction in Example 2.9 is not simplified. We can bring this correction into normalised form by the following sequence of simplification and reordering steps.

$$\begin{array}{ll}
[n\uparrow_0, (\uparrow_2, +\downarrow_3, n\uparrow_3,)\uparrow_4, +\uparrow_5, n/_6(,)\uparrow_8] & \\
[n\uparrow_0, (\uparrow_2, +/_3 n,)\uparrow_4, +\uparrow_5, n/_6(,)\uparrow_8] & \mid [+\downarrow_3, n\uparrow_3] \triangleright [+/_3 n] \\
[n\uparrow_0, (\uparrow_2, +/_3 n, +\uparrow_4,)\uparrow_4, n/_6(,)\uparrow_8] & \mid [)\uparrow_4, +\uparrow_5] \rightsquigarrow [+\uparrow_4,)\uparrow_4] \\
[n\uparrow_0, (\uparrow_2, n\uparrow_3,)\uparrow_4, n/_6(,)\uparrow_8] & \mid [+/_3 n, +\uparrow_4] \triangleright [n\uparrow_3] \\
[n\uparrow_0, (\uparrow_2, n\uparrow_3, n/_5(,)\uparrow_4,)\uparrow_8] & \mid [)\uparrow_4, n/_6(] \rightsquigarrow [n/_5(,)\uparrow_4] \\
\vdots & \vdots \\
[n/_2(,)\uparrow_4,)\uparrow_1, n\uparrow_1, (\uparrow_1, n\uparrow_0] &
\end{array}$$

It is easy to verify that the resulting correction ($\rho' = [n/_2(,)\uparrow_4,)\uparrow_1, n\uparrow_1, (\uparrow_1, n\uparrow_0,]$) is in *normalised form* and, also correct the word $++nn$ to $n + (n) + (n)$. To show that the

correction ρ' is *simplified*, we need to prove that no correction in $[\rho']_{\simeq}$ can be further simplified. Since we do not have a one-step simplification between two insert operations, the only possible simplification can be applied between the replace operation and one of the inserts. Therefore, it is sufficient to consider only the following corrections as corrections in $[\rho']_{\simeq}$ that can be possibly simplified:

$$[n/_2(,)\uparrow_4,)\uparrow_1,n\uparrow_1,(\uparrow_1,n\uparrow_0] \qquad [)\uparrow_4,n/_2(,)\uparrow_1,n\uparrow_1,(\uparrow_1,n\uparrow_0]$$
$$[)\uparrow_4,)\uparrow_1,n/_3(,n\uparrow_1,(\uparrow_1,n\uparrow_0] \qquad [)\uparrow_4,)\uparrow_1,n\uparrow_1,n/_4(,(\uparrow_1,n\uparrow_0]$$
$$[)\uparrow_4,)\uparrow_1,n\uparrow_1,(\uparrow_1,n/_5(,n\uparrow_0] \qquad [)\uparrow_4,)\uparrow_1,n\uparrow_1,(\uparrow_1,n\uparrow_0,n/_5(]$$

However, since none of these can be simplified, it is shown that ρ' is simplified. Note: As in Example 2.9, we use square brackets instead of round brackets for the corrections.

2.9 Error-correcting Parsing

An *error-correcting parser*[1, 13] doesn't reject a word w not in a language L, instead it computes a *correction* ρ (Sec. 2.8) such that $\rho(w) \in L$. For instance an error-correcting parser does not computes an arbitrary correction that leads to the target language; it computes the correction with the *minimum number of edit operations*. In other words, it computes the correction so that there is no other correction with a smaller *Levenshtein distance* [8] between the input word and the corrected word. Therefore, the *error correcting problem* for a word and a language is to find this correction in the parsing process and to return the corrected word or its syntax tree.

Error-correcting Parser based on the Earley Parser The error-correcting parser from [1] is based on the Earley Parser. The idea is to store for each Earley item, in addition to the conventional components of an Earley item (Definition 2.1), the number of editing operations k to create that Earley item. However, if there is more than one Earley item of the same form, only the item with the smaller number of operations is stored.

Definition 2.18. Let $G = (N, \Sigma, P, S)$ be a context-free grammar, $w \in \Sigma^*$ with $|w| = n$, $k \in \mathbb{N}_0$ and $A \to \gamma\alpha \in P$. Then $[A \to \gamma \bullet \alpha, j, k]$ is called an *Earley item with number of operations* with $A \in N$, $\gamma, \alpha \in (N \cup \Sigma)^*$ and $0 \leq j \leq n$.

Covering Grammar For *Earley items with number of operations*, the number of edit operations required to create this Earley item is counted. To create Earley items by edit operations, corresponding rules must be added to the Earley Parser for each edit operation (the way we go in Section 4 to calculate all corrections) or, like [1] the given grammar must be converted into a *covering grammar*. A covering grammar adds to the original grammar production rules that cover the cases where we apply an edit operation to the word. We call such a production rule a *terminal error production*. In addition, the original grammar is transformed so that these terminal error production rules can be derived.

Definition 2.19. Let $G = (N, \Sigma, P, S)$ be a context-free grammar. Then $G' = (N', \Sigma, P', S')$ is the *covering grammar* of G, where S' is the new start symbol ($S' \notin N$), $N' = N \cup \{S', H, I\} \cup \{E_a \mid a \in \Sigma\}$ and the production rules P' can be computed from G as follows:

- For each production rule, $A \to \gamma_0, a_0, \gamma_1, a_1, \ldots, a_m, \gamma_m$ in P so that $a_i \in \Sigma$ and $\gamma_i \in N^*$ for all $0 \leq i \leq m$, add the production rule $A \to \gamma_0, A_{a_0}, \gamma_1, A_{a_1}, \ldots, A_{a_m}, \gamma_m$ to P' where each A_{a_i} is a new nonterminal ($A_{a_i} \notin N$).
- For all terminal symbols $a \in \Sigma$, add the following production rules to P':

- $A_a \to a$ ▷ read a
- $A_a \to b$ for all $b \in \Sigma/\{a\}$ ▷ replace a by b
- $A_a \to Ha$ ▷ insert a prefix
- $I \to a$ ▷ insert a
- $E_a \to \varepsilon$ ▷ delete a

where H and I are new nonterminals ($H, I \notin N$). The red colored production rules are *terminal error production* rules.

- Add the following production rules to P'
  - $S' \to S$
  - $S' \to SH$
  - $H \to HI$
  - $H \to I$

The language of each covering grammar ($L(G')$) is Σ^* [1] so that any word over the alphabet can be derived from the covering grammar. The correction used in the derivation can be determined by the terminal error production rules entailed in the derivation of the word. This implies that the error correcting problem can be considered finding for a word the derivation that uses the minimum number of terminal error production rules of a covering grammar.

The usage of a production rule is described in the Earley Parser by a final Earley item. Such a final item can be used to derive a nonterminal in another item by applying the *completer* rule. This implies that to adapt the Earley Parser to count the number of used edit operations, we increment the counter of the item by 1 if we use a final Earley that represents a final error production rule to move the bullet one position to the right. Therefore, we adapt the completer rule as follows:

Completer (C): If a final Earley item $[A \to \alpha\bullet, j, l]$ in Q_i, add $[B \to \alpha' A\bullet\gamma', k, m]$ to Q_i for all Earley items $[B \to \alpha' \bullet A\gamma', k, p]$ in Q_j with $m = l+p+1$ if $A \to \gamma$ a terminal error production rule and $m = l+p$ else.

The application of a *scanner* rule leaves the number of edit operations k unchanged, and each new *Earley item with number of operations* created by a *predictor* rule starts with 0 already applied edit operations ($k = 0$). This small adjustment to the rules of the Earley Parser (Alg.1) and the above-mentioned storing of only the Earley item with the smallest number of edit operations leads to an error correcting parser. Since we know that $L(G') = \Sigma^*$, we can follow that the here presented adapted Earley Parser will always contain the element $[S' \to S\bullet, 0, k]$ in the Earley set $Q_{|w|}$, where k is the minimum number of edit operations to correct the input word to a word in the target language ($L(G)$) and S is the start symbol of the grammar. The syntax tree from the corrected word can be derived by the backward search on Earley sets to compute the syntax tree presented in Section 2.4.

The here presented error correcting parser solves the error correcting problem for a word of length n in time $\mathcal{O}\left(n^3\right)$ [1].

Example 2.11. The covering grammar of the context-free grammar G_{aexpr} defined in Example 2.3 is $G''_{aexpr} = \left(\{E, T, F, A_+, A_-, A_*, A_/, A_(, A_), A_n, S'\}, \{+, -, *, /, (,), n\}, P'', S'\right)$ with P'' as follows:

$$
\begin{array}{ll}
E \rightarrow\ EA_{+}T \mid EA_{-}T \mid T & A_{/} \rightarrow H/ \\
T \rightarrow\ TA_{*}F \mid TA_{/}F \mid F & A_{(} \rightarrow (\\
F \rightarrow\ A_{(}EA_{)} \mid A_{n} & A_{(} \rightarrow + \mid - \mid * \mid / \mid) \mid n \mid \varepsilon \\
A_{+} \rightarrow + & A_{(} \rightarrow H(\\
A_{+} \rightarrow - \mid * \mid / \mid (\mid) \mid n \mid \varepsilon & A_{)} \rightarrow) \\
A_{+} \rightarrow H+ & A_{)} \rightarrow + \mid - \mid * \mid / \mid (\mid n \mid \varepsilon \\
A_{-} \rightarrow - & A_{)} \rightarrow H) \\
A_{-} \rightarrow + \mid * \mid / \mid (\mid) \mid n \mid \varepsilon & A_{n} \rightarrow n \\
A_{-} \rightarrow H- & A_{n} \rightarrow + \mid - \mid * \mid / \mid (\mid) \mid \varepsilon \\
A_{*} \rightarrow * & A_{n} \rightarrow Hn \\
A_{*} \rightarrow + \mid - \mid / \mid (\mid) \mid n \mid \varepsilon & A_{n} \rightarrow + \mid - \mid * \mid / \mid (\mid) \mid n \\
A_{*} \rightarrow H* & I \rightarrow + \mid - \mid * \mid / \mid (\mid) \mid n \\
A_{/} \rightarrow / & S' \rightarrow E \mid EH \\
A_{/} \rightarrow + \mid - \mid * \mid (\mid) \mid n \mid \varepsilon & H \rightarrow HI \mid I
\end{array}
$$

where the red-colored production rules are terminal error production rules. We use the covering grammar to compute for the word $+$, which is obviously not contained in the language G_{aexpr}, the word with the smallest Levenshtein distance to it by the Earley Parser with the adjustments described above. The computed Earley items with the number of operations are outscored to the appendix (App. C, Fig. 23). We can observe that the Earley item $[S' \rightarrow\ E\bullet, 0, 1]$ are contained in the Earley Set Q_1. This implies that we can correct the input word by a single edit operation $(+/_0 n)$ to a word (n) in the language L_{aexpr}.

3 Indexless Corrections

The notion of corrections was introduced in Section 2.8. In Section 4 we investigate how we can compute all corrections for a word and a context-free grammar by using an adapted Earley Parser. For that, we need an equivalent formalization of corrections, where the edit operations *don't use indices* and we require an indicator that we *leave a character* of the word *unchanged* (*read operation*).

In this section, we present such an equivalent formalization of corrections, are called *indexless corrections*. To show that indexless corrections are equivalent to corrections, we first define the simplification for indexless corrections in the same way as for corrections, and present a *conversion function* (Sec. 3.1) to convert corrections to indexless corrections so that both correct a word to the same corrected word.

At least in Section 3.2 we consider a prefix order on corrections. Based on this order relation, we define an order relation on indexless corrections such that the order on corrections is maintained by the conversion to indexless corrections.

Basic Idea of Indexless Corrections For a *simplified* correction a word, we can observe that each character of the word can either be deleted, replaced by another letter or left unchanged (in this case we say that the character is read). For each insert operation of the correction, we can use its index and its position in the correction to determine exactly between which two characters of the word it applies, or whether it applies before the first character or after the last character. All the insert operations that apply between the same two characters together define a possible empty word.

Therefore, we can interpret each simplified correction as: insert the word defined by the insertion operations before the first character; delete or read the first character or replace it by another letter; insert the word defined by the insertion operations between the first and second character; and so on. So, for a word of length n we have exactly $2n + 1$ operations. The read, replace, or delete of each of the n characters and the insert of $n + 1$ words before each character and after the last character.

This previous high-level description is the basic idea of *indexless correction*, where an edit operation no longer needs an index because their *position* in an indexless correction *indicates* the *index* in the word for the edit operation. As mentioned above, an indexless correction for a word $w = a_0 \dots a_{n-1}$ of length n is a sequence of length $2n + 1$. Where at each even position i it is a word to insert before the character $a_{\frac{i}{2}}$ and at each odd position j it is a read, delete, or replacement operation of the character $a_{\frac{j+1}{2}}$.

Indexless Edit Operations As already mentioned above, the edit operations of indexless correction no longer require an index, since this is given implicitly by the position in the correction. For this reason, we call the edit operations of indexless correction *indexless edit operation*, where the meaning of these operations is left unchanged compared to conventional edit operations (Definition 2.10). Additionally the *read* operation (written as $a_i \rightarrow$), is an indexless edit operation that leaves a character unchanged. Furthermore, we allow for insert operations, instead of inserting only one symbol $a \in \Sigma$, to insert a word $x \in \Sigma^*$.

Definition 3.1. Let Σ be some alphabet, $\perp \notin \Sigma$. An *indexless edit operation* (written as τ) $\tau : \Sigma \cup \{\varepsilon\} \rightarrow \Sigma^* \cup \{\perp\}$ is defined as follow:

$$\tau(a) = \begin{cases} x, & \text{if } \tau = x\uparrow \text{ and } a = \varepsilon \\ \varepsilon, & \text{if } \tau = a\downarrow \\ b, & \text{if } \tau = a/b \\ a, & \text{if } \tau = a\rightarrow \\ \perp, & \text{otherwise} \end{cases}$$

		a_0		a_1		a_2		a_3	
$w =$		$+$		$+$		n		n	
	x_0	v_0	x_1	v_1	x_2	v_2	x_3	v_3	x_4
$\rho =$	$n\uparrow$	$+\rightarrow$	$(n\uparrow$	$+\downarrow$	$)+\uparrow$	$n/($	$\varepsilon\uparrow$	$n\rightarrow$	$)\uparrow$
$\rho(w) =$	n	$+$	$(n$		$)+$	$($		n	$)$

Table 3: Applying the indexless correction ρ from Example 3.1 to the word $++nn$.

where $b \in \Sigma$, $x \in \Sigma^*$ and $\perp$ is the undefined value that will be used if we try to perform an incorrect indexless edit operation.

As a convention, we write x for indexless insert operations and v for indexless read, replace or delete operations.

Indexless Correction We already know that an indexless correction for a word of length n is an *alternating sequence* of indexless insert operations and indexless read, replace, or delete operations of length $2n+1$. The resulting word from applying the indexless correction is obtained by concatenating the results of the individual indexless edit operations of the indexless corrections. Formally, an indexless correction is defined as follows:

Definition 3.2. Let Σ be an alphabet, $w \in \Sigma^*$ with $w = a_0 \dots a_{n-1}$ and $|w| = n$. Then the set of all indexless corrections C_w for w is defined as follows:

$$\begin{aligned} C_w = &\{\rho \mid \rho = (x_0, v_0, x_1, v_1 \dots, v_{n-1}, x_n), x_i = x\uparrow \text{ with } x \in \Sigma^* \\ &\text{and } v_i \in \{a_i\rightarrow, a_i\downarrow, a_i/b\} \text{ with } b \in \Sigma\} \end{aligned}$$

Definition 3.3. Let Σ be an alphabet, $w \in \Sigma^*$ and $\rho \in C_w$. The application of ρ onto w is a mapping $\rho(w) : \Sigma^* \rightarrow \Sigma^*$ that is defined as the concatenation of the results of the individual indexless edit operations.

$$\rho(w) = x_0(\varepsilon) \cdot v_0(a_0) \cdot x_1(\varepsilon) \cdot v_1(a_1) \dots v_{n-1}(a_{n-1}) \cdot x_n(\varepsilon)$$

Note that unlike corrections, the application of indexless corrections is defined only on exactly one word w. Since of the structure of the indexless correction, we have for all other words $w' \neq w$ that at least one $v_j(a_j)$ evaluates to $\perp$. Therefore, in the rest of this work we only consider indexless corrections that are applicable to a given word.

Example 3.1. For the word $++nn$ and the language L_{aexpr} defined in Example 2.3 we consider the indexless correction

$$\rho = (n\uparrow,\ \ +\rightarrow,\ \ (n\uparrow,\ \ +\downarrow,\ \ )+\uparrow,\ \ n/(,\ \ \varepsilon\uparrow,\ \ n\rightarrow,\ \ )\uparrow)$$

which leads to the correct word $n+(n)+(n)$. The process of applying this indexless correction is shown graphically in Table 3.

Simplified Indexless Corrections We have seen in Section 2.8 that some corrections are unnecessarily complex. By applying simplification steps, we were able to convert such a correction into a correction with a minimum number of edit operations. However, this definitions of simplification (Definition 2.15 and Definition 2.16) cannot be transferred directly

One-Step Simplification	Indexless Correction	Simpler Indexless Correction
$(b\uparrow_i, b\downarrow_i) \triangleright ()$	$\emptyset$	
$(b\uparrow_i, b\downarrow_{i+1}) \triangleright ()$	$(v_0\uparrow, \dots, v_j \cdot b\uparrow, \ b\downarrow, \dots, v_n\uparrow)$	$(v_0\uparrow, \dots, v_j\uparrow, \ b\rightarrow, \dots, v_n\uparrow)$
$(b\uparrow_i, b/_i d) \triangleright (d\uparrow_i)$	$\emptyset$	
$(b\downarrow_i, b\uparrow_i) \triangleright ()$	$(v_0\uparrow, \dots, v_j \cdot b\uparrow, \ b\downarrow, \dots, v_n\uparrow)$	$(v_0\uparrow, \dots, v_j\uparrow, \ b\rightarrow, \dots v_n\uparrow)$
$(b\downarrow_i, b\uparrow_i) \triangleright ()$	$(v_0\uparrow, \dots, b\downarrow, \ b \cdot v_j\uparrow, \dots, v_n\uparrow)$	$(v_0\uparrow, \dots, b\rightarrow, \ v_j\uparrow, \dots v_n\uparrow)$
$(b\downarrow_i, c\uparrow_i) \triangleright (b/_i c)$	$(v_0\uparrow, \dots, v_j \cdot c\uparrow, \ b\downarrow, \dots, v_n\uparrow)$	$(v_0\uparrow, \dots, v_j\uparrow, \ b/c, \dots, v_n\uparrow)$
$(b\downarrow_i, c\uparrow_i) \triangleright (b/_i c)$	$(v_0\uparrow, \dots, b\downarrow, \ c \cdot v_j\uparrow, \dots, v_n\uparrow)$	$(v_0\uparrow, \dots, b/c, \ v_j\uparrow, \ , \dots, v_n\uparrow)$
$(c/_i b, c\uparrow_i) \triangleright (b\uparrow_{i+1})$	$(v_0\uparrow, \dots, v_j \cdot c\uparrow, \ c/b, \dots, v_n\uparrow)$	$(v_0\uparrow, \dots, v_j\uparrow, \ c\rightarrow, \ b \cdot v_{j+1}\uparrow, \dots, v_n\uparrow)$
$(c/_i b, c\uparrow_{i+1}) \triangleright (b\uparrow_i)$	$(v_0\uparrow, \dots, c/b, \ c \cdot v_j\uparrow, \dots, v_n\uparrow)$	$(v_0\uparrow, \dots, v_{j-1} \cdot b\uparrow, \ c\rightarrow, \ v_j\uparrow, \dots, v_n\uparrow)$
$(c/_i b, b\downarrow_{i-1}) \triangleright (c\downarrow_i)$	$(v_0\uparrow, \dots, b\downarrow, \ \varepsilon\uparrow, \ c/b, \dots, v_n\uparrow)$	$(v_0\uparrow, \dots, b\rightarrow, \ \varepsilon\uparrow, \ c\downarrow, \dots, v_n\uparrow)$
$(c/_i b, b\downarrow_i) \triangleright (c\downarrow_i)$	$\emptyset$	
$(c/_i b, b\downarrow_{i+1}) \triangleright (c\downarrow_i)$	$(v_0\uparrow, \dots, c/b, \ \varepsilon\uparrow, \ b\downarrow, \dots, v_n\uparrow)$	$(v_0\uparrow, \dots, c\downarrow, \ \varepsilon\uparrow, \ b\rightarrow, \dots, v_n\uparrow)$
$(c/_i b, b/_i d) \triangleright (c/_i d)$	$\emptyset$	

Table 4: Indexless corrections that can be simplified (center) by a one-step simplification to a simpler indexless correction (right) and the corresponding one-step simplification for corrections (left).

from corrections to indexless correction. Since indexless corrections for a word of length n are always a sequence of $2n + 1$ indexless edits.

However, there are also indexless corrections that are unnecessarily complex. For example, let $\rho = (\varepsilon\uparrow, b\downarrow, b\uparrow)$. This correction delete the letter b and then reinsert it by the next indexless editi operation, but it would be easier to read the letter b and then not insert it. Since the individual edit operations are less complex. The considered simplification for indexless corrections is represented for corrections by the one-step simplification $(b\downarrow_i, b\uparrow_{i+1}) \triangleright ()$.

Most of the other cases of one-step simplification on corrections can be transferred to indexless corrections in the same way, and are shown in Table 4. Therefore, we call the transferred simplification rules *one-step simplifications* of indexless corrections.

Note: If we have a consecutive pair of indexless edit operations, where the first is a deletion operation and the second is an insertion operation. We cannot determine whether the insertion is before or after the deleted character, since this character is no longer in the intermediate word. Therefore, we need to consider both cases for simplifications on indexless correction. Note also that not all cases of one-step simplifications for corrections can also be covered for indexless corrections (the rows with $\emptyset$) because, as will be discussed later, these cases are already excluded by the definition of indexless corrections.

It is easy to verify that the presented one-step simplifications of indexless correction don't change the word on that the the indexless correction is applicable. This is because the one-step simplification only replaces a deletion or replace operation with a read operation of the same letter or causes a shortening of an insert word (x_i). Therefore it is sufficient to define the simplification of indexless corrections only of pairs of indexless corrections $\rho, \rho,' \in C_w$ for a word w. Remember that C_w is the set of all indexless corrections for w (Definition 3.2).

Definition 3.4. Let Σ be an alphabet, $w \in \Sigma^*$ and $\rho, \rho' \in C_w$. Then according to Table 4 ρ can be simplified to ρ' (written as $\rho \triangleright \rho'$) if ρ covers any center column case of Table 4, then ρ' has the form of the right column of the same row. A *simplification* is the reflexive and transitive closure of the one-step simplification.

		a_0		a_1		a_2		a_3	
$w =$		$+$		$+$		n		n	
	x_0	v_0	x_1	v_1	x_2	v_2	x_3	v_3	x_4
$\rho =$	$n\uparrow$	$+\rightarrow$	$(n)\uparrow$	$+\rightarrow$	$\varepsilon\uparrow$	$n/($	$\varepsilon\uparrow$	$n\rightarrow$	$)\uparrow$
$\rho(w) =$	n	$+$	(n)	$+$		$($		n	$)$

Table 5: Applying the simplified indexless correction ρ from Example 3.2 to the word $++nn$.

The definition of one-step simplification of indexless corrections allows us to bring any correction to its simplest form by repeatedly applying one-step simplification. We then say that the indexless correction is *simplified*.

Definition 3.5. An indexless correction ρ is *simplified* if we cannot further simplify it.

By the definition of simplified indexless corrections, another advantage of indexless corrections compared to corrections becomes apparent. Due to their prescribed form, there are no similar indexless corrections. Therefore, in contrast to corrections, only one indexless correction needs to be considered to determine whether the indexless correction is simplified or not.

Example 3.2. We can observe that the indexless correction from Example 3.1 is not in simplified form. Therefore, we bring this indexless correction into simplified form by applying the following one-step simplifications:

$$\begin{aligned}
&[n\uparrow,\ +\rightarrow,\ (n\uparrow,\ +\downarrow,\ )+\uparrow,\ n/(,\ \varepsilon\uparrow,\ n\rightarrow,\ )\uparrow] \\
&[n\uparrow,\ +\rightarrow,\ (n\uparrow,\ +/),\ +\uparrow,\ n/(,\ \varepsilon\uparrow,\ n\rightarrow,\ )\uparrow] && \mid (b\downarrow_i, c\uparrow_i) \triangleright (b/_i c) \\
&[n\uparrow,\ +\rightarrow,\ (n)\uparrow,\ +\rightarrow,\ \varepsilon\uparrow,\ n/(,\ \varepsilon\uparrow,\ n\rightarrow,\ )\uparrow] && \mid (c/_i b, c\uparrow_i) \triangleright (b\uparrow_{i+1})
\end{aligned}$$

It is easy to verify that the resulting indexless correction

$$[n\uparrow,\ +\rightarrow,\ (n)\uparrow,\ +\rightarrow,\ \varepsilon\uparrow,\ n/(,\ \varepsilon\uparrow,\ n\rightarrow,\ )\uparrow]$$

is in simplified form because it does not match any simplification case from Table 4. The process of applying this simplified indexless correction is shown graphically in Table 5. We can observe that both the simplified and the non-simplified correction lead to the same corrected word: $n + (n) + (n)$. Note: As in the examples for corrections (Sec. 2.8), we use square brackets instead of round brackets for the indexless corrections to avoid confusion with the alphabet round brackets symbols.

3.1 Corrections vs. Indexless Corrections

We can observe that the simplified correction from Example 2.10 and the simplified indexless correction from Example 3.2 applied to the word $++nn$ both result in $n + (n) + (n)$. It is therefore obvious to specify a *conversion function* (written as *conv*) which, for a fixed word w, converts a simplified correction ρ into simplified indexless corrections ρ' so that both applied to the same word result in the same corrected word ($\rho(w) = \rho'(w)$).

Observations on Indexless Corrections But before we define a conversion function, let us make some observations about corrections, indexless corrections, and their differences.

- At first, the definition of indexless corrections allows editing each character of the word only once, by deleting it or replacing it with another letter. This is not the case for corrections, where we can first replace the character and then delete the new character or replace the new character as well. But both pairs of operations can be simplified to simply deleting the character $((a/_i b, b\downarrow_i) \triangleright (a\downarrow_i))$ respectively by replacing the original character by the new character of the second replacement $((a/_i b, b/_i c) \triangleright (a/_i c))$.
- In addition, we can observe that in a correction, any letter that is inserted by some insert operation can subsequently be deleted or replaced. This cannot be covered by indexless corrections. However, the case described for corrections can also be simplified by doing nothing $((a\uparrow_i, a\downarrow_i) \triangleright ())$ or inserting directly the replaced character $((a\uparrow_i , a/_i b) \triangleright (b\uparrow_i))$.

Therefore, we can only convert corrections to indexless corrections if neither of the two cases described above occurs. We have already mentioned that both cases can be simplified, so for simplified corrections, these cases cannot occur. Therefore, we can only use simplified corrections to convert them to indexless corrections.

But we have already discussed in section 2.8 that it is sufficient to consider only normalizsd corrections. Since every correction ρ can be simplified to a simplified correction ρ', and that in $[\rho']_{\simeq}$ is a normalized correction. Remember that each normalised corrections is also simplified (Definition 2.17)

Definition 3.6. Let Σ be a alphabet, $w \in \Sigma^*$ and let ψ_w be the set of all normalised corrections defined for w:

$$\psi_w = \{\rho \mid \rho(w) \neq \perp \text{ and } \rho \text{ is a normalised correction}\}$$

In addition, let C'_w the set of all simplified indexless corrections for w:

$$C'_w = \{\rho \mid \rho \in C_w \text{ and } \rho \text{ is simplified}\}$$

Remember that C_w is the set of all indexless corrections for some word w (Definition 3.2). Then the conversion function *conv* defined by Algorithm 2 is a function that maps defined normalised corrections to simplified indexless corrections so that the application of both result in the same corrected word.

$$conv : \psi_w \to C'_w,\ \rho \mapsto \rho' \text{ such that} \rho(w) = \rho'(w)$$

Conversion Algorithm The *conversion algorithm*, presented in Algorithm 2, defines the converting function *conv* for a word w. At first, the algorithm sets all $2|w| + 1$ indexless edit operations to their default value. The default value for indexless insert operations is to insert the empty word $(\varepsilon\uparrow)$ and for the other position of the indexless correction, the read of the corresponding character $(a_i\rightarrow)$ because these default operations leave the word unchanged. The conversion algorithm then updates the default values, considering the edit operations of the normalised correction. Starting with the replace and read operations, the operation with index j is set as operation v_j of the indexless correction. Since a replace operation has no effect on the index of the following operations, in normalised corrections, the indexes of the delete operations are ordered from largest to smallest.

At last, we consider the insert operations of the normalised corrections. Here we must first determine between which two characters of the word the operation is applied by considering the number of deletion operations before this position. In addition, the order of

Algorithm 2 CONVERSION_ALGORITHM

Require: A correction $\rho = (b/_{i_0}c, \ldots, b/_{i_r}c, d{\downarrow_{j_0}}, \ldots, d{\downarrow_{j_m}}, e{\uparrow_{h_0}}, \ldots, e{\uparrow_{h_k}})$ in normalised form and an word $w = a_0 a_1 \ldots, a_{n-1}$ such that $\rho(w) \neq \perp$.
Ensure: An indexless correction.

$v_i \leftarrow a_i{\rightarrow}$ for all $0 \leq i < |w|$
$x_i \leftarrow \epsilon$ for all $0 \leq i \leq |w|$

for $l = 0$ to r **do**
    $b/_{i_l}c \leftarrow \rho[l]$ ▷ Where $\rho[l]$ is the l-th edit operation of the correction.
    $v_{i_l} \leftarrow b/c$
end for

for $l = 0$ to m **do**
    $a{\downarrow_{j_l}} \leftarrow \rho[l + n]$
    $v_j \leftarrow a_{j_l}{\downarrow}$
end for

for $l = 0$ to k **do**
    $e{\uparrow_{h_l}} \leftarrow \rho[l + n + m]$
    $j \leftarrow h_l, i \leftarrow 0$

    while $v_i < j$ **do**
        if v_i an indexless delete operation **then**
            $j \leftarrow j + 1$
        end if
        $i \leftarrow i + 1$
    end while
    $x_j \leftarrow e \cdot v_j$
end for

return $\left(x_0{\uparrow}, v_0, x_1{\uparrow}, v_1, \ldots, x_{|w|-1}{\uparrow}, v_{|w|-1}, x_{|w|}{\uparrow}\right)$

insertions in the normalised correction shows that when there are two insertions between the same characters, the later inserted letter is in front of the first inserted letter in the corrected word. This allows us to gradually determine all x_i of the indexless correction, so that the calculated indexless correction can finally be returned.

Example 3.3. We have already mentioned that the normalised correction $\rho = [n/_2(,)\uparrow_4,)\uparrow_1$, $n\uparrow_1, (\uparrow_1, n\uparrow_0]$ from Example 2.10 applied on the word $w = ++nn$ results in the same corrected word as the simplified indexless correction $\rho' = [n\uparrow, +\rightarrow, (n)\uparrow, +\rightarrow, \varepsilon\uparrow, n/(, \varepsilon\uparrow, n\rightarrow,)\uparrow]$ from Example 3.2 ($\rho(w) = \rho'(w)$).

In addition, we can observe that the conversion function *conv* maps for the word w the normalised correction ρ to the simplified indexless correction ρ' ($conv(\rho) = \rho'$). Note: As in the previous examples, we use square brackets instead of round brackets for corrections and indexless corrections.

Correctness Conversion Algorithm We have presented a conversion function *conv* to convert corrections in normalised form into simplified indexless corrections by using the conversion algorithm (Alg. 2). We want to show that for a fixed word w the conversion algorithm maps any defined normalised correction ρ to a simplified indexless correction ρ' so that applying both on the word w results in the same corrected word ($\rho(w) = \rho'(w)$).

We can observe that the conversion algorithm processes the operations of the normalised correction from left to right to build up the indexless correction.Therefore, the idea of the proof is to show that after any number of edit operations already applied from the normalised correction then the indexless correction so far computed by the conversion algorithm, leads to the same corrected word by applying both on w.

Theorem 3.1. *Let Σ be a alphabet, $w \in \Sigma^*$ with $w = a_0 \dots a_{n-1}$ and $|w| = n$. Each normalised correction ρ so that $\rho(w) \neq \perp$ is converted by the conversion function conv (defined by Algorithm 2) to an indexless correction $\rho' = conv(\rho)$ such that $\rho(w) = \rho'(w)$.*

Proof. Let $\rho = (b/_{i_0}c, \dots, b/_{i_r}c, d\downarrow_{j_0}, \dots, d\downarrow_{j_m}, e\uparrow_{h_0}, \dots, e\uparrow_{h_k})$ with $r, m, k \in \mathbb{N}$ such that $i_0 > \dots > i_r$; $j_0 > \dots > j_k$ and $h_0 \geq \dots \geq h_k$ be a correction in normalised form.

First, we look at some properties of normalized corrections that will help us in the rest of the proof.

- Since each normalised correction is also simplified (Definition 2.17) we know that for each position of the word there is at most either a replacement or a deletion operation. If this were not the case, the correction could be simplified by the one-step simplification $(c/_i b, b\downarrow_i) \triangleright (c\downarrow_i)$ (see Table 2), which would be a contradiction to the fact that the correction is simplified.
- Based on the order of the edit operations in normalized corrections, we can observe that each replacement operation with index i replaces the letter a_i. Since the length of the word is not changed by replacement operations and the deletion operations are also sorted in descending index order, each deletion operation with index i deletes the letter a_i.

We want to show now by an induction over the edit operations of the correction that the conversion function *conv* maps this correction to an indexless correction ρ', so that $\rho(w) = \rho'(w)$.

As a preparation for the induction, let $\rho_{0,l}$ be the *prefix correction* of ρ, which consists only of the first l operations of ρ. Additionally, let $\rho'_{0,l} = conv\,(\rho_{0,l})$. Therefore, we show by the following induction that for all $0 \leq l < (r + m + k)$: $\rho_{0,l}(w) = \rho'_{0,l}(w)$.

Base Case The base case is $l = 0$. Therefore, $\rho_{0,0}$ is the empty correction ($\rho_{0,0} = ()$), so by applying *conv* to the empty correction, we get the indexless correction $\rho'_{0,0} = (\varepsilon\uparrow, a_0\rightarrow, \dots a_{n-1}\rightarrow, \varepsilon\uparrow)$. We can see that applying both corrections leaves the input word unchanged ($\rho(w) = \rho'(w) = w$), so the base case is already proven.

Induction Hypothesis We assume for some l with $0 \leq l \leq (n + m + k)$ that $\rho_{0,l}(w) = \rho'_{0,l}(w)$ applies.

Induction Step In the induction step, we need to show that $\rho_{0,l+1}(w) = \rho'_{0,l+1}(w)$ also holds. To achieve this, we use the induction hypothesis that

$$\rho_{0,l}(w) = \rho'_{0,l}(w) = a'_0 \dots a'_{i-1} a'_i a'_{i+1} \dots a'_p$$

and add the next editing operation of ρ to $\rho_{0,l}$. Let $\tau = \rho_{0,l+1}[l+1]$ this next edit operation. This means that we only have to show that after adding τ the application of both corrections also leads to the same corrected word. Therefore, we divide the proof into the three different types of edit operations that τ can take and prove it separately for each type (replacement, deletion, and insertion).

- **Replacement** If $\tau = a'_i/_i b$ for some i with $0 \leq i \leq p$, then we have that $\rho_{0,l+1}(w) = a'_0 \dots a'_{i-1} b a'_{i+1} \dots a'_p$.

  Since τ is a replacement operation and due to the previously considered properties of normalized corrections, we know that the algorithm has not previously considered an operation that changes the letter a_i. Therefore, we have that $v_i = a'_i\rightarrow$, whereby we assume that $a_i = a'_i$.

  By considering τ in the conversion algorithm (Alg. 2, line 6) v_i is set to the indexless replacement operation a'_i/b. The rest of the indexless correction remains unchanged compared to $\rho'_{0,l}$, so that $\rho'_{0,l+1}(w) = a'_0 \dots a'_{i-1} b, a'_{i+1} \dots a'_p$ also applies.

- **Deletion** If $\tau = a'_i\downarrow_i$ for some i with $0 \leq i \leq p$, then we have that $\rho_{0,l+1}(w) = a'_0 \dots a'_{i-1} a'_{i+1} \dots a'_p$.

  For the same reasons as in the replacement case, we have that $v_i = a'_i\rightarrow$. Therefore, when τ is considered by the conversion algorithm, the read operation is replaced by a deletion operation of the same character ($v_j \leftarrow a'_i\downarrow$), so that $\rho'_{0,l+1}(w) = a'_0 \dots a'_{i-1} a'_{i+1} \dots a'_p$ also applies.

  This means that by considering τ, the conversion algorithms must extend the indexless correction by adding the letter b to the right inserting word x_j with $0 \leq j \leq |w|$ at the right place so that the letter b occurs after a'_{i-1} and before a'_i in the corrected word. Therefore, we first argue that the algorithm should choose to extend the correct x_j and that this also shows that b is added between a_{i-1} and a_i.

  In contrast to the replacement and deletion cases, the index i does not directly indicate between which two letters of the word w the insertion operation is performed. This position can be shifted by deletions before the letter a_i from the word w. Therefore, we must first determine how many letters in w are before a_i has already been deleted by $\rho_{0,l}$ (remember that a_i can also be replaced or deleted by $\rho_{0,l}$). This number must then be added to the position i to determine x_j, which needs to be extended by the letter b. This choosing of an x_j is done in lines 15 to 23 of the conversion algorithm (Alg. 2). Due to the order of the normalised corrections, no insertion operation in $\rho'_{0,l}$ has an effect on the x_j to be determined because they all have a greater or equal index. Note that this also indicates that no $a'_{i'}$ with $i' \leq i$ in $\rho_{0,l}(w)$ is the result of an insert operation. This implies that b must appear in the corrected word after a'_{i-1}.

  It remains to be shown that the letter b is inserted before a'_i in the corrected word. For that, we can use the fact that the order of normalised corrections implies that if

several insertions are performed between the same letters of w, the letter inserted last is the first in the corrected word. Therefore, if a'_i is also added by an insert operation between the same two characters, b will come before a'_i in the corrected word, and if a'_i is added by a later indexless edit operation, a'_i will also come after b in the corrected word.

Therefore, we have that $\rho'_{0,l+1}(w) = a'_0 \dots a'_{i-1} b a'_{i+1} \dots a'_p$ also applies.

So for all possible edit operations for τ we have that $\rho_{0,l+1}(w) = \rho'_{0,l+1}(w)$, which shows the correctness of the induction step.

Therefore, we have shown that the conversion function defined by Algorithm 2 is, for a fixed word, a mapping from normalised corrections to indexless corrections so that both results to the same corrected word.

Indexless Correction is in Simplified form It remains to be shown that the resulting indexless correction is also simplified. Since the indexless correction ρ' is in simplified form, none of the cases from Table 4 (centre) that can be simplified may occur. We show here by contradiction only the case that no indexless correction ρ' of the form $(v_0\uparrow, \dots, v_j \cdot b\uparrow, \quad b\downarrow, \dots, v_n\uparrow)$ can be occur if the correction ρ is in a simplified form because all other cases can be shown in the same way.

We assume that if $(v_0\uparrow, \dots, v_j \cdot b\uparrow, \quad b\downarrow, \dots, v_n\uparrow)$ is the result of the conversion function, then the normalised correction ρ must contain the edit operation $b\downarrow_i$ and as the last resulting insert operation for x_i the insert operation $b\uparrow_i$. Therefore, the normalised correction has the form $(\dots, b\downarrow_i, \dots, b\uparrow_{i'}, \dots)$, please note that the index i' of the insert operation can be changed by further deletion operations. This implies that either ρ or at least one correction in the equivalence class of ρ ($[\rho]_\simeq$) can be simplified by the simplification $(b\downarrow_i, b\uparrow_i) \triangleright ()$, so we have a contradiction to that ρ is in simplified form.

This shows that there can be no indexless correction in the form that we have considering here. It can also be shown, in the same way, that all other cases of indexless correction that can be simplified cannot occur if ρ is simplified. So we have that the conversion function maps normalised corrections to simplified indexless corrections. □

Inverse Conversion Function The conversion function *conv* allows us to convert normalised corrections to simplified indexless corrections. To be able to use indexless corrections instead of corrections in the rest of this work, we also need an *inverse conversion function* $conv^{-1}$ that converts simplified indexless corrections back into normalised corrections. It must also apply to the reverse conversion function that a correction ρ, which was mapped to indexless correction by the conversion function, is mapped back to the original correction ρ by the reverse conversion function ($\rho = conv^{-1}(conv(\rho))$).

But instead of defining such an inverse function and showing that it satisfies the requirements, we show that the conversion function is a bijective function, which implies that there must be such an inverse conversion function.

Bijectivity of the Conversion Function One could assume that the bijectivity follows from the correctness by showing that for each word pair w, w' there is at most one normalized correction ρ, so that $w' = \rho(w)$. It would then follow directly from the correctness of the algorithm that the conversion function defined by algorithm 2 is bijective. Unfortunately, this is not the case, as the following small counterexample shows. Let $w = a$ and let $\rho = (a\uparrow_0)$ and $\rho'(a\uparrow_1)$ be corrections, which are both obviously normalized. Then $\rho(w) = \rho'(w) = aa$ but $\rho \neq \rho$.

Nevertheless, it can be shown that the conversion function *conv* defined by Algorithm 2 is a *bijective* function. Therefore, we show that for all pairs of distinct normalised corrections in normalised form, the conversion function maps them to different simplified indexless

corrections (*surjective*). Afterwards we show that for a fixed word w, for each indexless correction $\rho' \in C_w$ in simplified form, there is a correction ρ in normalised form so that the conversion function maps ρ to ρ'(*injective*).

Lemma 3.2. *The conversion function conv (defined by Algorithm 2) is a injective function.*

Proof. To show that the conversion function is an injective function, we need to show that for any two distinct elements of the domain, the function maps them to distinct elements in the codomain.

Therefore, let Σ be a alphabet, $w \in \Sigma^*$ with $w = a_0 \dots a_{n-1}$ and let ρ_1, ρ_2 be two normalised corrections so that $\rho_1(w) \neq \bot$ and $\rho_2(w) \neq \bot$. Remember that each normalised correction is also simplified (Definition 2.17).

We need to show that for each pair $\rho_1 \neq \rho_2$ the conversion function maps ρ_1 and ρ_2 to different indexless corrections ($conv(\rho_1) \neq conv(\rho_2)$).

For that $\rho_1 \neq \rho_2$ to be true, the two corrections must be different in at least one edit operation. This means that at least one edit operation occurs in only one of the two corrections. Without loss of generality, we assume that τ is an edit operation that only occurs in correction ρ_1. Therefore, we will show in the following that for all possible types of edit operations, which can assume τ (replacement, deletion and insertion), it follows that the two indexless corrections $conv(\rho_1)$ and $conv(\rho_2)$ are distinct.

- **Replacement** If $\tau = a_i/_i b$ for some $b \in \Sigma \setminus \{a_i\}$, then we have that v_i for $conv(\rho_1)$ is a_i/b. Since τ is not in ρ_2 and the correction is in normalised form, the conversion function determines for $conv(\rho_2)$ that v_i is either $a_i\rightarrow$ or $a_i\downarrow$. Therefore, we have that $conv(\rho_1) \neq conv(\rho_2)$.

- **Deletion** If $\tau = a_i\downarrow_i$, then we have that v_i for $conv(\rho_1)$ is $a_i\downarrow$. For the same reasons as in the replacement case, we have for $conv(\rho_2)$ that v_i is either $a_i\rightarrow$ or a_i/b for some $b \in \Sigma \setminus \{a_i\}$. Therefore, we have that $conv(\rho_1) \neq conv(\rho_2)$.

- **Insertion** We can observe that each inserting word x_i is created by a sequence of insertion operations with the same index $(\dots, b_1\uparrow_j, b_2\uparrow_j, b_3\uparrow_j, \dots)$. Note that the order of the letters is reversed in the inserted word ($x_i = \dots b_3b_2b_1 \dots \uparrow$) and that j depends on the number of deletions in the normalised correction before the letter a_i.

  Every insert operation is part of such a sequence. Therefore, without loss of generality, we assume that τ is part of the sequence that produces the insert word x_i for $conv(\rho_1)$. Then it follows from the fact that τ is not in the correction ρ_2, that the inserting word x_i for $conv(\rho_2)$ is distinct to the inserting word x_i for $conv(\rho 1)$. Therefore, we have that $conv(\rho_1) \neq conv(\rho_2)$.

So we have shown that for all possible types of edit operations that τ ca be taken, that the resulting indexless corrections are distinct ($conv(\rho_1) \neq conv(\rho_2)$). This implies that the conversion function is an injective function. □

We have shown the injectivty of the conversion function; it remains to show its surjectivity.

Lemma 3.3. *The conversion function conv (defined by Algorithm 2) is a surjective function.*

Proof. To show that the conversion function is a surjective function, we need to show that for every element x in the codomain, there is an element in the domain that maps to x.

Therefore, let Σ be a alphabet, $w \in \Sigma^*$ with $w = a_0 \dots a_{n-1}$ and let ρ' a simplified indexless correction contained in C_w (see Definition 3.2). Then we need to show that they

are a normalised correction ρ such that the conversion function *conv* (Definition 3.6) maps ρ to ρ'.

We show this by specifying for each possible simplified indexless correction how a simplified correction ρ in normalised form can be calculated so that $conv(\rho) = \rho'$ applies. For the conversion algorithm, we can observe that each replacement and deletion operation defines exactly one v_i and each insert operation is part of one x_i. Therefore, the only thing that needs to be done is to reverse this mapping, which can be done by the following rule that already describes the inverse conversion function $conv^{-1}$.

For each simplified indexless correction $\rho' = (x_0, v_0, x_1, v_1 \ldots, v_{n-1}, x_n)$ we can reconstruct the normalised correction ρ as follows:

- For each v_i:
  - If $v_i = a_i/b$ for some $b \in \Sigma \setminus \{a\}$ then add $a_i/_i b$ to ρ.
  - If $v_i = a_i\downarrow$ then add $a_i\downarrow_i$ to ρ.
  - If $v_i = a_i\rightarrow$ then do nothing.
- For each $x_i = b_0 b_1 \ldots b_{m-1} b_m \uparrow$ with $b_j \in \Sigma$ for all $0 \leq j \leq m$: add the sequence $(b_m\uparrow i, b_{m-1}\uparrow i, \ldots, b_1\uparrow i, b_0\uparrow i)$ of insert operations to ρ.
- Bring the correction into normalised form.

Note that when adding edit operations to ρ, the indexes of the operations may have to be adjusted. It is easy to verify that the conversion of ρ ($conv(\rho)$) leads back to the indexless correction ρ'. So we have shown that the conversion function f is a surjective function. □

Since we know that the conversion function defined by the Algorithm 2 is an injective and surjective function, it follows directly that it is also a bijective function.

Corollary 3.4. *The conversion function conv (defined by Algorithm 2) is a bijective function.*

3.2 Order on Indexless Corrections

Some minimality definitions use a prefix ordering to define minimality. If the minimality definition requires that the corrections be in normalised form, only a few prefixes are possible. Therefore, it is often also allowed to choose prefixes for similar corrections. An example of such a minimality definition is that of [5], which we consider in Section 5.1.1 (Definition 5.3). However, this order cannot be transferred to indexless corrections. There are two reasons for this.

1. The above order don't use only normalised corrections, so we cannot convert all the corrections of the ordered set to simplified indexless corrections. To solve this, we analyze the above order and define an equivalent partial order relation $\prec_s$ only on the set of normalised corrections, called *scattered subcorrection* order (Definition 3.8).
2. Éach prefix ρ' of an indexless correction ρ with $\rho' \neq \rho$ for a word w leads to an invalid indexless correction. Because this prefix has at most $2n$ indexless editing operations, which violate the definition of indexless corrections (Definition 3.2).

To solve this, we define a partial order relation $\prec_i$ on indexless corrections (Definition 3.9). At least we show that the conversion function maps any pair of normalised corrections which are in order to each other by the *scattered subcorrection order* to a pair of simplified indexless corrections which are also in order to each other regarding the defined order on simplified indexless corrections ($\rho' \prec_s \rho \iff conv(\rho') \prec_i f(\rho)$) (Theorem 3.7).

Scattered Subcorrections At first, we define the prefix order of corrections mentioned above so that a correction ρ' is a subcorrection of another correction ρ' (written as $\rho' \prec_p \rho$) if and only if ρ' is a prefix of a correction in the equivalence class of ρ ($[\rho]_\simeq$). Remember, we write $a \blacktriangleleft b$ if a is a prefix of b (see Sec. 2.1).

Definition 3.7. Let ψ' the set of all corrections, then the relation $\preceq_p \subseteq \psi' \times \psi'$ is defined as:

$$\rho' \prec_p \rho \iff \exists \rho_1 \in [\rho]_\simeq \text{ so that } \rho' \blacktriangleleft \rho_1$$

The problem with this relation is that it is defined over all corrections because we cannot convert non-normalised corrections into indexless corrections. But from Section 2.8 we know that in the equivalence class of each correction there is also a correction in normalised form. Therefore, the idea of the following relation is to use the order defined above and reorder both corrections into a normalised form.

Definition 3.8. (cf. [5]) Let Σ be a alphabet, $w \in \Sigma^*$ and let ψ_w be the set of all normalised corrections defined for w:

$$\psi_w = \{\rho \mid \rho(w) \neq \bot \text{ and } \rho \text{ is a normalised correction}\}$$

Then the relation $\preceq_s \subseteq \psi_w \times \psi_w$ is defined as:

$$\rho' \prec_s \rho \iff \rho' \neq \rho \text{ and } \exists \rho_0 \in [\rho']_\simeq \; \exists \rho_1 \in [\rho]_\simeq \text{ such that } \rho_0 \blacktriangleleft \rho_1$$

of ρ.

Note that for all pairs of normalized corrections ρ and ρ' that are related to each other with respect to $\prec_s$ theirs don't correct the word w tho the same corrected ($\rho(w) \neq \rho'(w)$).

Example 3.4. As an example of scattered subcorrections we consider the correction $\rho = [n/_2(,)\uparrow_4,)\uparrow_1, n\uparrow_1, (\uparrow_1, n\uparrow_0]$ from Example 2.10. We have that $\rho' = [n/_2(,)\uparrow_4, n\uparrow_1, n\uparrow_0]$ is a scattered subcorrection of ρ ($\rho' \prec_s \rho$), because of $\rho' \blacktriangleleft [n/_2(,)\uparrow_4, n\uparrow_1, n\uparrow_0,)\uparrow_4, (\uparrow_3]$, which is in the equivalence class of ρ ($\in [\rho]_\simeq$).

As another example, we consider the simplified correction $\rho = (+\downarrow_0, +\downarrow_0, +\uparrow_1)$ in normalised form. We have that $\rho' = (+\uparrow_3)$ is a scattered subcorrection of ρ ($\rho' \prec_s \rho$) because $\rho' \blacktriangleleft (+\uparrow_3, +\downarrow_0, +\downarrow_0)$.

Note: As in previous examples, we also use square brackets for indexless corrections to avoid confusion with the alphabet symbols.

Observations on Scatterd Subcorrections For both scattered subcorrections of Example 3.4 we can observe that the correction is a *scattered subword* (see Sec. 2.1), where in the second example the index of the insert operation is shifted. This can be explained because we have named this order *scattered subcorrections*. The index shift can be explained by missing deletion operations in the scattered subcorrections, which implies that there are more letters before the insertion position and the index has to be adjusted accordingly. Therefore, instead of taking the detours via similar non-normalised corrections, the scattered subcorrection can also be calculated by selecting a scattered subword (Sec. 2.1) of the correction and adjusting the index for each insertion operation. The index must be increased by the difference from the deletion operation performed before this insertion position.

This also shows that the scattered subcorrection order is *transitive*. In addition, we can observe that the relation $\prec_s$ is *irreflexive* because of $\rho \neq \rho'$. Due to the equal length of all

τ \ ω	$u_i'\uparrow$	$b'\rightarrow$	$b'\downarrow$	b'/c'
$u_i\uparrow$	$\tau =_{ie} \omega$, if $u_i = u_i'$ $\tau \prec_{ie} \omega$, if $u_i \prec u_i'$ $\tau \succ_{ie} \omega$, if $u_i' \prec u_i$ $\tau \neq_{ie} \omega$, otherwise			
$b\rightarrow$		$\tau =_{ie} \omega$, if $b = b'$ $\tau \neq_{ie} \omega$, otherwise	$\tau \prec_{ie} \omega$, if $b = b'$ $\tau \neq_{ie} \omega$, otherwise	$\tau \neq_{ie} \omega$
$b\downarrow$		$\tau \succ_{ie} \omega$, if $b = b'$ $\tau \neq_{ie} \omega$, otherwise	$\tau =_{ie} \omega$, if $b = b'$ $\tau \neq_{ie} \omega$, otherwise	$\tau \neq_{ie} \omega$
b/c		$\tau \succ_{ie} \omega$, if $b = b'$ $\tau \neq_{ie} \omega$, otherwise	$\tau \neq \omega$	$\tau =_{ie} \omega$, if $b = b'$ and $c = c'$ $\tau \neq_{ie} \omega$, otherwise

Table 6: The order of the two indexless edit operations, τ and ω.

corrections of an equivalence class (see Definition 2.14), it follows that the relation is also *asymmetric*. This properties of the scattered subcorrection relation implies that the relation defines a *strict partial order* on the set of all defined corrections in normalised form for a fixed word w.

Corollary 3.5. *Let Σ be an alphabet and $w \in \Sigma^*$, then $(\psi_w, \prec_s)$ is a strict partial order.*

The above described equivalent calculation of the scattered subcorrections allows us to make the following assumptions about scattered subcorrections. Therefore, let ρ be a normalised correction, then for each scattered subcorrection ρ' ($\rho' \prec_s \rho$) we have that each edit operation contained in ρ' is also contained in ρ with possible adjustment of the indexes of the insert operations, and there is at least one edit operation τ which is only contained in ρ. We consider the different types of edit operations that τ can take and how this effects the indexless corrections $conv(\rho)$ and $conv(\rho')$, where $conv$ is the conversion function (Definition 3.6). These observations show how we must extend the ordering to indexless corrections, so that $conv(\rho')$ is less than $conv(\rho)$ if and only if ρ is less than ρ'.

- Let τ be the replacement operation $a_i/_i b$, then we have that v_i for $conv(\rho)$ is a_i/b and for $conv(\rho')$ v_i is $a_i\rightarrow$ because the correction is in simplified form. This means that for our partial order should be apply that: the read operation is less then each replace operation.
- Let τ be the deletion operation $a_i\downarrow_i$, then we have that v_i for $conv(\rho)$ is $a_i\downarrow$ and for $conv(\rho')$ v_i is $a_i\rightarrow$ because the correction is in simplified form. This means that for our partial order should be apply: the read operation is less then a deletion operation.
- Let τ be the insertion operation $b\uparrow_i$, then we have that x_i for $conv(\rho)$ inserts the word u_i ($x_i = u_i\uparrow$) and for $conv(\rho')$ x_i inserts the word u_i', where u_i' is a scattered subword of u_i ($u_i' \prec u_i$). This means that for our partial order should be apply: that an insertion operation $x\uparrow$ is less than an insertion operation $x'\uparrow$ if and only if x is a scattered subword of x'.

Order on Indexless Edit Operations These observations on scattered subcorrections leads to the definition of a partial order $\prec_{ie}$ on indexless edit operations, where we only allow to compare insertion operation with insertion operation and non-insertion operation with non-insertion operation. The order is presented for two indexless edit operations in Table 6. Remember that $\prec$ is the scattered subword order relation (see Sec. 2.1).

Order on Indexless Corrections Based on the order $\prec_{ie}$ for indexless edit operations, we can define an order $\prec_i$ for indexless correction. Above, we have only considered the case where we have omitted one edit operation from a correction to get a scattered subcorrection. But we can omit more than one edit operation to get a scattered subcorrection; this implies that we have the considered order one more than one indexless edit operation. Therefore, the simplified indexless correction ρ' is *less than or equal* to the simplified indexless correction ρ, if and only if for all edit operations at the same position in the correction this order is also satisfied.

Definition 3.9. Let Σ be a alphabet, $w \in \Sigma^*$ and let C'_w the set of all simplified indexless corrections for w:

$$C'_w = \{\rho \mid \rho \in C_w \text{ and } \rho \text{ is simplified}\}$$

Note that C_w is the set of all indexless correction for w (Definition 3.2). In addition, let $\rho = (x_0, v_0, x_1, v_1 \ldots, v_{n-1}, x_n)$ and $\rho' = \left(x'_0, v'_0, x'_1, v'_1 \ldots, v'_{n-1}, x'_n\right)$ be simplified indexless corrections contained in C'_w, then the relation $\preceq_i \subseteq C'_w \times C'_w$ is defined as:

$$\rho' \preceq_i \rho \iff (\forall_{i=0}^{n}\ x'_i \prec_{ie} x_i \text{ or } x'_i =_{ie} x_i) \text{ and } \left(\forall_{i=0}^{n-1}\ v'_i \prec_{ie} v_i \text{ or } v'_i =_{ie} v_i\right)$$

where we use the order on indexless edit operations $\prec_{ie}$ from Table 6.
The relation $\prec_i \subseteq \psi \times \psi$ is based on $\preceq_i$, but for at least one pair of indexless edit operations (x'_i, x_i) or (v'_i, v_i) it must satisfy that $x'_i \prec_{ie} x_i$ or $v'_i \prec_{ie} v_i$.

$$\rho' \prec_i \rho \iff \rho' \preceq_{ie} \rho \text{ and } \left(\exists_{i=0}^{n}\ x'_i \prec_{ie} x_i \text{ or } \exists_{i=0}^{n-1}\ v'_i \prec_{ie} v_i\right)$$

After we have defined the relations $\preceq_i$ and $\prec_i$, we look at some properties of these relations. It is obviously that the realization $\preceq_i$ is *reflexive* and that the relation $\preceq_i$ is *irreflexive*, because two indexless corrections in order must differ in at least one indexless edit operation. We know that two unequal simplified indexless corrections ρ and ρ' ($\rho \neq \rho'$) differ in at least one edit operation, which implies that at most either $\rho \prec_s \rho'$ or $\rho \succ_s \rho$ can apply. Therefore $\prec_i$ is *asymmetric* and $\preceq_s$ is *antisymmetric*. Due to the transitivity of scattered subwords relation $\prec$, it is obviously that the relation $\prec_{ie}$ on indexless edit operations is also transitive. From the transitivity of $\prec_{ie}$, the transitivity of the two relations $\prec_s$ and $\preceq_s$ can be followed. This properties implies that the relations define a *partial* and a *strict partial order* on the set of all simplified indexless corrections C'_w for a fixed word w.

Corollary 3.6. *Let Σ be a alphabet and $w \in \Sigma^*$, then $(C'_w, \preceq_i)$ is partial order and $(C'_w, \prec_i)$ is a strict partial order.*

Example 3.5. As an example of the partial order $\preceq_i$ on indexless corrections, we use the simplified indexless correction

$$\rho = (n\uparrow, +\rightarrow, (n)\uparrow, +\rightarrow, \varepsilon\uparrow, n/(, \varepsilon\uparrow, n\rightarrow,)\uparrow)$$

from Example 3.2 on the word $++nn$. For this correction, we have that the simplified indexless correction

$$\rho' = (n\uparrow, +\rightarrow, n\uparrow, +\rightarrow, \varepsilon\uparrow, n/(, \varepsilon\uparrow, n\rightarrow,)\uparrow)$$

are a subcorrection ($\rho' \prec_i \rho$) because for the indexless editing operations at position 3 in the corrections we have that $n\uparrow \prec_{ie} (n)\uparrow$ and all other positions the corrections are equal.

The Conversion Function is an Order Embedding It remains to show that the conversion function *conv* (Definition 3.6) obtains the order by mapping normalised corrections in simplified form to simplified indexless corrections. Therefore, for all defined pairs of corrections ρ and ρ' regarding some word it must hold that

$$\rho' \prec_s \rho \text{ if and only if } conv(\rho') \prec_i conv(\rho)$$

A function that fulfills such a condition is called an *order-embedding*.

Definition 3.10. (cf. [21, def. 1]) A function $f : (X, \prec_x) \to (Y, \prec_y)$ is an *order-embedding* if for all $u, v \in X$ it applies that:

$$u \prec_x v \text{ if and only if } f(u) \prec_y f(v)$$

Theorem 3.7. *Let Σ be alphabet, $w \in \Sigma$, let ψ_w be the set of all normalised correction defined for w, let C'_w the set of all simplified indexless corrections for w and let $conv : \psi_w \to C'_w$ the conversion function (Definition 3.6) defined for the word w.*

Then conv is an order-embedding from the partial ordered set $(\psi_w, \prec_s)$ into the partial ordered set $(C'_w, \prec_i)$.

Proof. To prove that the conversion function defines an order-embedding from $(\psi_w, \prec_s)$ into $(C'_w, \prec_i)$, we show that for any pair ρ and ρ' of normalised corrections in ψ_w, the following holds:

$$\rho \prec_s \rho' \text{ if and only if } conv(\rho) \prec_i conv(\rho')$$

We split the proof into showing first $\rho \prec_s \rho' \Rightarrow conv(\rho) \prec_i conv(\rho')$ and second $\rho \prec_s \rho' \Leftarrow conv(\rho) \prec_i conv(\rho')$. Therefore, let $\rho' = (\tau_0, \ldots, \tau_{j-1}, \tau_j, \tau_j, \ldots, \tau_m)$.

$\Rightarrow$: We know that every scattered subcorrection of the correction ρ' can be created by omitting at least one edit operation. Therefore, in the case $\rho \prec_s \rho'$, we can assume, without loss of generality, that for ρ we omit only the edit operation τ_j ($\rho' = (\tau_0, \ldots, \tau_{j-1}, \tau_j, \ldots, \tau_m)$), because of the transitivity of the subcorrection order. This implies that $conv(\rho)$ and $conv(\rho')$ only differ in one indexless edit operation. Let τ_i this indexless edit operation in $conv(\rho)$ and let τ'_i this indexless edit operation in $conv(\rho')$. We consider the different types of these edit operations and show that $\tau'_i \prec_{ie} \tau_i$ applies to all of them.

- **Replacement** If $\tau_j = a_j/_j b$ for some $b \in \Sigma \setminus \{a_j\}$, then we have that $\tau_i = a_j/b$ and $\tau'_i = a_j\rightarrow$. So we have that $\tau'_i \prec_{ie} \tau_i$.
- **Deletion** If $\tau_j = a_j\downarrow_j$, then we have that $\tau_j = a_j\downarrow$ and $\tau'_j = a_j\rightarrow$. So we have that $\tau'_i \prec_{ie} \tau_i$.
- **Insertion** If $\tau_j = b\uparrow_j$ for some $b \in \Sigma$, then we have that $\tau_j = xby\uparrow$, with $x, y \in \Sigma^*$, and $\tau'_j = xy\uparrow$. So we have that $\tau'_i \prec_{ie} \tau_i$.

So we have shown for all types of edit operations that τ_j can be taken, that we have $\tau_j \prec_{ie} \tau'_j$, and therefore it applies that: $\rho \prec_s \rho' \Rightarrow conv(\rho) \prec_i conv(\rho')$.

$\Leftarrow$: We prove this case by its contraposition: $\rho \not\prec_s \rho' \Rightarrow conv(\rho) \not\prec_i conv(\rho')$. That ρ is not a scattered subcorrection of ρ' ($\rho \not\prec_s \rho'$) we have either that $\rho' \prec_s \rho$ or that $\rho \neq_s \rho'$, where for the first we have already shown that $conv(\rho') \prec_i conv(\rho)$ and because $\prec_i$ is antisymmetric we have that $conv(\rho) \not\prec_i conv(\rho')$.

So it remains to show that if $\rho \neq_s \rho'$, which means that we have neither $\rho' \prec_s \rho$ nor $\rho \prec_s \rho'$, then $conv(\rho) \not\prec_i conv(\rho')$ holds. That for the two corrections ρ and ρ', which are normalised, $\rho \neq_s \rho'$ applies. Either the same letter of the word is replaced by another letter, or there is at least one edit operation contained in each correction that is not contained in the other correction. Considering the possible index shifts of the insert operations.

For the case where the same letter of the word is replaced by another letter, we assume that $a_i/_i b$ is contained in ρ and $a_i/_i b'$ is contained in ρ' with $b, b' \in \Sigma$ and $b \neq b'$. We have that v_i in $conv(\rho)$ is a/b and for $conv(\rho)$ is $v_i = a/b'$ because we have that $a/b \neq_{ie} a/b'$, this implies that $conv(\rho) \not\prec_i conv(\rho')$.

For the case where there is at least one edit operation contained in each correction that is not contained in the other correction. We have that for one pair of indexless edit operations $\prec_i$ and for another pair $\succ_i$ holds, or if both edit operations are an insertion between the same letters, we have that for the corresponding insertion words that $\neq_i$ holds. Therefore, we also have for this case that $conv(\rho) \not\prec_i conv(\rho')$.

So we have shown that $\rho' = (\tau_0, \ldots, \tau_{j-1}, \tau_j, \tau_j, \ldots, \tau_m)$ also applies and therefore the conversion function is an order-embedding. □

We have shown that the conversion function is an order embedding. We can use indexless corrections instead of corrections in the rest of this work, because we can convert any normalised correction into an equivalent simplified indexless correction so that the order of the corrections is preserved.

Example 3.6. In Example 3.4 we have shown that the correction $\rho' = [n/_2(,)\uparrow_4, n\uparrow_1, n\uparrow_0]$ is a scattered subcorrection of $\rho = [n/_2(,)\uparrow_4,)\uparrow_1, n\uparrow_1, (\uparrow_1, n\uparrow_0]$ ($\rho' \prec_i \rho$). In addition, in Example 3.3 we have shown that $conv(\rho) = [n\uparrow, +\rightarrow, (n)\uparrow, +\rightarrow, \varepsilon\uparrow, n/(, \varepsilon\uparrow, n\rightarrow,)\uparrow]$. Applying the conversion function to the correction ρ' we get the indexless correction $conv(\rho') = (n\uparrow, +\rightarrow, n\uparrow, +\rightarrow, \varepsilon\uparrow, n/(, \varepsilon\uparrow, n\rightarrow,)\uparrow)$, for which we have in Example 3.5 shown that it is a subcorrection of $f(\rho)$ ($f(\rho') \prec_i f(\rho)$). Therefore, we have an order embedding for the pair of corrections ρ and ρ' into indexless corrections.

4 Computing all Corrections

In Section 2.9 we have presented how we can compute *one* correction (the correction with the minimal number of edit operations) for a word and a context-free grammar. In this section, we will consider how we can compute *all corrections* for a word and a context-free grammar. Note that in general the set of all corrections is infinite. Therefore, we need a data structure that can store this infinite set and an terminating algorithm to compute such a data structure. The data structure we present in this section (Sec. 4.1) is a *correction shared packed parse forest* (CSPPF), which is an adaptation of shared packed parse forest (Sec. 2.5) to store indexless corrections. The basic idea is, that we label the *leaves* of the CSPPF with *indexless edit operations*, such that recurring sequences of insertions can be represented by cycles in the CSPPF. Then each syntax tree contained in a CSPPF represents exactly one indexless correction.

In addition to CSPPF itself, we present in Section 4.2 an adaptation of the generalized Earley Parser, called *all-corrections Earley Parser*, to compute a CSPPF instead of an SPPF. For this algorithm, we prove in Section 4.3 that it works correctly and compute a CSPPF that contains all indexless corrections for a given word and a context-free grammar. Finally we analyse in Section 4.4 the complexity of the algorithm and show that the algorithm computes an CSPPF of at most $\mathcal{O}\left(n^3\left(|N| + \sum_{A\to\gamma\in P}|\gamma|\right)\right)$ nodes and $\mathcal{O}\left(n^3\left(|N| + \sum_{A\to\gamma\in P}|\gamma|\right)\right)$ edges in time $\mathcal{O}\left(n^3\left(\sum_{A\to\gamma\in P}|\gamma|\right)\left(|P| + \sum_{A\to\gamma\in P}|\gamma|\right) + n^2|P|^2\right)$ for a word of length n and a context-free grammar $G = (N, \Sigma, P, S)$.

Idea to adapt the Earley Parser Before we introduce the all-corrections Earley Parser (Sec. 4.3), we introduce an adaptation of the Earley Parser from Section 2.3 to compute corrections. The purpose of this adaptation of the Earley Parser is to give a first idea of the all-corrections Earley Parser, and to show the requirements that correction shared packed parser forests (Sec. 2.5) must fulfill in order to store an infinite set of corrections.

To adapt the Earley Parser to compute corrections, we extend the Earley items and store, in addition to the common attributes of an item, a correction. For example, the Earley item $[E \to E + \bullet T, 2, (n\uparrow_3)]$ indicates that the algorithm has added an n to position 3 of the word to be able to parse $E+$. In addition, to the three Earley rules: *scanner*, *predictor*, and *completer* we add three new rules: *replacement*, *deletion*, and *insertion*. Each of these new rules perform an edit operation on the word. Note instead of adding the three new rules, it is also possible to use the covering grammar as in [1] for the given context-free grammar.

Let $G = (N, \Sigma, P, S)$ be a context-free grammar, $w \in \Sigma^*$ with $w = a_0 a_1 \dots a_{n-1}$ and $|w| = n$, let Q_i and Q_j be Earley sets with $0 \leq i \leq n$ and $0 \leq j \leq n$, and let $\alpha, \gamma, \gamma \in (N \cup \Sigma)^*$, $b \in \Sigma$, $B \in N$. Then the Earley rules that we add to adapt the Early Parser are formally as follows:

Replacement (R): If $[A \to \alpha \bullet b\gamma', j, \rho]$ in Q_i and $a_i \neq b$, add $[A \to \alpha b \bullet \gamma', j, \rho \cdot a_i/_i b]$ to Q_{i+1}.

Deletion (D): If $[A \to \alpha \bullet \gamma, j, \rho]$ in Q_i, add $[A \to \alpha \bullet \gamma, j, \rho \cdot a_i \downarrow_i]$ to Q_{i+1}.

Insertion (I): If $[A \to \alpha \bullet b\gamma', j, \rho]$ in Q_i, add $[A \to \alpha b \bullet \gamma', j, \rho \cdot b\uparrow_i]$ to Q_i.

Intuitively, the *replacement* rule is used when the next terminal symbol in the Earley item is different to the next symbol of the word. Then, the symbol of the word can be replaced with the next symbol in the current Earley item, and thus the parser can scan the replaced letter. The *delete* rule can be applied to any Earley item to delete the next symbol of the

word. This means that the parser does not have to scan any more letter at the current position and the Earley item can be moved to the next Earley set. The *insertion* rule can be applied to any Earley item for which the sentential form to the right of the bullet points starts with a terminal symbol. This terminal symbol can be inserted before we scan the next symbol of the word, and therefore doesn't need to be scan by the parser as next.

In addition to the new rules, the existing rules for calculating the corrections must also be adapted slightly. The *completion* rule must concatenate the corrections of the two Earley items used. The Earley items generated by the *predictor* rule are annotated with an empty correction sequence, and the *scanner* rule leaves the corrections unchanged. With all rules, however, it should be noted that when adding a new operation or concatenating two corrections, the indices of the individual operations must be adjusted accordingly. For example, $(a{\downarrow_1}) \cdot (b{\uparrow_3}) = (a{\downarrow_1}, b{\uparrow_2})$.

With the adapted version of the Earley Parser briefly presented here, the calculated *corrections* ρ can be taken from the *final Earley items* of the new start symbol S' in the last Earley set Q_n with a pointer back to position 0 of the input word ($(S' \to S\bullet, 0, \rho) \in Q_n$). This approach has *two major problems*:

1. Not all corrections are computed.
2. The algorithm only terminates if the target language is finite, which is not true in practice for the most used context-free languages. Because in the case that the target language contains an infinite set of words, at any time during parsing further Earley items with corrections that correct the input word to a corrected word that have not yet been taken into account can be added at any time during parsing.

The first problem can be solved by storing only *normalised corrections* or *indexless corrections*, since we have already discussed in Section 2.8 and Section 3, it is sufficient to consider only one of this types of corrections.

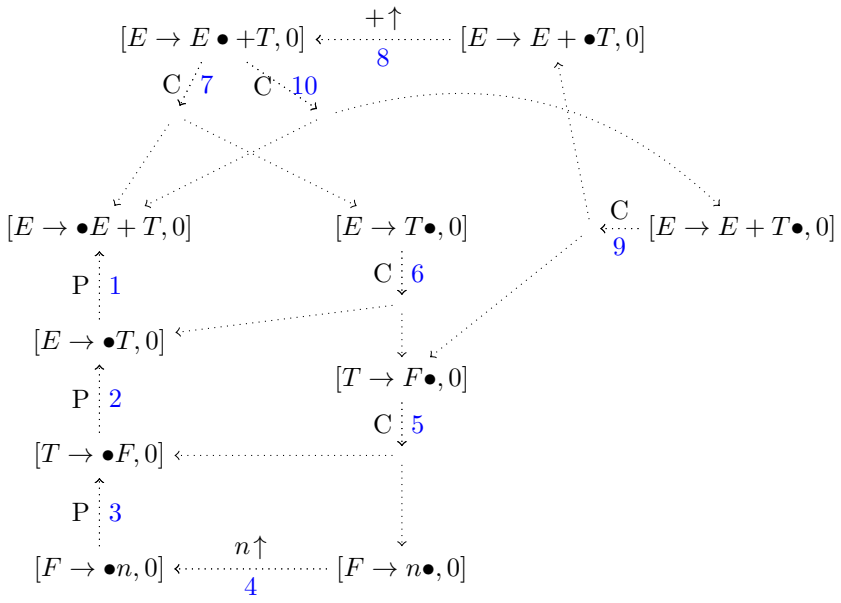
The second problem can be solved as for the CYK-Algorithm version to compute minimal corrections [3] by storing only an Earley items with minimal corrections. However, this approach requires a lot of comparisons between the corrections in order to filter out only the minimal corrections at each step. Furthermore, the parser is strongly adapted to the minimality definition used and only terminates when the set of minimal corrections is finite. In addition, this approach is only possible if pairs of not yet fully calculated corrections are comparable regarding the used definition of minimality. And at least it is only possible to compute all minimal corrections and not all corrections.

Therefore, we use the idea introduced here to adapt the Earley Parser with the three new rules (*replacement*, *deletion*, and *insertion*) to compute in Section 4.2 a data structure (correction shared packet parse forest) independent of the used definition of minimality, which contains all indexless corrections for a word and a context-free grammar.

4.1 CSPPF: Correction Shared Packet Parse Forest

In this section we define *correction shared packet parse forest* (Definition 4.1) as a data structure for storing a possible *infinite set* of *indexless corrections* for a word and a context-free grammar. Therefore, we first discuss what properties a data structure needs to store an infinite set of corrections. Building on the definition of CSPPFs we consider which corrections are contained in a correction shared packed parse forest.

Properties of a Data Structure for Storing Infinite Corrections For a word over some alphabet Σ of length n, we already know from the definition of indexless corrections (Definition 3.2) that every indexless correction has exactly length $2n + 1$ and at every even position there is a deletion, replace, or read operation. At each even position, we can perform $|\Sigma| - 1$ different replacement operations that don't lead to the undefined value ($\perp$).



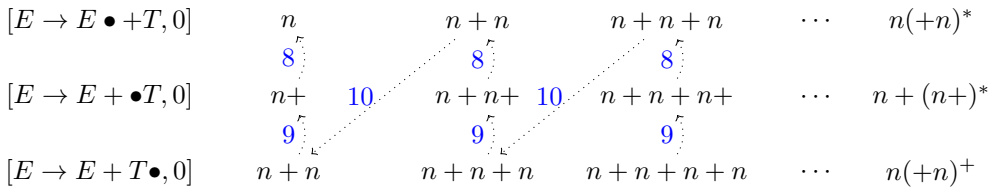


Figure 9: (Top) A set of Earley items contained in the Earley set Q_0 for the language L_{aexpr} from Example 2.3 and an Earley Parser version with the additional insertion rule. The dotted outgoing edges from an item indicate which rule (label of the edge: C = *completer*, P = *predictor*, and $x\uparrow$ as the *insertion* the terminal symbol x) creates the Earley item. (Down) For three of the Earley items above, all words that can be inserted before the first letter of any input word and stored with the corresponding Earley item. In addition, the dotted edges indicate which rule must be applied to which item/correction to get this correction. Note: Because we consider insertion for the first letter of the input word, these Earley items are the same for each word.

Therefore, at each even position, we can perform $|\Sigma| + 1$ different edit operations so that we have for an input word of length n, regardless of insertions, only $(|\Sigma| + 1)^n$ different combinations of defined indexless corrections. However, we can *insert an infinite* set of different insert words *at each odd position* of an indexless correction.

For example, for the input word $+n$ and the context-free grammar G_{aexpr} from Example 2.3, we can insert at position 0 every word $x_0 \in L(G_{aexpr})$ such that the corrected word $x_0 + n \in L(G_{aexpr})$. But why is it possible to insert an infinite set of inserts here? To answer this, we look at the Earley Parser with corrections with the additional insertion rule already defined above, and consider how all insertion words (x_0) at position 0 that are contained in the regular expression $n(+n)^*$ are computed in the parsing process. All Earley items that can contain such an insertion as an indexless correction or are used to compute the Earley item with such a correction are shown in Figure 9.

In this simple example, we can observe that after some time, the Earley items are repeated, and only the corrections stored in addition to the Earley items become longer respectively define an regular expression. But we can observe that the insertions stored in the Earley items are created by cyclic repetitions of applying the same rules to the same Earley items. This indicates that the possible insertions don't have to be stored directly in the Earley items but can also be determined by unwinding loops. This makes it possible to store an infinite set of indexless corrections in a finite data structure.

Correction Shared Packet Parse Forest A *correction shared packet parse forest* is a data structure to store an (possible) infinite number of corrections for a word w and a context-free grammar G and is an adaptation of shared packed parse forest (Sec. 2.5) to store indexless correction.

For this purpose, the *leaves* of a CSPPF are labeled with *indexless edit operations* (Definition 3.1), so that each syntax tree contained in a CSPPF represents an indexless correction. Whereby we only allow the insertion of single letters An insert word is formed by concatenating of the leaves of the single inserted letters. The reason for this is explained after the following definition of CSPPFs.

Definition 4.1. Let $G = (N, \Sigma, P, S)$ be a context-free grammar, $w \in \Sigma^*$ with $w = a_0 a_1 \dots a_{n-1}$ and $|w| = n$. A *correction shared packed parse forest* (CSPPF) for w and G is a labelled directed graph (V, E), where V is the set of vertices and E is the set of edges. The set of vertices V of an CSPPF is a subset of the 4 disjoint set of (possible) nodes: *edit nodes* (written as V_E) *symbol nodes* (written as V_S), *intermediate nodes* (written as V_i) and *packed nodes* (written as V_p), defined as follows;

$$
\begin{aligned}
V \subseteq\ & (V_E \cup V_{INS} \cup V_S \cup V_I \cup V_P) \\
V_E = {} & \{(x, i, i+1) \mid 0 \le i < n-1 \text{ and } x \in \{a_i\rightarrow, a_i\downarrow\}\} \\
& \cup \{a_i \not b \mid b \in \Sigma \setminus \{a_i\} \text{ and } 0 \le i < n-1\} \\
& \cup \{(b\uparrow, i, i) \mid b \in \Sigma \cup \{\varepsilon\} \text{ and } 0 \le i \le n\} \\
V_S = {} & \{(A, i, j) \mid A \in N \text{ and } 0 \le i \le j \le n\} \\
V_I = {} & \{(A \rightarrow \alpha \bullet \beta, i, j) \mid A \in N, \alpha \in (N \cup \Sigma)^*, \beta \in (N \cup \Sigma)^+, \text{ such that } A \rightarrow \alpha\beta \in P \\
& \quad \text{and } 0 \le i \le j \le n\} \\
V_P = {} & \{(A \rightarrow \alpha \bullet \beta, i, k, j, 0) \mid A \in N, \alpha, \beta \in (N \cup \Sigma)^* \text{ such that} \\
& \quad A \rightarrow \alpha\beta \in P \text{ and } 0 \le i \le k \le j \le n\} \\
& \cup \{(A \rightarrow \alpha \bullet \beta, i, k, j, 1) \mid A \in N, \alpha, \beta \in (N \cup \Sigma)^* \text{ such that} \\
& \quad A \rightarrow \alpha\beta \in P, 0 \le i \le k < n \text{ and } j = k+1\}
\end{aligned}
$$

We call the pair (i, j) contained in symbol nodes, packed nodes and intermediate nodes and the pair $(i, i+1)$ in the leaf nodes the *extent* of this nodes. The natural number k of packed nodes is called the *pivot*.

The set of edges E of an CSPPF is a subset of 6 disjoint sets of (possible) edges, where each of them describes edges from a particular node type to a particular node type. The different sets of edges are: edges from symbol nodes to packed nodes (written as E_{SP}); intermediate nodes to packed nodes (written as E_{IP}); packed nodes to symbol nodes (written as E_{PSR} and E_{PSL}); packed nodes to edit nodes (written as E_{PER}) and packed nodes to intermediate nodes (written as E_{PIL}).

$$
\begin{aligned}
E \subseteq\ & (E_{SP} \cup E_{IP} \cup E_{PSR} \cup E_{PER} \cup E_{PSL} \cup E_{PIL}) \\
E_{SP} = \{&(u,v) \mid u = (A,i,j) \in V_S, v = (A \to \alpha \bullet \varepsilon, i, k, j, z) \in V_P \text{ with } 0 \le z \le 1 \\
&\text{ such that } i \le k \le j \text{ and } (z = 1 \to j = k+1)\} \\
E_{IP} = \{&(u,v) \mid u = (A \to \alpha \bullet \beta, i, j,) \in V_S, v = (A \to \alpha \bullet \beta, i, k, j, z) \in V_P \text{ with} \\
&0 \le z \le 1 \text{ such that } i \le k \le j \text{ and } (z = 1 \to j = k+1)\} \\
E_{PSR} = \{&(u,v) \mid u = (A \to \alpha \bullet \beta, i, k, j, 0) \in V_P, v = (A', k, j) \in V_S, \gamma \in (N \cup \Sigma)^* \\
&\text{ such that } \alpha = \gamma A'\} \\
E_{PER} = \{&(u,v) \mid u = (A \to \alpha \bullet \beta, i, k, j, 0) \in V_P, v = (a_{i'} \to, i', i'+1) \in V_E, \gamma \in (N \cup \Sigma)^* \\
&\text{ such that } \alpha = \gamma a_{i'}, k = i' \text{ and } j = i'+1\} \\
\cup\ \{&(u,v) \mid u = (A \to \alpha \bullet \beta, i, k, j, 0) \in V_P, v = (a_{i'} \not b, i', i'+1) \in V_E, \gamma \in (N \cup \Sigma)^* \\
&\text{ such that } \alpha = b\gamma, k = i' \text{ and } j = i'+1\} \\
\cup\ \{&(u,v) \mid u = (A \to \alpha \bullet \beta, i, k, j, 0) \in V_P, v = (b\uparrow, i', i') \in V_E, \gamma \in (N \cup \Sigma)^* \\
&\text{ such that } \alpha = \gamma b \text{ amd } k = j = i'\} \\
\cup\ \{&(u,v) \mid u = (A \to \alpha \bullet \beta, i, k, j, 1) \in V_P, v = (a_{i'}\downarrow, i', i'+1), \\
&\text{ such that } k = i' \text{ and } j = i'+1\} \\
E_{PSL} = \{&(u,v) \mid u = (A \to \alpha \bullet \beta, i, k, j, 0) \in V_P, v = (A', i, k) \in V_S, \gamma \in (N \cup \Sigma)^* \\
&\text{ such that } \alpha = A'\gamma\} \\
E_{PIL} = \{&(u,v) \mid u = (A \to \alpha \bullet \beta, i, k, j, 0) \in V_P, v = (A \to \alpha' \bullet \beta', i, k) \in V_I, \gamma \in (N \cup \Sigma)^* \\
&\text{ such that }, \alpha = \alpha'\gamma \text{ and } \beta' = \gamma\beta\} \\
\cup\ \{&(u,v) \mid u = (A \to \alpha \bullet \beta, i, k, j, 1) \in V_P, v = (A' \to \alpha \bullet \beta, i, j-1) \in V_I\}
\end{aligned}
$$

We can observe that the definition of CSPPFs largely follows the definition of SPPFs (Definition 2.5). Thus, symbol nodes remain unchanged and only minor adjustments are made to intermediate nodes and packed nodes. Therefore, the meaning of nodes of this type remains unchanged. In addition, we can also observe that a CSPPF is also a *bipartite* graph, where one independent set is the set of packed nodes and the other independent set are all nodes that are non of type packed node. The main difference is that we use edit nodes instead of leave nodes in CSPPFs.

Due to the similarity of CSPPFs to SPPFs, we also use the visualisation conventions for SPPFS from Section 2.5 for CSPPFs. In addition, in the following we only consider the changes to the individual node types of CSPPFs compared to SPPFs.

- **Intermediate nodes** (V_I): The only small difference for intermediate nodes is that we allow intermediate nodes in CSPPFs where α has length 0 ($\alpha = \varepsilon$). As we will see in Section 4.2, this adjustment is necessary to be able to represent indexless corrections that consist of consecutive deletion operations.
- **Edit nodes** (V_E): An edit node can only appear as the right successor of a packed node and represent the execution of an indexless edit operation on w with respect to

G and, unlike all other node types, edit nodes have no successors. Therefore all syntax trees stored in a CSPPF have only leaves of this type.

We have that the extent for edit nodes that represent an replacement, deletion or read operation is $(i, i+1)$ and the extent of nodes that represent insertion operations is (i, i). Note that only insert words of length 1 can be used to label a node.

The reason for the different extents is that with an insert operation, no letter of the word is scanned. Therefore, after an insert operation, we also have to derive the same letter of the input word as before the insert operation. This is not the case for the other operations; here we have to scan the next letter of the input word.

The reason we can only label nodes with insertions of only 1 terminal symbol is that the insertion rule only considers the next terminal symbol that the item requires. Note that longer insertions are the result of *concatenating* the *inserted letters* of *leaves* labeled with the *same extent* from left to right.

One might get the impression that it would make sense to label leaves with longer inserts. However, this is not the case, as all nodes in between are also required in a CSPPF, that contains all indexless corrections, because every symbol and intermediate node also has at least one child family with a read operation node, an delete operation node or a replacement operation node. This would therefore only lead to more nodes in the CSPPF. Furthermore, this type of representation of indexless insertion operations has the additional advantage that we can easily deal with production rules of the form $A \to \varepsilon$ (with $A \in N$). In this case, we can insert the letter ε, as concatenating a word with ε leaves the word unchanged.

- **Packed nodes** (V_P): Packed nodes in CSPPFs are tuples of length 5, whereas in SPPFs they are only defined as tuples of length 4. The simple reason for this is that there must be packed nodes with the same labelling for the definition as 4 tuples, as otherwise each family cannot be represented by exactly one packed node. More detailed for a packed node $(A \to \alpha \bullet \beta, i, k, j, z)$ with $A \in N, \alpha, \beta \in (N \cup \Sigma)^*$, $0 \leq z \leq 1$ and $\alpha = \gamma x$ with $\gamma \in (N \cup \Sigma)^*$ and $x \in (N \cup \Sigma)$ this packed node can have the following right successor nodes depending on k, j and x.
  - If $x \in \Sigma$ and $k = j$ then the right successor node can only be represent an insertion.
  - If $x \in N$ and $k \neq j - 1$ then the right successor node can only be of type symbol node.
  - If $x \in N$ and $k = j - 1$ then the right successor can be of type symbol node or represent an deletion operation.
  - If $x \in \Sigma \setminus \{a_{j-1}\}$ and $k = j - 1$ then the right successor can represent and replacement operation or an deletion operation.
  - If $x = a_{j-1}$ and $k = j-1$ then the right successor can represent an read operation or an deletion operation.

  This shows that in the case $k = j-1$ the right successor node can be a node representing a deletion operation and another node depending on x. Therefore, the same packed node can represent two different families. To solve this problem with z, another value has been added to the definition of packed nodes in CSPPFs.

  Where in the case that $z = 0$ the right child cannot be a node that represent a deletion operation and in the case that $z = 1$ the right child can only be a node that represent a deletion operation. This means that each packed node again represents exactly one family.

Indexless Corrections contained in a CSPPF We have mentioned above that each syntax tree contained in a CSPPF represents an indexless correction. It remains to be clarified which syntax trees are contained in a CSPPF and which indexless corrections such a syntax tree represents.

The existence of a syntax tree in a correction shared packed parse forest is defined as the existence of a syntax tree in a SPPF (Definition 2.6) with the only difference that the leaves are labeled with indexless edit operations instead of letters of the word.

To determine for a syntax tree contained in a CSPPF which indexless correction is represented by that syntax tree, we consider the leaf front of this syntax tree. To determine which form has a leaf front of a syntax tree, let's look at some of the properties of a CSPPF. We know from the definition of CSPPF (Definition 4.1) that for a packed node with two children, extent (i, j) and pivot k the left successor node has the extent (i, k) and the right successor node has the extent (k, j). Remember that in order to obtain a syntax tree from an SPPF, we have selected exactly one packed node as successor node for each intermediate node and each symbol node (Definition 2.6). These two facts imply that for all adjacent leaves of any syntax tree, the extent of the left leaf is (j, k) and the extent of the right leaf is (k, i) with $0 \leq j \leq k \leq i \leq n$. In addition, the definition of CSPPF requires that for all leaves representing an insert operation, the extent must be (i, i), and for all leaves representing any other edit operation, the extent must be $(i, i + 1)$. This means that for each syntax tree in a CSPPF for a word of length n the leaves front read from left to right are of the following form:

- A possibly empty sequence of nodes represent an insertion operation with extent $(0, 0)$.
- A edit node $(x, 0, 1)$ with $x \in \{a_0\rightarrow, a_0\downarrow, a_0/b\}$ with $b \in \Sigma \setminus \{a_0\}$.
- A possibly empty sequence of nodes represent an insertion operation with extent $(1, 1)$.
- $\cdots$
- A edit node $(x, n, n-1)$ with $x \in \{a_{n-1}\rightarrow, a_{n-1}\downarrow, a_{n-1}/b\}$ with $b \in \Sigma \setminus \{a_{n-1}\}$.
- A possibly empty sequence of nodes represent an insertion operation with extent (n, n).

More formally, by the following corollary.

Corollary 4.1. *Let $G = (N, \Sigma, P, S)$ be a context-free grammar, $w \in \Sigma^*$ with $w = a_0 a_1 \dots a_{n-1}$, let ψ be a CSPPF for w and G. Then the indexless edit operations of leaf front of each syntax tree τ contained in ψ ($\tau \; \xi \; \psi$) is ordered from left to right of the following form:*

$$x_{0_0}\uparrow, \dots, x_{0_{m_0}}\uparrow, v_0, x_{1_0}\uparrow, \dots, x_{1_{m_1}}\uparrow, v_1, \dots, v_{n-1}, x_{n_0}\uparrow, \dots, x_{n_{m_n}}\uparrow$$

with $v_i \in \{a_i\downarrow, a_i\rightarrow, a_i/b\}$ for $0 \leq i < n$, $b \in \Sigma$ and $x_{i_j} \in \{c\uparrow\}$ with $c \in \Sigma \cup \{\varepsilon\}$.

It is obviously that the leaf front of a syntax tree contained in a CSPPF *defines an indexless correction*. To do this, all previously independent insert operations on the same positron simply need to be concatenated into one insert word. For example the leaf front from Corollary 4.1 defines the following indexless correction:

$$(x_{0_0} \dots x_{0_{m_0}}\uparrow, v_0, x_{1_0} \dots x_{1_{m_1}}\uparrow, v_1, \dots, v_{n-1}, x_{n_0} \dots x_{n_{m_n}}\uparrow)$$

Therefore, all indexless corrections contained in a CSPPF are defined by the leaf fronts of the syntax trees in that CSPPF.

Definition 4.2. Let ψ be a CSPPF, then $C(\psi)$ is the set of all indexless corrections contained ψ.

$$C(\psi) = \{\rho \mid \text{they are an syntax tree } \tau \ \xi \ \psi \text{ and the leaf front of } \tau \text{ defines } \rho\}$$

Example 4.1. An example of a correction shared packed parse forest for the word + and the context-free grammar G_{aexpr} from Example 2.3 is shown in Figure 10. As this CSPPF can look confusing at first glance, let us take a closer look at some of its parts.

First we look at the symbol node $(E, 0, 0)$ on the left in the center. For this node, we can consider that from its subtree we can derive all words of the language $L(G_{aexpr})$ as insertions before the first letter of the input word because the loops in the subtree always return to this node.

A simple correction is the correction $(\varepsilon\uparrow, +/n, \varepsilon\uparrow)$ which can be derived from the forest as follows: $(E, 0, 1) \to (T, 0, 1) \to (F, 0, 1) \to (T, 0, 1) \to (+/n, 0, 1)$. Next to this correction, we have the correction $(\varepsilon\uparrow, +\downarrow, n\uparrow)$, which can be simplified to the previous correction. Such places where the CSPPF contains a non-simplified correction and the same simplified correction can be found at often in the CSPPF. Note: In Section 5.3 we consider how to remove such non-simplified corrections from an CSPPF.

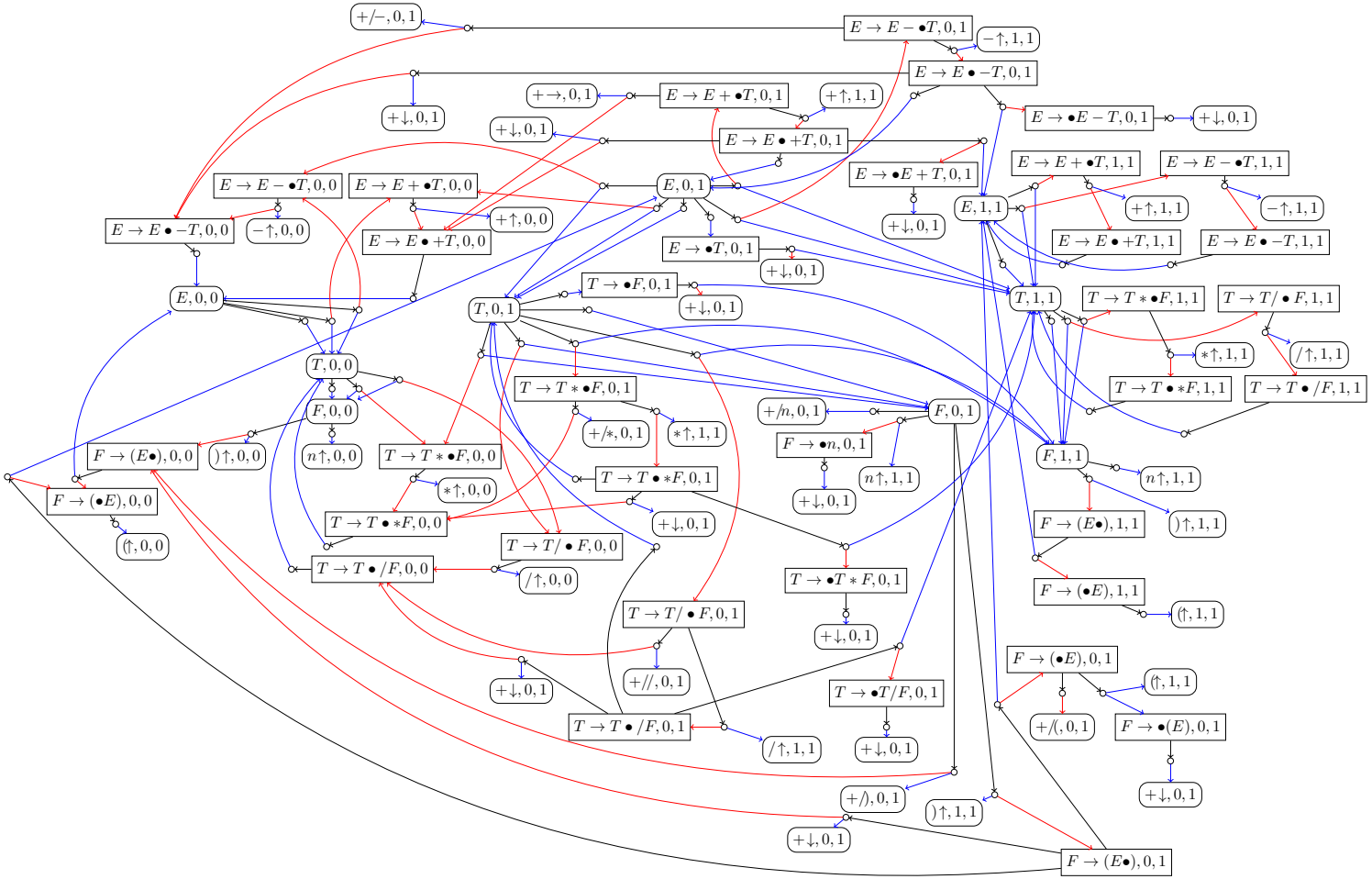


Figure 10: A correction shared packed parse forest for the word + and the context-free grammar G_{aexpr}.

4.2 All-Correction Earley Parser

We have introduced correction shared packed parse forests as a data structure to store a possible infinite set of indexless corrections. In this section, we will look at how we can compute a CSPPF that contains all indexless corrections for a word and a context-free grammar.

At the beginning of Section 4 we presented a way to compute corrections by adding the rules *replacement*, *deletion*, and *insertion* to the conventional rules *completer*, *scanner*, and *predictor* of the Earley Parser and storing the correction as part of an Early item. One of the major problems with this approach is that the algorithms don't terminate if the set of correction is infinite. This is because we can computed an infinite number of Earley items with corrections for the reason that, for each item without correction, we have to store a possible infinite number of corrections with that item. However, with CSPPFs we now know a way to store an infinite number of corrections. This allows us to store only one node from the CSPPF for each Earley item, instead of an infinite number of corrections. So that the infinite number of corrections is represented by the subtrees below that node.

Therefore, the *all-correction Earley Parser* presented in this section uses *Earley items with nodes*(Definition 2.7) as the generalised Earley Parser(Sec. 2.6), with the small difference that the nodes are from a CSPPF instead of an SPPF. Therefore, when we refer to Earley item in the rest of this section, we always mean a *Earley item with node*.

In addition, the idea is to adapt the generalised Earley Parser (Alg. 12) to compute a CSPPF instead of an SPPF by incorporating the ideas from the *replacement*, *deletion*, and *insertion* rules from above into the algorithm. By applying the individual rules, the CSPPF is built bottom-up by the all-correction Earley parser.

Building a CSPPF But before we look at the algorithm in detail, let us look at the process of building up the CSPPF by using one of the *replacement*, *deletion*, or *insertion* rules. Figure 11 presents these rules and introduces the build process for each rule graphically. The rules in Figure 11 differ from the rules presented at the beginning of Section 4 only in that the Earley items store nodes of a CSPPF instead of corrections. Therefore, for each rule, we will only look at the CSPPF build process on the right of Figure 11. All three rules are applied to the Earley item, which belongs to the lower left node. This means that both the item and the corresponding node must already exist. The rule then creates a new node that represents the edit operation, the lower right node, and an node, that has the other two nodes as child family and to which the Earley item created by the rule belongs. However, it should be noted that all nodes are only created if they are not already contained in the CSPPF. If this is the case, only the corresponding edges are added. The build process for the *scanner*, *completer*, and *predictor* rules remains unchanged for CSPPF compared to SPPF (see Sec. 2.6).

Example 4.2. For a better understanding of the CSPPF construction process, we will look at some steps of this process for the context-free grammar G_{aexpr} from Example 2.3 and the word n. The steps considered in the following result in the CSPPF shown in Figure 12, where the numbered dotted boxes contain exactly the nodes of the corresponding step. Note that all the nodes we create here are included in the CSPPF of Figure 10 and all the calculated *Earley items with node* of the all-correction Earley Parser are shown in Figure 24.

1. We assume that, based on the initial Earley item, all *Earley items with node* resulting solely from the application of a series of *predictor* rules are already contained in the Earley set Q_0. Therefore, the item $[F \to \bullet, 0, null]$ is also contained in Q_0. We apply the *insertion rule* to this item. This adds the edit node $(n\uparrow, 0, 0)$ to the CSPPF and the Earley item $[S \to n\bullet, 0, v]$ to the Earley set Q_n. As the added item is a final item, the node $v = (F, 0, 0)$, which must also be added to the CSPPF, is of type Symbol Node. Finally, a child family must be added to the node v, which consists of the node

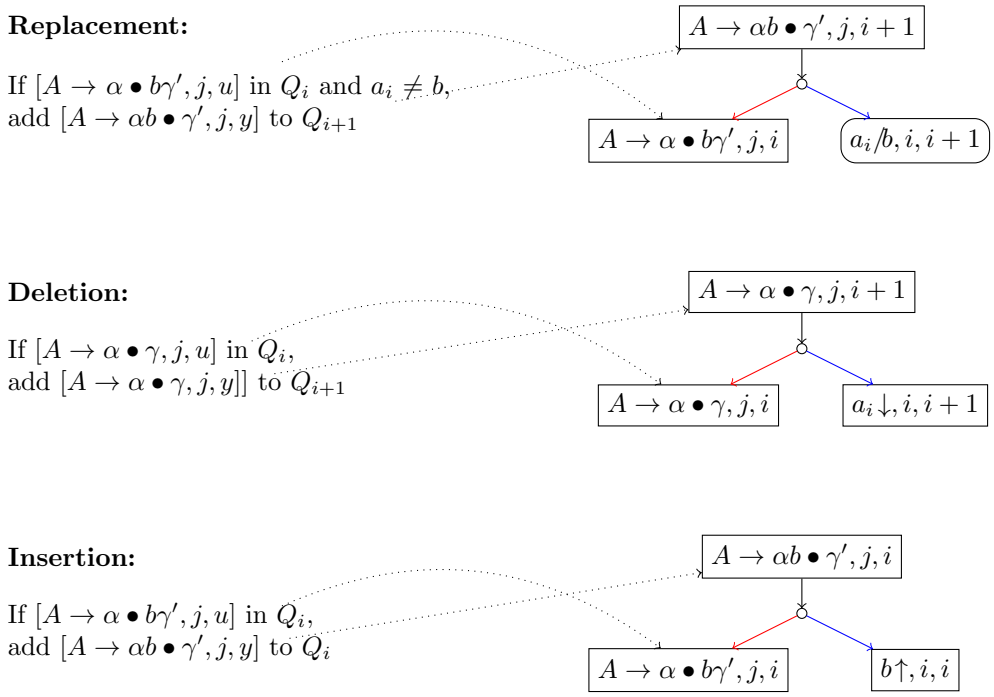


Figure 11: (Left) The rules *replacement*, *deletion*, and *insertion*. (Right) The corresponding nodes of a CSPPF for these rules. The dotted edges from left to right indicate the node in the CSPPF stored in the Earley item with node.

of the item $[F \to \bullet, 0, null]$ and the node of the edit operation. As the item node is the dummy node *null*, the family only consists of the node $(n\uparrow, 0, 0)$.

2. By assuming from the previous step that all items that only emerge from the initial item by applying a series of *predictor* rules are contained in the set Q_0. We know that the item $[T \to \bullet F, 0, null]$ must also be contained in Q_0. We also created the Earley item $[S \to n\bullet, 0, (F, 0, 0)]$ in the previous step. We apply the *completer* rule to these two items. The adjustments to the CSPPF resulting from the application of a *completer* rule are explained graphically in Figure 7. In this case, we must add the Earley item $[T \to F\bullet, 0, (T, 0, 0)]$ to the Earley set Q_0 and the node $(T, 0, 0)$ to the CSPPF. As in the previous step, the item $[T \to \bullet F, 0, null]$ is assigned to the dummy node *null*. Therefore, the child family of the node $(T, 0, 0)$ only consists of the node $(F, 0, 0)$.

3. In this step, the *completer* rule is applied to the final item $[S \to n\bullet, 0, (F, 0, 0)]$ created in the previous step and the item $[T \to \bullet T * F, 0, 0, null]$ contains in the Earley set Q_0. This adds the Earley item $[T \to T \bullet *F, 0, 0, v]$ with $v = (T \to T \bullet *F, 0, 0)$ to the Earley set Q_0 and the node v, which, since the item is not a final element, is of node type intermediate, is added to the CSPPF. In the child family of the node v there is only the node $(T, 0, 0)$.

4. We apply the *deletion* rule to the item $[T \to T \bullet *F, 0, 0, (T \to T \bullet *F, 0, 0)]$ created in the previous step. To do this, the edit node $(+\downarrow, 0, 1)$ must be added to the CSPPF. In addition, the Earley item $[T \to T \bullet *F, 0, 1, (T \to T \bullet *F, 0, 1)]$ must be added to the Earley set Q_1 and the corresponding node with the child family, consisting of $(T \to T \bullet *F, 0, 0)$ and $(+\downarrow, 0, 1)$, must be added to the CSPPF. In this step we have to add the newly created Earley item to the Earley set Q_1, because due to the deletion operation the first letter + of the word has been corrected.

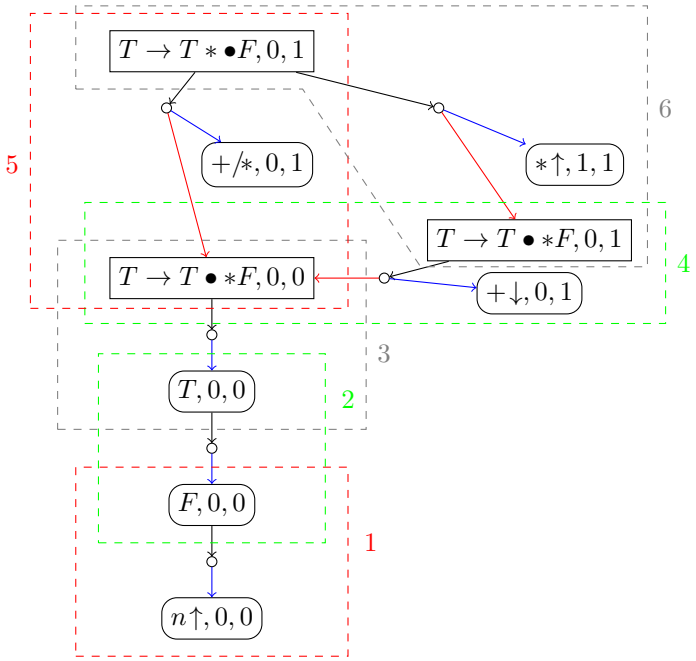


Figure 12: Example of the build-up process of a CSPPF for the word w and the grammar G_{aexpr} defined in Example 2.3.

5. In this step, we apply the *replacement* rule to the Earley item created in step 3. This results in the Earley item $[T \to T * \bullet F, 0, 1, v]$ with $v = (T \to T * \bullet F, 0, 1)$, which must be added to the set Q_1. In addition, the edit node $(+/*, 0, 1)$ and the node v with the child family consisting of $(T \to T \bullet *F, 0, 0)$ and $(+/*, 0, 1)$ must be added to the CSPPF.

6. We apply the *insertion* rule to the Earley item $[T \to T \bullet *F, 0, 1, (T \to T \bullet *F, 0, 1)]$ that we created in step 4. This results in the Earley item $[T \to T * \bullet F, 0, 1, (T \to T * \bullet F, 0, 1)]$. This item is already contained in Q_1 and the node $(T \to T * \bullet F, 0, 1)$ is already contained in the CSPPF. Therefore, we only need to add the edit node $(* \uparrow, 1, 1)$ to the CSPPF and add this node together with the node $(T \to T \bullet *F, 0, 1)$ as another child family of $(T \to T * \bullet F, 0, 1)$.

Algorithm After introducing the construction of CSPPF and the idea of the all-correction Earley Parser to compute a *correction* shared packed parse forest forest for a context-free grammar $G = (N, \Sigma, P, S)$ and a word $w \in \Sigma^*$. We look at the algorithm in detail. The all-correction Earley Parser is presented in Algorithm 3. The algorithm uses the auxiliary functions COMPUTE_EARLEY_SET (Alg. 5), INIT_NEXT_EARLEY_SET (Alg. 6), ADD_ITEM (Alg. 4) and MAKE_NODE (Alg. 7), which we will discuss in detail below. But first, we consider the meaning of the sets used by the algorithm.

- $N_\Sigma = \Sigma(N \cup \Sigma)^*$ is the set of all sentential forms over the grammar alphabet that *starts with a terminal* symbol.
- $\Sigma_N = N(N \cup \Sigma)^* \cup \{\varepsilon\}$ is the set of sentential forms that *starts with a nonterminal* or is empty.
- Q_i is the Earley set i.

- R is a queue of Earley items that still need to be considered in step i of the algorithm (computation of the Earley set Q_i).
- N is a subset of the current Earley set (Q_i), containing all Earley items that begin with a nonterminal to the right of the bullet ($N = \{[A \to \gamma \bullet \alpha, j, u] \mid [A \to \gamma \bullet \alpha, j, u] \in Q_i \text{ and } \alpha \in N_\Sigma\}$).
- N' is a subset of the set of all Earley items beginning with a nonterminal to the right of the bullet that are contained in the next Earley set ($N' \subseteq \{[A \to \gamma \bullet \alpha, j, u] \mid [A \to \gamma \bullet \alpha, j, u] \in Q_{i+1} \text{ and } \alpha \in N_\Sigma\}$).
- V is the set of nodes computed at the current step of the all-corrections Earley Parser (computation of the Earley set Q_i).
- The set H is a set of tuples of nonterminals and symbol nodes $H \subseteq \{(A, v) \mid A \in N, v = (A, i, i) \text{ with } 0 \leq i \leq n\}$. The occurrence of a tuple (A, v) in H represents that by computing the i-th Earley set we have computed an final Earley item $[A \to \alpha\bullet, j, u]$ for which $j = i$ applies. This means that all Earley items that are subsequently calculated for the Earley set $Q_i i$ can possibly be completed with the final item. However, as these items are not yet available in the current Earley set, we save the node and the nonterminal of the final item in the set H in order to check for all newly calculated Earley items in the Earley set Q_i whether it can be completed by an element in H.

Note: The sets N and N' aren't necessary. Instead, the Earley sets Q_i and Q_{i+1} can be used by filtering out the corresponding items. Another optimization is that the set V, which is used to check if a node already exists, only contains nodes calculated at the current step. This is because at step i we can only compute symbol or intermediate with extent (j, i) for some $j \leq i$.

Algorithm 3 ALL_CORRECTION_EARLEY_PARSER

Require: A context-free grammar $G = (N, \Sigma, P, S)$ and a word $w = a_0 a_1 \dots a_{n-1}$
Ensure: The CSPPF that contains all indexless corrections ρ corresponding to w and G such that $\rho(w) \in L(G)$

```
1: Q_i ← ∅ for all 0 ≤ i ≤ n
2: R, N', V, H ← ∅
3:
4: for all S → δ ∈ P do
5:     ADD_ITEM([S → •δ, 0, null], 0, true)
6: end for
7:
8: for i = 0 to n do
9:     H ← ∅, R ← Q_i, N ← N'
10:    N' ← ∅
11:
12:    COMPUTE_EARLEY_SET()
13:
14:    INIT_NEXT_EARLEY_SET()
15: end for
16:
17: return (S, 0, n)
```

After initializing the described sets, the algorithm (Alg. 3) adds for each production rule ($S \to \delta \in P$) with the start symbol on the left side an Earley item ($[S \to \bullet\delta, 0, null]$) with the dummy node *null* assigned to the first Earley set (Q_0). If the right side of the bullet starts with a nonterminal ($\delta \in N_\Sigma$), the algorithm adds this item additionally to the set N'.

Algorithm 4 ADD_ITEM

Require: An Earley Item with Node $[B \rightarrow \ \alpha \bullet \beta, h, u]$, the index of an Earley Set j, a boolean *init*

```
1:
2: if [B → α • β, h, u] ∉ Q_j then
3:     add [B → α • β, h, u] to Q_j
4:
5:     if ¬init then
6:         add [B → α • β, h, u] to R
7:     end if
8: end if
9:
10: if β ∈ N_Σ then
11:     if init then
12:         add [B → α • β, h, u] to N'
13:     else
14:         add [B → α • β, h, u] to N
15:     end if
16: end if
```

This step is similar to the generalized version of the Earley Parser, which also does not add a new start symbol (see App. A), with the difference that we have outsourced the adding processes to the auxiliary function ADD_ITEM (Alg. 4).

We have already observed for the Earley Parser (Sec. 2.3) that by applying the *predictor* and *completer* rules, the newly created Earley item is added to the current Earley set (Q_i) and by applying the *insertion* rule, the new item is added to the next Earley set (Q_{i+1}). For the new rules, we can see that the *insertion* rule creates an Earley item in the current Earley set(Q_i), and the *replacement* and *deletion* rules create items in the next Earley set (Q_{i+1}). Therefore, for each step of the parser, we can split the computation of the Earley items into two steps. First, all items in the current set are calculated. This is done by the auxiliary function COMPUTE_EARLEY_SET (Alg. 5). The second step is the calculation of the items of the next set. This is done by the auxiliary function INIT_NEXT_EARLEY_SET (Alg. 6). We repeat this procedure for all positions of the input word, starting from position 0.

At this point, we have calculated all Earley sets and the corresponding CSPPF. We can observe that we have for each production rule $S \rightarrow \gamma \in P$ a final Earley item $[S \rightarrow \gamma\bullet, 0, (S, 0, n)]$ in the last Earley set (Q_n). This is because we can correct every input word to at least one target word if the target language is not empty. Therefore, the node $(S, 0, n)$ is always contained in the computed CSPPF. In addition, the subtree of this node contains the set of all corrections. Therefore, we can always return the node $(S, 0, n)$ by our algorithm.

Auxiliary Functions The auxiliary function ADD_ITEM (Alg. 4) checks if a given *Earley item with node* is already contained in a certain Earley set. If it is not, it adds the item to that Earley set. If the variable $init = false$, then the algorithm also add the item to the queue R. The variable *init* is false iff it is called by the function COMPUTE_EARLEY_SET, otherwise it is false. This means that each Earley item is only added to the queue R once by the algorithm. This shows that the all-correction Earley parser terminates. In addition, the ADD_ITEM function fills the previously described sets N and N' with all Earley items that fulfil the condition for these sets.

The auxiliary function COMPUTE_EARLEY_SET (Alg. 5) takes every Earley item from a queue R of the current set and tries to apply on it the *predictor*, *completer*, or *insertion* rules and build up the SPPF bottom up as already described above by using the auxiliary

Algorithm 5 COMPUTE_EARLEY_SET

```
1: while R ≠ ∅ do
2:     Λ = [B → α • β, j, u] ← R.pop()
3:
4:     if β = Cβ′ then                                          ▷ with C ∈ N
5:         for all C → δ ∈ P do
6:             ADD_ITEM([C → •δ, i, null], i, false)            ▷ Predictor Rule
7:
8:             if (C, v) ∈ H with v = (C, i, i) then            ▷ Completer Rule
9:                 y ← MAKE_NODE(B → αC • β′, j, i, u, v)
10:                ADD_ITEM([B → αC • β, h, y], i, false)
11:            end if
12:        end for
13:
14:    else if β = bβ′ then                                     ▷ with b ∈ Σ
15:        u ← (b↑, i, i)                                       ▷ Insertion Rule
16:        V ← V ∪ {u}
17:        y ← MAKE_NODE(B → αb • β, j, i, u, v)
18:        ADD_ITEM([B → αb • β, j, y], i, false)
19:
20:    else if Λ is a final item then
21:        if u = null then
22:            if (B, j, i) ∉ V then
23:                add (B, j, i) to V
24:            end if
25:
26:            u ← (B, i, i)
27:
28:        end if
29:
30:        if j = i then
31:            add (B, u) to H
32:
33:            v ← (ϵ↑, i, i)
34:            if B → ϵ ∈ P and u does not have a family of children v then
35:                V ← V ∪ {v}
36:                add (v) as a child family of u
37:            end if
38:        end if
39:
40:        for all [A → τ • Bδ, k, z] ∈ Q_j do                  ▷ Completer Rule
41:            y ← MAKE_NODE(A → τB • δ, k, i, z, u, V)
42:            ADD_ITEM([A → τB • δ, k, y], i, false)
43:        end for
44:    end if
45: end while
```

Algorithm 6 INIT_NEXT_EARLEY_SET

$V \leftarrow \{\}$

while $N \neq \emptyset$ **do**
  $[B \rightarrow \ \alpha \bullet b\beta, h, u] \leftarrow N.pop()$

  if $b = a_i$ **then** ▷ Scanner Rule
    $v \leftarrow (a_i \rightarrow, i, i+1)$
    $V \leftarrow V \cup \{v\}$
    $y \leftarrow \mathit{MAKE\_NODE}(B \rightarrow \alpha a_i \bullet \beta, h, i+1, u, v)$
    ADD_ITEM($[B \rightarrow \ \alpha a_i \bullet \beta, h, y], i+1, true$)

  else ▷ Replacement Rule
    $v \leftarrow (a_i / b, i, i+1)$
    $V \leftarrow V \cup \{v\}$
    $y \leftarrow \mathit{MAKE\_NODE}(B \rightarrow \alpha b \bullet \beta, h, i+1, u, v)$
    ADD_ITEM($[B \rightarrow \ \alpha b \bullet \beta, h, y], i+1, true$)
  end if
end while

$R \leftarrow Q_i$
while $R \neq \emptyset$ **do**
  $[B \rightarrow \ \alpha \bullet \beta, h, u] \leftarrow R.pop()$ ▷ Deletion Rule
  $v \leftarrow (a_i \downarrow, i, i+1)$
  $V \leftarrow V \cup \{v\}$
  $y \leftarrow \mathit{MAKE\_NODE}(B \rightarrow \alpha \bullet \beta, h, i+1, u, v)$
  ADD_ITEM($[B \rightarrow \ \alpha \bullet \beta, h, y], i+1, true$)
end while

Algorithm 7 MAKE_NODE

Require: A 5−tupel $(B \rightarrow \alpha \bullet \beta, j, i, u, v)$ where B is a nonterminal, α, β are sentential forms, $j, i \in \mathbb{N}_0$ with $j \leq i$ and the SPPF nodes u, v
Ensure: A SPPF node

if $\beta = \epsilon$ **then**
  $s \leftarrow B$ ▷ Symbol node
else
  $s \leftarrow (B \rightarrow \alpha \bullet \beta)$ ▷ Intermediate node
end if

$y \leftarrow (s, j, i)$
if $y \notin V$ **then**
  add y to V
end if

if $u = null$ and y does not have a family of children (v) **then**
  add (v) as child family of y
else if $u \neq null$ and and y does not have a family of children (u, v) **then**
  add (u, v) as child family of y
end if
return y

function MAKE_NODE (Alg. 7). In addition, we use the set H that stores only the final Earley items of the current Earley set that also start at the same index as the index of the Earley set itself. This has the advantage that we don't have to filter out all items that fulfill this condition. In addition, for each production rule $B \to \varepsilon$ with $B \in N$, for each symbol node of the form (B, i, i), we add the insert of ε as a child family of that node.

The frequently used auxiliary function MAKE_NODE (Alg. 7) in the function COMPUTE_EARLEY_SET, checks whether the predecessor node created by applying a rule already exists. If not, the function adds a new node to the CSPPF. The function also adds a new family to the created or existing node for the two given input nodes. A big difference to the MAKE_NODE algorithm for SPPF (Alg. 15) is that it will always create the top node, even if α of the function call is a terminal symbol ($\alpha \in \Sigma$). In this case, the MAKE_NODE algorithm for SPPF does not create a new intermediate node as the top node because, from the nonterminal we can only derive the scanning of this letter. But in our case, we have more possibilities to derive this letter by scanning, replacing, or inserting it. So we need to create a new intermediate node. We also need to create a node for $\alpha = \varepsilon$. This is because otherwise, no consecutive delete operations can be represented. If not the first operation that moves the bullet point one to the right (all non-deletion rules), would have to include all delete operations performed up to that point in the child family. This, in turn, would contradict the CSPPF definition that packed nodes may only have a maximum of two children. Therefore, we also create an intermediate node for $\alpha = \varepsilon$, which also justifies the adaptation of the definition of intermediate nodes for CSPPF (Definition 4.1) compared to SPPF.

At last, we consider the auxiliary function INIT_NEXT_EARLEY_SET (Alg. 6). Since this function is called after the COMPUTE_EARLEY_SET function, we know that all Earley items have already been calculated and included in the current Earley set, and that all of these items starting with a nonterminal are included in the set N. For all items contained in N, we can use either the *scanner* or *replacement* rule, depending on whether the terminal symbol to the right of the bullet point matches the currently viewed letter of the input word. In addition to all items in the current Early set, we can apply the *deletion* rule. Therefore, the INIT_NEXT_EARLEY_SET applies the rules to the described Earley items and calculates the first Earley items of the next Earley set. In addition, the function builds the CSPPF as described above.

Example 4.3. In Example 4.2 we have already looked at some steps of the build process for the CSPPF for the context-free grammar G_{aexpr} and the word n. The resulting CSPPF, containing all indexless corrections for G_{aexpr} and n, after all the steps of the all-correction Earley Parser we have already shown in Figure 10. The *Earley items with nodes* of the CSPPF and Earley sets computed by the all-correction Earley Parser are outsourced to the appendix (App. D, Fig. 24).

4.3 Correctness

We have introduced correction shared packed parse forests (Sec. 4.1) and presented an algorithm (Sec. 4.2) to compute such a CSPPF for a given context-free grammar and a word over the grammar alphabet. We want to show that the algorithm works correctly and computes the set of *all* indexless corrections for the given input. For the sake of simplicity, we first show the *soundness* (Lemma 4.2) of the algorithm and then its *completeness* (Lemma 4.3).

Note that in this section we do not show that the algorithm terminates because in Section 4.4 we consider the runtime of the all-correction Earley Parser and show that it requires at most cubic time with respect to the length of the word (Theorem 4.6), which directly implies that it also terminates.

Soundness To show the soundness of the algorithm, we need to prove that any indexless correction ρ contained in a CSPPF computed by Algorithm 3 for a word w and a context-free language L corrects the word to a target word contained in the given language ($\rho(w) \in L$). The idea of the proof is to show that from the syntax tree that defines the indexless correction (see Sec. 4.1), we can construct a syntax tree for the corrected word. The existence of a syntax tree for the corrected word shows that the word is contained in the target language. The proof of the following lemma shows how we can extract the syntax tree of the word $\rho(w)$ from the syntax tree for ρ that are contained in the CSPPF. Remember that C_w is the set of all indexless corrections for a word w (Definition 3.2).

Lemma 4.2. *Let $G = (N, \Sigma, P, S)$ be context-free grammar with $|L(G)| > 1$, $w \in \Sigma^*$ and let ψ be the CSPPF constructed by Algorithm 3 for w and G. Then it applies for each indexless correction $\rho \in C_w$ that:*

$$\rho \text{ is contained in } \psi \Rightarrow \rho(w) \in L(G)$$

Proof. We have already discussed in Section 4.1 that each syntax tree contained in a CSPPF defines exactly one indexless correction by its leaf front. So let τ be a syntax tree contained in ψ ($\tau \, \xi \, \psi$) which defines the indexless correction ρ. In this proof, we first show how a syntax tree τ' for the word $\rho(w)$ can be constructed from τ. Then we show that the constructed syntax tree τ' is a valid syntax tree for $\rho(w)$ by showing that all *Earley rules* that construct the CSPPF follow the rules of grammar G, such that τ' represents a derivation of $\rho(w)$ regarding G.

For the syntax tree τ it can be observed that it differs from a conventional syntax tree only in that the leaves are labeled with indexless edit operations instead of letters of the word. Therefore, τ' can be constructed from τ by renaming the leaves to the resulting letter of the application of the edit operation. The only exceptions are deletion operations. Since the letter deleted by a deletion operation has no letter to match in the corrected word, leaves representing a delete operation must be removed from the syntax tree.

More detailed, all leaf nodes v of the syntax tree τ must be edited as follows:

- If $v = x\uparrow$ with $x \in \Sigma \cup \{\varepsilon\}$ then relabel v to x.
- If $v = a_i/x$ with $0 \leq i < n$ and $x \in \Sigma$ then relabel v to x.
- If $v = a_i\rightarrow$ with $0 \leq i < n$ relabel v to a_i
- If $v = a_i\downarrow$ with $0 \leq i < n$ then delete the leaf v.

After we have presented how a syntax tree τ' can be constructed from τ, we now have to show that the syntax tree τ' is also a derivation of the word $\rho(w)$ with respect to the grammar G. For this purpose, we show that all Earley rules used in Algorithm 3 to construct the CSPPF ψ follow the production rules P of the grammar G, and thus τ' is a valid syntax tree with respect to G.

- **Scanner:** Regarding the current derivation status of a production rule $A \rightarrow \alpha \bullet a\gamma'$ in which the letter a must be parsed next, represented by an *Earley item with node*. Can the letter a be parsed with the *scanner* rule, so that after applying the rule of we have the status $A \rightarrow a\alpha \bullet \gamma'$ of the derivative. This rule creates an edit node in the CSSPPF describing that the letter a was parsed (see Fig. 7). This edit node is a leaf of the syntax tree if it is contained in the syntax tree τ. However, since the letter a is contained in both $\rho(w)$ and w, we only need to rename the leaf from $a\rightarrow$ to a.
- **Insertion:** Similar to the previous *scanner* rule case with the only small difference that the considered letter a is only contained in $\rho(w)$. Therefore, by the insertion rule an insertion edit node are add to the CSPPF (see Fig. 11), which represent the derivation of the letter a in the word $\rho(w)$.

- **Replacement:** Similar to the two previous cases with the difference that the letter a is contained in $\rho(w)$, but another letter x with $x \in \Sigma$ must be parsed as the next from the word w. Therefore, the corresponding replacement edit node are added to the CSPPF (see Fig. 11), which represent the derivation of the letter a in the word $\rho(w)$.
- **Predictor:** This rule only selects production rules, which can then be derived by the other rules, and does not contribute to the structure of the CSPPF.
- **Completer:** Regarding the current derivation status of a production rule $A \to \alpha \bullet A\gamma'$ in which the nonterminal A must be derived as next, represented by an *Earley item with node*. Can the *completer* rule add all derivatives of the nonterminal A represented by a node to the CSPPF (see Fig. 7) so that the nonterminal no longer needs to be considered next ($A \to \alpha A \bullet \gamma'$).
- **Deletion:** The *deletion* rule only deletes a symbol of the word w that is not contained in the word $\rho(w)$. But we can observe that it doesn't change the current derivation status of a production rule (see fig. 11). This means that no deletion edit node is contained in a valid syntax tree of $\rho(w)$ which explains why we have to remove these nodes from the syntax tree of τ to get τ'.

Thus we have shown for all Earley rules that they follow the production rules of G, and thus any converted syntax tree τ' is a valid syntax tree with respect to the grammar G. Since we have a valid τ' syntax tree for $\rho(w)$ with respect to G, it also follows that $\rho(w) \in L(G)$ holds. □

Completeness After having shown the soundness of our algorithm, we also need to show its completeness. Therefore, we need to show that our algorithm computes a correction shared packed parse forest for a word w and a context-free grammar G that contains all indexless correction ρ for w such that $\rho(w) \in L(G)$. We already know that each indexless correction is defined by the leaves of the syntax tree contained in the correction shared packed parse forest. The idea of the proof is to show that the leaves are added to the CSPPF in such a way that the syntax tree is contained in the root node $((S, 0, n))$ of the CSPPF.

Lemma 4.3. *Let $G = (N, \Sigma, P, S)$ be context-free grammar with $|L(G)| > 1$, $w \in \Sigma^*$ and let ψ be the CSPPF constructed by Algorithm 3 for w and G. Then it applies for each indexless correction $\rho \in C_w$ that:*

$$\rho(w) \in L(G) \Rightarrow \rho \text{ is contained in } \psi$$

Proof. Since ρ is an indexless correction contained in C_w we know from Definition 3.2 that ρ must have the following from:

$$\rho = (x_0, v_0, x_1, v_1 \ldots, v_{n-1}, x_n)$$

where $x_i = x\uparrow$ with $x \in \Sigma^*$ and $v_i \in \{a_i \to, a_i \downarrow, a_i / b\}$ with $b \in \Sigma$. Remember that the set C_w is the set of all indexless correction for a word w (Definition 3.2). In addition, we have already discussed in Section 4.1 that each syntax tree contained in a CSPPF defines exactly one indexless correction by its leaf front. Therefore, in the following proof we need to show that a syntax tree τ with the following leaf front is contained in ψ:

$$x_{0_0}\uparrow, \ldots, x_{0_{m_0}}\uparrow, v_0, x_{1_0}\uparrow, \ldots, x_{1_{m_1}}\uparrow, v_1, \ldots, v_{n-1}, x_{n_0}\uparrow, \ldots, x_{n_{m_n}}\uparrow$$

where for all $0 \leq i \leq$ applies that $x'_i = x_{i_0}, \ldots, x_{m_i}$ such that $x'_i\uparrow = x_i$.

We show this by an induction over the edit operations of ρ, where we consider all insertion words x_i character by character for the sake of simplicity in the induction. Because we all add them character by character in the all-correction Earley Parser by adding edit nodes.

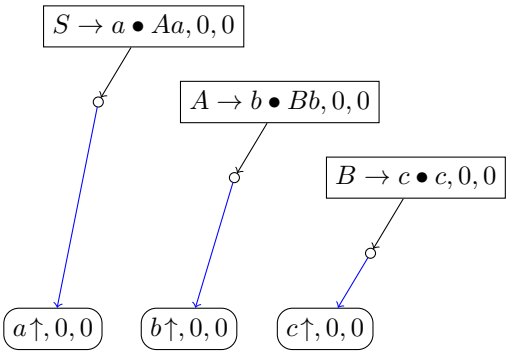


Figure 13: Exemplary illustration of a series of intermediate nodes in the set C_l.

But before we start with the briefing, we make some preparations. First, we look at some observations about the indexless insertion operations x_i. If $x_i = \varepsilon\uparrow$ then there is no leaf in the syntax tree τ that is part of x_i. Conversely, the leaf front that is part of x_i $(x_{i_0}, \ldots, x_{m_i})$ can have additional nodes that insert an epsilon $(\varepsilon\uparrow)$. However, these leaves have made no productive contribution to the indexless correction. Therefore, we will add these leaves whenever necessary in the following. However, for the sake of simplicity, we will not consider them as leaves of the syntax tree described above because these leaves have no productive contribution to the indexless correction. Second, we define some helpful definitions:

- Let ρ_j be the indexless edit operation that define the $j-$th leaf ordered from left to right in the syntax tree τ. Note: this is either an insertion of a single character of x_i or an operation v_i.
- Let $\rho_{0,i}$ be a sequence of indexless edit operations defined as $\rho_{0,j} = (\rho_0, \ldots, \rho_j)$. Note that this sequence define an indexless edit operation.
- In other words, ρ_j is defined by the first j leaves of the syntax tree of τ (remember, we consider the syntax tree of τ without possible insertion of ε).

  Let us consider from which parts of the CSPPF we can derive these leaf front. First to mention are symbol nodes, from which we can derive a syntax tree with exactly this leaf front.

  However, this leaf front can also be derived from series of intermediate nodes. We will consider this case in more detail. Let $(v_0, v_1, \ldots, v_h)$ be such a series of intermediate nodes. Then a syntax tree can be derived from v_0 which has the leaf front $(\rho_0, \ldots \rho_m)$ with $m \leq j$. From v_1 we can derive a syntax tree with the leaf front $(\rho_{m+1}, \ldots \rho_{m'})$ with $m \leq m' \leq j$. And so on so that we can finally derive a syntax tree from v_h that has the leaf front $(\ldots, \rho_j)$. This means that $\rho_{0,i}$ can also be derived from this series of intermediate nodes. For example, we have an intermediate node v_0 of the form $(A \to \alpha \bullet B\beta', 0, k)$ representing the indexless correction up to some point m, but the subtree representing the indexless correction after m is incomplete at the current point in the algorithm, so this subtree cannot be combined into a family with v_0 using the completer rule. In this case, the intermediate node v_1 represents the current state of the subtrees being built. Fig. 13 represents an exemplary series of nodes that together represent the insertion of *abc*.

  Let C_l be the set of all *symbol nodes* in ψ and all *series of intermediate nodes* in ψ

from which *we can derive the sequence* $\rho_{0,l}$. More formally defined as:

$$
\begin{aligned}
C_l = \{v \mid & \text{ we can derive a syntax tree from } v \text{ with the leaf front } \rho_{0,l}\} \cup \\
& \{(v_0, v_1, \ldots, v_h) \mid \forall_{j=0}^{h-1}, \exists k, \exists m, \exists m' \exists l' \quad v_j = (A \to \alpha \bullet B\beta', j', k), \\
& \quad v_{j+1} = (B \to \alpha' \bullet \beta, k, i) \\
& \quad \text{and } (\text{we can derive a syntax tree from } v_j \text{ with the leaf front } \rho_{m',m}) \\
& \quad \text{and } (\text{we can derive a syntax tree from } v_{j+1} \text{ with the leaf front } \rho_{m+1,l'}) \\
& \quad \text{and } (j = 0 \to m' = 0) \text{ and } (j = h-1 \to l' = l)\}
\end{aligned}
$$

- We can observe that all symbol nodes in C_l have the same extent $(0, j')$ with $0 \leq j \leq n$. In addition, it can be seen that for all series of intermediate nodes the first node has the extent $(0, m)$ and the last node has the extent (m', j') with $0 \leq m \leq m' \leq j \leq n$. Therefore, the extent of series of intermediat nodes can also be interpreted as $(0, j')$.

  Therefore, all elements in C_l have the same extent $(0, j')$. This means that the all-correction Earley parser is in step j', calculation of the Earley set $Q_{j'}$, at which we can at earliest determine C_l. Therefore, after determining of C_l, we can assume that all Earley sets Q''_j with $0 \leq j'' < j'$ have already been fully calculated by all-correction Earley parser. We will need this property later for our induction hypothesis.

  Therefore, let $S(C_l)$ be the step of the all-correction Earley parser at which we can determine C_l at the earliest.

We have now made all the preparations to start with the proof of induction. In the following proof of induction we show that for all $\rho_{0,j}$ with $0 \leq j \leq n + (\sum_{i=0}^{n} |x| \text{ with } x_i = x\uparrow)$ it holds that the set C_j contains at least one element. In other words, after each edit operation of the indexless correction ρ added to the CSPPF, there is a element in C_l from which we can derive a leaf front describing the correction ρ up to the current position. So that after we have added all edit operations we know that if C_l is not empty, the correction ρ can be derived. Note that to show that ρ is contained in ψ, it must also be shown that the node $(S, 0, n)$ is contained in the last computed C_l.

Base Case As a base case, we show that if ρ_0 is added to the CSPPF, there is at least one element in C_0. Note that ρ_0 is the leftmost node of the syntax tree τ. The node ρ_0 is an edit note that represents either an *insertion*, *deletion*, *replacement*, or *read* operation, depending on the indexless correction ρ. Therefore, to create such a node, we need to apply one of these rules to an Earley item that is currently assigned with the dummy node *null*. Because if the item is already assigned with a non-dummy node, then that node will be set as the left neighbor of the added node, and so the edit node cannot be the leftmost node. Therefore, the question is whether the all-correction Earley Parser can compute such an Earley item contained in the first Earley set (Q_0) so that we can apply one of the rules *deletion*, *replacement*, *insertion*, or *scanner* to create the leftmost leaf of the indexless correction. To accomplish this, we consider the different cases of indexless edit operations that can be represented by the ρ_0.

- **Deletion** The deletion rule can be applied to all Earley items. It is therefore sufficient that there is at least one Earley item in the first Earley set. This condition is always fulfilled because we have given a that $|L(G)| > 0$.
- **Insertion** Since we want to add an insertion node of the form $(b\uparrow, 0, 1)$ with $b \in \Sigma$ to ψ as the leftmost node.

  They are an *Earley item with node* of the form $[A \to \bullet b\gamma, nuill]$ with $A \in N$ $\gamma \in (N \cup \Sigma)^*$ and $A \to b\gamma \in P$ in the first Earley set (Q_0).

Such an item is included in the first Earley set either if we can derive it directly from the start symbol (line 4 to line 6 in Alg. 3) or through a series of *predictor* rules (where if there is a production rule of the form $A \to \varepsilon$ with $A \in N$, then a series of combinations of predictor, insert epsilon, and completer is also possible). Or, to put it more simply, at least one word in $L(G)$ starts with the letter b.

This implies that if there is not such an item contained in the first Earley set, we cannot create such an edit node. In this case, however, $\rho(w)$ would also lead to a word that is not included in $L(G)$ since $\rho(w)$ would begin with the letter b, which would lead to a contradiction with the condition $\rho(w) \in L$.

- **Replacement** Same argumentation as for insertion, with the additional requirement that we can only use the replacement rule if $a_0 \neq b$ because we cannot replace the letter b by itself.
- **Read** Same argumentation as for insertion and replacement, but in this case, it must also apply that $b = a_i$.

Thus, for all possible types of indexless editing operations, which can be ρ_0, we have shown that our algorithm creates such a leaf, which shows the correctness of the base case.

Note that the most edit nodes are in the child families of more than one node, each created by a different *Earley item with node*, to which we can apply the corresponding rule. Therefore, we have a set of nodes from that we can derive only the added edit node (the set C_0).

Induction Hypothesis We assume that the set C_l has already been calculated by the all-correction Earley Parser and contains at least one element (symbol node or series of intermediate nodes). In addition, we assume that all Earley sets $Q_{j'}$ with $0 \leq j' < S(C_l)$ have already been fully calculated.

Induction Step Based on the set C_l, we need to show that the set C_{l+1} also contains at least one element. To accomplish this, we need to show that it is possible to add the leave ρ_{i+1} as the next leave of the syntax tree τ, to the CSPPF. But before, let us look at how we deal with the fact that the set C_l contains both symbol nodes and series of intermediate nodes.

Therefore, let $(v_0, v_1, \ldots, v_h)$ be a series of intermediate nodes in the set C_l. Then we know that each v_i represents the root of a subtree that is not yet fully constructed. The first subtree we need to complete is subtree v_h. So we add the next leaf to the right of v_h. Therefore, we must only consider the node v_h in order to add the leaf ρ_{l+1}. This means that the argument for adding ρ_{l+1} is the same for both types because we only have to consider one node in both cases.

As in the base case, we split the proof into the different types of editing operations that can be represented ρ_{l+1}. But here we have grouped together the edit operations *insertion*, *replacement*, and *read* because we have already seen in the base case that all three follow the same argumentation. This is the same here.

- **Deletion** In the base case, we have already mentioned that we can apply the *deletion* rule to all Earley items. In addition, we can observe that an intermediate node or symbol node is only in the CSPPF if at least one Earley item is assigned to that node. This means that we can apply the deletion rule to an item that is assigned to a node (a single node or rightmost node of a series of nodes) in the set C_l to add as next the edit node ρ_{l+1} that represents an *deletion* operation.
- **Insertion, Replacement, Read**

  We consider what is required to add the leaf ρ_{l+1} to the right of an intermediate node and what is required to add this leaf to the right of a symbol node. To finally show

that one of the two opportunities is always possible, and therefore there is at least one element in the set C_{l+1}.

For us to be able to apply one of the *scanner*, *insertion*, or *replacement* rules to add ρ_{l+1} as the next leaf of τ to ψ, we need an *Earley item with node* in the current Earley set, for which the sentential form to the right of the bullet starts with the nonterminal that we want to scan, insert, read, or replace. We consider how we can calculate such an element from a symbol or an intermediate node.

- **Intermediate node** For each *intermediate node* of the form $(A \to \alpha\bullet\beta, j, i)$ there exists an *Earley item with node* $[A \to \alpha \bullet \beta, j, v]$ in the Earley set Q_i associated with that intermediate node, where β can start with either a nonterminal or a terminal.

  If it starts with a *terminal* symbol, we can only apply the rule if that terminal matches the terminal we want to insert, replace, or read. In the other case where β starts with a *nonterminal*, we can apply the predictor rule to this item to create an Earley item that starts with the required nonterminal in front. Note that the item does not necessarily have to be generated by the simple application of a *predictor* rule. It can also be generated by the multiple application of the *predictor* rule, or if there is a rule of the form $A \to \varepsilon$, there can also be *completer* rules in the series of *predictor* rules to possibly generate the desired Earley item.
- **Symbol node** For each *symbol node* of the form (A, j, i) we have at least one *final Earley item with node* $([A \to \alpha\bullet, j, v])$ for the nonterminal A in the Earley set Q_i. So we can apply the *completer* rule to all combinations of the final Earley item with node and Earley items in the Earley set Q_j that start with the nonterminal A to the right of the bullet point. To create one of the required items on which we can apply either the *scanner*, *replacement*, or *insertion* rule as next.

  If $j < i$ applies, we can assume that Q_j has already been fully calculated based on the induction hypothesis. In the case $j = i$, we add the Earley item to the set H in the auxiliary function COMPUTE_EARLEY_SET, so that later, for each new item added to the Earley set Q_i, the algorithm can check if it can apply the completer rule to the new and the final item. These two facts show that the algorithm applies the *completer* rule to all possible combinations of the final Earley item and any other item that can be created for the grammar G. Note that the simple application of the *completer* rule may not be sufficient to create the required Earley item; rather, the simple application may entail a number of other possible rules to create the required item.

So far, we know that it is *possible* to create an Earley item from any element in the set C_l, so we can add the leaf ρ_{l+1} to the CSPPF as the next leaf of the syntax tree. However, it remains to be shown that such an Earley item is also created from at least one of these elements in C_l.

On closer inspection of the Earley items used to construct the part of the CSPPF from which the syntax tree of the indexless correction can be derived, it is noticeable that it is precisely these Earley items that build up the syntax tree of $\rho(w)$ when $\rho(w)$ and G are given as input into the Earley Parser (Sec. 2.3).Note that the indices in the Earley items and in which items the sets are contained do not have to match. This is because the use of *deletion* and *insertion* operations can cause the indices to shift. The reason we get the same Earley items is that the all-corrections Earley Parser follows all derivations of the target language by correcting the word w whenever we would not have been able to follow the derivation otherwise. This means that we also follow the derivation of the word $\rho(w)$ by calculating the same Earley items. Since $\rho(w) \in L$ applies, there must be an Earley item of the required form in the Earley Parser for us

to be able to derive the required letter by the *scanner* rule. This means that this item must also exist in the all-correction Earley Parser at the current step. This indicates that we can add the leave ρ_{l+1} as the next of the syntax trees to the CSPPF, and so we have at least one element in the set C_{l+1}.

Thus, for all possible edit operations for ρ_{l+1}, we have shown that we can add an corresponding edit node to the CSPPF, such that we can derive the sequence $\rho_{0,l+1}$ from the CSPPF. This implies that there is at least one element in the set C_{l+1}, which shows the correctness of our induction step.

Conclusion We have shown the correctness of the base case and of the induction step. We know that for each indexless correction, there is an element in the set C_l that represents the indexless correction ρ. Please note that this does not directly imply that the indexless correction ρ is contained in the CSPPF τ. For this to be true, the root node $(S, 0, n)$ of the CSPPF must be contained in the set $C_{l'}$, where l' is the number of leaves of the syntax tree τ that represent this indexless correction.

We can observe that the extent of each symbol node in the set $C_{l'}$ is $(0, n)$, because each syntax tree we can derive from a symbol node in $C_{l'}$ represents the indexless correction ρ. In addition, we know that $\rho(w) \in L$ holds, so we can assume that if C_l contains all derivations, it also contains one from the start symbol. From this we can follow: that $(S, 0, n) \in C_{l'}$ hold.

Therefore, we have shown that the syntax tree representing the indexless correction ρ is contained in the CSPPF by constructing it by the all-correction Earley Parser (Alg. 3). □

The soundness and completeness of the CSPPF computed by Algorithm 3 for some context-free grammar and word over the grammar alphabet shows that our algorithm computes a CSPPF that contains exactly the set of all indexless corrections that correct the word to a word in the language.

Theorem 4.4. *Let $G = (N, \Sigma, P, S)$ be context-free grammar with $|L(G)| > 1$, $w \in \Sigma^*$ and let ψ be the CSPPF constructed by Algorithm 3 for w and G. Then it applies for each indexless correction $\rho \in C_w$ that:*

$$\rho(w) \in L(G) \iff \rho \text{ is contained in } \psi$$

Proof. The correctness of this theorem and so of the all-correction Earley Parser (Alg. 3) follows directly from its *soundness* (Lemma 4.2) and its *completeness* (Lemma 4.3). □

4.4 Analysis

Looking at our CSPPF example in Fig. 10, at first glance it looks like the computed correction shared packed parse forest is huge in relation to the given word w and context-free grammar G. Therefore, in this subsection, we analyze the size of the correction shared packed parse forests and look at the runtime of Algorithm 3 to compute it.

First, we show that the each CSPPF has at most a cubic number of nodes and edges regarding the length of the word (Theorem 4.5). Second, we show that the required runtime of our algorithm is also cubic regarding the length of the word (Theorem 4.6).

Size of CSPPF To determine the number of nodes of a correction shared packed parse forest for w and G, we consider the number of nodes for each node type of CSPPF regarding the length of the word.

We can observe that the majority of the nodes are of type packed node, because we have with i, j and k three variables depending on the word length. Whereas for all other node types we have with i and j only two variables regarding the word length. This explains why there is a maximum of $\mathcal{O}(n^3)$ nodes in a CSPPF for a word of length n, if we consider the size of the grammar G as a constant factor.

The fact that we already know the number of packed nodes helps us a lot in determining the number of edges. Because we know that every CSPPF is a bipartite graph and that one of the independent sets is the set of packed nodes. Furthermore, each packed node has at most one predecessor and at most two successors. Therefore, each node of the one independent set has at most 3 edges, so we have at most $\mathcal{O}(n^3)$ edges in a correction shared packed parse forest. The analysis of the size of a CSPPF, briefly introduced here, is considered in detail in the following theorem, with size of the grammar as an additional input parameter.

Theorem 4.5. *Let $G = (N, \Sigma, P, S)$ be context-free grammar $w \in \Sigma^*$ with $|w| = n$. Then each CSPPF for w and G has at most:*

$$\mathcal{O}\left(n^3\left(|P| + \sum_{A \to \gamma \in P} |\gamma|\right)\right) \text{ nodes}$$

and

$$\mathcal{O}\left(n^3\left(|P| + \sum_{A \to \gamma \in P} |\gamma|\right)\right) \text{ edges.}$$

Proof. The number of nodes is the sum of the number of nodes of the different node types. Therefore, we split the proof into different parts so that we can consider the number of nodes for each type independently. First, we consider edit nodes, then symbol nodes, then intermediate nodes, and at least packed nodes. At the end, we add up the individual results to show the promised number of nodes and edges.

Edit nodes From the definition of edit nodes, we already know that if the node represents a replace, delete or read operation, then the extent of the edit node must be $(i, i+1)$ with $0 \leq i < n$. Therefore, for each of these operations, we can only have n different extents for a word of length n. For each extent, there can only be one different delete and one different read operation, but $|\Sigma| - 1$ different replace operations.

For edit nodes representing an insert operation, we know that the extent must be of the form (i, i) with $0 \leq i \leq n$. So there are $n+1$ different extents, and for each of these we can either insert a nonterminal from the alphabet or ε. Therefore, the CSPPF contains at most $(n+1)(|\Sigma|+1)$ different edit nodes representing an insertion operation. So we have

$$(n+1)(|\Sigma|+1) + n + n + n(|\Sigma|-1) = 2n|\Sigma| + 2n + |\Sigma| + 2$$

different edit nodes.

Symbol nodes A symbol node is of the form (A, i, j) with $A \in N$ and $0 \leq i \leq j \leq n$. For each j we can have $j+1$ different values for i, so we can have for each nonterminal $\sum_{j=0}^{n} j + 1$ different extents, which can be simplified by the sum formula of arithmetic series[5] to $\frac{(n+1)(n+2)}{2}$.

So we have

$$\frac{(n+1)(n+2)}{2}|N| = \left(\frac{n^2}{2} + \frac{3n}{2} + 1\right)|N|$$

different symbol nodes.

[5]Sum formula of arithmetic series: $\sum_{k=1}^{n} k = \frac{k(k+1)}{2}$

Intermediate nodes We look at the number of intermediate nodes. Again, for each production rule of the grammar, we have $\frac{(n+1)(n+2)}{2}$ different extents. In addition, for each production $A \to \gamma$ we have $|\gamma|$ different ways of splitting γ into α and β, so that $\alpha\beta = \gamma$ with $\alpha \in (N \cup \Sigma)^*$ and $\beta \in (N \cup \Sigma)^+$. A CSPPF can have an intermediate node for any combination of a possible extent and a split of a production rule. Therefore, a CSPPF can have at most

$$\frac{(n+1)(n+2)}{2} \sum_{A \to \gamma \in P} |\gamma| = \left(\frac{n^2}{2} + \frac{3n}{2} + 1\right) \left(\sum_{A \to \gamma \in P} |\gamma|\right)$$

different intermediate nodes.

Packed nodes The number of packed nodes remains to be determined. We have packed nodes that have a delete operation node as successor and packed nodes that do not have a delete operation node as successor. First we consider the packed nodes that have no delete operation nodes as successors. For this nodes we know from the definition that $0 \leq i \leq k \leq j \leq n$ must hold. So we first calculate the number of different ways of choosing a combination of i, j and k such that $0 \leq i \leq k \leq j \leq n$ by the following formula.

$$\begin{aligned}
&\sum_{j=0}^{n} \sum_{i=0}^{i} \sum_{k=0}^{j-i} 1 \\
=&\sum_{j=0}^{n} \sum_{i=0}^{j} (j - i - 1) \\
=&\sum_{j=0}^{n} \left(\sum_{i=0}^{j} i + \sum_{i=0}^{j} -i + \sum_{i=0}^{j} 1 \right) \\
=&\sum_{j=0}^{n} \left(j(j+1) - \frac{i(j+1)}{2} + j + 1 \right) \\
=&\sum_{j=0}^{n} \left(\frac{j^2}{2} + \frac{ji}{2} + 1 \right) \\
\overset{(1)}{=}&\frac{1}{2} \sum_{j=0}^{n} j^2 + \frac{3}{2} \sum_{j=0}^{n} 3j + \sum_{j=0}^{n} 1 \\
=&\frac{n(n+1)(2n+1)}{12} + \frac{3n(n+1)}{4} + n + 1 \\
=&\frac{2n^3 + 12n^2 + 10n}{12} + n + 1 \\
=&\frac{n^3}{6} + n^2 + \frac{11n}{6} + 1
\end{aligned}$$

Whereby in (1) we have used the formula for the sum of squares[6]. For each combination of i, j and k and each production rule $A \to \gamma$ we have $|\gamma| + 1$ different ways of splitting γ into α and β, such that $\alpha\beta = \gamma$ with $\alpha, \beta \in (N \cup \Sigma)^*$. This implies we have

$$\left(\frac{n^3}{6} + n^2 + \frac{11n}{6} + 1\right) \left(\sum_{A \to \gamma \in P} |\gamma| + 1\right)$$

[6]Formula for sum of squares: $\sum_{k=0}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$

different packed nodes that have no deletion operation node as successor.

Second, we need to determine the number of packed nodes that have a deletion operation node as a successor. For these nodes we have the constraint that $j = k + 1$ must hold. Therefore we have the following number of combinations to choose i, j and k:

$$\sum_{i=0}^{n-1} i + 1 = \frac{n^2}{2} + \frac{3n}{2} + 1 - n + 1 = \frac{n^2}{2} + \frac{n}{2} + 2$$

As for packed nodes without a deletion operation node as successor, we have for each combination of i, j and k $\left(\sum_{A \to \gamma \in P} |\gamma| + 1\right)$ opportunities for $A \to \alpha \bullet \beta$. This implies we have

$$\left(\frac{n^2}{2} + \frac{n}{2} + 2\right)\left(\sum_{A \to \gamma \in P} |\gamma| + 1\right)$$

different packed nodes that have a deletion operation node as successor.

Therefore, we have at most the following number of different packed nodes in a CSPPF.

$$\begin{aligned}
&\left(\frac{n^3}{6} + n^2 + \frac{11n}{6} + 1 + \frac{n^2}{2} + \frac{n}{2} + 2\right)\left(\sum_{A \to \gamma \in P} |\gamma| + 1\right) \\
&= \left(\frac{n^3}{6} + \frac{3n^2}{2} + \frac{7n}{3} + 3\right)\left(|P| + \sum_{A \to \gamma \in P} |\gamma|\right)
\end{aligned}$$

Number of Nodes We have now determined, for all node types, the maximum number of nodes that a CSPPF can contain for w and G. We can combine these intermediate results to obtain the maximum number of nodes that a CSPPF computed by Algorithm 3 can have. So a CSPPF can have at most the following number of nodes:

$$\begin{aligned}
&\underbrace{2n|\Sigma| + 2n + |\Sigma| + 2}_{\text{edit nodes}} \\
&+ \underbrace{(\frac{n^2}{2} + \frac{3n}{2} + 1)|N|}_{\text{symbol nodes}} \\
&+ \underbrace{\left(\frac{n^2}{2} + \frac{3n}{2} + 1\right)\left(\sum_{A \to \gamma \in P} |\gamma|\right)}_{\text{intermediate nodes}} \\
&+ \underbrace{\left(\frac{n^3}{6} + \frac{3n^2}{2} + \frac{7n}{3} + 3\right)\left(|P| + \sum_{A \to \gamma \in P} |\gamma| + 1\right)}_{\text{packed nodes}}
\end{aligned}$$

It is obvious that the number of packed nodes has the most impact on the total number of nodes. Therefore, in an asymptotic view, we have the promised $\mathcal{O}\left(n^3\left(|P| + \sum_{A \to \gamma \in P} |\gamma|\right)\right)$ number of nodes.

Number of Edges To determine the number of edges in an correction shared packed parse forest, we use the fact that a CSPPF is a bipartite graph with symbol nodes, edit nodes and intermediate nodes as one independent set and packed nodes as the other independent set. This implies that it is sufficient to determine the number of edges for each packed node.

We know that each packed node has exactly one incoming edge and one or two outgoing edges. Therefore each packed node can have at most 3 edges, so that in an asymptotic view a CSPPF for w and G has at most $\mathcal{O}\left(n^3\left(|P| + \sum_{A\to\gamma\in P}|\gamma|\right)\right)$ edges. □

Time of the Algorithm At last, we analyze the runtime of the all-correction Earley Parser (Alg. 3) presented in Section 4.2 to compute all indexless corrections for a context-free grammar and a word of length n, that correct the word into the grammar language. We already know that the runtime of the original Earley Parser [4] is cubic regarding the length of the word. The proof of the runtime of the Earley Parser [4, Sec. 5] shows that for each Earley set (we have $n+1$ Earley sets), we can have $\mathcal{O}(n)$ different Earley items, if we consider the size of the grammar as constant. Because for all items $[A \to \gamma \bullet a, j]$ applies: $j \leq i \leq n$. Each item is considered once by the Earley Parser (Alg. 1). For each item, we can only apply the *predictor* rule and the *scanner* rule at most once, whereas the *completion* rule can be applied at most $\mathcal{O}(n)$ (size of the referenced Earley set). This results in the total cubic runtime.

The additional storage of a CSPPF node for an Earley item has no effect on the number of different Earley items. We can also observe that all new rules (*insertion*, *replacement*, and *deletion*) as well as the construction of the CSPPF can be performed in constant time. This indicates that the all-correction Earley Parser has also a cubic runtime.

Since, in contrast to [4], we also consider the grammar as an input parameter, the following theorem shows the runtime of the algorithm regarding a word of length n and a context-free grammar. However, the structure of the proof is not very different from that in [4].

Theorem 4.6. *Let $G = (N, \Sigma, P, S)$ a context-free grammar, $w \in \Sigma^*$ with $|w| = n$. Then a CSPPF that contains all indexless corrections ρ such that $\rho(w) \in L(G)$ is computed by Algorithm 3 in time:*

$$\mathcal{O}\left(n^3\left(\sum_{A\to\gamma\in P}|\gamma|\right)\left(|P| + \sum_{A\to\gamma\in P}|\gamma|\right) + n^2|P|^2\right).$$

Proof. For a word of length n we have $n+1$ different Earley sets. An *Earley item with node* (Definition 2.7) contained in the Earley set Q_i is of the form $[A \to \gamma \bullet \alpha, j, u]$ where $A \to \gamma\alpha \in P$, $0 \leq j \leq i \leq n$ and u is a node of the constructed CSPPF or the dummy node *null*. Whereby the possible nodes in the CSPPF are already uniquely described by $A \to \gamma \bullet \alpha, j$ and the assignment to the set i. This implies that we can have at most $\mathcal{O}\left((n+1)\sum_{A\to\gamma\in P}|\gamma| + 1\right)$ different Earley items for each Earley set.

Each Earley item is considered only twice by the all-correciont Earley Parser (Alg. 3). Once in the auxiliary function COMPUTE_EARLEY_SET (Alg. 5) to apply the *predictior*, *completer*, and *insertion* rules to it, and once in the function INIT_NEXT_EARLEY_SET (Alg. 6) to apply the *deletion*, *scanner*, and *replacement* rules to it. We can observe that the application of a single rule, as a step in the construction of the CSPPF by that rule (auxiliary function MAKE_NODE, Alg. 7), can be performed in constant time. Therefore, we need to consider how many times each rule can be applied to the same Earley item.

We can observe that the rules *insertion*, *replacement*, *deletion*, and *scanner* can only be applied once to the same Earley item. Whereas the *predictor* rule can be applied at most once for each production rule of the grammar and the *completer* rule can be applied at most once for each item, excluding the final items in the set (Q_j) referenced by the item. We already know that each Earley set has at most $\mathcal{O}\left(n\sum_{A\to\gamma\in P}|\gamma|\right)$ different non-final Earley items, so we have the following asymptotic runtime of our algorithm (where Pr = *predictor*,

C = *completer*, I = *insertion*, D = *deletion*, R = *replacement*, and S = *scanner*).

$$
\begin{aligned}
&\mathcal{O}\left(\underbrace{(n+1)}_{\text{sets}} \cdot \underbrace{n\left(\sum_{A\to\gamma\in P} |\gamma|+1\right)}_{\text{items per set}} \cdot \left(\underbrace{|P|}_{\text{Pr}} + \underbrace{n\left(\sum_{A\to\gamma\in P} |\gamma|\right)}_{\text{C}} + \underbrace{1}_{\text{I, D, R, S}}\right)\right)\\
=&\mathcal{O}\left(\left(n^2|P| + n^2 \sum_{A\to\gamma\in P} |\gamma|\right) \cdot \left(|P| + n\left(\sum_{A\to\gamma\in P} |\gamma|\right)\right)\right)\\
=&\mathcal{O}\left(n^3\left(\sum_{A\to\gamma\in P} |\gamma|\right)^2 + n^3|P|\left(\sum_{A\to\gamma\in P} |\gamma|\right) + n^2|P| \sum_{A\to\gamma\in P} |\gamma| + n^2|P|^2\right)\\
=&\mathcal{O}\left(n^3\left(\sum_{A\to\gamma\in P} |\gamma|\right)^2 + n^3|P|\left(\sum_{A\to\gamma\in P} |\gamma|\right) + n^2|P|^2\right)\\
=&\mathcal{O}\left(n^3\left(\sum_{A\to\gamma\in P} |\gamma|\right)\left(|P| + \sum_{A\to\gamma\in P} |\gamma|\right) + n^2|P|^2\right)
\end{aligned}
$$

Which shows the promised runtime of Algorithm 3. □

Disambiguous Grammars In [4] it was additionally shown that the Earley Parser for an *disambiguous grammar* only needs a runtime of $\mathcal{O}(n^2)$ instead of $\mathcal{O}(n^3)$ regarding a word of length n. This is because, for a disambiguous grammar, each Earley item can only be created in exactly one way. Consequently, each item can only be completed once using the *completer* step (using a different Earley item to move the point one position to the right). As a result, the factor n in the completer rule is omitted for each Earley item.

But in the all-correction Earley Parser (Alg. 3) we can always create each Earley item in more than one way (except $w = \varepsilon$), even for disambiguous grammars. This is because instead of using the *replacement* or *scanner* rule, we can also use the *insertion* rule and *deletion* rule to create the same Earley item as by using the *replacement* or *scanner* rule. But these different ways are absolutely necessary; otherwise, our algorithm would compute a CSPPF that does not include all indexless corrections.

Therefore, the runtime for the all-correction Earley Parser to compute a CSPPF remains unchanged, even for disambiguous grammars, in contrast to the original Earley Parser.

5 Filter on CSPPF

We have show in Section 4 how we can compute all corrections for a word w and a context-free grammar G, by the all-correction Earley Parser to compute a CSPPF that stores all indexless corrections ρ for w and G such that $\rho(w) \in L(G)$. We now want to consider how we remove certain corrections from the set of all corrections, represented by a correction shared packed parse forest.

Therefore, we define in Section 5.1 *minimality filters* on CSPPF, these *remove all non-minimal corrections* for some minimality definition so that only the *minimal corrections remain* in a CSPPF. For that the procedure is equivalent to the procedure for disambiguation filters on SPPF (Sec. 2.7). We then consider a specific minimality definition (Sec. 5.1.1) defined in [5], which is specifically designed for training the formulation of hypotheses in biology classrooms [6]. This definition filters out all corrections ρ for which at least one subcorrection of this correction in the equivalence of ρ exists (Definition 5.3).

With this theoretical background of minimality filters and one specific definition of minimality, we introduce in the second part of this section how the concrete filter for this definition of minimality can be implemented in practice. For the sake of simplicity, we divide this filter into three subfilters (*no-loop filter*, *simplification filter*, *no-super correction filter*), whose composition results in the wanted filter. The following task is implemented using the corresponding filters:

1. **No-loop filter** (Sec. 5.2): Removes all syntax trees from a CSPPF that result from *unwinding* a loop.
2. **Simplification filter** (Sec. 5.3): Removes all *non-simplified* indexless corrections from a CSPPF.
3. **No-super correction filter** (Sec. 5.4): Removes all corrections for which a *scattered subcorrection* is also included in the CSPPF.

In addition, all three filters can be used as standalone definition of minimality or as part of the implementation of any other definition of minimality. It should be mentioned that the filters from [9] can also be suitable for the implementation of further minimality definitions.

5.1 Minimality Filters

We define a *minimality filter* on CSPPFs in an equivalent form as a disambiguation filter (Definition 2.8) on the set of syntax trees (typically represented as an SPPF). Note that for a CSPPF ψ, the set of all indexless corrections contained in ψ is defined as $C(\psi)$. (Definition 4.2)

Definition 5.1. Let $G = (N, \Sigma, P, S)$ be a context-free grammar, $w \in \Sigma^*$ and let Γ be the set of all CSPPFs corresponding to G and w. Then a *minimality filter* is a function f that maps the CSPPF ψ ($\psi \in \Gamma$) to a CSPPF ψ' ($\psi' \in \Gamma$) such that: $C(\psi) \supseteq C(\psi')$.

In Section 2.7 we have mentioned that a disambiguation filters can also be generated by a relation over SPPFs (cf. Definition 2.9). In the following we show that the generation of a disambiguation filter by a relation can be transferred to minimality filters.

Lemma 5.1. *Let Σ be a alphabet, $w \in \Sigma^*$, let Γ be the set of all CSPPFs corresponding to w, and let $\prec \subseteq \Gamma \times \Gamma$ be a relation. Then the filter $f^{\prec}$ generated by the relation $\prec$:*

$$f^{\prec}(\psi) = \{\tau \ \xi \ \psi \mid \neg\exists \tau' \ \xi \ \psi : \tau' \prec \tau\}$$

with $\psi \in \Gamma$, is a minimality filter.

Proof. To show that the filter $f^{\prec}$ is a minimality filter, we have to show that

$$C(\psi) \supseteq C\left(f^{\prec}(\psi)\right)$$

holds.

It is obviously that every syntax tree contained in $f^{\prec}(\psi)$ is also contained in ψ. In addition, we have considered in Section 4.1 that each syntax tree in a CSPPF defines exactly one indexless correction through its leaf front (Corollary 4.1). This implies that all indexless corrections contained in $C(f^{\prec}(\psi))$ are also contained in $C(\psi)$ and so $f^{\prec}$ is a minimality filter. □

Lemma 5.1 allow us to use functions as well as relations to define the set of minimal corrections. For example, we can define filters that filter out all indexless corrections from a CSPPF that are of a certain length, produce a certain correction, the corresponding syntax tree has a certain form, and so on. But consider that we can also define minimality filters by *non-computational functions* that we can never implement in practice.

Furthermore, due the structure of CSPPFs, *not all* definable and commutable minimality filters are *applicable* to CSPPFs to remove a certain set of syntax trees from the CSPPF. An example of this is given at the beginning of Section 5.3, where we attempt to remove an non-simplified correction from the CSPPF shown in Figure 18. We see there that this correction cannot be removed from the CSPPF without also removing another correction. We solve the problem by reconstructing the CSPPF so that the non-simplified indexless corrections can be removed. However, to ensure that the reconstructed CSPPF remains a syntactically correct CSPPF according to Definition 4.1, the underlying context-free grammar must be changed. Note that after applying this filter, neither the node set nor the edge set of the resulting CSPPF is a subset of the original CSPPF.

5.1.1 A Definition of Minimality

To give an example of a more complex minimality filter, we consider the definition of minimality from [5] and follow from the definition of minimality the definition of a minimality filter. The definition of minimality from [5] is defined for corrections (Sec. 2.8) for a fixed word w and a target language L as follows: a correction ρ is a minimal correction if and only if:

1. The correction is in normalised form.
2. The correction correct the word w to a word in the target language ($\rho(w) \in L$), such that each intermediate word, produced by the prefix of the correction or by a prefix of a similar correction, is already contain in the target language.

More formally:

Definition 5.2. (cf. [5, Def. 14]) (p-Minimality) Let Σ be a alphabet, $L \subseteq \Sigma^*$ and $w \in \Sigma^*$, then a correction ρ is *potentially minimal* (p-minimal) if ρ is simplified and $\rho(w) \in L$ and $\rho'(w) \notin L$ for every prefix ρ' of ρ.

Definition 5.3. (cf. [5, Def. 16]) (Minimality) Let Σ be a alphabet, $L \subseteq \Sigma^*$ and $w \in \Sigma^*$, then a correction ρ is *minimal* if ρ is p-minimal according to w and L, ρ is in normalised form, and there exists no similar correction that is not p-minimal.

From the definition of minimality and the definition of scattered subcorrection (Definition 3.8) it immediately follows that a normalised correction is a minimal correction if and only if no scattered subcorrection exists that already leads into the target language.

Corollary 5.2. *Let Σ be a alphabet, $L \subseteq \Sigma^*$ and $w \in \Sigma^*$, then a correction ρ is a minimal correction if and only if ρ is in normalised form and there are no scattered subcorrection ρ' ($\neg\exists\rho'$ such that $\rho' \prec_i \rho$) corresponding to w and L (Definition 3.8) so that ρ' leads into the language ($\rho'(w) \in L$).*

We have shown in Section 3.2 that the conversion function *conv* (Definition 3.6) is an order embedding (Theorem 3.7) from the ordered set of normalised corrections into the ordered set of simplified indexless corrections. Therefore, we can use simplified indexless corrections and the partial order $\prec_i$ on the set of simplified indexless corrections (Definition 3.9) to implement the definition of minimality (Definition 5.3) by a minimality filter. However, due to the problems considered in the following, the relation $\prec_i$ still needs to be adjusted slightly. We call the adjusted order $\prec_m$.

A problem with the partial order $\prec_i$ is that they are only defined for simplified indexless corrections. However, a minimality filter defined by a relation requires a relation that is defined for all indexless corrections, not only for the simplified ones. However, due to the definition of minimality considered here, only simplified corrections can be minimal. Furthermore, we know from Section 3 that every simplified correction in normalised form can be converted into a simplified indexless correction, and that for every non-simplified indexless correction there is also a simplified indexless correction. Therefore, we extend the partial order $\prec_i$ on indexless corrections to the partial order $\prec_m$ on indexless corrections by setting each indexless correction ρ' that is *not* in a simplified form in relation ($\rho' \prec_m \rho$) to the corresponding simplified correction ρ. Because for the filter generated by a relation, only an element is minimal if there is no smaller element, and so by this extension, no non-simplified correction can be minimal.

Furthermore, the partial order $\prec_i$ on indexless corrections does not take into account whether the ordered corrections leads into a target language L or not. For example, we assume that for a word w and the indexless corrections ρ and ρ' applies: $\rho \prec_i \rho'$,$\rho(w) \notin L$ and $\rho'(w) \in L$. Then ρ is a reason that ρ' is removed by the filter $f^{\prec_i}$. However, since CSPPFs only contain corrections ρ for which $\rho(w) \in L$ applies. For the relation to $\prec_m$, it is therefore sufficient to define it only on the set of of indexless corrections regarding w, for which $\rho(w) \in L$ also applies, so that the case described above cannot occur.

Definition 5.4. Let $G = (N, \Sigma, P, S)$ be a context-free grammar, $w \in \Sigma^*$ and let ψ be the set of all indexless corrections ρ that correct w into $L(G)$:

$$\psi = \{\rho \mid \rho(w) \in L(G)\}$$

In addition, let ψ' be the subset of ψ that contains only the simplified indexless corrections

$$\psi' = \{\rho \mid \rho \in \psi \text{ and } \rho \text{ is simplified}\}$$

Then the relation $\prec_m \subseteq \psi \times \psi$ is defined as:

$$\begin{aligned}\prec_m = &\{(\rho', \rho) \mid \rho', \rho \in \psi' \text{ and } \rho' \prec_i \rho\} \ \cup \\ &\{(\rho', \rho) \mid \rho' \in \psi', \rho \in \psi \text{ and } \rho \triangleright \rho'\}\end{aligned}$$

Note that we write $\rho \triangleright \rho'$ iff ρ can be simplified to ρ' (Definition 3.4). The relation $\prec_m$ can be used to define a minimality filter $f^{\prec_m}$ which filters out exactly those indexless corrections that are not minimal according to the minimal definition used here. Therefore, applying the filter $f^{\prec_m}$ to a CSPPF ψ that contains all indexless corrections for a word and a language results in a CSPPF that contains all minimal indexless corrections in simplified form.

We can observe that a correction ρ is by Definition 5.3 minimal for a word and a context-free language if and only if the equivalent indexless correction $conv(\rho)$ (where *conv* is the

minimal correction	minimal indexless correction	$\rho(w)$
$[+/_0 n]$	$[\varepsilon\uparrow,\ +/n,\ \varepsilon\uparrow]$	n
$[+/_0(,\ )\uparrow_1,\ n\uparrow_1]$	$[\varepsilon\uparrow,\ +/(,\ n)\uparrow]$	(n)
$[+/_0),\ n\uparrow_0,\ (\uparrow_0]$	$[(n\uparrow,\ +/),\ \varepsilon\uparrow]$	(n)
$[n\uparrow_1,\ n\uparrow_0]$	$[n\uparrow,\ +\rightarrow,\ n\uparrow]$	$n+n$

Table 7: All minimal corrections for the word + and the context-free language L_{aexpr} from Example 2.3 (left). As well as all minimal indexless corrections equivalent to the minimal corrections (Centre). And the resulting words by applying these corrections to the word + (right). Note: To avoid confusion with the alphabet symbols (and), we use square brackets instead of round brackets for the corrections and indexless corrections.

conversion function) is a minimal indexless correction corresponding to the minimality filter $f^{\prec_m}$ applied to ψ. Therefore, we can use the minimality filter to implement the considered minimality definition in practice.

Corollary 5.3. *Let $G = (N, \Sigma, P, S)$ be a context-free grammar, $w \in \Sigma^*$ and let ψ be the set of all indexless corrections ρ that correct w into $L(G)$. Then for a correction ρ we have that:*

$$\rho \text{ is minimal} \iff conv(\rho) \in f^{\prec_m}(\psi)$$

Example 5.1. For the word + and the context-free language L_{aexpr} from Example 2.3 we can determine 4 different minimal corrections by using the minimality definition considered in this section (Definition 5.3), which are presented in the left column of Table 7. For each of these corrections, the middle column of the Table 7 shows the equivalent indexless correction we get by converting the minimal correction with the conversion function (Definition 3.6).

Therefore, the correction shared packed parse forest that we get by applying the minimality filter $f^{\prec_m}$ to the CSPPF of Example 4.1 shown in Figure 10, which contains all minimal indexless corrections for the word + and the context-free language L_{aexpr}, must contain exactly these 4 indexless corrections. These resulting CSPPF are shown in Figure 14.

5.2 No-Loop Filter

We have defined with $f^{\prec_m}$ a minimality filter. We now want to consider how this filter can be implemented in practice. As mentioned above, we divide this into three steps, as shown in Figure 15. In this section, we look at the first step with the *no-loop filter*. The first step is the removal of all indexless corrections from the given correction shared packed parse forest, for which at least one insertion word x_i is created by unwinding a cycle from the CSPPF. This has the advantage that, after applying this filter to a correction shared packet parse forest, the resulting CSPPF only contains a finite number of corrections. Therefore, in this section, we will first explain that no correction that can be derived from a CSPPF by unwinding a cycle is a minimal correction, using the minimality Definition 5.3. Second, we consider how to remove all these corrections from a given CSPPF in practice by using the minimality filter $f_{\text{no\_loop}}$ (Alg. 8).

Loops in CSPPF A *loop* or *cycle* in a CSPPF is a sequence of symbol and intermediate nodes $v_0, v_1, \ldots, v_k$ so that for each v_i with $i < k$ the node v_{i+1} is in at least one child family of v_i and $v_0 = v_k$. We can observe that the extent of all nodes in a loop is the same. This implies that, in addition to the nodes in the cycle, only insert operation nodes (or nodes from which only insert operation leaves can be derived) can be included in the derived

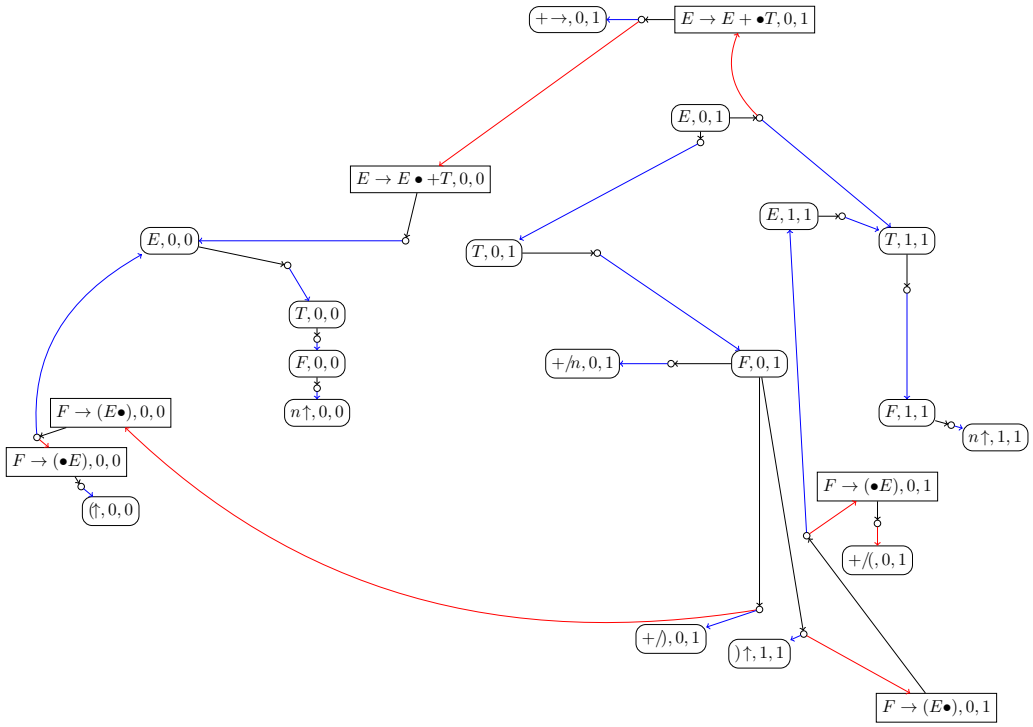


Figure 14: The correction shared packed parse forest resulted from applying the minimality filter $f^{\prec_m}$ to the CSPPF of Figure 10.

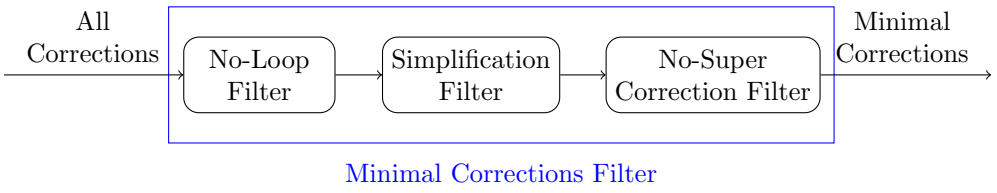


Figure 15: Realization of the minimality filter for the definition of minimality defined in Sec. 5.1.1.

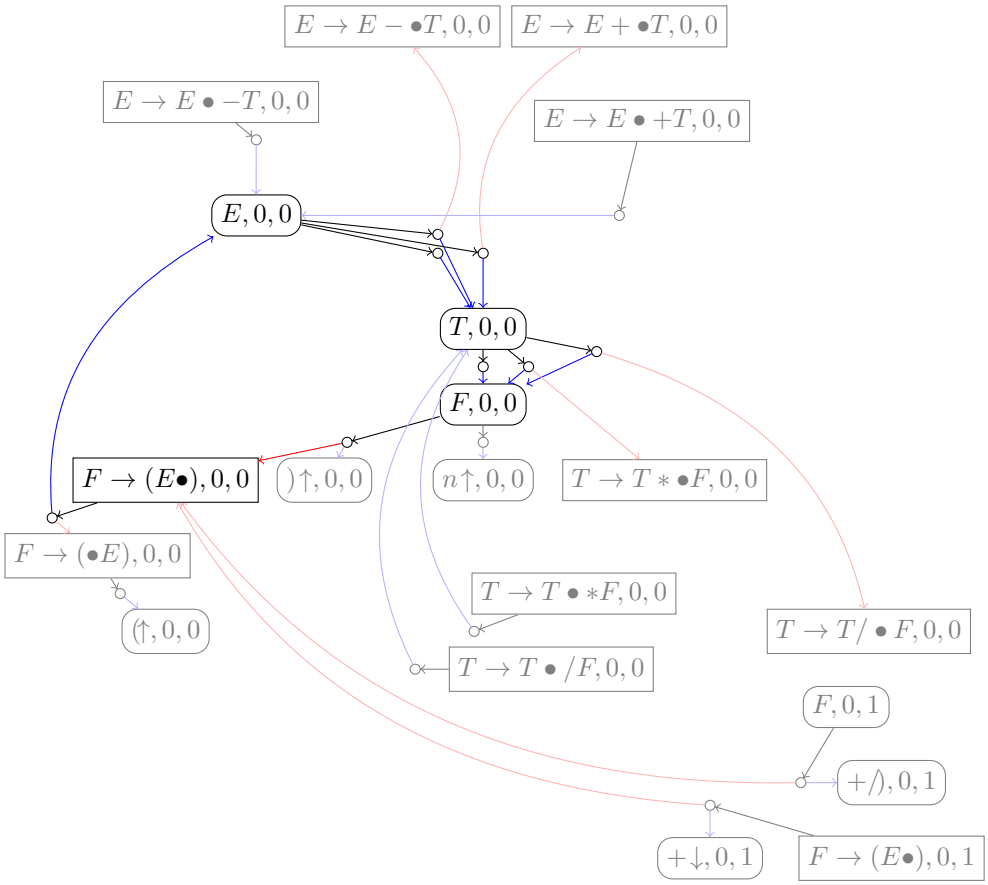


Figure 16: An induced subgraph of the CSPPF from Figure 10, represents an exemplary loop. The non-grayed nodes are contained in the loop, and the grayed nodes are either the predecessor or a successor node of a node in the loop.

families. Since the definition of CSPPF (Definition 4.1) requires that for each symbol node representing an indexless read, replacement, or deletion operation, its extent is of the form $(i, i+1)$. Therefore, for a family containing such a node, the other node in the family cannot be part of a loop.

The sequence $(E, 0, 0)$, $(T, 0, 0)$, $(F, 0, 0)$, $(F \to (E\bullet), 0, 0)$, $(E, 0, 0)$ is an example of a loop from the CSPPF presented in Figure 10, where we omit for the sake of simplicity packed nodes also contained in this loop. The induced subgraph, consisting of the nodes of the cycle, the participating nodes of a derived family, and the nodes that lead into the cycle, are shown in Figure 16. We can observe that for each unrolling of the loop, the leaves $((\uparrow, 0, 0)$ and $()\uparrow, 0, 0)$ are also derived through the derived families of the cycles. Therefore, the corresponding indexless correction of the syntax tree, which results from unrolling the exemplary loop k-times, is for the first insertion word x_0 of the following form:

$$x_0 = \underbrace{((\ldots((}_{k\text{-times}} \quad x_0' \quad \underbrace{))\ldots))}_{k\text{-times}}$$

where x_0' is the word inserted after the unrolling. We can observe that x_0' is a scattered subword of x_0 ($x_0' \prec x_0$). This implies that any indexless correction ρ that starts with the indexless edit operation $x_0\uparrow$ cannot be minimal (using the minimality Definition 5.3) because the same indexless correction ρ' with the only difference that the first operation is $x_0'\uparrow$ is a subcorrection ($\rho' \prec_i \rho$). This simple introductory example shows that any indexless correction resulting from unrolling a loop cannot be minimal.

Idea of the No-Loop Filter The *no-loop filter* (written as $f_{\text{no\_loop}}$) is defined by Algorithm 8. The algorithm reconstructs a given CSPPF ψ such that the resulting CSPPF is acyclic and contains exactly the indexless correction in ψ that can be derived without unwinding a loop. Note that there are other ways to implement the no-loop filter besides the presented reconstruction, such as redefining which syntax trees are contained in a correction shared packed parse forest (Definition 2.6) by excluding the unwinding of loops.

To implement the no-loop filter, we consider exactly one loop that is contained in the CSPPFs and remove the loop by rebuilding the CSPPFs. Before we restart the algorithm recursively to remove the next loop. For the loop considered in the current step, we create a copy of the loop for each symbol or intermediate node contained in the loop and removing this loop from the resulting CSPPF. In each copy, the loop is completely unwound once, starting from the corresponding node but without returning to this node. The corresponding node is the only node that has predecessor nodes that are not contained in the loop. Therefore, we only need to create a copy loop of symbol or intermediate nodes that are contained in at least one family of a node outside the loop. The child families of the nodes in the copy remain unchanged from the original loop, except for the last node of the unwinding. Here we need to remove the child family, leading back to the first node of the copy.

Once we have done this for the considered loop, we will look at the next loop and reconstruct the CSPPF for that loop in the same way. We repeat this until there no more loops in the correction shared packed parse forest. Therefore, the CSPPF resulting from the described reconstruction of all loops is an acyclic directed graph. This implies that it contains a finite number of indexless corrections, since no loop can unwind.

Algorithm Here we present the recursive implemented algorithm (Alg. 8) that defines the no-loop filter ($f_{\text{no\_loop}}$) in detail. The algorithm requires a CSPPF given as a directed graph (V, E), where V is the set of nodes and E is the set of edges.

The first step of the algorithm is the finding of a loop in the graph; this is outscored to the auxiliary function FIND_LOOP. This auxiliary function will not be considered in detail because it can be implemented by a simple depth-first search. Note: To optimize the search, it is sufficient to search the subgraphs of nodes with extent (i, i), since we have already discussed that a loop can only occur in these parts of a CSSPF. We assume that the FIND_LOOP function returns the set of nodes C that are contained in a loop of the CSPPF and if there are no loops in it then the function returns the empty set. Therefore, we know that if this set is empty, the algorithm has removed all loops from the CSPPF and can return it.

In the other case, where we have identified a loop, we remove this loop from the CSPPF and create the copies as described above. Therefore, for each symbol and intermediate node, we first check whether it is contained in at least one family of a node outside the loop. If it isn't, then we can't reach that node from outside the loop. This implies we don't need to consider it as the starting node of the loop unwinding and, therefore, don't need a copy of the loop for that node.

For nodes v that we can reach from outside the loop, we start the unwinding of the loop from that node v. So we first create a set V_v containing all the nodes we need to consider in the following for the current unwinding. Thereafter, we create a new node that represents the entry to the unwinding. Therefore, we set this node as the predecessor of all nodes that have the node v in the original CSPPF as their successor. To uniquely identify the newly created nodes, we label them with a triple (u, v, C), where u is the original node, C is the loop we are considering, and v is the start of the unwind. To compute the child families of this node, we use the auxiliary function ADD_FAMILIES that we present in Algorithm 9.

Note: In order to ensure that the resulting CSSPF is a syntactically valid CSPPF according to Definition 4.1, the context-free grammar on which the CSPPF is based must be slightly adapted due to the new nodes. To do this, the described triples must be set as the set of nonterminals and the production rules must be adapted with regard to the new

Algorithm 8 NO_LOOP_FILTER

Require: A CSPPF $\psi = (V, E)$
Ensure: $f_{\text{no\_loop}}(\psi)$

$C \leftarrow$ FIND_LOOP(V, E)

if $C = \emptyset$ **then**
    return ψ
end if

$V' \leftarrow V \setminus C$
$E' \leftarrow E \setminus \{(u, v) \mid (u, v) \in E \text{ and } (u \in C \text{ or } v \in C)\}$
$C' \leftarrow C \setminus \{v \mid v \in C \text{ and } v \text{ is an packed node}\}$
$O_C \leftarrow \{v \mid v \in C \text{ and } v \text{ has a child family for whom all nodes are not contained in } C'\}$

for all $v \in C'$ **do**
    $I_v \leftarrow \{(u, v') \mid (u, v') \in E \text{ and } v' = v\}$

    if $I_v \neq \emptyset$ **then**
        $V_v \leftarrow C' \setminus \{v\}$
        Add (v, v, C) to V'

        for all $(u, v') \in I_v$ **do**
            Add $(u, (v, v, C))$ to E'
        end for

        ADD_FAMILIES(v, v, C, V_v, O_C)

        while $(V_v \cap O_C) \neq \emptyset$ **do**
            $u \leftarrow C.NEXT()$
            $V_v \leftarrow V_v \setminus \{u\}$
            Add (u, v, C) to V'
            ADD_FAMILIES(u, v, C, V_v, O_C)
        end while
    end if
end for

return (V', E')

Algorithm 9 ADD_FAMILIES

Require: The symbol and intermediate nodes u, v, the loop C and the sets V_v and O_C

for all child families (u', v') of u **do**
  if $u' \neq v$ and $v' \neq v$ **then**
    $u'' \leftarrow$ **if** $u' \in C$ **then** (u', v, C) **else** u'
    $v'' \leftarrow$ **if** $v' \in C$ **then** (v', v, C) **else** u'

    if $(u' \notin C$ and $v' \notin C)$ or $(V_v \cap O_C) \neq \emptyset$ **then**
      Add (u'', v'') as child family of (u, v, C)
    end if
  end if
end for

for all child families (u') of u **do**
  if $u' \neq v$ **then**
    $u'' \leftarrow$ **if** $u' \in C$ **then** (u', v, C) **else** u'
    if $u' \notin C$ or $(V_v \cap O_C) \neq \emptyset$ **then**
      Add (u'') as child family of (u, v, C)
    end if
  end if
end for

nonterminals.

The ADD_FAMILIES function considers for an original node u all child families and checks which of these families must also be owned by the corresponding copies. The copy must contain the child families in which the start node v of the unwinding is not included. In addition, the child nodes must also be renamed if they are nodes of the loop.

Once we have considered the start node v of the unwind, we look at the remaining nodes of the loop in order of their occurrence in the loop based on the start node. This is done by the NEXT auxiliary function, which we will not discuss further due to its simplicity. For each of these nodes, we create a new node in the described form above and also add the child families with the auxiliary function ADD_FAMILIES.

We can observe that if the last node in the unwinding process has no child family for that all nodes are not contained in the loop, then the copied node has no children but also does not represent an indexless edit operation because we cannot leave the loop from this node. To avoid creating such nodes, we use the sets O_C and V_v. The set O_C contains the loop that has at least one child family, for which all nodes are not contained in the loop. In other words, these are nodes through which the loop can be exited. Therefore, if there are no more nodes to consider for the current unwinding process that are also in the set O_c (which means that $(V_v \cap O_C) \neq \emptyset$ applies), the algorithm stops the current unwinding. Otherwise, it creates in the next steps a node as described above. Furthermore, from the last node in $V_v \cap O_C$ the copy doesn't have a child family where a node is part of the loop. So that we can, from the last node of the unwinding, leave the loop.

After the algorithm has completed the unwinding process for all intermediate and symbol nodes, the algorithm is called recursive again with the reconstructed CSPPF to resolve the next loop.

Example 5.2. As an example of the no-loop filter, we consider the loop from our introductory example shown in Figure 16. For this loop, we can observe that only the nodes $(E, 0, 0)$, $(T, 0, 0)$ and $(F \to (\bullet E), 0, 0)$ are in a family of nodes outside the loop. So we need 3 copies of the loop, one for each of the three nodes. In addition, we can observe that the node $(F, 0, 0)$ is the only symbol or intermediate node in the loop, which with the family

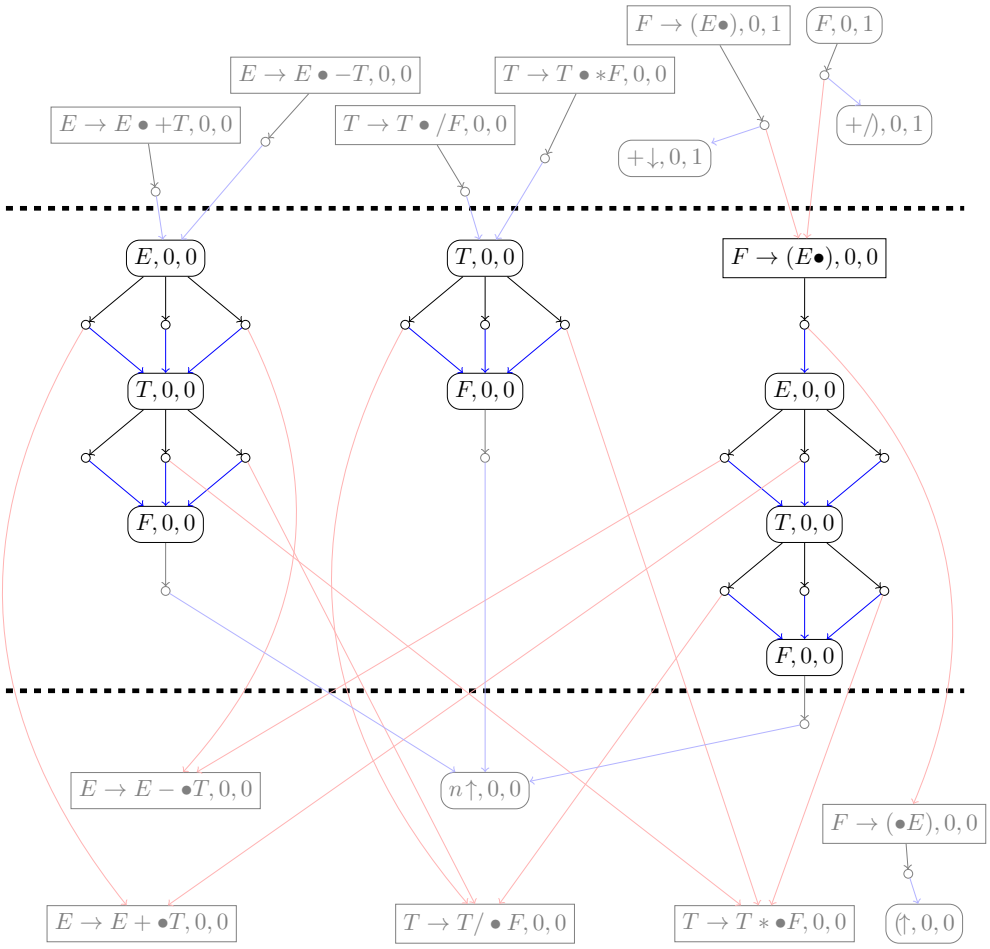


Figure 17: The reconstruction by the no-loop filter of the loop considered in Figure 16. The greyed nodes are either the predecessor or a successor of nodes contained in the original loop. The non-greyed nodes are copies of the nodes in the original loop. For the sake of clarity, these nodes are not labeled with triples as in the algorithm, since in this case, the corresponding triple can be derived from the figure.

$((n\uparrow, 0, 0))$ has a child family for which all the nodes are outside the loop. Therefore, the algorithm (Alg. 8) unwinds each of the three copies only up to this node.

The CSPPF reconstructed by the algorithm regarding the considered loop is shown in Figure 17. Note that this example introduces only one recursive step of the no-loop filter.

5.3 Simplification Filter

We have removed all loops from a CSPPF by the non-loop filter. We can filter out the set of minimal corrections from the current CSPPF by comparing all the indexless corrections stored in it, because the current CSPPF contains only a finite number of indexless corrections. However, this would lead to many comparisons of indexless corrections, which estimate makes the algorithm estimate slow. Therefore, we want to remove further indexless corrections from the CSPPF for which we already know that they cannot be minimal regarding the used definition of minimality (Definition 5.3). Such indexless corrections are all that are not in simplified form.

Therefore, in this section, we present a filter called *simplification filter* (written as f_{simp}), which removes all non-simplified indexless corrections from a CSPPF.

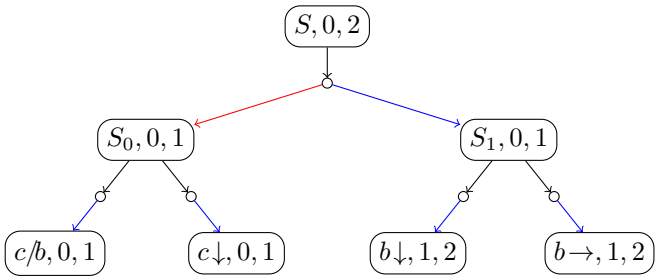


Figure 18: An exemplary CSPF that shows that we cannot remove edges or nodes from it to remove the indexless corrections.

Simplification and CSPPF But before we introduce the simplification filter, let us make some observations on the on-step simplifications for indexless corrections (Definition 3.4). For all one-step simplifications in Table 4, we can observe that only neighboring indexless edit operations can be simplified. On closer inspection, we can also see that if the indexless correction is represented by a syntax tree in a CSPPF, only adjacent leaves can be simplified. Adjacent leaves are pairs of leaves with extents (i, j) and (i', j') such that $j = i'$. If there is at least one adjacent pair of indexless edit operations in the syntax tree that represents the indexless correction, that can be simplified (Definition 3.4), that indexless correction is not simplified.

From Lemma 4.3 we know that if we compute a correction shared packed parse forest for a word and a language with Algorithm 3, the CSPPF contains all indexless corrections that correct the word to a word in the language. We can also observe that this is not changed by the previous application of the non-loop filter. Therefore, all simplified indexless corrections are already included in the CSPPF. This means that there is no need to simplify non-simplified corrections because the resulting simplified correction is already contained in the CSPPF. Therefore, we can easily remove all non-simplified indexless corrections from the correction shared packed parse forest.

The only problem is: How can we remove all these non-simplified indexless corrections from the CSPPF without also removing a simplified indexless correction? This is a challenging task because, due to the concept of node sharing in CSPPFs we have that in the subtree below a node, there can be parts of simplified and non-simplified indexless corrections. Therefore, we can neither remove this node nor the edges leading to it. An example of this issue is the CSPPF shown in Figure 18. This correction shared packed parse forest contains the simplified indexless corrections $(\varepsilon\uparrow, c/b, \varepsilon\uparrow, b\rightarrow, \varepsilon\uparrow)$, $(\varepsilon\uparrow, c\downarrow, \varepsilon\uparrow, b\downarrow, \varepsilon\uparrow)$, $(\varepsilon\uparrow, c\downarrow, \varepsilon\uparrow, b\rightarrow, \varepsilon\uparrow)$ and the non-simplified indexless correction $(\varepsilon\uparrow, c/b, \varepsilon\uparrow, b\downarrow, \varepsilon\uparrow)$. We can observe that removing any edge or node that is contained in the syntax tree of the non-simplified indexless correction results in at least one simplified correction also no longer being contained in the CSPPF. Therefore, we cannot remove nodes or edges from the CSPPF to remove all non-simplified indexless corrections from it. Instead, we need to restructure the CSPPF to remove the non-simplified indexless corrections.

Idea of the Simplification Filter We already know that only adjacent leaves can be simplified. Let u, v be such a pair of leaves that can be simplified. Let (i, j) be the extent of the node u and let (i', j') be the extent of the node v, then since they are adjacent leaves, $j = i'$ must apply. Because of the properties of a CSPPF, we can observe that for exactly one packed node x in the CSPPF, we have that u can be the rightmost leaf of at least one syntax tree in the left subtree of x and v is the leftmost leaf of at least one syntax tree in the right subtree of x. In addition, we have that for the extent i_p, j_p of x it must hold that: $i_p \geq i$ and $j \leq j_p$ and that the pivot element has the value j (remember $j = i'$).

Note that the nodes u and v represent indexless edit operations. Therefore, the idea of

the simplification filter is to create a copy for each symbol, packed and intermediate node for every possible combination of pairs of edit operations and assign each syntax tree to the corresponding copy of the node. This allows us to remove the families (packed nodes) for which the pair of edit operation can be simplified.

We have already discussed above that it is sufficient to remove the non-simplified indexless correction from the CSPPF because the simplified one is also already contained in the CSPPF. It is therefore sufficient to know whether a syntax tree can be simplified or not, without knowing how the simplification looks like. This means that we can combine the different leftmost and rightmost leaves into groups so that we already have a constant number of copies for each symbol and intermediate node.

Groups of Leaves To determine which groups of leftmost and rightmost nodes in a subtree are needed, we consider the Table 4 with the one-step simplifications for indexless corrections. We can observe that for the *leftmost node* we require the groups:

- Delete the current character (written as: $a_i\downarrow$).
- Replace the current character with another letter (written as: a_i/b).
- Insert words starting with the last character (written as: $a_{i-1}\uparrow$).
- Insert words starting with a letter unequal to the last character (written as: $b\uparrow$ with $b \neq a_{i-1}$).
- Others (written as: e).

Except for the other group, all of them can possibly lead to a one-step simplification. In addition, we can observe that for the *rightmost node* we need the groups:

- Insertion of words ending with the next character (written as: $a_{j+1}\uparrow$).
- Insertion of words ending with a character unequal to the next character (written as: $d\uparrow$ with $d \neq a_{j+1}$).
- Deletion of the current character (written as: $a_j\downarrow$).
- Replacement of the current character by the next character (written as: a_j/a_{j+1}).
- Others (e).

This gives us 5 different groups for the leftmost leaf and 5 different groups for the rightmost leaf. Therefore, we require at most $5 \cdot 5 = 25$ copies for each symbol or intermediate node to represent all different types of syntax trees. In practice, all 25 copies are rarely needed, exactly when there is at least one syntax tree for each of the 25 cases that can be derived from this node.

The copies of the nodes make it possible to assign each subtree of the original CSPPF to exactly one copy. The assignment can be calculated bottom-up from the leaves. First, we determine the right and left groups for all edit nodes. Note that each edit node can have exactly one right and one left group, defined by its indexless edit operation. For families of size 2, the group membership can then be determined by the left group of the left subtree and the right group of the right subtree. For families with only one node, the group memberships are transferred. Note that this changes the structure of the CSPPF because the same family of the original CSPPF can appear as the child of more than one copy, each of them represents a disjoint set of syntax trees from the original family.

This division of the syntax trees according to the properties of the left and right nodes enables us to subsequently remove the syntax trees that can be simplified.

x \ y	$a_i\downarrow$	a_i/c	$a_{i-1}\uparrow$	$b\uparrow$	e
$a_{j+1}\uparrow$	$(b\uparrow_i, b\downarrow_{i+1}),\ (b\downarrow_i, c\uparrow_i)$	$(c/_i b, c\uparrow_i)$	✓	✓	✓
$d\uparrow$	$(b\downarrow_i, b\uparrow_i)$	✓	✓	✓	✓
$a_j\downarrow$	✓	✓	$(b\downarrow_i, b\uparrow_i)$	$(b\downarrow_i, b\uparrow_i)$	✓
a_j/a_{j+1}	$(c/_i b, b\downarrow_{i+1})$	✓	$(c/_i b, c\uparrow_{i+1})$	✓	✓
e	✓	✓	✓	✓	✓

Table 8: Allowed (green cells) and not-allowed pairs (red cells) in simplified CSPPF. Where $a_i\downarrow$, a_i/c with $c \in \Sigma$, $a_{i-1}\uparrow$, $b\uparrow$ with $b \neq a_{i-1}$, e are the left group of the right subtree and $a_{j+1}\uparrow$, $d\uparrow$ with $d \in \Sigma \setminus \{a_{j+1}\}$, $a_j\downarrow$, a_j/a_{j+1}, e are the right groups of the left subtree. In addition, for the not-allowed families, the cell content indicates which simplification rule from Table 4 is covered.

Removing of Non-Simplified Indexless Corrections Starting with the restructured correction shared packed parse forest, the syntax trees that represent a non-simplified correction can be easily removed. Therefore, we remove any family where the right group property of the left subtree and the left group property of the right subtree cover a case that can be simplified. The allowed pairs (green cells) and the pairs that need to be removed (red cells) are shown in Table 8.

Algorithm As mentioned above, the algorithm (Alg. 10) that defines the simplification filter (f_{simp}) combines the two considered steps of restructuring the required CSPPF ψ and removing not-allowed families.

In order to be able to store a unique node for each of the described copies in the resulting simplified CSPPF ψ' ($\psi' = \{\rho \mid \rho \in \psi \text{ and } \rho \text{ is simplified}\}$), we label the nodes of ψ' with triples (y, v, x). Where v is a node of the original CSPPF ψ and y describe the group of the leftmost leaves ($y \in \{a_i\downarrow, a_i/c, a_{i-1}\uparrow, b\uparrow, e\}$) and x is the group of the rightmost leaves ($x \in \{a_{j+1}\uparrow, d\uparrow, a_j\downarrow, a_j/a_{j+1}, e\}$). Note In order for the resulting CSPPF to be a syntactically valid CSPPF according to Definition 4.1, the underlying grammar must be slightly modified by storing the group information in the nonterminals and adjusting the production rules accordingly.

At first, the algorithm starts to create a new node for each edit node of ψ that has the group of the left- and rightmost node described above. To achieve this, we use the auxiliary functions LEFT_GROUP and RIGHT_GROUP, which easily define the group the edit nodes. Note: There is exactly one left group and one right group for each edit node. Therefore, only one copy is required for each edit node.

In addition, all edit nodes are set as marked (add to set M). All symbol and intermediate nodes of ψ are added to the unmarked set R. All packed nodes of ψ are added to the P_1 or P_2, where the packed node with 1 successor added to P_1 and the packed node with 2 successors are added to P_2. Then the algorithm loops until all nodes have been considered once. This means that all symbol and intermediate nodes are selected ($R = \emptyset$) and all packed nodes are removed from the sets P_1 or P_2 ($P_1 = P_2 = \emptyset$).

We consider a packed node in P_1 or P_2 if all successors nodes are set as marked. Then we create the necessary copies, described in detail below, and remove them from P_1 or P_2. A symbol or an intermediate node is marked if and only if the algorithm does not have to be considered a successor packed node.

In other words, we build the new CSPPF bottom-up, starting with the edit nodes in ψ. Then, based on the new edit nodes, we successively compute the necessary copies for

Algorithm 10 SIMPLIFICATION_FILTER

Require: A CSPPF ψ
Ensure: A simplified CSPPF $\psi' = \{\rho \mid \rho \in \psi \text{ and } \rho \text{ is simplified}\}$

$V, M \leftarrow \emptyset$

for v is a leave in ψ **do**
    $V \leftarrow V \cup \{(\text{LEFT\_GROUP}(v), v, \text{RIGHT\_GROUP}(v))\}$
    $M \leftarrow M \cup \{v\}$
end for

$R \leftarrow \{v \mid v\text{is a symbol or intermediate node in } \psi\} \setminus M$
$P_2 \leftarrow \{(u, v) \mid (u, v) \text{ is a familiy/ packed node in } \phi\}$
$P_1 \leftarrow \{v \mid (v) \text{ is a familiy/ packed node in } \phi\}$

while $R \neq \emptyset$ **do**
    for v in P_1 **do**
        if $v \in M$ **then**
            $P_1 \leftarrow P_2 \setminus \{v\}$

            for (y, v, x) in V **do**
                $V \leftarrow V \cup \{(x, z, y)\}$ ▷ Where (v) is a child family of z
                add $((y, v, x))$ as child family of (x, z, y)
            end for
        end if
    end for

    for (u, v) in P_2 **do**
        if $u \in M$ and $v \in M$ **then**
            $P_2 \leftarrow P_2 \setminus \{(u, v)\}$

            for (y_u, u, x_u) in V **do**
                for (y_v, v, x_v) in V **do**
                    if ALLOWED_FAMILY(x_u, y_v) **then**
                        $V \leftarrow V \cup \{(y_u, z, x_v)\}$ ▷ Where (u, v) is a child family of z
                        add $((y_u, u, x_u), (y_v, v, x_v))$ as child family of (x, z, y)
                    end if
                end for
            end for
        end if
    end for

    for v in R **do**
        if no child family of v is in P_1 or P_2 **then**
            $R \leftarrow R \setminus \{v\}$
        end if
    end for
end while

for (y, s, x) in V **do** ▷ Where s is the root node of ψ
    add $((y, s, x))$ as child family of s' ▷ s' is the new root node of ψ'
end for

return S'

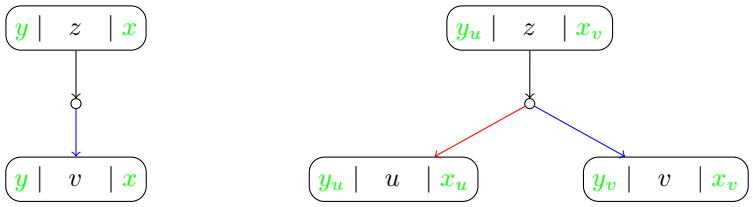


Figure 19: The building process of the simplification filter. On the left for families of size one and on the right for families of size two. Where on the right it must apply that $(x_v.Y_)$ is an allowed pair by Table 8

all predecessor nodes. Where we only consider a packed node when all subtrees have been fully computed. This is possible because in our case, the CSPPF has no loops, due to the previously used no-loop filter (Sec. 5.2). In the case that we use the simplification filter as a standalone filter, we need to adapt the algorithm to also handle CSPPFs with loops. But since we know from Section 5.2 that loops can only occur as a sequence of insertions and that the groups are computed by the first and last inserted characters, we don't need to worry about the internal structure of the loops in the adjustment.

After we have determined the order in which we build the new CSPPF ψ', we look at the exact structure of the nodes and edges of ψ'. In the build process, we distinguish between packed nodes in P_1 and packed nodes in P_2. In addition to the following description, Figure 19 introduces the build process graphically. For packed nodes in P_1 both, the rightmost group and the leftmost group are transferred from the child of the packed node to the predecessor. This is shown on the left in Figure 19. For packed nodes in P_2, for all combinations of copies of the left child node and the right child node, we first check whether the combination of the right group of the left child x_u and the left group of the right child y_v is a valid combination by Table 8. This check is performed in the algorithm by the ALLOWED_FAMILY auxiliary function. If we have a valid combination, the groups of the predecessor node will be the left group of the left child node and the right group will be the right group of the right's child node. This is shown on the right in Figure 19. In the other case, when we don't have a valid combination, we don't add the children of the packed node as a family of the predecessor.

At the last step of the algorithm, we add a new root node (S') that has every existing copy of the old root node as a child family.

Example 5.3. As an example of the simplification filter, we apply it ($f_{simp}(\psi)$) on the CSPPF (ψ) from the introductory example in Figure 18. The result is shown in Figure 20.

5.4 No-Super Correction Filter

The last step of implementing the minimality filter $f^{\prec_m}$ is the removal of all corrections from a CSPPF for that, regarding the relation $\prec_m$ (Definition 5.4), are a subcorrection is contained in the CSPPF. This is realized by the no-super correction filter (written as f_{no_super}) that we present in this section. This filter compares each indexless correction ρ with all other indexless corrections ρ' that are contained in a CSPPF to check if ρ is a scatterd super correction of ρ' ($\rho \succ_m \rho'$). If this is the case, then we know that ρ cannot be a minimal correction regarding Definition 5.3 and can therefore be removed. This means that at the end of the algorithm, only the minimal corrections remain.

Since the filter compares each indexless correction with all other indexless corrections, we can only apply the super correction filter to CSPPFs that contain only a finite number of indexless corrections. For a CSPPF with an infinite number of indexless corrections, the algorithm doesn't terminate. We already know that a correction shared packed parse forest

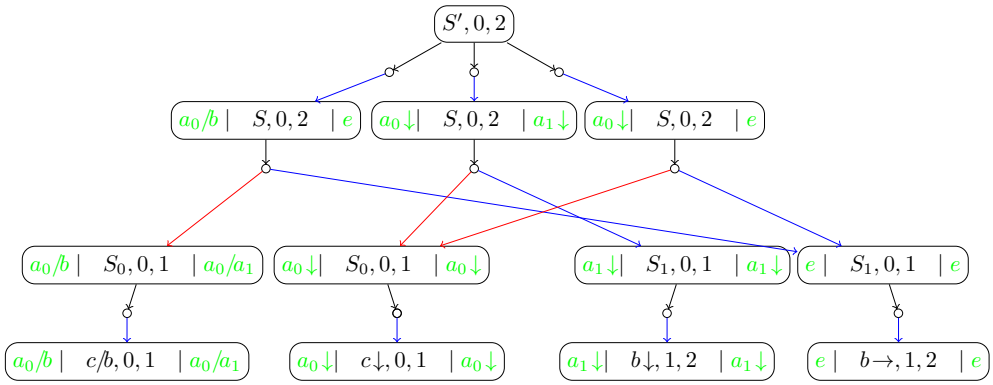


Figure 20: The simplified correction shared packed parse forest for the CSPPF from Figure 18.

contains a finite number of corrections if and only if it is acyclic. This means that the no-super correction filter is applicable in our use case to implement $f^{\prec_m}$ because the CSPPF is already acyclic due to the prior application of the no loop filter (Sec. 5.2).

Algorithm We have already mentioned that the idea of the algorithm that defines the no-super correction filter is the comparison of each correction with all other corrections. Therefore, the implementation is straightforward. Nevertheless, we will briefly discuss some optimizations and the functionality of this algorithm (Alg. 11) below.

Firstly, remember that $C(\psi)$ for a CSPPF ψ is the set of all corrections that are contained in this CSPPF (Definition 4.2). For all indexless corrections ρ that are in $C(\psi)$ we check if they is a correction ρ' in $C(\psi)$ that is a scattered sub-correction ($\rho' \prec_m \rho$). If we cannot find such a correction, then we know that this correction is minimal and can be added to the set of minimal corrections M. In addition, we also compute the set of scattered super-corrections of the current considered corrections. Since these corrections cannot be minimal and therefore do not need to be considered by the algorithm in the following. To calculate this, we use the fact that all indexless corrections in a CSPPF have the same length. The algorithm first compares the indexless correction at position 0, then at position 1, and so on. At each step, the algorithm groups each correction into one of three sets: E_c the set of indexless corrections up to the current position equal to c; B_c the set of indexless corrections up to the current position greater than c; and S_c the set of indexless corrections up to the current position less than c. Therefore, after comparing all indexless edit operations, the set B_c contains all scattered super-corrections, and the current considered indexless correction c is minimal if and only if the set S_c is empty. Because the algorithm has not found a correction that is at all indexless edit operation less than or equal to the currently considered indexless correction and smaller by at least one edit operation. Therefore, there is no scattered sub-correction in CSPPF.

For the sake of simplicity and since no further filters are subsequently applied in our use case, the algorithm returns the set of minimum corrections instead of a CSPPF. But note that it is quite simple to compute for the set of minimal correction a CSSPF at the end of the algorithm, so that the algorithm defines a minimality filter regarding Definition 5.1. By building a CSPPF from the syntax trees according to the principles of packing and splitting nodes. Care must be taken to ensure that no further syntax trees are added to the CSPPF as a result of packing.

Example 5.4. As an example for the no-super correction filter, we consider the CSPPF ψ presented in Figure 21. We can see that this CSPPF is similar to the CSPPF of Example 5.1 shown in Figure 14, which contains all minimal corrections for the word n and the context-

Algorithm 11 NO_SUPER_CORRECTION_FILTER

Require: A acyclic CSPPF ψ for a word of length n

Ensure: The set of indexless corrections contained in ψ, so there are no scattered sub correction contained in ψ.

$C \leftarrow C(\psi)$
$M \leftarrow \emptyset$

for $c \in C$ **do**
  $E_c \leftarrow C \setminus \{c\}$
  $B_c, S_c \leftarrow \emptyset$

  for $i = 0$ to $2n$ **do**
    for $c' \in E_c$ **do**
      if $c[i] \neq_{ie} c'[i]$ **then** ▷ where $c[i]$ is the $i-$th indexless edit operation of c
        $E_c \leftarrow E_c \setminus \{c'\}$
      else if $c[i] \prec_{ie} c'[i]$ **then**
        $E_c \leftarrow E_c \setminus \{c'\}$
        $B_c \leftarrow B_c \cup \{c'\}$
      else if $c[i] \succ_{ie} c'[i]$ **then**
        $E_c \leftarrow E_c \setminus \{c'\}$
        $S_c \leftarrow S_c \cup \{c'\}$
      end if
    end for

    for $c' \in S_c$ **do**
      if $c[i] \neq_{ie} c'[i]$ or $c[i] \prec_{ie} c'[i]$ **then**
        $S_c \leftarrow S_c \setminus \{c'\}$
      end if
    end for

    for $c' \in B_c$ **do**
      if $c[i] \neq_{ie} c'[i]$ or $c[i] \succ_{ie} c'[i]$ **then**
        $B_c \leftarrow B_c \setminus \{c'\}$
      end if
    end for
  end for

  if $S_c = \emptyset$ **then**
    $M \leftarrow M \cup \{c\}$
  end if

  $C \leftarrow C \setminus B_c$
end for

return M

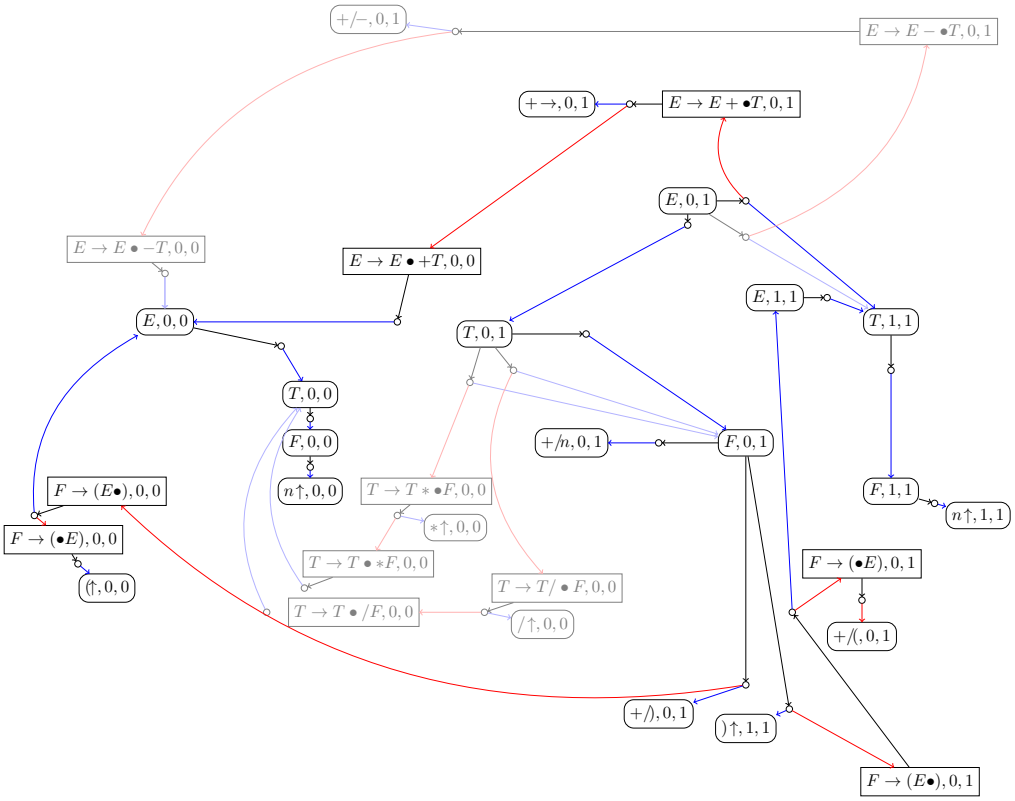


Figure 21: A CSPPF to introduce the functionality of the super-correction filter. This filter removes all the grayed nodes and edges of the CSPPF, so that after applying the filter, only the corrections consisting of non-grayed nodes and edges remain.

free language L_{aexpr}, except that we have added some additional nodes and edges (the gray nodes and edges). Therefore, we have besides the four minimal corrections

$$[\varepsilon\uparrow,\ +/n,\ \varepsilon\uparrow],\ [\varepsilon\uparrow,\ +/(,\ n)\uparrow],\ [(n\uparrow,\ +/),\ \varepsilon\uparrow] \text{ and } [n\uparrow,\ +\rightarrow,\ n\uparrow]$$

the following additionally corrections are contained in the CSPPF ψ. Where we have to avoid confusion between the terminal symbol / and the replacement operation, we use : instead of the terminal symbol /.

$$[n\uparrow,\ +/-,\ n\uparrow],\ [n*\uparrow,\ +/n,\ \varepsilon\uparrow],\ [n:\uparrow,\ +/n,\ \varepsilon\uparrow],\ [n*\uparrow,\ +/(,\ n)\uparrow]$$

$$[n/\uparrow,\ +/(,\ n)\uparrow],\ [n*(n\uparrow,\ +/),\ \varepsilon\uparrow] \text{ and } [n:(n\uparrow,\ +/),\ \varepsilon\uparrow]$$

But for all of this additional corrections ρ, it holds that they are minimal correction that are a scattered subcorrection of ρ. For example, we have that $[n\uparrow,\ +\rightarrow,\ n\uparrow] \prec_s [n\uparrow,\ +/-,\ n\uparrow]$ or that $[\varepsilon\uparrow,\ +/(,\ n)\uparrow] \prec_m [n*\uparrow,\ +/n,\ \varepsilon\uparrow]$. Therefore, all these additional corrections are removed from the CSPPF by the no-super correction filter, so that the set of minimal corrections remains.

Comment on the Minimality Filters The three presented filters f_{no_loop}, f_{simp} and f_{no_super} introduce how a more complex filter definition can be implemented in practice. In the considered case, the minimality filter $f^{\prec_m}$ defined by the relation $\prec_m$ (Definition 5.4) be realised by the concatenation of the presented filters:

$$f^{\prec_m}(\psi) = f_{no\_super}\left(f_{simp}\left(f_{no\_loop}(\psi)\right)\right)$$

where ψ is some CSPPF, typically computed by the all-corrections Earley Parser (Alg. 3) for a word and context-free grammar.

However, it should be noted that we have not proven the correctness of the filters. To show that the composition of the three filters in the presented order implements the filter $f^{\prec_m}$, it would have to be shown that each non-minimal correction is removed from a given CSPPF without also removing a minimal correction. More detailed for a CSPPF ψ we need to shown that none of the three filters removes a minimal correction in ψ. In addition, we need to show that every non-minimal correction is removed by at least one of the three filters.

It should be noted that both the no-loop filter and the simplification filter use the structure of the CSPPF to filter out a set of non-minimal corrections. The implementation of the no-super-correction filter presented in this work doesn't use the properties of the structure. But in an optimized version of the filter, it probably makes sense to use the structure as well. This is because two corrections that have the same indexless edit operation at the same position also use the same leaves of the CSPPF. This means that many comparisons can be saved in an optimized version by using the structure. Note, that in most of the visualizations of CSPPFs in this work, we have often printed the same edit node more than once for the sake of clarity, even though it is the same node.

These structure properties show the advantages of CSPPFs to store a set of indexless corrections and allow the definition of filters in a simple way. This makes it possible to implement further minimality filters for CSPPFs with practical benefits in addition to the filters presented here.

For example, to use different filters in certain parts of the correction. For example, we want to use a different filter inside a loop body of an algorithm than in the rest of the algorithm code. This is easily realized if the used grammar has a nonterminal that represents the loop body and from which the body code can be derived. Then we can apply the filter we want to use inside a loop body to all subtrees below the symbol nodes of this nonterminal.

6 Conclusion

The main goal of this work was to find an algorithm that computes the set of all *minimal corrections* for a given *definition of minimality* in an *efficient way*.

To achieve this, we have presented the all-correction Earley Parser(Alg. 3), an algorithm that computes the set of all corrections for a context-free language and a word that corrects the word into the language. Instead of corrections, however, the algorithm uses indexless corrections, which we have presented in this work. We have shown that indexless corrections are an equivalent formalization of corrections. Indexless corrections have the advantage over corrections that they have a fixed length regarding a word and that they can express that a letter of the word is not changed.

Due to these advantages of indexless corrections, we were able to present a data structure with correction shared packed parse forest (CSPPF) that can store a possible infinite set of indexless corrections and can be calculated by the all-correction Earley Parser. We have shown that for a word and a context-free language, the CSPPF computed by the all-correction Earley parser contains all indexless corrections that correct the word into the language. In addition, we have shown that the all-correction Earley Parser requires only cubic time with respect to the length of the word.

However, the aim of this work was not to calculate the set of all corrections, but to calculate the set of minimal corrections for some definition of minimality. Therefore, we have presented minimality filters in order to filter out exactly the set of minimal corrections for some definition of minimality from the set of all corrections stored in a CSPPF. Finally, we have shown how such a minimality filter can be implemented algorithmically for the definition of minimality used for teaching hypothesis formulation in the biology classroom.

Further Research In the following, we will discuss some interesting further research questions that have arisen from this work.

- Probably the most important further work consists of implementing the theoretical algorithm presented in this work in order to be able to apply it in practice. Furthermore, this algorithm can then be compared with existing algorithms on benchmark datasets.

- In this work, we have only introduced how the definition of minimality from Section 5.1.1 can be implemented by composing three filters, without proving its correctness or investigating its complexity. Therefore, both remain as further work.

  However, even without fully analyzing the complexity of the filter, it can be seen that filtering out the minimal corrections for this minimality definition takes significantly more time than calculating all corrections. It is therefore advisable to investigate whether the filters can be implemented in a more efficient way. Furthermore, it should be investigated which definitions of minimality can be implemented by efficiently computable minimality filters.

- One of the main motivations for investigating corrections are didactic applications. In addition, it would be desirable to look for other fields of application of corrections and to investigate what are meaningful definitions of minimal corrections in these fields of application.

- In [7, Sec. 4.1] are defined some proprieties for disambiguation filters, that can be transferred to minimality filters. This immediately raises the question of which properties the filters defined in this work fulfill. In particular, which filter pairs are commutative? Assuming that the property would apply to all pairs of the filters f_{no_super}, f_{simp} and f_{no_loop} from Section 5. This would mean that any order of these filters would lead to the same result. Thus, the minimality definition, which is mainly considered in this work, would be calculated by any order of the filters.

Furthermore, it is useful to determine in which language class of the Chomsky hierarchy the language of all words that are removed by a filter lies to gain further insight into the filters (see [7]). Since this indicates which minimality definitions are calculable by a concatenation of filters through the closure properties of the language classes.

- In Section 2.7 we have mentioned, for reasons of complexity, that disambiguation rules should be implemented as early as possible in the parse process. For the same reasons, it should be investigated whether certain filters can also be implemented in earlier phases of the parse process. However, it should be mentioned that this represents a strong adaptation of the algorithm to the filter used, and thus the advantage of the modularity of this approach is largely lost.

- We can observe that if the CSPPF computed by the all-correction Earley Parser (Alg. 3) for some context-free grammar and some word of length n contains the symbol node (S, i, i), for some $0 \leq i \leq n$, where S is the start symbol of the context-free grammar. Then we can derive from (S, i, i) as an insertion word of an indexless correction all words that contain the language of the grammar. It therefore makes sense to simplify the underlying subtree or store it more efficiently. This possibility is shown below for certain types of grammar.

  The symbol node (S, i, i) is included in the computed correction shared packed parse forest for each i if and only if the start symbol occurs on at least one right side of a productive production rule because through a series of *insertion*, *deletion*, *prediction*, and *completion* operations, we have an Earley item with the required production rule, from which we can predict the nonterminal S in the Earley set Q_i. The items predicted in this way are the starting point for building the syntax tree below the node (S, i, i).

  Therefore, if we have a production rule in which the start symbol occurs on the right side, then we have in the CSPPF computed by the all-corrections Earley parser n copies of the same subtree expected by the adjustments of the extent. This leads to several possible optimizations of the algorithm for this case that need to be investigated.

  One of them is that instead of n copies, only one subtree needs to be saved, and filters possible only need to be applied to one subtree. However, the greatest potential for optimization lies in the calculation of all Earley items. Here we can observe that for every item $[A \to \gamma \bullet \alpha, j]$ in the Earley set Q_i there is an Earley item $[A \to \gamma \bullet \alpha, j+1]$ in the set Q_{i+1} because we know that the start symbol is predicted in every Earley set. This suggests that many computational steps can be saved. It is therefore advisable to investigate how to optimize the computation of all corrections for grammars with this property using the all-correction Earley Parser.

- In this work we have only considered how to adapt the general Earley parser to compute a CSPPF containing all corrections instead of a shared packed parse forest. But there are more variants to compute a SPPF: generalized LR parsers (left-to-right, leftmost derivation) like RIGLR[16] and BRNGLR[18] and generalized LL parsers (left-to-right, rightmost derivation) like GLL[17]. The question remains whether these algorithms can be adapted to compute a correction shared packed parse forest and, if so, whether this will lead to more efficient algorithms, especially in practice. However, it should be mentioned that the current bottleneck is not in the calculation of all corrections but in the implementation of the filters, especially when using the minimality definition from Section 5.1.1.

References

[1] Alfred Aho and Thomas Peterson. "A minimum distance error-correcting parser for context-free languages". In: *SIAM Journal on Computing* 1.4 (1972), pp. 305–312. DOI: 10.1137/0201022.

[2] John Aycock and Nigel Horspool. "Directly-executable Earley parsing". In: *International Conference on Compiler Construction*. Springer. 2001, pp. 229–243. DOI: 10.1007/3-540-45306-7_16.

[3] Bruse, Florian and Kablowski, Stefan and Lange, Martin. "Error-Correcting Parsing – This Time We Want All!" In: *Theorietag Automaten und Formale Sprachen* (2023). URL: https://theorietag2023.mpi-sws.org/abstracts/112MartinLange.pdf.

[4] Jay Earley. "An efficient context-free parsing algorithm". In: *Communications of the ACM* 13.2 (1970), pp. 94–102. DOI: 10.1145/362007.362035.

[5] Stefan Kablowski. *Computing All Minimal Corrections for a Word to Match a Context-Free Description*. B.sc. thesis, Univ. of Kassel, Germany, Faculty of Electr. Eng. and Comp. Sci., 2022. URL: https://www.uni-kassel.de/eecs/tifm/abschlussarbeiten/abg.

[6] Marit Kastaun, Monique Meier, Norbert Hundeshagen, and Martin Lange. "ProfiLL–Professionalisierung durch intelligente Lehr-Lernsysteme". In: *Bildung, Schule, Digitalisierung* (2020), pp. 357–363.

[7] Paul Klint and Eelco Visser. "Using filters for the disambiguation of context-free grammars". In: *Proc. ASMICS Workshop on Parsing Theory*. Milan, Italy. 1994, pp. 1–20.

[8] Vladimir Levenshtein. "Binary codes capable of correcting deletions, insertions, and reversals". In: *Soviet physics doklady*. Vol. 10. 8. 1966, pp. 707–710.

[9] José Nuno Macedo and João Saraiva. "Expressing disambiguation filters as combinators". In: *Proceedings of the 35th annual ACM symposium on applied computing*. 2020, pp. 1348–1351. DOI: 10.1145/3341105.3374123.

[10] Martin Lange. *Formale Sprachen und Logik*. 2019.

[11] Christoph Meinel and Martin Mundhenk. *Mathematische Grundlagen der Informatik*. Springer, 2011.

[12] Robert Moore. "Removing left recursion from context-free grammars". In: *1st Meeting of the North American Chapter of the Association for Computational Linguistics*. 2000.

[13] Sanguthevar Rajasekaran and Marius Nicolae. "An error correcting parser for context free grammars that takes less than cubic time". In: *Language and Automata Theory and Applications*. Springer, 2016, pp. 533–546. DOI: 10.1007/978-3-319-30000-9_41.

[14] Bram van der Sanden. "Parse Forest Disambiguation". MA thesis. Master Thesis, 2014. URL: https://pure.tue.nl/ws/portalfiles/portal/46998704/784691-1.pdf.

[15] Elizabeth Scott. "SPPF-style parsing from Earley recognisers". In: *Electronic Notes in Theoretical Computer Science* 203.2 (2008), pp. 53–67. DOI: https://doi.org/10.1016/j.entcs.2008.03.044.

[16] Elizabeth Scott and Adrian Johnstone. "Generalized bottom up parsers with reduced stack activity". In: *The Computer Journal* 48.5 (2005), pp. 565–587. DOI: 10.1093/comjnl/bxh102.

[17] Elizabeth Scott and Adrian Johnstone. "GLL parse-tree generation". In: *Science of Computer Programming* 78.10 (2013), pp. 1828–1844. DOI: https://doi.org/10.1016/j.scico.2012.03.005.

[18] Elizabeth Scott, Adrian Johnstone, and Rob Economopoulos. "BRNGLR: a cubic Tomita-style GLR parsing algorithm". In: *Acta informatica* 44 (2007), pp. 427–461. DOI: 10.1007/s00236-007-0054-z.

[19] Masaru Tomita. *Efficient parsing for natural language: a fast algorithm for practical systems*. Vol. 8. Springer Science & Business Media, 1985.

[20] Mark Van den Brand, Jeroen Scheerder, Jurgen Vinju, and Eelco Visser. "Disambiguation filters for scannerless generalized LR parsers". In: *Compiler Construction*. Springer. 2002, pp. 143–158. DOI: 10.1007/3-540-45937-5_12.

[21] Ivan Vendrov, Ryan Kiros, Sanja Fidler, and Raquel Urtasun. "Order-Embeddings of Images and Language". In: *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2016. URL: http://arxiv.org/abs/1511.06361.

[22] Daniel Younger. "Recognition and parsing of context-free languages in time n3". In: *Information and control* 10.2 (1967), pp. 189–208. DOI: https://doi.org/10.1016/S0019-9958(67)80007-X.

A Algorithm of the Generalised Earley Parser

In Section 2.5 we have already considered, for a word and a context-free grammar, the generalized Earley Parser [15] to compute a shared packed parse forest that contains all syntax trees for the word. Here we present the generalized Earley Parser algorithm in detail and discuss some differences/optimizations to the original Earley Parser [4] and the all-correction Earley Parser to compute a correction shared packed parse forest (Sec. 4.2). The generalized Earley Parser algorithm is presented in Algorithm 12. This algorithm uses the auxiliary functions COMPUTE_EARLEY_SET (Alg. 13), INIT_NEXT_EARLEY_SET (Alg. 14) and MAKE_NODE (Alg. 15).

Algorithm 12 GENERALIZED_EARLEY_PARSER

Require: A context-free grammar $G = (N, \Sigma, P, S)$ and a word $w = a_0 a_1 \dots a_{n-1}$
Ensure: The SPPF representing all Syntax Trees corresponding to w and G, if $w \in L(G)$

```
 1: Q_i ← ∅ for all 0 ≤ i ≤ n
 2: R, N', V, H ← ∅
 3:
 4: for all S → δ ∈ P do
 5:     if β ∈ Σ_N then
 6:         add [S → •δ, 0, null] to Q_0
 7:     end if
 8:
 9:     if β = a_0 δ' then
10:         add [S → •δ, 0, null] to N'
11:     end if
12: end for
13:
14: for i = 0 to n do
15:     H ← ∅, R ← Q_i, N ← N'
16:     N' ← ∅
17:
18:     COMPUTE_EARLEY_SET()
19:
20:     v ← (a_i, i, i + 1)
21:     V ← {v}
22:
23:     INIT_NEXT_EARLEY_SET()
24: end for
25:
26: if [S → γ•, 0, u] ∈ Q_n then
27:     return u
28: else
29:     return w ∉ L(G)
30: end if
```

At first, the algorithm does not add a new start symbol like the original Earley Parser because we would rather not have any nonterminal in a syntax tree that is not in the given grammar. Instead, for each production rule $(S \to \delta)$ with the start symbol (S) on the left, we add an Earley item $([S \to \bullet\delta, 0, null])$ assigned with the dummy node to the first Earley set (Q_0).

Secondly, we can observe that the application of a *predictor* or *completer* rule creates a new Earley item in the same Earley set as the item to which the rule is applied. In addition, we can observe that the *scanner* rule creates a new Earley item in the next Earley set.

Algorithm 13 COMPUTE_EARLEY_SET

```
1: while R ≠ ∅ do
2:     Λ = [B → α • β, j, u] ← R.pop()
3:
4:     if β = Cβ′ then
5:         for all C → δ ∈ P do
6:             if δ ∈ Σ_N and [C → •δ, i, null] ∉ Q_i then                ▷ Predictor Rule
7:                 add [C → •δ, i, null] to Q_i and R
8:             end if
9:
10:            if δ = a_i δ′ then
11:                add [C → •δ, i, null] to N
12:            end if
13:
14:            if (C, v) ∈ H with v = (C, i, i) then                     ▷ Completer Rule
15:                y ← MAKE_NODE(B → αC • β′, j, i, u, v)
16:                if β ∈ Σ_N and [B → αC • β, j, y] ∉ Q_i then
17:                    add [B → αC • β, j, y] to Q_i and R
18:                end if
19:
20:                if β = a_i δ′ then
21:                    add [B → αC • δ, j, y] to N
22:                end if
23:            end if
24:        end for
25:
26:    else if Λ is a final item then
27:        if u = null then
28:            if (B, j, i) ∉ V then
29:                add (B, j, i) to V
30:            end if
31:            y ← (B, j, i)
32:            if y does not have a family of children (e) then
33:                add (e) as child family of y
34:            end if
35:        end if
36:
37:        if j = i then
38:            add (B, u) to H
39:        end if
40:
41:        for all [A → τ • Bδ, k, z] ∈ Q_j do                           ▷ Completer Rule
42:            y ← MAKE_NODE(A → τB • δ, k, i, z, u, V)
43:            if δ ∈ Σ_N and [A → τB • δ, k, y] ∉ Q_i then
44:                add [A → τB • δ, k, y] to Q_i and R
45:            end if
46:
47:            if δ = a_i δ′ then
48:                add [A → τB • δ, k, y] to N
49:            end if
50:        end for
51:    end if
52: end while
```

Algorithm 14 INIT_NEXT_EARLEY_SET

while $N \neq \emptyset$ **do**
    $+[B \rightarrow \ \alpha \bullet a_i\beta, h, u] \leftarrow N.pop()$
    $y \leftarrow MAKE\_NODE(B \rightarrow \alpha a_i \bullet \beta, h, i+1, u, v)$
    if $\beta \in \Sigma_N$ **then**
        add $[B \rightarrow \alpha a_i \bullet \beta, h, y]$ to Q_{i+1}
    end if

    if $\beta = a_{i+1}\beta'$ **then**
        add $[B \rightarrow \alpha a_i \bullet \beta, h, y]$ to N'
    end if
end while

Algorithm 15 MAKE_NODE

Require: A 5−tupel $(B \rightarrow \alpha \bullet \beta, j, i, u, v)$ where B is a nonterminal, α, β are sentential forms, $j, i \in \mathbb{N}_0$ with $j \leq i$ and the SPPF nodes u, v
Ensure: A SPPF node
if $\beta = \epsilon$ **then**
    $s \leftarrow B$ ▷ Symbol node
else
    $s \leftarrow (B \rightarrow \alpha \bullet \beta)$ ▷ Intermediate node
end if

if $\alpha = \Sigma$ and $\beta \neq \epsilon$ **then**
    $y \leftarrow v$
else
    $y \leftarrow (s, j, i)$
    if $y \notin V$ **then**
        add y to V
    end if

    if $u = null$ and y does not have a family of children (v) **then**
        add (v) as child family of y
    else if $u \neq null$ and and y does not have a family of children (u, v) **then**
        add (u, v) as child family of y
    end if
end if

return y

Therefore, we split the calculation into two steps: (I) applying the *predictor* and *completer* rules until no new Earley items are created, (II) applying the scanner rule to all items in the current Earley set. The algorithm repeats this procedure for each Earley set. The first step is outscored to the auxiliary function COMPUTE_EARLEY_SET (Alg. 13) and the second step to the auxiliary function INIT_NEXT_EARLEY_SET (Alg. 14).

We can also observe that if the sentential form to the right of the bullet point for an Earley item begins with a terminal symbol. Then we can only apply the *scanner* rule to it if the terminal symbol matches the next letter of the word to be read. In the other case, where the sentential form on the right starts with a nonterminal or is empty, we can possibly apply the *predictor* or *completer* rule to it. Therefore, we define $\Sigma_N = A(N \cup \Sigma)^* \cup \{\varepsilon\}$ with $A \in N$ and add a newly created Earley item to the Earley set only if the sentential form on the right of the bullet point is contained in N_Σ. Otherwise, if the sentence form starts with the next letter of the word, we add the Earley item to the set (N) of items on which we can apply the scanner rule.

Auxiliary functions The auxiliary function COMPUTE_EARLEY_SET takes from a queue (R) every Earley item of the current set and tries to apply on it the *predictor* or *completer* rule and build up the SPPF bottom-up as already described in Section 2.5 by using the auxiliary function MAKE_NODE (Alg. 15). In addition, we use a set H that stores only the final Earley items of the current Earley set that also start at the same index as the Earley set itself. For an item to be included in this set, there must be a production rule $A \to \varepsilon$ with $A \in N$. All Earley items created by the *completer* rule based on this production rule are final Earley items, but with the dummy node *null*. So we add ε ad child familiy to the nodes that represent such items. The use of the set H has the advantage that we don't have to filter out all items that fulfill this condition.

The INIT_NEXT_EARLEY_SET auxiliary function applies the *scanner* rule to all items contained in the precomputed set N. Thus filling the next Earley set and the next set of Earley items for which the sentential form to the right of the bullet starts with the letter of the word after the current scanned letter. Like the COMPUTE_EARLEY_SET function, the INIT_NEXT_EARLEY_SET function uses the MAKE_NODE auxiliary function to build up the tree.

In both auxiliary functions, we use the MAKE_NODE function, which checks whether the predecessor node created by a rule already exists. If it doesn't exist, the function adds a new node to the SPPF. Additionally, the function adds a new family to the created or existing node for the two given nodes. The differences between the MAKE_NODE function presented here for SPPF and the MAKE_NODE function for CSPPF (Alg.7) were already discussed in Section 4.2.

At least, the algorithm to compute the SPPF. It checks if for the start symbol (S) there is a final Earley item in the last Earley set (Q_n) that has a pointer back to index 0. If this is true, the algorithm returns the corresponding node as the root of the SPPF.

The attentive reader will have noticed that there possible is more than one final Earley item in Q_n of this form. But the good news is that they are all associated with the same node $(S, 0, n)$.

B Earley Items of Example 2.7

Q_0		
$[E \to \bullet E + T$	$, 0, \ null]$	
$[E \to \bullet E - T$	$, 0, \ null]$	
$[E \to \bullet T$	$, 0, \ null]$	
$[T \to \bullet T * F$	$, 0, \ null]$	**P**
$[T \to \bullet T / F$	$, 0, \ null]$	**P**
$[T \to \bullet F$	$, 0, \ null]$	**P**
$[F \to \bullet (E)$	$, 0, \ null]$	**P**
$[F \to \bullet n$	$, 0, \ null]$	**P**

Q_1			
$[F \to n\bullet$	$, 0, \ (F, 0, 1)]$		**S**
$[T \to F\bullet$	$, 0, \ (T, 0, 1)]$		**C**
$[E \to T\bullet$	$, 0, \ (E, 0, 1)]$		**C**
$[T \to T \bullet * F$	$, 0, \ (T \to T \bullet * F, 0, 1)]$		**C**
$[T \to T \bullet / F$	$, 0, \ (T \to T \bullet / F, 0, 1)]$		**C**
$[E \to E \bullet + T$	$, 0, \ (E \to E \bullet + T, 0, 1)]$		**C**
$[E \to E \bullet - T$	$, 0, \ (E \to E \bullet - T, 0, 1)]$		**C**

Q_2			
$[T \to T * \bullet F$	$, 0, \ (T \to T * \bullet F, 0, 2)]$		**S**
$[F \to \bullet (E)$	$, 2, \ null]$		**P**
$[F \to \bullet n$	$, 2, \ null]$		**P**

Q_3			
$[F \to (\bullet E)$	$, 2, \ (F \to (\bullet E), 2, 3)]$		**S**
$[E \to \bullet E + T$	$, 3, \ null]$		**P**
$[E \to \bullet E - T$	$, 3, \ null]$		**P**
$[E \to \bullet T$	$, 3, \ null]$		**P**
$[T \to \bullet T * F$	$, 3, \ null]$		**P**
$[T \to \bullet T / F$	$, 3, \ null]$		**P**
$[T \to \bullet F$	$, 3, \ null]$		**P**
$[F \to \bullet (E)$	$, 3, \ null]$		**P**
$[F \to \bullet n$	$, 3, \ null]$		**P**

Q_4			
$[F \to n\bullet$	$, 3, \ (F, 3, 4)]$		**P**
$[T \to F\bullet$	$, 3, \ (T, 3, 4)]$		**C**
$[E \to T\bullet$	$, 3, \ (E, 3, 4)]$		**C**
$[T \to T \bullet * F$	$, 3, \ (T \to T \bullet * F, 3, 4)]$		**C**
$[T \to T \bullet / F$	$, 3, \ (T \to T \bullet / F, 3, 4)]$		**C**
$[F \to (E\bullet)$	$, 2, \ (F \to (E\bullet), 2, 4)]$		**C**
$[E \to E \bullet + T$	$, 3, \ (E \to E \bullet + T, 3, 4)]$		**C**
$[E \to E \bullet - T$	$, 3, \ (E \to E \bullet - T, 3, 4)]$		**C**

Q_5			
$[E \to E + \bullet T$	$, 3, \ (E \to E + \bullet T, 3, 5)]$		**S**
$[T \to \bullet T * F$	$, 5, \ null]$		**P**
$[T \to \bullet T / F$	$, 5, \ null]$		**P**
$[F \to \bullet (E)$	$, 5, \ null]$		**P**
$[F \to \bullet n$	$, 5, \ null]$		**P**

Q_6			
$[F \to n\bullet$	$, 5, \ (F, 5, 6)]$		**S**
$[T \to F\bullet$	$, 5, \ (T, 5, 6)]$		**C**
$[E \to E + T\bullet$	$, 3, \ (E, 3, 6)]$		**C**
$[T \to T \bullet * F$	$, 5, \ (T \to T \bullet * F, 5, 6)]$		**C**
$[T \to T \bullet / F$	$, 5, \ (T \to T \bullet / F, 5, 6)]$		**C**
$[E \to T\bullet$	$, 5, \ (E, 5, 6)]$		**C**
$[F \to (E\bullet)$	$, 2, \ (F \to (E\bullet), 2, 6)]$		**C**
$[E \to E \bullet + T$	$, 3, \ (E \to E \bullet + T, 3, 6)]$		**C**
$[E \to E \bullet - T$	$, 3, \ (E \to E \bullet - T, 3, 6)]$		**C**

Q_7			
$[F \to (E)\bullet$	$, 2, \ (F, 2, 7)]$		**S**
$[T \to T * F\bullet$	$, 0, \ (T, 0, 7)]$		**C**
$[E \to T\bullet$	$, 0, \ (E, 0, 7)]$		**C**
$[T \to T \bullet * F$	$, 0, \ (T \to T \bullet * F, 0, 7)]$		**C**
$[T \to T \bullet / F$	$, 0, \ (T \to T \bullet / F, 0, 7)]$		**C**
$[E \to E \bullet + T$	$, 0, \ (E \to E \bullet + T, 0, 7)]$		**C**
$[E \to E \bullet - T$	$, 0, \ (E \to E \bullet - T, 0, 7)]$		**C**

Figure 22: The Earley sets for the word $n * (n + n)$ and the context-free grammar G_{aexpr} with the additional information to which node of the SPPF (Fig. 6) the item belongs to. The red color of an Earley item indicates that this Earley item is a final item, and the blue letters next to the items indicate the rule by which the item is generated.

C Earley Items of Example 2.11

Q_0					
$[S' \to \bullet E, 0, 0]$		$[I \to \bullet /, 0, 0]$	P	$[A_+ \to \bullet +, 0, 0]$	P
$[S' \to \bullet EH, 0, 0]$		$[I \to \bullet (, 0, 0]$	P	$[A_+ \to \bullet (, 0, 0]$	P
$[E \to \bullet EA_+T, 0, 0]$	P	$[I \to \bullet), 0, 0]$	P	$[A_+ \to \bullet -, 0, 0]$	P
$[E \to \bullet EA_-T, 0, 0]$	P	$[I \to \bullet n, 0, 0]$	P	$[A_+ \to \bullet *, 0, 0]$	P
$[E \to \bullet T, 0, 0]$	P	$[F \to A_(\bullet EA_), 0, 1]$	C	$[A_+ \to \bullet /, 0, 0]$	P
$[T \to \bullet TA_*F, 0, 0]$	P	$[F \to A_n \bullet, 0, 1]$	C	$[A_+ \to \bullet), 0, 0]$	P
$[T \to \bullet TA_/F, 0, 0]$	P	$[T \to F \bullet, 0, 1]$	C	$[A_+ \to \bullet n, 0, 0]$	P
$[T \to \bullet F, 0, 0]$	P	$[E \to T \bullet, 0, 1]$	C	$[A_+ \to \bullet H +, 0, 0]$	P
$[F \to \bullet A_(EA_), 0, 0]$	P	$[T \to T \bullet A_*F, 0, 1]$	C	$[A_+ \to \bullet \epsilon, 0, 0]$	P
$[F \to \bullet A_n, 0, 0]$	P	$[T \to T \bullet A_/F, 0, 1]$	C	$[A_- \to \bullet -, 0, 0]$	P
$[A_(\to \bullet (, 0, 0]$	P	$[S' \to E \bullet, 0, 1]$	C	$[A_- \to \bullet (, 0, 0]$	P
$[A_(\to \bullet +, 0, 0]$	P	$[S' \to E \bullet H, 0, 1]$	C	$[A_- \to \bullet +, 0, 0]$	P
$[A_(\to \bullet -, 0, 0]$	P	$[E \to E \bullet A_+T, 0, 1]$	C	$[A_- \to \bullet *, 0, 0]$	P
$[A_(\to \bullet *, 0, 0]$	P	$[E \to E \bullet A_-T, 0, 1]$	C	$[A_- \to \bullet /, 0, 0]$	P
$[A_(\to \bullet /, 0, 0]$	P	$[F \to A_(E \bullet A_), 0, 2]$	C	$[A_- \to \bullet), 0, 0]$	P
$[A_(\to \bullet), 0, 0]$	P	$[A_* \to \bullet *, 0, 0]$	P	$[A_- \to \bullet n, 0, 0]$	P
$[A_(\to \bullet n, 0, 0]$	P	$[A_* \to \bullet (, 0, 0]$	P	$[A_- \to \bullet \epsilon, 0, 0]$	P
$[A_(\to \bullet \epsilon, 0, 0]$	P	$[A_* \to \bullet +, 0, 0]$	P	$[A_- \to \bullet H -, 0, 0]$	P
$[A_(\to \bullet H (, 0, 0]$	P	$[A_* \to \bullet -, 0, 0]$	P	$[A_) \to \bullet), 0, 0]$	P
$[A_n \to \bullet n, 0, 0]$	P	$[A_* \to \bullet /, 0, 0]$	P	$[A_) \to \bullet (, 0, 0]$	P
$[A_n \to \bullet (, 0, 0]$	P	$[A_* \to \bullet), 0, 0]$	P	$[A_) \to \bullet +, 0, 0]$	P
$[A_n \to \bullet +, 0, 0]$	P	$[A_* \to \bullet n, 0, 0]$	P	$[A_) \to \bullet -, 0, 0]$	P
$[A_n \to \bullet -, 0, 0]$	P	$[A_* \to \bullet H *, 0, 0]$	P	$[A_) \to \bullet *, 0, 0]$	P
$[A_n \to \bullet *, 0, 0]$	P	$[A_* \to \bullet \epsilon, 0, 0]$	P	$[A_) \to \bullet /, 0, 0]$	P
$[A_n \to \bullet /, 0, 0]$	P	$[A_/ \to \bullet /, 0, 0]$	P	$[A_) \to \bullet n, 0, 0]$	P
$[A_n \to \bullet), 0, 0]$	P	$[A_/ \to \bullet (, 0, 0]$	P	$[A_n \to \bullet \epsilon, 0, 0]$	P
$[A_n \to \bullet \epsilon, 0, 0]$	P	$[A_/ \to \bullet +, 0, 0]$	P	$[A_) \to \bullet H), 0, 0]$	P
$[A_n \to \bullet Hn, 0, 0]$	P	$[A_/ \to \bullet -, 0, 0]$	P	$[T \to TA_* \bullet F, 0, 2]$	C
$[H \to \bullet HI, 0, 0]$	P	$[A_/ \to \bullet *, 0, 0]$	P	$[T \to TA_/ \bullet F, 0, 2]$	C
$[H \to \bullet I, 0, 0]$	P	$[A_/ \to \bullet), 0, 0]$	P	$[E \to EA_+ \bullet T, 0, 2]$	C
$[I \to \bullet +, 0, 0]$	P	$[A_/ \to \bullet n, 0, 0]$	P	$[E \to EA_- \bullet T, 0, 2]$	C
$[I \to \bullet -, 0, 0]$	P	$[A_/ \to \bullet \epsilon, 0, 0]$	P		
$[I \to \bullet *, 0, 0]$	P	$[A_/ \to \bullet H /, 0, 0]$	P		

(a) Earley set Q_0.

Q_1					
$[A_(\rightarrow +\bullet, 0, 0]$	S	$[I \rightarrow \bullet(, 1, 0]$	P	$[A_/ \rightarrow \bullet/, 1, 0]$	P
$[A_n \rightarrow +\bullet, 0, 0]$	S	$[I \rightarrow \bullet), 1, 0]$	P	$[A_/ \rightarrow \bullet(, 1, 0]$	P
$[I \rightarrow +\bullet, 0, 0]$	S	$[I \rightarrow \bullet n, 1, 0]$	P	$[A_/ \rightarrow \bullet+, 1, 0]$	P
$[A_* \rightarrow +\bullet, 0, 0]$	S	$[A_(\rightarrow \bullet(, 1, 0]$	P	$[A_/ \rightarrow \bullet-, 1, 0]$	P
$[A_/ \rightarrow +\bullet, 0, 0]$	S	$[A_(\rightarrow \bullet+, 1, 0]$	P	$[A_/ \rightarrow \bullet*, 1, 0]$	P
$[A_+ \rightarrow +\bullet, 0, 0]$	S	$[A_(\rightarrow \bullet-, 1, 0]$	P	$[A_/ \rightarrow \bullet), 1, 0]$	P
$[A_- \rightarrow +\bullet, 0, 0]$	S	$[A_(\rightarrow \bullet*, 1, 0]$	P	$[A_/ \rightarrow \bullet n, 1, 0]$	P
$[A_) \rightarrow +\bullet, 0, 0]$	S	$[A_(\rightarrow \bullet/, 1, 0]$	P	$[A_/ \rightarrow \bullet\epsilon, 1, 0]$	P
$[F \rightarrow A_(\bullet EA_), 0, 1]$	C	$[A_(\rightarrow \bullet), 1, 0]$	P	$[A_/ \rightarrow \bullet H/, 1, 0]$	P
$[F \rightarrow A_n\bullet, 0, 1]$	C	$[A_(\rightarrow \bullet n, 1, 0]$	P	$[F \rightarrow A_(\bullet EA_), 1, 1]$	C
$[H \rightarrow I\bullet, 0, 1]$	C	$[A_(\rightarrow \bullet\epsilon, 1, 0]$	P	$[H \rightarrow \bullet HI, 1, 0]$	P
$[T \rightarrow TA_* \bullet F, 0, 2]$	C	$[A_(\rightarrow \bullet H(, 1, 0]$	P	$[H \rightarrow \bullet I, 1, 0]$	P
$[T \rightarrow TA_/ \bullet F, 0, 2]$	C	$[A_n \rightarrow \bullet n, 1, 0]$	P	$[F \rightarrow A_n\bullet, 1, 1]$	P
$[E \rightarrow EA_+ \bullet T, 0, 1]$	C	$[A_n \rightarrow \bullet(, 1, 0]$	P	$[A_+ \rightarrow \bullet+, 1, 0]$	P
$[E \rightarrow EA_- \bullet T, 0, 2]$	C	$[A_n \rightarrow \bullet+, 1, 0]$	P	$[A_+ \rightarrow \bullet(, 1, 0]$	P
$[E \rightarrow \bullet EA_+T, 1, 0]$	P	$[A_n \rightarrow \bullet-, 1, 0]$	P	$[A_+ \rightarrow \bullet-, 1, 0]$	P
$[E \rightarrow \bullet EA_-T, 1, 0]$	P	$[A_n \rightarrow \bullet*, 1, 0]$	P	$[A_+ \rightarrow \bullet*, 1, 0]$	P
$[E \rightarrow \bullet T, 1, 0]$	P	$[A_n \rightarrow \bullet/, 1, 0]$	P	$[A_+ \rightarrow \bullet/, 1, 0]$	P
$[T \rightarrow F\bullet, 0, 1]$	C	$[A_n \rightarrow \bullet), 1, 0]$	P	$[A_+ \rightarrow \bullet), 1, 0]$	P
$[A_(\rightarrow H \bullet (, 0, 1]$	C	$[A_n \rightarrow \bullet\epsilon, 1, 0]$	P	$[A_+ \rightarrow \bullet n, 1, 0]$	P
$[A_n \rightarrow H \bullet n, 0, 1]$	C	$[A_n \rightarrow \bullet Hn, 1, 0]$	P	$[A_+ \rightarrow \bullet H+, 1, 0]$	P
$[H \rightarrow H \bullet I, 0, 1]$	C	$[S' \rightarrow E\bullet, 0, 1]$	C	$[A_+ \rightarrow \bullet\epsilon, 1, 0]$	P
$[S' \rightarrow EH\bullet, 0, 2]$	C	$[S' \rightarrow E \bullet H, 0, 1]$	C	$[A_- \rightarrow \bullet-, 1, 0]$	P
$[A_* \rightarrow H \bullet *, 0, 1]$	C	$[E \rightarrow E \bullet A_+T, 0, 1]$	C	$[A_- \rightarrow \bullet(, 1, 0]$	P
$[A_/ \rightarrow H \bullet (, 0, 1]$	C	$[E \rightarrow E \bullet A_-T, 0, 1]$	C	$[A_- \rightarrow \bullet+, 1, 0]$	P
$[A_+ \rightarrow H \bullet +, 0, 1]$	C	$[F \rightarrow A_(E \bullet A_), 0, 2]$	C	$[A_- \rightarrow \bullet*, 1, 0]$	P
$[A_- \rightarrow H \bullet -, 0, 1]$	C	$[A_* \rightarrow \bullet*, 1, 0]$	P	$[A_- \rightarrow \bullet/, 1, 0]$	P
$[A_) \rightarrow H\bullet), 0, 1]$	C	$[A_* \rightarrow \bullet(, 1, 0]$	P	$[A_- \rightarrow \bullet), 1, 0]$	P
$[F \rightarrow \bullet A_(EA_), 1, 0]$	P	$[A_* \rightarrow \bullet+, 1, 0]$	P	$[A_- \rightarrow \bullet n, 1, 0]$	P
$[F \rightarrow \bullet A_n, 1, 0]$	P	$[A_* \rightarrow \bullet-, 1, 0]$	P	$[A_- \rightarrow \bullet\epsilon, 1, 0]$	P
$[T \rightarrow \bullet TA_*F, 1, 0]$	P	$[A_* \rightarrow \bullet/, 1, 0]$	P	$[A_- \rightarrow \bullet H-, 1, 0]$	P
$[T \rightarrow \bullet TA_/F, 1, 0]$	P	$[A_* \rightarrow \bullet), 1, 0]$	P	$[T \rightarrow TA_* \bullet F, 0, 2]$	C
$[T \rightarrow \bullet F, 1, 0]$	P	$[A_* \rightarrow \bullet n, 1, 0]$	P	$[T \rightarrow TA_/ \bullet F, 0, 2]$	C
$[E \rightarrow T\bullet, 0, 1]$	C	$[A_* \rightarrow \bullet H*, 1, 0]$	P	$[E \rightarrow EA_+ \bullet T, 0, 2]$	C
$[T \rightarrow T \bullet A_*F, 0, 1]$	C	$[A_* \rightarrow \bullet\epsilon, 1, 0]$	P	$[E \rightarrow EA_- \bullet T, 0, 2]$	C
$[T \rightarrow T \bullet A_/F, 0, 1]$	C				
$[I \rightarrow \bullet+, 1, 0]$	P				
$[I \rightarrow \bullet-, 1, 0]$	P				
$[I \rightarrow \bullet*, 1, 0]$	P				
$[I \rightarrow \bullet/, 1, 0]$	P				

(b) Earley set Q_1.

Figure 23: The Earley sets for the word + and the covering grammar G''_{aexpr} defined in Example 2.11 with the additional information for each Earley item, the minimum number of edit operations required to produce that item. The red color of an Earley item indicates that this Earley item is a final item; items are surrounded by a box representing terminal error production rules, and the blue letters next to the items indicate the rule by which the item is generated. Subfigure 23a presents Earley Set Q_0 and Subfigure 23b represents the set Q_1.

D Earley Items of Example 4.3

Q_0					
$[E \to \bullet E + T$	$,0,$	$null]$			
$[E \to \bullet E - T$	$,0,$	$null]$			
$[E \to \bullet T$	$,0,$	$null]$			
$[T \to \bullet T * F$	$,0,$	$null]$	**P**		
$[T \to \bullet T / F$	$,0,$	$null]$	**P**		
$[T \to \bullet F$	$,0,$	$null]$	**P**		
$[F \to \bullet (E)$	$,0,$	$null]$	**P**		
$[F \to \bullet n$	$,0,$	$null]$	**P**		
$[F \to (\bullet E)$	$,0,$	$(F \to (\bullet E), 0, 0)]$	**I**		
$[F \to n\bullet$	$,0,$	$(F, 0, 0)]$	**I**		
$[T \to F\bullet$	$,0,$	$(T, 0, 0)]$	**C**		
$[E \to T\bullet$	$,0,$	$(E, 0, 0)]$	**C**		
$[T \to T \bullet * F$	$,0,$	$(T \to T \bullet * F, 0, 0)]$	**C**		
$[T \to T \bullet / F$	$,0,$	$(T \to T \bullet / F, 0, 0)]$	**C**		
$[E \to E \bullet + T$	$,0,$	$(E \to E \bullet + T, 0, 0)]$	**C**		
$[E \to E \bullet - T$	$,0,$	$(E \to E \bullet - T, 0, 0)]$	**C**		
$[F \to (E\bullet)$	$,0,$	$(F \to (E\bullet), 0, 0)]$	**C**		
$[T \to T * \bullet F$	$,0,$	$(T \to T * \bullet F, 0, 0)]$	**I**		
$[T \to T / \bullet F$	$,0,$	$(T \to T / \bullet F, 0, 0)]$	**I**		
$[E \to E + \bullet T$	$,0,$	$(E \to E + \bullet T, 0, 0)]$	**I**		
$[E \to E - \bullet T$	$,0,$	$(E \to E - \bullet T, 0, 0)]$	**I**		
$[F \to (E)\bullet$	$,0,$	$(F \to (F, 0, 0)]$	**I**		
$[T \to T * F\bullet$	$,0,$	$(T, 0, 0)]$	**C**		
$[T \to T / F\bullet$	$,0,$	$(T, 0, 0)]$	**C**		
$[E \to E + T\bullet$	$,0,$	$(E, 0, 0)]$	**C**		
$[E \to E - T\bullet$	$,0,$	$(E, 0, 0)]$	**C**		

Q_1					
$[F \to (\bullet E)$	$,0,$	$(F \to (\bullet E), 0, 1)]$	**R, D, I**		
$[F \to n\bullet$	$,0,$	$(F, 0, 1)]$	**R, D, I**		
$[T \to T * \bullet F$	$,0,$	$(T \to T * \bullet F, 0, 1)]$	**R, D, I**		
$[T \to T / \bullet F$	$,0,$	$(T \to T / \bullet F, 0, 1)]$	**R, D, I**		
$[E \to E + \bullet T$	$,0,$	$(E \to E + \bullet T, 0, 1)]$	**S, D, I**		
$[E \to E - \bullet T$	$,0,$	$(E \to E - \bullet T, 0, 1)]$	**R, D, I**		
$[F \to (E)\bullet$	$,0,$	$(F, 0, 1)]$	**R, D, I**		
$[E \to \bullet E + T$	$,0,$	$(E \to \bullet E + T, 0, 1)]$	**D**		
$[E \to \bullet E - T$	$,0,$	$(E \to \bullet E - T, 0, 1)]$	**D**		
$[E \to \bullet T$	$,0,$	$(E \to \bullet T, 0, 1)]$	**D**		
$[T \to \bullet T * F$	$,0,$	$(T \to \bullet T * F, 0, 1)]$	**D**		
$[T \to \bullet T / F$	$,0,$	$(T \to \bullet T / F, 0, 1)]$	**D**		
$[T \to \bullet F$	$,0,$	$(T \to \bullet F, 0, 1)]$	**D**		
$[F \to \bullet (E)$	$,0,$	$(F \to \bullet (E), 0, 1)]$	**D**		
$[F \to \bullet n$	$,0,$	$(F \to \bullet n, 0, 1)]$	**D**		
$[T \to F\bullet$	$,0,$	$(T, 0, 1)]$	**D, C**		
$[E \to T\bullet$	$,0,$	$(E, 0, 1)]$	**D, C**		
$[T \to T \bullet * F$	$,0,$	$(T \to T \bullet * F, 0, 1)]$	**D, C**		
$[T \to T \bullet / F$	$,0,$	$(T \to T \bullet / F, 0, 1)]$	**D, C**		
$[E \to E \bullet + T$	$,0,$	$(E \to E \bullet + T, 0, 1)]$	**D, C**		
$[E \to E \bullet - T$	$,0,$	$(E \to E \bullet - T, 0, 1)]$	**D, C**		
$[F \to (E\bullet)$	$,0,$	$(F \to (E\bullet), 0, 1)]$	**D, C**		
$[T \to T * F\bullet$	$,0,$	$(T, 0, 1)]$	**D, C**		
$[T \to T / F\bullet$	$,0,$	$(T, 0, 1)]$	**D, C**		
$[E \to E + T\bullet$	$,0,$	$(E, 0, 1)]$	**D, C**		
$[E \to E - T\bullet$	$,0,$	$(E, 0, 1)]$	**D, C**		
$[T \to T * \bullet F$	$,1,$	$null]$	**P**		
$[T \to T / \bullet F$	$,1,$	$null]$	**P**		
$[E \to E + \bullet T$	$,1,$	$null]$	**P**		
$[E \to E - \bullet T$	$,1,$	$null]$	**P**		
$[E \to \bullet E + T$	$,1,$	$null]$	**P**		
$[E \to \bullet E - T$	$,1,$	$null]$	**P**		
$[E \to \bullet T$	$,1,$	$null]$	**P**		
$[F \to \bullet (E)$	$,1,$	$null]$	**P**		
$[F \to \bullet n$	$,1,$	$null]$	**P**		
$[T \to \bullet T * F$	$,1,$	$null]$	**P**		
$[T \to \bullet T / F$	$,1,$	$null]$	**P**		
$[T \to \bullet F$	$,1,$	$null]$	**P**		
$[F \to (\bullet E)$	$,1,$	$(F \to (\bullet E), 1, 1)]$	**I**		
$[F \to n\bullet$	$,1,$	$(F, 0, 1)]$	**I**		
$[T \to F\bullet$	$,1,$	$(T, 1, 1)]$	**C**		
$[E \to T\bullet$	$,1,$	$(E, 1, 1)]$	**C**		
$[T \to T \bullet * F$	$,1,$	$(T \to T \bullet * F, 1, 1)]$	**C**		
$[T \to T \bullet / F$	$,1,$	$(T \to T \bullet / F, 1, 1)]$	**C**		
$[E \to E \bullet + T$	$,1,$	$(E \to E \bullet + T, 1, 1)]$	**C**		
$[E \to E \bullet - T$	$,1,$	$(E \to E \bullet - T, 1, 1)]$	**C**		
$[F \to (E\bullet)$	$,1,$	$(F \to (E\bullet), 1, 1)]$	**C**		
$[T \to T * \bullet F$	$,1,$	$(T \to T * \bullet F, 1, 1)]$	**I**		
$[T \to T / \bullet F$	$,1,$	$(T \to T / \bullet F, 1, 1)]$	**I**		
$[E \to E + \bullet T$	$,1,$	$(E \to E + \bullet T, 1, 1)]$	**I**		
$[E \to E - \bullet T$	$,1,$	$(E \to E - \bullet T, 1, 1)]$	**I**		
$[F \to (E)\bullet$	$,1,$	$(F, 1, 1)]$	**I**		
$[T \to T * F\bullet$	$,1,$	$(T, 1, 1)]$	**C**		
$[T \to T / F\bullet$	$,1,$	$(T, 1, 1)]$	**C**		
$[E \to E + T\bullet$	$,1,$	$(E, 1, 1)]$	**C**		
$[E \to E - T\bullet$	$,1,$	$(E, 1, 1)]$	**C**		

Figure 24: The Earley sets for the word + and the context-free grammar G_{aexpr} with the additional information to which node of the CSPPF (Fig. 10) the item belongs to. The red color of an Earley item indicates that this Earley item is a final item, and the blue letters next to the items indicate the rule(s) by which the item is generated.