UNIKASSEL VERSITÄT | ELEKTROTECHNIK INFORMATIK

Nutzergesteuerte Fehlerkorrektur für kontextfreie Sprachen

Bachelorarbeit
am Fachgebiet Theoretische Informatik / Formale Methoden
der Universität Kassel

Autor:	Tobias Stuhldreier
Erstprüfer:	Prof. Dr. Martin Lange
Zweitprüfer:	Prof. Dr. Joel Greenyer
Betreuer:	M.Sc. Maurice Herwig

Eingereicht am: 19. Januar 2026

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich diese Arbeit selbstständig verfasst habe und alle verwendeten Quellen und Hilfsmittel angegeben habe. Für das Umschreiben einiger Textabschnitte wurde KI verwendet, was allerdings nicht den Inhalt der Arbeit verändert hat.

____

Tobias Stuhldreier

Kassel, 19. Januar 2026

Inhaltsverzeichnis

1 Einleitung 1

2 Grundlagen 4
- 2.1 Mathematische Grundlagen . . . 4
- 2.2 Kontextfreie Sprachen . . . 5
- 2.3 Earley Parser . . . 6
- 2.4 Shared Packed Parse Forest (SPPF) . . . 8
- 2.5 Corrections . . . 11
  - 2.5.1 Corrections in einem SPPF speichern . . . 13
- 2.6 Berechnung aller Corrections . . . 15
  - 2.6.1 All Correction Earley Parser . . . 15
  - 2.6.2 Berechnung des SPPF . . . 17

3 Nutzergesteuerte Fehlerkorrektur im SPPF 21
- 3.1 Nutzergesteuerte Traversierung im SPPF . . . 21
- 3.2 Verfahren zur Traversierung im SPPF . . . 22
  - 3.2.1 Startknoten berechnen und Traversierung nach unten . . . 23
  - 3.2.2 Pfade im SPPF verfolgen . . . 26
- 3.3 Korrektheit und Vollständigkeit . . . 31
- 3.4 Analyse und Probleme . . . 32

4 Nutzergesteuerte Fehlerkorrektur mit dem Earley Parser 33
- 4.1 Optimierte Traversierung im SPPF . . . 33
  - 4.1.1 Lokale Berechnung des SPPF . . . 33
- 4.2 Vom SPPF zurück zum Earley Parser . . . 34
- 4.3 Nutzergesteuerter Earley Parser . . . 35
  - 4.3.1 Auswahlregeln des nutzergesteuerten Earley Parser . . . 35
  - 4.3.2 Algorithmus zum Verfahren . . . 36
  - 4.3.3 Beispiel zum Verfahren . . . 40
- 4.4 Korrektheit und Vollständigkeit . . . 41
- 4.5 Analyse . . . 43

5 Fazit und Ausblick 44
- 5.1 Resultate . . . 44
- 5.2 Fazit . . . 44
- 5.3 Ausblick . . . 45

1 Einleitung

Fehlerkorrigierendes Parsing kontextfreier Sprachen besitzt eine hohe Relavanz in zahlreichen Anwendungen der Informatik. Dazu zählen etwa Anwendungen in interaktiven Entwicklungsumgebungen (Behebung von Syntaxfehlern), Anwendungen bei der Verabeitung natürlicher Sprache (fehlerhafte Spracheingaben oder Texteingaben) oder auch in interaktiven Lernsystemen (Schaffung eines zyklischen Lernprozess durch Feedback auf fehlerhafte Lösungen wie in [1]).

Im fehlerkorrigierenden Parsing wird eine möglicherweise fehlerhafte Eingabe w bezüglich einer kontextfreien Sprache betrachtet. Klassische Parsingverfahren wie der *CYK* Algorithmus [3] oder der *Earley Parser* [2] verwerfen solche fehlerhaften Eingaben. Ziel bei der Fehlkerkorretur, ist es sogenannte Corrections bezüglich der Eingabe zu berechnen. Eine Correction ist dabei eine Sequenz von Edit Operationen, etwa dem Einfügen, Löschen oder Ersetzen von Zeichen in der Eingabe. Im Allgemeinen existieren sehr viele mögliche Corrections, die in ihrer Länge, als auch in ihrer Struktur stark variieren können. Dabei stellt sich die Frage, welche dieser Corrections im jeweiligen Kontext der Anwendung als geeignet anzusehen ist. Dies soll anhand des folgenden fehlerhaften Python Programms gezeigt werden.

```
1    for i in range(0,n)
2        m += i
```

Das Programm ist syntaktitsch fehlerhaft, da nach dem Schleifenkopf ein Doppelpunkt fehlt. Eine minimale Correction besteht darin, genau dieses Symbol einzufügen. Alternativ könnten jedoch auch umfangreiche Änderungen am Programmcode vorgenommen werden, die zwar zu einem neuen syntaktisch korrekten Programm führen, aber offensichtlich nicht der Intention des Nutzers entsprechen. Ziel beim fehlerkorrigierenden Parsing ist es daher, nicht nur syntaktische Fehler zu beheben, sondern eine Correction zu bestimmen, die eben dieser Intention entspricht.

Bekannte Verfahren beschränken die Menge der Corrections zum Beispiel durch eine maximale Anzahl an Manipulationen der Eingabe w. Ein anderer Ansatz ist, eine effiziente und endliche Repräsentation aller Corrections zu berechnen. Dazu werden in [4], anhand einer Erweiterung des Earley Parsers, alle möglichen Corrections in kubischer Laufzeit berechnet und in der Datenstruktur des Shared Packed Parse Forest(SPPF) [5] gespeichert. In einem nächsten Schritt kann dann die Menge aller Corrections durch so genannte Filter auf Ebene des Parse Forest eingeschränkt werden [6].

Motivation für diese Arbeit ist es, wie in [6] vorgeschlagen, diesen Filterprozess interaktiv zu gestalten. Dabei sollen Nutzereingaben iterativ in den Auswahlprozess eingebunden werden, sodass am Ende eine konkrete Correction aus der Menge aller Corrections bestimmt wird. Aus dieser Überlegung ergibt sich zunächst die folgende Forschungsfrage:

1. Wie lässt sich aus der Menge aller Corrections in einem interaktiven Verfahren eine konkrete Correction bestimmen?

Der erste Teil der Arbeit beschäftigt sich mit der effizienten Berechnung und Repräsentation aller Corrections in einem SPPF (Kapitel 2), sowie der nutzergesteuerten Traversierung im SPPF (Kapitel 3). Ziel dieser Traversierung ist es, anhand von Nutzereingaben eine konkrete Correction aus dem SPPF zu extrahieren. Im Anschluss gilt es, das entwickelte Verfahren hinsichtlich der Effizienz als auch anhand des konzeptionellen Aufwands zu untersuchen. Insbesondere da dergesamte SPPF berechnet werden muss, bevor die erste Interaktion mit dem Nutzer stattfinden kann. Dies motiviert die zweite zentrale Forschnugsfrage dieser Arbeit:

2. Wie kann das Verfahren zur nutzergesteuerten Fehlerkorrektur optimiert werden und möglichst früh in den Parsingprozess integriert werden?

Mit dieser Frage beschäftigt sich Kapitel 4. Während der Formalisierung zur Optimierung des Verfahrens zeigte sich, dass eine Traversierung des SPPF nicht notwendig ist und eine nutzergesteuerte Fehlerkorrektur direkt auf der Ebene des Earley Parsers realisiert werden kann.

Diese Erkenntnis wird durch die Arbeit von Klint und Visser gestützt, in der Mehrdeutigkeiten kontextfreier Grammatiken durch formale Regeln eindeutig gemacht werden [6]. Eine kontextfreie Grammatik ist mehrdeutig, wenn für ein Wort mehrere Syntaxbäume existieren. Ein Syntaxbaum repräsentiert dabei die syntaktische Struktur eines Wortes. Ein Beispiel hierfür ist eine mehrdeutige kontextfreie Grammatik für die Menge der arithmetischen Ausdrücke. Für das Wort $a+b\cdot c$ könnte ein Syntaxbaum die Auswertungsfolge $(a+b)\cdot c$ vorgeben und ein zweiter Syntaxbaum die Auswertungsreihenfolge $a+(b\cdot c)$.

Klint und Visser setzen die Eindeutigkeitsregeln durch Filter um, die auf der Ebene eines Parse Forest angewandt werden. Zudem wird angeregt diese Regeln so früh wie möglich im Parsingprozess einzuführen [6]. Dieses Konzept lässt sich direkt auf den Anwendungsfall der vorliegenden Arbeit übertragen. Abbildung 1 zeigt dazu das angepasste Schaubild aus [6] zum Parsingprozess mehrdeutiger Grammatiken. In unserem Fall entspricht der *Sentence* einem (möglicherweise fehlerhaften) Eingabewort. Der *Parse Forest* ist hier ein SPPF, der alle Corrections für das Eingabewort enthält. Die Eindeutigkeitsregeln (*Disambiguation Rules*) werden in dieser Arbeit durch interaktive Nutzereingaben definiert. Daraus entsteht dann eine konkrete Correction, die im Schaubild einem *Tree* entspricht.

Im ersten Teil der Arbeit werden diese Nutzereingaben als Filter auf der Ebene des SPPF interpretiert. Im zweiten Teil hingegen werden die Nutzereingaben direkt in den Earley Parser integriert. Diese Optimierung und frühe Einbindung der Nutzereingabe in den Parsingprozess, entsprechen den zentralen Gedanken dieser Arbeit.

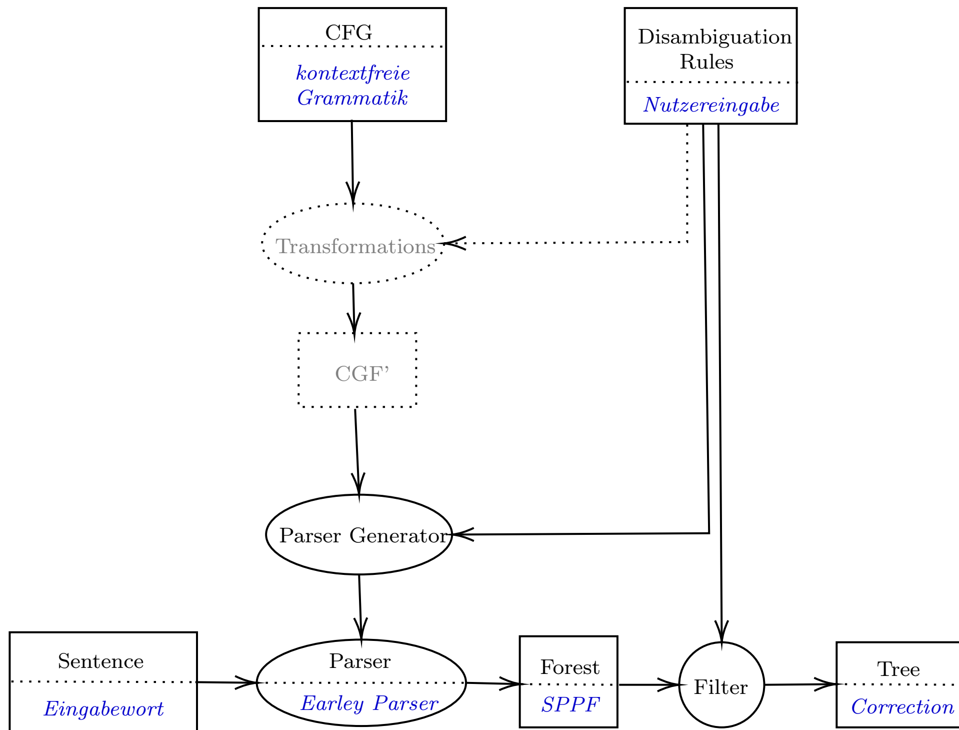

Abbildung 1: Schaubild aus [6] zu den verschiedenen Phasen im Parsingprozess mehrdeutiger Grammatiken, angepasst auf den Anwendungsfall zur nutzergesteuerten Fehlerkorrektur

2 Grundlagen

Dieses Kapitel führt die grundlegenden Konzepte ein, die für den weiteren Verlauf der Arbeit benötigt werden. Nachdem wir in Kapitel 2.1 die benötigten mathematischen Grundlagen definiert haben, werden in Kapitel 2.2 zunächst kontextfreie Sprachen und dazu zugehörige Begriffe eingeführt. Zentral ist hier die Definition eines Syntaxbaums als strukturelle Repräsentation von Ableitungen. Zudem wird in Kapitel 2.3 erklärt, wie mit Hilfe des Earley Parsers [2] entschieden werden kann, ob ein Wort in der Sprache einer kontextfreien Grammatik liegt. Die Definitionen und Notationen sind dabei alle aus der Vorlesung *Formale Sprachen und Logik, 2024* [8] und dem Buch *Mathematische Grundlagen der Informatik* [9] entnommen. Anschließend wird in Kapitel 2.4 erläutert wie alle möglichen Syntaxbäume eines Wortes bezüglich einer kontextfreien Grammatik effizient in der Datenstruktur des *Shared Packed Parse Forest* (SPPF) gespeichert werden können. Darauf aufbauend wird in Kapitel 2.5 der Begriff der Correction formal definiert und erklärt, wie alle Corrections in einem SPPF repräsentiert werden können. Die Definitionen dazu stammen aus [4], [5], [7] und [12]. Abschließend wird in Kapitel 2.6 erläutert, wie alle Corrections mit einer Erweiterung des Earley Parsers [4] berechnet werden können. Diese Grundlagen bilden die Basis für den späteren Teil der Arbeit und das Verfahren zur nutzergesteuerten Fehlerkorrektur.

2.1 Mathematische Grundlagen

Alphabet, Wort und Sprache Ein *Alphabet* Σ ist eine endliche, nicht-leere Menge von Symbolen, die im folgenden auch *Terminale* genannt werden. Ein *Wort* w über einem Alphabet Σ ist eine endliche Folge an Zeichen $w = a_0a_1...a_{n-1}$ mit $a_i \in \Sigma$ für alle $0 \leq i < n$. Dabei definieren wir ε als das *leere Wort*. Für ein Wort w bezeichnen wir die Länge $|w|$ als die Anzahl der Zeichen in w mit $|w| = |a_0a_1...a_{n-1}| = n$. Zudem setzen wir $|\varepsilon| = 0$. Die Menge aller Wörter über Σ bezeichnen wir als Σ^*. Eine *Sprache* L über Σ ist dann eine Teilmenge $L \subseteq \Sigma^*$.

Teilwörter Ein *Teilwort* von w ist eine Folge von Zeichen, die in w aufeinander folgen. Formal ist ein Teilwort von $w = a_0a_1...a_{n-1}$ also eine Folge $a_ia_{i+1}...a_j$, wobei $i \leq j < n$.

Konkatenation von Wörtern Seien $v = a_0...a_{n-1}, w = b_0...b_{m-1} \in \Sigma^*$ zwei Wörter. Ihre *Konkatenation* ist definiert als das Wort $v \cdot w = a_0a_1...a_{n-1}b_0...b_{m-1}$. Wir schreiben im folgenden auch einfach vw statt $v \cdot w$.

Grammatik Eine (allgemeine) Grammatik ist ein 4-Tupel $G = (N, \Sigma, P, S)$, wobei:

- N für die nicht-leere, endliche Menge der *Nonterminale* steht
- Σ das *Alphabet* ist mit $N \cap \Sigma = \emptyset$
- P die Menge an *Produktionen* ist mit $P \subseteq (N \cup \Sigma)^+ \times (N \cup \Sigma)^*$
- $S \in N$ das *Startsymbol* ist.

Ableitungen Wir nennen in folgendem ein $\alpha \in (N \cup \Sigma)^*$ auch *Satzform*. Eine Grammatik G beschreibt dann eine Ableitungsrelation $\Rightarrow_G$ auf Satzformen wie folgt:

$$\alpha\gamma\beta \Rightarrow_G \alpha\delta\beta, \text{ falls } (\gamma, \delta) \in P$$

Falls die Grammatik aus der Anwendung klar wird, nutzen wir auch einfach $\Rightarrow$, anstatt $\Rightarrow_G$. Wir schreiben dann $\alpha \Rightarrow^n \beta$, falls es $\alpha_0, ..., \alpha_n$ gibt mit $\alpha = \alpha_0 \Rightarrow \alpha_1 \Rightarrow ... \Rightarrow \alpha_n = \beta$ für alle $n \in \mathbb{N}$. Zudem schreiben wir $\alpha \Rightarrow^* \beta$, falls es ein $n \in \mathbb{N}$ gibt mit $\alpha \Rightarrow^n \beta$. Für den Fall $n > 0$ schreiben wir dann auch $\alpha \Rightarrow^+ \beta$. Wir nennen eine Grammatik *zyklisch*, falls ein $A \in N$ existiert mit $A \Rightarrow^+ A$. Die von der Grammatik $G = (N, \Sigma, P, S)$ erzeugte Sprache ist dann $L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$.

2.2 Kontextfreie Sprachen

Kontextfreie Sprachen sind in verschiedenen Bereichen der Informatik von zentraler Bedeuting. Insbesondere im Compilerbau und bei Parsingverfahren spielen sie eine große Rolle. In diesem Abschnitt werden wir kontextfreie Sprachen als Sprachklasse definieren und relevante Begriffe für sie einführen. Unter einer Sprachklasse verstehen wir dabei eine Menge an Sprachen.

Kontexfreie Sprachen Sei zunächst $G = (N, \Sigma, P, S)$ eine Grammatik. Wir nennen G kontextfrei, falls $P \subseteq N \times (N \cup \Sigma)^*$ gilt. Wir nennen eine Sprache L kontextfrei, falls es eine kontextfreie Grammatik G gibt, die diese Sprache erzeugt ($L = L(G)$). Wir bezeichnen die Menge aller kontextfreien Sprachen als die Sprachklasse **CFL** (*context free languages*). Wir betrachten dazu folgendes Beispiel einer einfachen kontextfreien Grammatik.

Beispiel 1. Sei $G_{reg} = (\{S, C, T\}, \{+, (,), a, b\}, P, S)$ eine Grammatik mit P wie folgt definiert:

$$\begin{aligned} S &\to S + C \mid C \\ C &\to CT \mid T \\ T &\to (S) \mid a \mid b \end{aligned}$$

Diese Grammatik ist offensichtlich kontextfrei und erzeugt die Sprache $L_{reg} = L(G_{reg})$ der regulären Ausdrücke, wobei diese über $\Sigma = \{a, b\}$ geformt sind und zur Vereinfachung keinen Kleene Stern enthalten. Wir werden im weiteren Verlauf der Arbeit einige Male auf dieses Beispiel zurückgreifen.

Syntaxbäume Im konkreten Anwendungsfall reicht es oft nicht festzustellen, ob ein Wort w in der Sprache $L(G)$ einer Grammatik G enthalten ist oder nicht. Ebenso wichtig ist, wie dieses Wort abgeleitet wurde. Insbesondere interessiert uns, welche Nonterminale jeweils abgeleitet wurden und durch welche Produktionsregeln sie ersetzt wurden. Ein *Syntaxbaum* soll also eine strukturelle Repräsentation der Ableitung für ein Wort bezüglich einer kontextfreien Grammatik darstellen.

Definition 1. Sei $G = (N, \Sigma, P, S)$ eine kontextfreie Grammatik mit $w \in L(G)$. Ein Syntaxbaum für w und G ist ein Baum mit Beschriftungen aus $(N \cup \Sigma \cup \{\varepsilon\})$ mit folgenden Eigenschaften:

- Der Wurzelknoten ist mit dem Startsymbol S beschriftet.
- Die Blätter sind jeweils mit Terminalen oder dem leeren Wort ε beschriftet.
- Jeder innere Knoten, ist mit $A \in N$ beschriftet, wobei seine Kinder jeweils mit $x_0, ..., x_n$ für $A \to x_0...x_n \in P$ beschriftet sind.

Wird die Blätterfront eines Syntaxbaums bezüglich w und G von links nach rechts gelesen, so ergibt sich das Wort w. Die Grammatik G ist *eindeutig*, falls es für jedes $w \in L(G)$ genau einen Syntaxbaum gibt. Falls nicht, so nennen wir G mehrdeutig.

Beispiel 2. Wir betrachten die Grammatik G_{reg} aus Beispiel 1 und das Wort $w = a + ab \in L(G_{reg})$. Das Wort lässt sich dann wie folgt ableiten:

$$S \Rightarrow S + C \Rightarrow C + C \Rightarrow T + C \Rightarrow a + C \Rightarrow a + CT \Rightarrow a + TT \Rightarrow a + aT \Rightarrow a + ab$$

Ein Syntaxbaum für w ist in Abbildung 2 zu sehen. Da die Grammatik eindeutig ist, gibt es nur genau diesen Syntaxbaum für das Wort w.

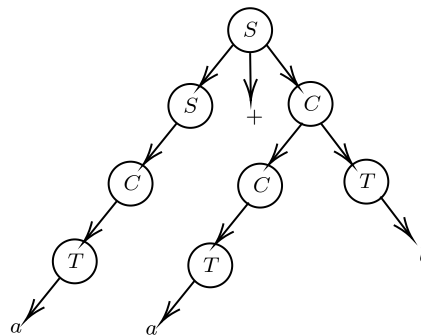

Abbildung 2: Syntaxbaum für das Wort $w = a + ab$ bezüglich der Grammatik G_{reg} aus Beispiel 1.

2.3 Earley Parser

Das Wortproblem für kontextfreie Sprachen stellt die Frage, ob ein Wort in einer kontextfreien Sprache enthalten ist oder nicht. Für ein Wort w mit der Länge $|w| = n$ kann das Wortproblem mit dem *Earley Parser* in Laufzeit $\mathcal{O}(n^3)$ entschieden werden [2]. Für unsere Zwecke genügt eine vereinfachte Version [10] des Earley Parsers aus [2] ohne *Lookahead*, da wir diesen in unserer angepassten Version des Algorithmus nicht verwenden können.

Der Earley Parser nutzt dynamische Programmierung, um zu entscheiden, ob ein Wort

in einer kontextfreien Sprache enthalten ist. Dazu werden so genannte *Earley Items* verwendet, die den aktuellen Zustand im Parsingprozess beschreiben.

Sei dazu $G = (N, \Sigma, P, S)$ eine kontextfreie Grammatik und $w = a_0...a_{n-1} \in \Sigma^*$ ein Wort, wobei $n \in \mathbb{N}$ gilt. Sei zudem $A \rightarrow \alpha\beta \in P$ eine Produktion der Grammatik, wobei $A \in N$ und $\alpha, \beta \in (N \cup \Sigma)^*$.

Definition 2. Wir nennen $[A \rightarrow \alpha \bullet \beta, i]$ ein *Earley Item*, wobei wir $\bullet \notin (N \cup \Sigma)$ fordern. Falls $\beta = \varepsilon$ gilt, so nennen wir das Item ein *finales* Earley Item.

Dabei beschreibt die Position von $\bullet$ in der Produktionsregel, welcher Teil α bereits verarbeitet wurde und welcher Teil β noch vom Parser verarbeitet werden muss. Der Index i beschreibt den Startpunkt eines Teilwortes $a_i...a_j$ für $i \leq j \leq n$, welches von dieser Produktion aus abgeleitet wird.

Definition 3. Ein *Earley Set* Q_j ist eine Menge an Earley Items, wobei $0 \leq j \leq n$ gilt.

Ein Earley Set Q_j und die darin enthaltenen Earley Items fassen den Zustand des Parsingprozess zusammen, nachdem das $j-$te Zeichen im Eingabewort w verarbeitet wurde. Zur Berechnung der Earley Sets, und damit zur Fortführung des Parsingprozess, werden Regeln angewandt, die im Folgenden definiert werden. Seien dazu jeweils $A, B \in N$ und α, β und γ Satzformen.

Definition 4 (Predictor Regel). Falls $[A \rightarrow \alpha \bullet B\beta, i]$ in Q_j enthalten ist, füge $[B \rightarrow \bullet\gamma, j]$ zu Q_j hinzu für alle $B \rightarrow \gamma \in P$.

Durch die Anwendung der Predictor Regel erweitern wir das Earley Set um alle mögliche Produktionen aus P, die als nächstes in Frage kommen.

Definition 5 (Completer Regel). Falls Q_j ein finales Item $[A \rightarrow \alpha\bullet, i]$ enthält, füge $[B \rightarrow \beta A\bullet\gamma, k]$ zu Q_j hinzu für alle Items $[B \rightarrow \beta\bullet A\gamma, k] \in Q_i$, wobei $0 \leq k \leq i \leq j \leq n$.

Das finale Earley Item in Q_j sagt aus, dass die Produktion $A \rightarrow \alpha$ abgeschlossen ist. Die Anwendung der Completer Regel vervollständigt nun diese Produktion und verbindet das Item mit allen Items aus vorherigen Earley Sets Q_i, die als nächstes die Abarbeitung des Nonterminals A fordern.

Definition 6 (Scanner Regel). Falls $[A \rightarrow \alpha \bullet b\beta, i]$ in Q_j enthalten ist und $a_j = b$ gilt, füge $[A \rightarrow \alpha b \bullet \beta, i]$ zu Q_{j+1} hinzu.

Die Scanner Regel vergleicht das nächste Zeichen a_j vom Eingabewort w mit dem vom Parser erwarteten Zeichen b.

Der Earley Parser beginnt mit der Initialisierung des Earley Sets Q_0. Dazu wird für jede Produktion $S \rightarrow \alpha \in P$ vom Startsymbol das Earley Item $[S \rightarrow \bullet\alpha, 0]$ zu Q_0 hinzugefügt. Anschließend werden iterativ die Predictor Regel und die Completer Regel auf Q_0 angewandt. Dies wird so lange fortgesetzt, bis sich Q_0 nicht mehr verändert.

Ist Q_0 nicht leer, so wird die Scanner Regel angewandt, um das nächste Earley Set Q_1 zu initialisieren. Dieser Prozess wird anschließend für alle Earley Sets Q_i für $0 \leq i \leq n$ wiederholt, wobei n die Länge des Eingabewortes ist. Nach der Berechnung von Q_n gilt nun $w \in L(G)$ genau dann, wenn ein finales Earley Item der Form $[S \to \alpha\bullet, 0]$ in Q_n enthalten ist.

2.4 Shared Packed Parse Forest (SPPF)

Als nächstes wollen wir eine Datenstruktur betrachten, die es uns ermöglicht alle Syntaxbäume für ein Wort w bezüglich einer kontextfreien Grammatik G effizient zu repräsentieren. Wir erinnern uns dazu, dass w mehrere Syntaxbäume besitzen kann, falls G merhdeutig ist. Gilt zudem, dass G zyklisch ist, können sogar unendlich viele Syntaxbäume für w existieren. Ein *Shared Packed Parse Forest* [7], [4] und [5] (kurz SPPF) ist ein gerichteter Graph, aus dem alle möglichen Syntaxbäume zu einem Wort extrahiert werden können.

Der Name der Datenstruktur lässt sich durch die beiden Konzepte *Sharing* und *Packing* erklären

- *Sharing* bedeutet, dass gleiche Teilbäume nur einmal gespeichert werden und auf den gleichen Knoten verwiesen wird. Solche Knoten werden wir im Folgenden als *Symbol Nodes* und *Intermediate Nodes* definieren.
- *Packing* bedeutet, dass alle möglichen Ableitungen für ein Teilwort in einem Knoten zusammengefasst werden. Dieser Knoten besitzt dann als Nachfolger so genannte *Packed Nodes*, die jeweils eine alternative Ableitung repräsentieren.

Im weiteren Verlauf der Arbeit werden wir eine Fallunterscheidung über die Art des betrachteten Knoten im SPPF benötigen. Im Folgenden werden wir also die Knotenmenge des SPPF formal definieren. Für die Kantenmenge reicht es uns, nur intuitiv zu erläutern wie die Vorgänger und Nachfolger für die verschiedenen Knotenarten aussehen.

Definition 7. ([5], [7]) Sei $G = (N, \Sigma, P, S)$ eine kontextfreie Grammatik und $w = a_0 a_1 ... a_{n-1} \in \Sigma^*$ ein Wort. Ein *Shared Packed Parse Forest* (SPPF) ist ein beschrifteter, gerichteter Graph $F = (V, E)$, wobei V die Knotenmenge und E die Kantenmenge ist. Die Knotenmenge V ergibt sich dann als Teilmenge der vier disjunkten Mengen V_L (Blätter), V_S (Symbol Nodes), V_I (Intermediate Nodes) und V_P (Packed Nodes) wie folgt:

$$\begin{aligned}
V &\subseteq (V_L \cup V_S \cup V_I \cup V_P) \\
V_L &= \{(a_i, i, i+1) \,|\, 0 \leq i < n-1\} \cup \{(\varepsilon, i, i) \,|\, 0 \leq i < n\} \\
V_S &= \{(A, i, j) \,|\, A \in N \text{ und } 0 \leq i \leq j < n\} \\
V_I &= \left\{ (A \to \alpha \bullet \beta, i, j) \,\middle|\, \begin{array}{l} A \in N,\ \alpha, \beta \in (N \cup \Sigma)^+, \\ A \to \alpha\beta \in P,\ 0 \leq i \leq j < n \end{array} \right\} \\
V_P &= \left\{ (A \to \alpha \bullet \beta, i, k, j) \,\middle|\, \begin{array}{l} A \in N,\ \alpha \in (N \cup \Sigma)^+,\ \beta \in (N \cup \Sigma)^*, \\ A \to \alpha\beta \in P,\ 0 \leq i \leq k \leq j < n \end{array} \right\}
\end{aligned}$$

Das Paar (i, j) in den Knoten nennen wir den *Extent* der Knoten. Die Zahl k in den Packed Nodes nennen wird *Pivot*. Wir betrachten nun die intuitive Bedeutung der Knotenarten und werden dabei erläutern, wie sich die Kantenmenge ergibt:

- **Blätter** (V_L): Für jedes Zeichen a_i im Wort w existiert genau ein Blatt im SPPF. Blätter besitzen keine Nachfolger, da sie die Ableitung für ein einzelnes Terminalsymbol beschreiben. Ein Blatt hat eine Menge an Packed Nodes als Vorgänger.

- **Symbol Nodes** (V_S): Ein Symbol Node (A, i, j) beschreibt alle möglichen Ableitungen für das Teilwort $a_i...a_j$ vom Nonterminal A aus. Daraus können wir ableiten, dass der Symbol Node $(S, 0, n)$ der Startknoten des SPPF ist. Jeder Symbol Node kann mehrere Packed Nodes als Nachfolger haben. Die Anzahl hängt davon ab, wie viele Syntaxbäume bezüglich der Ableitung des Teilworts von A existieren.

- **Intermediate Nodes** (V_I): Ein Intermediate Node $(A \to \alpha \bullet \beta, i, j)$ beschreibt alle möglichen Ableitungen für das Teilwort $a_i...a_j$ von der Satzform α aus. Folglich geht es hier um den linken Teil der Produktion $A \to \alpha\beta \in P$. Genau wie Symbol Nodes besitzen Intermediate Nodes immer eine Menge an Packed Nodes als Nachfolger. Intermediate Nodes sorgen dafür, dass der SPPF eine binäre Struktur erhält. Dies hat den Vorteil, dass die Größe der Datenstruktur nur kubisch in Abhängigkeit des Eingabeworts w anwächst [5]. Es gilt zu beachten, dass die binäre Struktur sich auf die Anzahl der Nachfolgerknoten eines Packed Nodes bezieht. Wie bereits erwähnt können Symbol Nodes und Intermediate Nodes mehr als zwei Nachfolger besitzen.

- **Packed Nodes** (V_P): Ein Packed Node $(A \to \alpha \bullet \beta, i, k, j)$ repräsentiert eine Alternative das Teilwort $a_i...a_j$ von der Satzform α aus abzuleiten. Jeder Packed Node hat einen eindeutigen Vorgänger. Falls $\beta \neq \varepsilon$ gilt, ist das der Intermediate Node $(A \to \alpha \bullet \beta, i, j)$, ansonsten der Symbol Node (A, i, j).

  Ein Packed Node besitzt zwei Nachfolger, ein linkes und ein rechtes Kind. Falls $\alpha = \gamma\delta$ gilt für $\gamma \in (N \cup \Sigma)^*$ und $\delta \in (N \cup \Sigma)^+$, steht das linke Kind für die Ableitung des Teilwortes $a_i...a_{k-1}$ von der Satzform γ aus. Bei dem linken Kind handelt es sich immer um einen Symbol Node oder um einen Intermediate Node. Es gilt zu beachten, dass das linke Kind für den Fall $i = k$ nicht existieren muss. Das ist der Fall, wenn $\gamma = \varepsilon$ gilt.

  Das rechte Kind existiert immer. Es steht für die Ableitung des Teilwortes $a_k...a_j$ von der Satzform δ aus. Bei dem rechten Kind handelt es sich stets um einen Symbol Node oder um ein Blatt.

Wir definieren noch folgende Begriffe für Vorgänger und Nachfolger der einzelnen Knotenarten, da im weiteren Verlauf der Arbeit immer wieder darauf eingegangen wird.

Definition 8. Sei $F = (V, E)$ ein SPPF und sei $v \in V$ ein Symbol Node oder ein Intermediate Node. Wir bezeichnen die Menge aller Vorgänger von v als $\textsc{Pred}(v)$ und die Menge aller Nachfolger von v als $\textsc{Succ}(v)$.

Definition 9. Sei $F = (V, E)$ ein SPPF und sei $u \in V$ ein Packed Node. Dann bezeichnen wir den eindeutigen Vorgänger von v als $\textsc{Parent}(v)$. Das linke Kind von v definieren wir als $\textsc{Left}(v)$ und das rechte Kind als $\textsc{Right}(v)$.

Syntaxbäume berechnen Für ein Wort $w \in L(G)$ bezüglich einer kontextfreien Grammatik lassen sich alle möglichen Syntaxbäume extrahieren. Diese können durch folgende Schritte bestimmt werden.

1. Zunächst muss vom Startknoten $(S, 0, n)$ aus für jeden Symbol und Intermediate Node genau einer der Packed Nodes als Nachfolger ausgewählt werden.

2. Als nächstes müssen die Zyklen in den einzelnen Teilgraphen behandelt werden. Dazu können wir diese beliebig oft abrollen. Jede Anzahl an Abrollungen für einen Zyklus erzeugt dann einen seperaten Syntaxbaum für w. Wir erinnern uns dazu, dass Zyklen durch Ableitungen der Form $A \Rightarrow^+ A$ zustande kamen. Die Abrollung von Zyklen bedeutet, dass Knoten im SPPF mehrfach besucht werden.

3. Zuletzt müssen noch die einzelnen Bäume in tatsächliche Syntaxbäume umgewandelt werden. Dazu werden alle Intermediate Nodes und Packed Nodes entfernt. Anschlißend müssen die Kinder des entfernten Knoten seinm Vorgänger als Nachfolger hinzugefügt werden. Zudem muss jeder Symbol Node (A, i, j) in den Knoten (A) umbenannt werden.

Visualisierung und Beispiel In der Visualisierung eines SPPF werden wir Symbol Nodes und Blätter als Rechtecke mit abgerundeten Kanten zeichnen. Intermediate Nodes werden als Rechtecke gezeichnet. Da die Beschriftung eines Packed Nodes eindeutig aus seinem Vorgänger abgeleitet werden kann, zeichnen wir diese im weiteren Verlauf einfach als Kreise und verwerfen die Beschriftung. Zuletzt wird die Kante eines Packed Node zu seinem linken Kind in roter Farbe dargestellt, und die Kante zu seinem rechten Kind in blauer Farbe. Ein Beispiel für die Visualisierung eines SPPF ist in Abbildung 3 zu sehen.

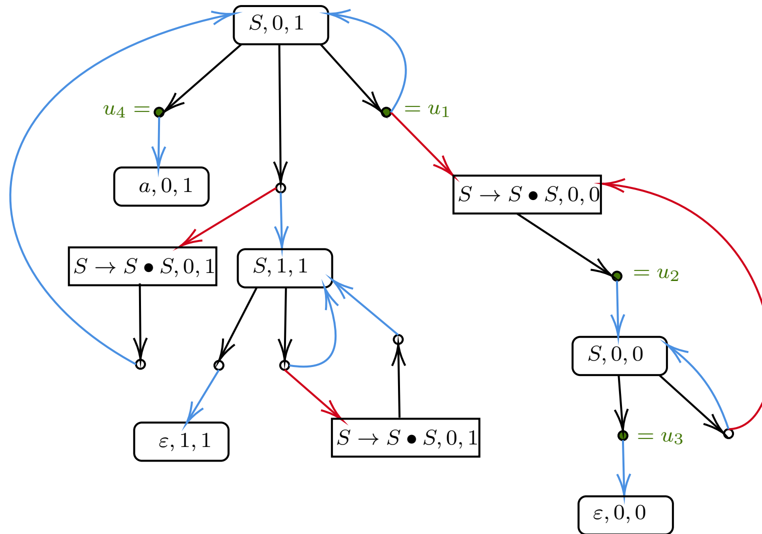

Abbildung 3: SPPF für $G = (\{S\}, \{a\}, \{S \to SS \mid a \mid \varepsilon\}, S)$ und das Wort $w = a$

Aus dem SPPF lässt sich der Syntaxbaum aus Abbildung 4 extrahieren. Dazu besuchen wir vom Startknoten aus die Packed Nodes u_1, u_2, u_3 und u_4 in dieser Reihenfolge. Für die Berechnung muss der Zyklus $(S, 0, 1) \to u_1 \to (S, 0, 1)$ genau einmal abgerollt werden, um als nächstes u_4 zu besuchen.

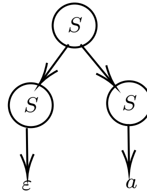

Abbildung 4: Ein Syntaxbaum bezüglich des SPPF aus Abbildung 3.

2.5 Corrections

Bis jetzt wurde nur der Fall betrachtet, dass ein Wort w in der Sprache einer kontextfreien Grammatik G enthalten ist. Als nächstes betrachten wir den Fall, dass w nicht in $L(G)$ enthalten ist. Bekannte Parsingverfahren, wie zum Beispiel der Earley Parser aus [2], verwerfen in diesem Fall die Eingabe. In den nächsten Abschnitten wollen wir erarbeiten, wie so genannte *Corrections* bezüglich w berechnet werden können, anstatt dass die Eingabe verworfen wird. Eine Correction ρ soll dabei eine Manipulation des ursprünglichen Eingabewortes $w \notin L(G)$ definieren, sodass die Anwendung der Correction $\rho(w)$ in die Sprache $L(G)$ fällt. Dazu werden wir zunächst den Begriff der Correction formal definieren. In dem darauf folgenden Abschnitt betrachten wir dann, wie wir alle Corrections berechnen können.

Um den Earley Parser so anzupassen, dass er alle möglichen Corrections berechnet, benötigen wir zunächst eine formale Definition einer Correction. Dazu definieren wir den Begriff der *Edit Operation* als eine Manipulation eines einzelnen Zeichens über einem Alphabet.

Definition 10. Sei Σ ein Alphabet. Eine Edit Operation ist eine Abbildung $\tau : \Sigma \cup \{\varepsilon\} \rightarrow \Sigma^* \cup \{\perp\}$, die wie folgt definiert ist:

$$\tau(a) = \begin{cases} b, & \text{falls } \tau = b^{\uparrow} \text{ und } a = \varepsilon \\ \varepsilon, & \text{falls } \tau = a^{\downarrow} \\ b, & \text{falls } \tau = a^{/b} \\ a, & \text{falls } \tau = a^{\leftarrow} \\ \perp, & \text{sonst.} \end{cases}$$

wobei $b \in \Sigma, x \in \Sigma^*$ gilt und $\perp$ der undefinierte Wert ist, falls es sich bei τ um keine der anderen Edit Operationen handelt. Wir führen zudem die Konvention ein, dass wir x für eine Folge von Insert Operationen nutzen ($x \in \{a^{\uparrow} \mid a \in \Sigma\}^*$) und v für eine Read Operation, Delete Operation oder Replace Operation ($v \in \{a^{\leftarrow}, a^{\downarrow}, a^{/b} \mid a, b \in \Sigma\}$). Wir führen die Konvention für x als Folge von Insert Operationen ein, da wir in einem Wort vor jedem Zeichen beliebig viele Einfügungen zulassen wollen.

Auf diese Weise lässt sich der Begriff der Correction wie folgt als eine Sequenz mit einer festen Länge definieren.

Definition 11. Eine Correction für ein Wort w der Länge n ist eine Folge $\rho = \langle x_0, v_0, x_1, v_1, ..., v_{n-1}, x_n \rangle$ der Länge $2n + 1$, wobei $x_i \in \{a^{\uparrow} \mid a \in \Sigma\}^*$ für $0 \leq i \leq n$ und $v_j \in \{a^{\leftarrow}, a^{\downarrow}, a^{/b} \mid a, b \in \Sigma\}$ für $0 \leq j < n$.

Diese Definition ermöglicht es uns, eine Corrections als ein Wort ρ über dem Alphabet $\hat{\Sigma} := \{a^{\uparrow}, a^{\leftarrow}, a^{\downarrow}, b^{/a} \mid a, b \in \Sigma\}$ zu modellieren. Dadurch wird im Folgenden die Menge aller Corrections zu einer kontextfreien Sprache. Diese Defintionen orientieren sich an [4], [7] und [12]. Das korrigierte Wort resultiert dann aus der Anwendung der Werte der einzelnen Edit Operationen. Formal definieren wir dies wie folgt.

Definition 12. Sei Σ ein Alphabet mit $w \in \Sigma^*$ und $\rho = \langle x_0, v_0, x_1, v_1, ..., v_{n-1}, x_n \rangle$ eine Correction. Die Anwendung von ρ auf w ist eine Abbildung $\rho(w) : \Sigma^* \rightarrow \Sigma^*$ mit $\rho(w) = x_{0_0}(\varepsilon) \cdot x_{0_1}(\varepsilon) \cdot ... \cdot x_{0_m}(\varepsilon) \cdot v_0(a_0) \cdot ... \cdot v_{n-1}(a_{n-1}) \cdot x_n(\varepsilon) \cdot x_{n_1}(\varepsilon) \cdot ... \cdot x_{n_m}(\varepsilon)$

Die Anwendung einer Correction ρ bezieht sich also auf genau ein Wort w. Wir definieren die Menge aller Corrections dann wie folgt.

Definition 13. Sei Σ ein Alphabet mit $w \in \Sigma^*$ und $w = a_0 a_1 .. a_{n-1}$. Sei zudem L eine kontextfreie Sprache. Die Menge aller Corrections $C_w(L)$ bezüglich w und L ist dann wie folgt definiert:

$$\begin{aligned} C_w(L) = \big\{\rho \mid \rho = (x_0, v_0, x_1, v_1, \ldots, v_{n-1}, x_n), x_i \in \{a^{\uparrow} \mid a \in \Sigma\}^*, \\ v_i \in \{a_i^{\leftarrow}, a_i^{\downarrow}, a_i^{/b} \mid b \in \Sigma\} \text{ für } 0 \leq i < n \text{ und } \rho(w) \in L\big\}. \end{aligned}$$

Ist die Zielsprache L klar, so bezeichnen wir diese Menge im weiteren Verlauf auch einfach als C_w.

Beispiel 3. Wir betrachten das Wort $w = ++$ und die Sprache L, die durch die Grammatik G_{reg} aus Beispiel 1 erzeugt wird. Sei dann die Correction $\rho = \langle a^{/+}, +^{\leftarrow}, (^{\uparrow}b^{\uparrow})^{\uparrow}\rangle$ gegeben. Daraus resultiert dann die Anwendung $\rho(w) = a + (b) \in L$ und somit auch $\rho \in C_w(L)$.

Folgendes Theorem dient uns im nächsten Abschnitt dazu, eine algorithmische Lösung für das Finden aller Corrections zu erarbeiten.

Theorem 1. (Theorem 1 aus [4]) Sei Σ ein Alphabet, $L \in \Sigma^*$ eine kontextfreie Sprache und $w \in \Sigma^*$ ein Wort.
Dann ist $C_w(L)$ eine kontextfreie Sprache über $\widehat{\Sigma} = \{a^{\uparrow}, a^{\rightarrow}, a^{\downarrow}, a^{/b} \mid a, b \in \Sigma\}$

Dieses Resultat ermöglicht es uns nun, alle Corrections bezüglich eines Wortes und einer kontextfreien Grammatik in einem SPPF zu speichern und in Kapitel 2.6 mit Hilfe einer Erweiterung des Earley Parsers [4] zu berechnen.

2.5.1 Corrections in einem SPPF speichern

Dazu betrachten wir zunächst, wie wir die Datenstruktur des SPPF anpassen müssen, um alle Corrections darin zu speichern. In [7] wird detailliert erklärt, wie ein SPPF bezüglich Corrections definiert werden kann. Für unsere Zwecke reichen die folgenden zwei Anpassungen der Datenstruktur.

1. Wir passen die Definiton 7 für Intermediate Nodes $(A \to \alpha \bullet \beta, i, j)$ an und erlauben auch $a = \varepsilon$, um aufeinanderfolgende Deletion Operationen im SPPF repräsentieren zu können.

2. Alle Blätter im SPPF sind nun mit Edit Operationen beschriftet.

In Abbildung 5 ist ein Beispiel eines SPPF gegeben, der alle Corrections bezüglich der Grammatik $G = (\{S, C\}, \{a, c\}, \{S \to aS, S \to C, C \to c\}, S)$ und dem Wort $w = a$ enthält. Da die Blätter nun mit Edit Operationen beschriftet sind, repräsentiert ein Syntaxbaum aus dem SPPF genau eine Correction. Diese lässt sich durch die Bläterfront des Syntaxbaums von links nach rechts ablesen. In Abbildung 6 ist ein Syntaxbaum zu sehen. Dieser lässt sich aus dem SPPF über die Packed Nodes u_1, u_2, u_3 und u_4 extrahieren. Die Correction des Syntaxbaum ist dann $\rho = \langle a^{\leftarrow}, c^{\uparrow}\rangle$.

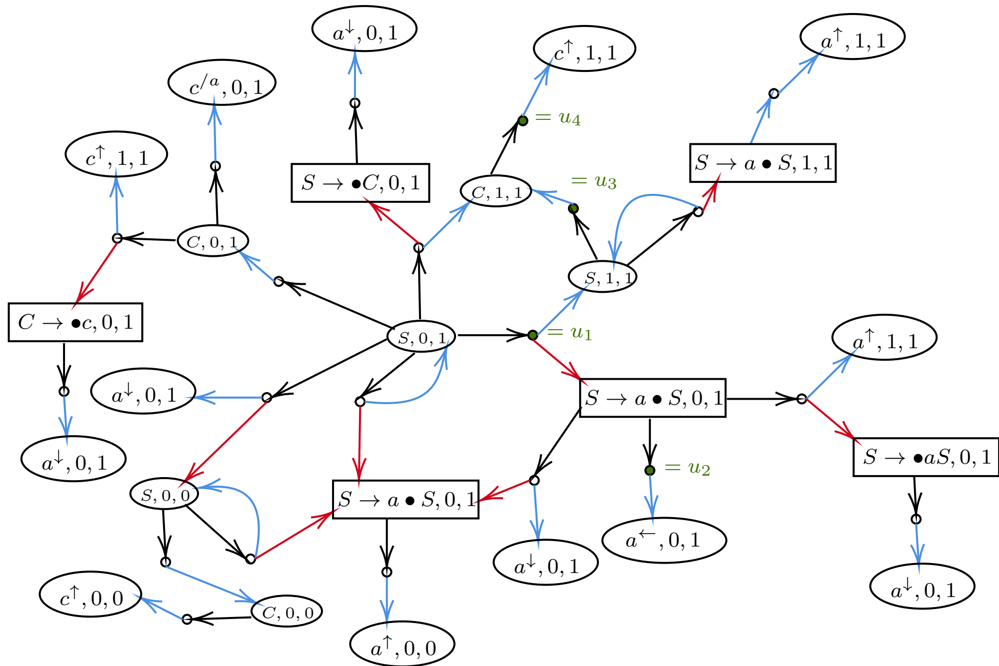

Abbildung 5: SPPF für $C_w(L)$, wobei $L = L(G)$ durch die kontextfreie Grammatik $G = (\{S, C\}, \{a, c\}, \{S \to aS, S \to C, C \to c\}, S)$ erzeugt wird und $w = a$ gilt.

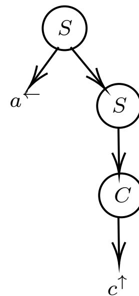

Abbildung 6: Ein Syntaxbaum bezüglich des SPPF aus Abbildung 5

2.6 Berechnung aller Corrections

Nachdem wir im letzten Abschnitt gezeigt haben, wie wir bezüglich einer kontextfreien Grammatik und einem Wort alle Corrections in einem SPPF speichern können, lernen wir nun einen Algorithmus zur Berechnung aller Corrections kennen. Dazu definieren wir zunächst das entsprechende Problem aus [4] wie folgt:

Gegeben: eine kontextfreie Sprache $L \subseteq \Sigma^*$ und ein Wort $w \in \Sigma^*$.
Berechne: die Menge aller Corrections $C_w(L)$.

Dabei beschränken wir uns auf kontextfreie Sprachen, da nach [12] das Problem bereits für kontextsensitive Sprachen unentscheidbar ist. Die Idee des Algorithmus ist dabei eine Erweiterung des Earley Parser aus [2], die es uns ermöglicht, einen SPPF bezüglich einer kontextfreien Grammatik und einem Wort zu konstruieren, der alle Corrections für w enthält. Der Parsing Algorithmus AllCep wurde in [4] und [7] entwickelt. In Kapitel 2.6.1 werden wir dieses Verfahren besprechen. Anschließend wird noch ein Algorithmus zur Konstruktion des SPPF in Kapitel 2.6.2 vorgestellt.

2.6.1 All Correction Earley Parser

Wir werden jetzt dem Earley Parser aus [2] neue Regeln hinzufügen, die es uns ermöglichen, aus den Earley Sets einen SPPF zu berechnen. Dazu wird der klassische Earley Parser um eine mögliche Manipulation des aktuell betrachteten Zeichen im Eingabewort erweitert. Unter einer solchen Manipulation verstehen wir hier eine Anwendung einer Edit Operation aus Definiton 10. Weiterhin werden die bestehenden Regeln dahingehend erweitert, dass zusätzlich ein Knoten des SPPF in einem Earley Item gespeichert wird. Dieser Knoten korrespondiert dann mit dem gleichen Parsingzustand wie das Earley Item. Wir werden die Berechnung der Earley Sets und die Berechnung des SPPF dabei voneinander trennen. Dies dient dem besseren Verständnis der vorgestellten Verfahren.

Hierfür werden zunächst einige Defintionen eingeführt, die für die Erweiterung des Algorithmus benötigt werden. Sei dazu $G = (N, \Sigma, P, S)$ eine kontextfreie Grammatik und $w = a_0...a_{n-1} \in \Sigma^*$ ein Wort, wobei $n \in \mathbb{N}$ gilt. Sei zudem $A \to \alpha\beta \in P$ eine Produktion der Grammatik, wobei $A \in N$ und $\alpha, \beta \in (N \cup \Sigma)^*$. Zuletzt sei noch der SPPF F bezüglich w und G gegeben.

Definition 14. Wir nennen $[A \to \alpha \bullet \beta, j, v]$ ein Earley Item mit Knoten. Für v gilt $v = \mathit{null}$ oder v ist ein Symbol Node, oder ein Intermediate Node in F.

In diesem Kapitel sehen wir v zunächst als einen Platzhalter im Earley Item an, der später zur Berechnung eines Knotens im SPPF genutzt wird. Für die Bestimmung der Earley Sets werden nun die Regeln für den AllCep Parser eingeführt.

Definition 15 (Predictor Regel). Falls $[A \to \alpha \bullet B\beta, i, v]$ in Q_j enthalten ist, füge $[B \to \bullet\gamma, j, \mathit{null}]$ zu Q_j hinzu für alle $B \to \gamma \in P$.

Bei der Anwendung der Predictor korrespondiert kein Knoten aus dem SPPF mit dem neuen Earley Item. Der Grund dafür ist, dass mit diesen Items noch keine tatsächliche Ableitung beziehungsweise Correction eines Teilwortes von w assoziiert ist.

Definition 16 (**Completer Regel**)**.** Falls Q_j ein finales Item $[A \to \alpha \bullet, i, v'']$ enthält, füge $[B \to \beta A \bullet \gamma, k, v']$ zu Q_j hinzu für alle Items $[B \to \beta \bullet A\gamma, k, v] \in Q_i$, wobei $0 \leq k \leq i \leq j \leq n$.

Definition 17 (**Scanner Regel**)**.** Falls $[A \to \alpha \bullet b\beta, i, v]$ in Q_j enthalten ist und $a_j = b$ gilt, füge $[A \to \alpha b \bullet \beta, i, v']$ zu Q_{j+1} hinzu.

Bis jetzt wurden lediglich die bereits bekannten Regeln aus Kapitel 2.3 des klassischen Earley Parsers definiert. Folgende Regeln erweitern den Earley Parser nun zum AllCep Parser aus [4].

Definition 18 (**Insertion Regel**)**.** Falls $[A \to \alpha \bullet b\beta, i, v]$ in Q_j enthalten, füge $[A \to \alpha b \bullet \beta, i, v']$ zu Q_j hinzu.

Falls das betrachtete Earley Item fordert, dass als nächstes das Zeichen b im Eingabewort gelesen werden muss, können wir dieses Zeichen auch in w vor a_j einfügen. Zu beachten ist, dass hierbei a_j noch nicht vom Parser verarbeitet wurde. Aus diesem Grund wird das neue Item auch wieder zu Q_j hinzugefügt.

Definition 19 (**Replacement Regel**)**.** Falls $[A \to \alpha \bullet b\beta, i, v]$ in Q_j enthalten ist und $b \neq a_i$ gilt, füge $[A \to \alpha b \bullet \beta, i, v']$ zu Q_{j+1} hinzu.

Falls, das betrachtete Early Item fordert, dass als nächstes das Zeichen $b \neq a_j$ in w gelesen wird, wollen wir ermöglichen, dass das fehlerhafte Zeichen a_j durch b ersetzt werden kann. Dies wird durch die Anwendung der Replacement Regel gewährleistet.

Definition 20 (**Deletion Regel**)**.** Falls $[A \to \alpha \bullet \beta, i, v]$ in Q_j enthalten ist, füge $[A \to \alpha \bullet \beta, i, v']$ zu Q_{j+1} hinzu.

Zuletzt wollen wir ermöglichen, dass zu jedem Zeitpunkt das aktuell betrachtete Zeichen a_j gelöscht werden kann. Das Löschen dieses Zeichens entspricht dem Überspringen der Verarbeitung von a_j, weshalb das betrachtete Earley Item einfach in das nächste Earley Set Q_{j+1} kopiert wird.

Der Algorithmus zur Lösung des Problems AllCEP ist in 1 gegeben. Eingabe ist eine kontextfreie Grammatik $G = (N, \Sigma, P, S)$, sowie ein Wort $w = a_0 a_1 ... a_{n-1}$. Zu Beginn wird in Zeile 0 das Earley Set Q_0 initialisiert, indem für jede Produktion $S \to \alpha \in P$ das Earley Item $[S \to \bullet\alpha, 0, null]$ hinzugefügt wird. Anschließend durchläuft der Algorithmus in Zeile 1 jede Position j im Wort von 0 bis n und berechnet das Earley Set Q_j durch die Anwendung der oben definierten Regeln.

Zunächst werden in der Schleife in Zeile 4 iterativ so lange die Predictor Regel, die Completer Regel und die Insertion Regel angewandt, bis sich die Menge Q_j nicht mehr

verändert. Anschließend werden ab Zeile 15 noch die Scanner Regel, die Replacement Regel und die Deletion Regel angewandt. Diese Regeln fügen bereits Earley Items in Q_{j+1} hinzu.

Ist das letzte Earley Set Q_n berechnet, so besteht die Ausgabe des Algorithmus aus den berechneten Earley Sets Q_j für alle $1 \leq j \leq n$.

Algorithm 1 AllCEP Parsing

Eingabe: Kontext freie Grammatik $G = (N, \Sigma, P, S)$ und Wort $w = a_0 a_1 ... a_{n-1}$
Ausgabe: Earley Sets Q_j für alle $0 \leq j \leq n$

1: Füge $[S \rightarrow \bullet\alpha, 0, null]$ zu Q_0 hinzu für alle $S \rightarrow \alpha \in P$
2: **for all** $0 \leq j \leq n$ **do**
3: $Q'_j = \emptyset$
4:
5: **while** $Q'_j \neq Q_j$ **do**
6: $Q'_j = Q_j$
7: **if** $[A \rightarrow \alpha \bullet B\beta, i, v] \in Q_j$ **then** ▷ Predictor Regel
8: Füge $[B \rightarrow \bullet\gamma, j, null]$ zu Q_j hinzu für alle $B \rightarrow \gamma \in P$
9: **if** Q_j enthält finales Item $[A \rightarrow \alpha\bullet, i, v'']$ **then** ▷ Completer Regel
10: Füge $[B \rightarrow \beta A \bullet \gamma, k, v']$ zu Q_j hinzu für alle Items $[B \rightarrow \beta \bullet A\gamma, k, v] \in Q_i$
11: **if** $[A \rightarrow \alpha \bullet b\beta, i, v] \in Q_j$ **then** ▷ **Insertion** Regel
12: Füge $[A \rightarrow \alpha b \bullet \beta, i, v']$ zu Q_j hinzu.
13:
14: **if** $[A \rightarrow \alpha \bullet b\beta, i, v] \in Q_j$ mit $b = a_j$ **then** ▷ Scanner Regel
15: Füge $[A \rightarrow \alpha b \bullet \beta, i, v']$ zu Q_{j+1} hinzu.
16: **if** $[A \rightarrow \alpha \bullet b\beta, i, v] \in Q_j$ mit $b \neq a_j$ **then** ▷ **Replacement** Regel
17: Füge $[A \rightarrow \alpha b \bullet \beta, i, v']$ zu Q_{j+1} hinzu.
18: **if** $[A \rightarrow \alpha \bullet \beta, i, v] \in Q_j$ **then** ▷ **Deletion** Regel
19: Füge $[A \rightarrow \alpha \bullet \beta, i, v']$ zu Q_{j+1} hinzu.

2.6.2 Berechnung des SPPF

Im Folgenden wird nun erklärt wie der SPPF, mit Hilfe der Earley Items und den darin enthaltenen Platzhaltern v, v' und v'', berechnet werden kann. Sei dazu $G = (N, \Sigma, P, S)$ eine kontextfreie Gramatik und $w = a_0 ... a_{n-1} \in \Sigma^*$ ein Wort. Weiterhin sei für jedes $0 \leq j \leq n$ das Earley Set Q_j gegeben, das durch den AllCep Parser berechnet wurde.

Wir betrachten dazu ein Earley Item $[A \rightarrow \alpha \bullet \beta, i, j, v] \in Q_j$ bezüglich der Produktion $A \rightarrow \alpha\beta \in P$. Der Platzhalter v kann nun eine der folgenden drei Formen annehmen.

1. Der Knoten entspricht dem Intermediate Node $(A \rightarrow \alpha \bullet \beta, i, j)$.

2. Der Knoten entspricht dem Symbol Node (A, i, j). Dies ist der Fall, wenn das Earley Item final ist, also $\beta = \varepsilon$ gilt.
3. Der Platzhalter v ist *null* und somit mit keinem Knoten im SPPF assoziiert. Dies ist nach Anwendung der Predictor Regel der Fall.

Im ersten Fall enthält der Teilgraph des Knoten v alle Syntaxbäume für das Teilwort $a_i ... a_j$, die sich aus dem α-Teil der Produktion $A \to \alpha\beta$ ergeben. Wie in Kapitel 2.5 erläutert, repräsentiert ein Syntaxbaum hier eine konkrete Correction für das Teilwort. Im zweiten Fall enthält der Teilgraph des Knoten $v = (A, i, j)$ dann alle Corrections des Teilwortes $a_i ... a_j$, welches vom Nonterminal A abgeleitet wird. Dabei ist zu beachten, dass dieser Symbol Node mit mehreren Earley Items assoziiert sein kann, sofern es für das Nonterminal A mehrere Produktionen in P gibt.

Abbildung 7 visualisiert nun die Berechnung der Kanten für alle Regeln, bei denen eine Edit Operation auf ein Zeichen im Wort angewandt wird. In allen Fällen wird hier zusätzlich ein Blatt erzeugt, welches mit der entsprechenden Edit Operation beschriftet ist. Die beteiligten Knoten werden über einen erstellten Packed Node miteinander verbunden.

Die Kantenberechnung anhand der Completer Regel ist in Abbildung 8 dargestellt. Hier werden bereits bestehende Teilgraphen im SPPF miteinander verbunden. Zu beachten ist, dass alle rot markierten Knoten in den Abbildungen auch *null* sein können.

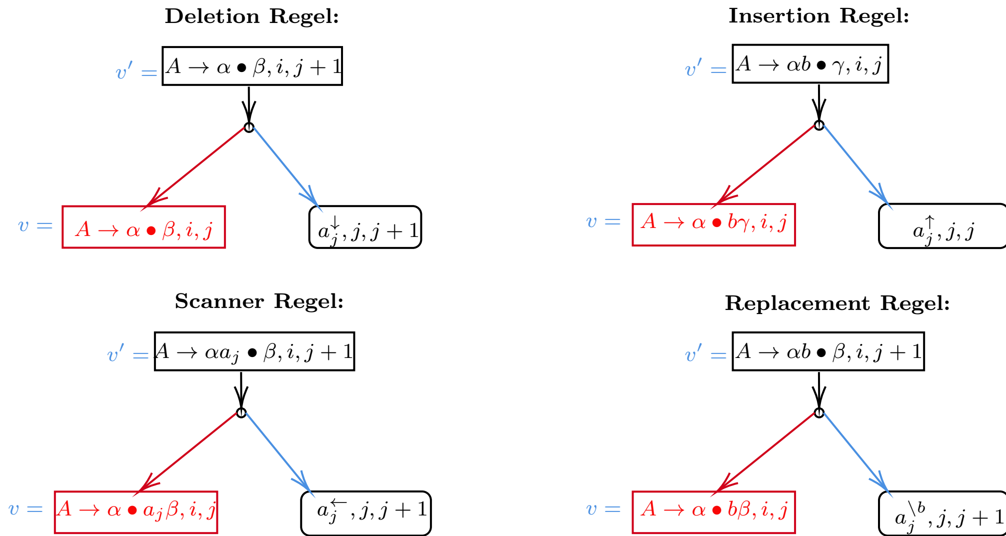

Abbildung 7: Visualisierung der Kantenberechnung für den SPPF bei denen jeweils ein Blatt mit einer Edit Operation erstellt wird. Die rot markierten Knoten können auch *null* sein.

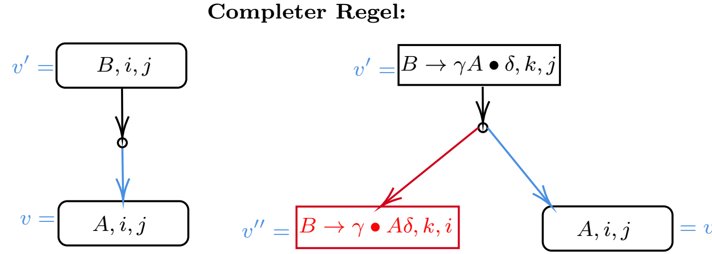

Abbildung 8: Visualisierung der Kantenberechnung für den SPPF bei der Anwendung der Completer Regel für alle $0 \leq k \leq i$. Der rot markierte Knoten kann auch *null* sein.

Das beschriebene Verfahren ist in Algorithmus 2 angegeben. Wir iterieren hierbei in Zeile 1 über alle Early Items $I := \bigcup_{j=1}^{n} Q_j$ und berechnen jeweils die Knoten und Kanten wie zuvor beschrieben. Dazu definieren wir noch die Anwendung der Funktion F.`addFamily`(v, v', v''). Eine Visuialisierung ist in Abbildung 9 gegeben. Hierbei wird dem SPPF ein Packed Node u mit dem Vorgänger v hinzugefügt. Ferner erhält u den Knoten v' als linkes Kind und den Knoten v'' als rechtes Kind.

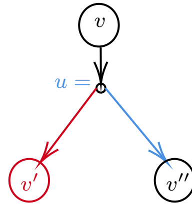

Abbildung 9: Visualisierung der Funktion `addFamily`. Zu beachten ist, dass der linke Knoten von u auch *null* sein kann).

Betrachtet man, wie für Parsingverfahren üblich, die Grammatik nicht als Teil der Eingabe, so ergibt sich die Korrektheit und die Laufzeit für den Algorithmus aus folgendem Lemma.

Lemma 1. (Proposition 2 aus [4]) Für eine gegebene kontextfreie Sprache $L(G)$ und ein Wort $w \in \Sigma^*$ berechnet AllCep $C_w(L(G))$ in Zeit $\mathcal{O}(n^3)$.

Auch die Größe des konstruierten SPPF ist mit diesen Voraussetzungen durch $\mathcal{O}(n^3)$ beschränkt [7]. Diese wird maßgeblich durch die Anwendung der Completer Regel in Zeile 15 der Kantenberechnung bestimmt.

Algorithm 2 Berechnung des SPPF

Eingabe: Kontextfreie Grammatik $G = (N, \Sigma, P, S)$, Wort $w = a_0 a_1 ... a_{n-1}$ und Earley-Items aus $I := \bigcup_{j=1}^{n} Q_j$

Ausgabe: SPPF F mit berechneter Knoten- und Kantenmenge

1: **for all** $[s, i, v] \in I$ und $0 \leq i \leq j \leq n$ **do** ▷ v kann auch *null* sein
2:     **if** $j < n$ **then**
3:         F.`addFamily`$((s, i, j+1), v, (a_j^{\downarrow}, j, j+1))$ ▷ Kanten für Deletion Regel
4:
5:     **if** $s = A \to \alpha \bullet b\beta$ **then**
6:         F.`addFamily`$((A \to \alpha b \bullet \beta, i, j), v, (b^{\uparrow}, j, j))$ ▷ Kanten für Insertion Regel
7:
8:         **if** $b = a_i$ und $j < n$ **then** ▷ Kanten für Scanner Regel
9:             F.`addFamily`$((A \to \alpha b \bullet \beta, i, j+1), v, (b^{\leftarrow}, j, j+1))$
10:         **else** ▷ Kanten für Replacement Regel
11:             **if** $j < n$ **then**
12:                 F.`addFamily`$((A \to \alpha b \bullet \beta, i, j+1), v, (b^{/a_j}, j, j+1))$
13:
14:     **if** $s = A \to \alpha \bullet$ **then** ▷ Item ist final und $v = (A, i, j)$
15:         **for all** $[s', k, v'] \in I$ mit $0 \leq k \leq i$ **do** ▷ Kanten für Completer Regel
16:             **if** $s' = B \to \gamma A \bullet \delta$ und $\gamma \neq \varepsilon$ **then**
17:                 F.`addFamily`$(v', (B \to \gamma \bullet A\delta, k, j), v)$
18:             **if** $s' = B \to A \bullet \delta$ oder $v' = (B, i, j)$ und $B \to A \in P$ **then**
19:                 F.`addFamily`$(v', null, v)$

3 Nutzergesteuerte Fehlerkorrektur im SPPF

In diesem Kapitel wird das Konzept der nutzergesteuerten Traveriserung eines SPPF dargestellt. Ziel ist es, aus der Menge aller Correction eine konkrete Correction zu bestimmen, indem der Nutzer ihm vorgeschlagene Edit Operationen auswählt. Die Correction wird dabei sukzessive von links nach rechts, beginnend bei der ersten Edit Operation, aufgebaut. Dazu traversiert der vorgestellte Algorithmus den SPPF schrittweise und bestimmt für jede Iteration alle nächsten möglichen Edit Operationen. In Anlehnung an das angepasste Schaubild 1 von Klint und Visser aus [6] findet hier der interakive Filterprozess auf Ebene des Parse Forest (SPPF) statt. Zunächst wird die Traversierung im SPPF motiviert und erläutert, bevor die dabei auftretenden Probleme analysiert werden. Auf Basis dieser Analyse und in Verbindung mit den Erkenntnissen aus [6] liegt es nahe, im folgenden Kapitel 4 eine optimierte nutzergesteuerte Fehlerkorrektur einzuführen, bei der bereits auf der Ebene des Earley Parsers die Menge aller Correction eingeschränkt wird.

3.1 Nutzergesteuerte Traversierung im SPPF

Aus Lemma 1 in Kapitel 2.6 wissen wir, dass sich mit Hilfe des Verfahren zum AllCep Parsing (Algorithmus 1) und der anschließenden Kantenberechnung (Algorithmus 2) ein SPPF konstruieren lässt, der alle möglichen Corrections bezüglich einer kontextfreien Grammatik G und einem Wort w enthält. Ziel ist es nun, ausgehend vom Wurzelknoten $(S, 0, |w|)$, mit Hilfe einer Traversierung und anhand von Nutzereingaben genau eine Corrections aus diesem SPPF zu extrahieren.

Um die Intuition der nutzergesteuerten Traversierung zu verstehen, ist es zunächst hilfreich sich zu erinnern, in welcher Form diese Corrections im SPPF repräsentiert sind. Jeder im SPPF enthaltene Syntaxbaum steht dabei für genau eine Correction. Ein solcher Syntaxbaum kann berechnet werden, indem ausgehend vom Wurzelknoten $(S, 0, |w|)$ für jeden Symbol Node und Intermediate Node genau ein Packed Node ausgewählt wird. Die zugehörige Correction ergibt sich anschließend, indem die Blätterfront dieses Syntaxbaums von links nach rechts gelesen wird

Wie in Kapitel 2.5.1 erläutert, sind die Blätter des SPPF mit Edit Operationen gemäß Definition 10 beschriftet. Diese Edit Operationen entsprechen intuitiv Manipulationen im Eingabewort w, wie etwa dem Einfügen, Ersetzen, Lesen oder Löschen einzelner Zeichen. Der SPPF enthält somit alle möglichen Sequenzen solcher Edit Operationen, die w in ein Wort aus der Sprache $L(G)$ überführen. Hier setzt die Idee für ein entsprechendes Verfahren an. Ausgangspunkt ist dabei der vollständig berechnete SPPF.

1. Zunächst sollen dem Nutzer alle möglichen ersten Edit Operationen einer Correction vorgeschlagen werden. Dazu müssen alle Blätter bestimmt werden, die in einem Syntaxbaum ganz links stehen, da diese die erste Edit Operation einer Correction repräsentieren. Wir können diese Blätter ausgehend vom Wurzelknoten erreichen, indem für jeden Packed Node ausschließlich sein linkes Kind traversiert wird.

2. Der Nutzer wählt anschließend aus diesen Vorschlägen eine Edit Operation aus.

3. Daraufhin folgt ein interaktiver Filterprozess, bei dem alle Blätter verworfen werden, deren Beschriftung nicht der vom Nutzer gewählten Edit Operation entspricht. Auf diese Weise werden alle Syntaxbäume ausgeschlossen, die nicht mit der getroffenen Auswahl übereinstimmen.

4. Im folgenden Schritt müssen die nächsten möglichen Edit Operationen bestimmt werden. Dazu wird der SPPF ausgehend von den verbleibenden Blättern so traversiert, dass genau die Blätter erreicht werden, die mit einer Edit Operation beschriftet sind, welche als nächstes in einer Correction folgen kann.

5. Dieser iterative Prozess wird solange wiederholt, bis in der Traversierung wieder der Wurzelknoten erreicht wird. Das Ergebnis ist dann eine konkrete Correction, deren Sequenz von Edit Operationen vollständig durch Nutzereingaben bestimmt wurde.

Die folgenden Kapitel formalisieren diese Idee und entwickeln daraus ein konkretes Verfahren.

3.2 Verfahren zur Traversierung im SPPF

Wir werden jetzt das Vorgehen bei der nutzergesteuerten Traversierung eines SPPF im Detail erarbeiten. Ein Verfahren ist in Algorithmus 3 gegeben. Eingabe ist eine kontextfreie Grammatik $G = (N, \Sigma, P, S)$, ein Wort $w \in \Sigma^*$ mit $w = a_0 a_1 ... a_{n-1}$ und der berechnete SPPF F bezüglich w und G (siehe Kapitel 2.6). Ausgabe ist eine konkrete Correction $\rho \in C_w(L(G))$.

Wir beginnen dabei mit der Initialisierung der Correction $\rho = \langle\rangle$ in Zeile 1. Zu beachten ist, dass ρ zu diesem Zeitpunkt noch keine Correction im Sinne von Definition 11 darstellt. Anschließend initialisieren wir in Zeile 2 die Variable `path`. Diese Variable dient der Verfolgung eines Pfades im SPPF vom Wurzelknoten $(S, 0, n)$ aus.

Algorithm 3 Nutzergesteuerte Traversierung im SPPF

Eingabe: Kontextfreie Grammatik $G = (N, \Sigma, P, S)$; Wort $w = a_0...a_{n-1} \in \Sigma^*$; SPPF F bezüglich G und w;
Ausgabe: Correction $\rho \in C_w(L(G))$

```
 1: ρ = ⟨⟩
 2: path ← ε
 3: root ← (S, 0, n)
 4: currentNodes ← TraverseDown(root, path)
 5: while True do
 6:     nextOperation, filteredNodes ← ChooseNextOperation(currentNodes)
 7:     if rootReached = 1 and nextOperation = STOP then
 8:         break
 9:     ρ.append(nextOperation)
10:     rootReached ← 0
11:     currentNodes ← GetNextNodes(filteredNodes, rootReached)
12: return ρ
```

3.2.1 Startknoten berechnen und Traversierung nach unten

Der erste Schritt ist nun die Berechnung der Startknoten. Unter einem Startknoten verstehen wir einen Packed Node, dessen rechtes Kind mit einer möglichen ersten Edit Operation beschriftet ist. Im Algorithmus entspricht dieser Schritt der Berechnung der Menge `currentNodes` in Zeile 4. Diese Menge enthält Einträge der Form (u, π), wobei u ein Startknoten ist und π ein Pfad von Packed Nodes vom Wurzelknoten $(S, 0, n)$ aus zum Knoten u ist. Die Notwendigkeit der Speicherung dieses Pfades π wird in Abschnitt 4 im Beispiel 4 verdeutlicht.

Wir können nun alle Startknoten und die entsprechenden Pfade, wie zuvor in Kapitel 4 beschrieben, mit Hilfe einer Tiefensuche vom Wurzelknoten $(S, 0, n)$ aus berechnen. Ein Verfahren dazu traversiert für jeden Packed Node nur sein linkes Kind (falls beide Kinder existieren) und ist in der Hilfsfunktion TraverseDown 4 angegeben. Wir betrachten zunächst folgendes Beispiel zur Veranschaulichung.

Beispiel 4. In Abbildung 10 ist ein Teilgraph des SPPF bezüglich G_{reg} (siehe Grammatik G_{reg} aus 1) und dem Wort $w = +$ zu sehen. Der Pfad $\pi = u_1 \cdot u_2 \cdot u_3 \cdot u_4 \cdot u_5 \cdot u_6$ und der Packed Node u_6 werden in Zeile 4 als Tupel (u_6, π) in die Menge `currentNodes` aufgenommen. Zu beachten ist, dass auf dem Pfad zwar ein Zyklus vorliegt, sich aber keine Packed Nodes in π wiederholen.

Da u_6 das Blatt $(a^\uparrow, 0, 0)$ als rechtes Kind besitzt, wird dem Nutzer in der ersten Iteration (unter anderem) vorgeschlagen, ein a vor dem $+$ in w einzufügen (siehe Zeile 6).

Ferner erkennen wir die Notwendigkeit der Verfolgung des Pfades π. Dazu kann der Symbol Node $(T, 0, 0)$ betrachtet werden. Hätten wir hier nicht die Information gespeichert,

dass wir vom Knoten u_5 gekommen sind, wäre nicht klar, welcher der drei Vorgänger auf dem traversierten Pfad zu $(S, 0, 1)$ liegt. Insbesondere der Vorgänger mit dem linken Kind $(C \to C \bullet T, 0, 0)$ stellt ein Problem dar, da hier $(T, 0, 0)$ das rechte Kind dieses Knotens ist. Folglich wäre also auf diesem Pfad $(a^{\uparrow}, 0, 0)$ kein Blatt, welches eine mögliche erste Edit Operation einer Correction ρ darstellt.

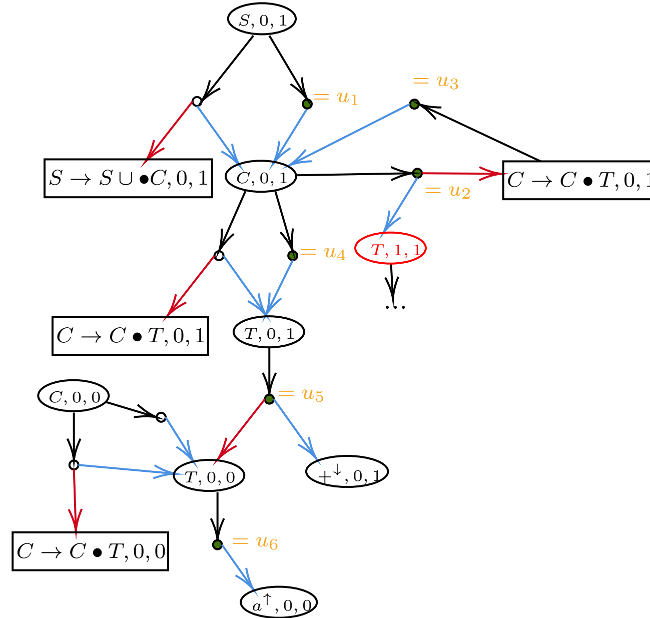

Abbildung 10: Visualisierung der Berechnung eines Pfades $\pi = u_1 \cdot u_2 \cdot u_3 \cdot u_4 \cdot u_5 \cdot u_6$ an Packed Nodes vom Wurzelknoten $(S, 0, n)$ aus zum Knoten u_6. Zur Übersicht wurde nur der für das Beispiel relevante Teilgraph des SPPF angegeben.

Im Allgemeinen berechnet sich die Menge `currentNodes` durch das Verfahren TRAVERSEDOWN (siehe Algorithmus 4) anhand folgender Regeln.

Definition 21 (TRAVERSEDOWN Regeln)**.** Sei $u \in$ ein Knoten im SPPF und π ein Pfad vom Wurzelknoten $(S, 0, n)$ zum Knoten u.

1. Wenn u ein Packed Node ist, RIGHT(u) ein Blatt ist und LEFT(u) nicht existiert, dann gebe $\{(u, \pi)\}$ zurück (siehe Zeile 8) und nehme damit das Paar in die Menge `currentNodes` auf.

2. Wenn u ein Packed Node ist und LEFT(u) existiert, dann traversiere LEFT(u) rekursiv (siehe Zeile 13).

3. Wenn u ein Packed Node ist, LEFT(u) nicht existiert und RIGHT(u) ein Symbol Node oder ein Intermediate Node ist, dann traversiere RIGHT(u) rekursiv (siehe Zeile 11).

4. Wenn u ein Symbol Node oder ein Intermediate Node ist, dann traversiere alle Nachfolger aus $\textsc{succ}(u)$ rekursiv (siehe Zeile 20)

Nach Anwendung dieser Regeln wurden auf dem Pfad π zum Knoten u keine rechten Kinder eines Packed Nodes traversiert, falls das linke Kind existierte. Das Blatt steht also ganz links im betrachteten Teilgraphen und somit auch ganz links in einem entsprechenden Syntaxbaum. Folglich repräsentiert das Blatt nun eine mögliche erste Edit Operation in der Correction ρ. Insbesondere finden wir für jeden Syntaxbaum im Teilgraphen diese Blätter, da wir aufgrund der vierten Regel alle möglichen Packed Nodes besuchen.

Algorithm 4 TraverseDown

Eingabe: Knoten `node` der nach unten traversiert werden soll; Pfad `path`, der alle Packed Nodes auf einem Pfad vom Wurzelknoten $(S, 0, n)$ zu `node` speichert.

Ausgabe: Menge `nextNodes` an Tupeln $(u, \texttt{path})$, wobei $\textsc{right}(u)$ ein Blatt ist, welches mit einer möglichen nächsten Edit Operation beschriftet ist; `path` enthält alle Packed Nodes auf dem Pfad vom Wurzelknoten $(S, 0, n)$ zum Knoten u.

```
 1: if node wurde bereits in path traversiert then
 2:     return ∅
 3:
 4: nextNodes ← ∅
 5: if node is PackedNode then
 6:     path ← path · node
 7:
 8:     if SUCC(RIGHT(node)) = ∅ and LEFT(node) = null then
 9:         return {(node,path)}
10:     else
11:         if LEFT(node) ≠ null then
12:             nextNode ← LEFT(node)
13:         else
14:             nextNode ← RIGHT(node)
15:
16:         nextPath ← COPY(path)
17:         nextNodes ← TraverseDown(nextNode, nextPath)
18:
19: else
20:     for all child ∈ SUCC(node) do
21:         nextPath ← COPY(path)
22:         _nextNodes ← TraverseDown(child, nextPath)
23:         nextNodes ← nextNodes ∪ _nextNodes
24:
25: return nextNodes
```

3.2.2 Pfade im SPPF verfolgen

Nachdem in Zeile 4 `currentNodes` für die erste Iteration der `while`-Schleife berechnet wurde, wählt der Nutzer nun eine Edit Operation, die anhand dieser Menge vorgeschlagen wird. Entsprechend der Nutzereingabe filtert ChooseNextOperation nun die Einträge $(u, \pi) \in$ `currentNodes`. Dabei werden in der Menge `filteredNodes` nur Einträge aufgenommen, bei denen die Beschriftung von $\textsc{right}(u)$ der ausgewählten Edit Operation entspricht (siehe Zeile 6). Ein Verfahren zu ChooseNextOperation ist in Algorithmus 5 angegeben.

Algorithm 5 ChooseNextOperation

Eingabe: Menge `currentNodes` aus Paaren (u, π), wobei u ein Packed Node ist und π ein Pfad vom Wurzelknoten $(S, 0, n)$ aus zum Knoten u ist; Variable `rootReached` $\in \{0, 1\}$, die angibt, ob einer der Knoten den Wurzelknoten als Vorgänger besitzt.
Ausgabe: Die gewählte nächste Edit Operation `nextOperation`; Menge `filteredNodes`, die die anhand der Nutzereingabe gefilterten Einträge aus `currentNodes` enthält.

```
 1: ℰ ← ∅                                    ▷ Menge möglicher nächster Edit-Operationen
 2:
 3: for all (u, π) ∈ currentNodes do
 4:     ℰ ← ℰ ∪ {RIGHT(u)}                   ▷ Extrahiere Edit Operation aus Blatt
 5:
 6: if rootReached = 1 then
 7:     ℰ ← ℰ ∪ {STOP}                       ▷ Nutzer soll Correction abschließen können
 8:
 9: nextOperation ← UserChoose(ℰ)
10:
11: filteredNodes ← { (u, π) ∈ currentNodes | RIGHT(u) = nextOperation }
12:
13: return nextOperation, filteredNodes
```

Nach dem Filterprozess wird in Zeile 9 die gewählte Edit Operation der Correction ρ angehangen. Anschließend muss, ausgehend von den Einträgen in `filteredNodes`, die Menge `currentNodes` für die nächste Iteration bestimmt werden (siehe Zeile 11). Dieser Schritt entspricht der Berechnung aller nächsten möglichen Edit Operationen. Dazu betrachten wir zunächst folgendes Beispiel.

Beispiel 5. Wir schauen uns erneut den Teilgraphen aus Abbildung 10 des SPPF bezüglich der Grammatik G_{reg} aus Beispiel 1 und des Wortes $w = +$ an.

Das Tupel (u_6, π) wurde dabei in die Menge `currentNodes` aufgenommen. Wir nehmen also an, der Nutzer wählt $a^{\uparrow}$ als nächste Edit Operation. Folglich wird das Tupel auch in die Menge `filteredNodes` aufgenommen. Als nächstes muss für den Pfad π also der Packed Node u bestimmt werden, für den Right(u) mit einer nächsten möglichen Edit Operation beschriftet ist. Da $\textsc{right}(u_5)$ mit der Edit Operation $+^{\downarrow}$ beschriftet ist,

wird das Tupel (u_5, π') für $\pi' = u_1 \cdot u_2 \cdot u_3 \cdot u_4 \cdot u_5$ in die Menge `currentNodes` aufgenommen.

Wir nehmen wieder an, dass sich der Nutzer für die passende Edit Operation $+^{\downarrow}$ entscheidet. Entsprechend wird also auch (u_5, π') in `filteredNodes` übernommen. Es gilt nun erneut die nächsten möglichen Edit Operation bezüglich des Pfades π' zu bestimmen. Wir betrachten zunächst u_4. Für diesen Packed Node existiert $\text{LEFT}(u_4)$ aufgrund der Produktion $C \to T$ nicht. Folglich fahren wir mit dem Packed Node u_3 fort. Auch hier existiert $\text{LEFT}(u_3)$ nicht. Wir betrachten also den Packed Node u_2. Für diesen Knoten existiert $\text{LEFT}(u_2)$, aber $\text{RIGHT}(u_2)$ ist in diesem Fall der Symbol Node $(T, 1, 1)$ und kein Blatt (siehe rot markierter Knoten in Abbildung 10).

In diesem Fall müssen wir den Teilgraphen des SPPF mit Wurzelknoten $(T, 1, 1)$ nach unten traversieren, um alle nächsten möglichen Edit Operationen zu bestimmen. Dazu ist der relevante Teil in Abbildung 11 gegeben. Die Traversierung nach unten entspricht in diesem Fall einem Aufruf des Verfahrens $\text{TRAVERSEDOWN}(u, \pi'')$ für $u = (T, 1, 1), \pi'' = u_1 \cdot u_2$ (siehe Algorithmus 4).

Es gilt $\text{SUCC}((T, 1, 1)) = \{u'_1, u'_2, u'_3\}$. Nach Fall 4 aus Definiton 21 müssen wir nun alle Nachfolger in einer Tiefensuche traversieren. Wir betrachten zunächst u'_1. Für diesen Knoten existiert $\text{LEFT}(u'_1)$ nicht und $\text{RIGHT}(u'_1)$ ist ein Blatt. Nach Fall 1 in Definiton 21 nehmen wir also das Tupel $(u'_1, \pi'' \cdot u'_1)$ in `currentNodes` auf. Wir traversieren nun u'_2 weiter nach unten. Nach Fall 2 in Definiton 21 gehen wir als nächstes zum Intermediate Node $[T \to (S\bullet), 1, 1]$. Dieser Knoten hat nur den Nachfolger u'_4. Für diesen Knoten traversieren wir wieder nach Fall 2 den Intermediate Node $[T \to (\bullet S), 1, 1]$ und anschließend den Packed Node u'_5. Hier müssen wir dann das Paar $(u'_5, \pi'' \cdot u'_2 \cdot u'_4 \cdot u'_5)$ in die Menge `currentNodes` aufnehmen. Als letztes betrachten wir noch den Knoten u'_3. Wie zuvor beschrieben, nehmen wir auch das Paar $(u'_3, \pi'' \cdot u'_3)$ in die Menge `currentNodes` auf.

Wählt der Nutzer in der folgenden Iteration die Edit Operation $b^{\uparrow}$, so befindet sich das Tupel $(u'_3, u_1 \cdot u_2 \cdot u'_3)$ in `filteredNodes`. Für den Packed Node u'_3 kommen wir bereits von seinem rechten Kind, weswegen wir u_2 als nächstes betrachten. Da hier kein linkes Kind existiert, ist der Knoten u_1 als nächstes relevant. Hier existiert ebenfalls kein linkes Kind und $(S, 0, 1)$ ist der Vorgängerknoten. Wir setzen daher die Variable `rootReached` auf 1.

Dies führt dazu, dass dem Nutzer vorgeschlagen wird, die Correction ρ abzuschließen. Wird nun die Option `STOP` vom Nutzer gewählt, so wird in Zeile 8 die `while`-Schleife verlassen und $\rho = \langle a^{\uparrow}, +^{\downarrow}, b^{\uparrow} \rangle$ zurückgegeben. Wir sehen, dass in diesem Fall $\rho(+) = ab \in L(G_{reg})$ gilt und somit ρ auch eine wohldefinierte Correction darstellt.

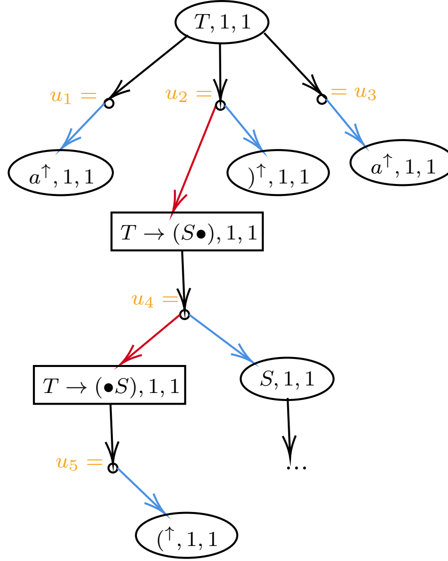

Abbildung 11: Relevanter Teilgraph des SPPF bezüglich $w = \cup$ und G_{reg} zur Traversierung des Knoten $(T, 1, 1)$ nach unten.

Ein Verfahren GetNextNodes, welches den beschriebenen Vorgang im Allgemeinen umsetzt ist in Algorithmus 6 angegeben. Das Verfahren hält sich dabei an folgende Regeln bei der Pfadverfolgung.

Definition 22 (Regeln zur Pfadverfolgung)**.** Sei $u \in$ ein Knoten im SPPF und $\pi = u_1 \cdot ... \cdot u_m \cdot u$ ein Pfad von Packed Nodes vom Wurzelknoten $(S, 0, n)$ aus zum Knoten u. Sei zudem $m > 0$.

1. Wenn $\text{RIGHT}(u_m)$ ein Blatt ist, dann füge $(u_m, u_1 \cdot ... \cdot u_m)$ zu `nextNodes` hinzu (siehe Zeile 21).

2. Wenn $\text{RIGHT}(u_m)$ ein Symbol Node ist, der noch nicht nach unten traversiert wurde, dann füge die Ausgabe von TraverseDown$(u_m, u_1 \cdot ... \cdot u_m)$ zu `nextNodes` hinzu (siehe Zeile 23).

3. Wenn $\text{LEFT}(u_m)$ nicht existiert oder $\text{RIGHT}(u_m)$ bereits nach unten traversiert wurde, dann wiederhole den Vorgang für das Paar $(u_{m-1}, u_1 \cdot ... \cdot u_{m-1})$ (siehe Zeile 9).

Wir benötigen die nächsten möglichen Edit Operationen. Ausgangspunkt ist der Packed Node u für den Right(u) mit der letzten gewählten Edit Operation beschriftet ist. Falls Right(u_m) ebenfalls ein Blatt ist, so ist Right(u_m) auch mit ener nächsten möglichen Edit Operation beschriftet. Deshalb fügen wird in der ersten Regel der Eintrag $(u_m, u_1 \cdot ... \cdot u_m)$ zu `nextNodes` hinzu.

Kommen wir in der Traversierung von links unten und ist Right(u_m) ein Symbol Node

v, so müssen die nächsten Edit Operation aus dem Teilgraphen von v berechnet werden. Deswegen wird in der zweiten Regel der Knoten $\textsc{Right}(u_m)$ nach unten traversiert. Kommen wir in der Traversierung von rechts unten, oder existiert der Knoten $\textsc{Left}(u_m)$ nicht, muss anhand der dritten Regel zum Packed Node u_{m-1} nach oben traversiert werden und der beschriebene Vorgang wiederholt werden.

Wurde ein Packed Node erreicht dessen Teilgraph bereits vollständig traversiert wurde und für den sein Vorgänger dem Wurzelknoten $(S, 0, n)$ entpspricht, kann die Correction abgeschlossen werden.

Algorithm 6 GETNEXTNODES

Eingabe: Eine Menge `nodes` an Tupeln (`node, path`), wobei `node` ein Packed Node ist und `path` der eindeutige Pfad vom Wurzelknoten $(S, 0, n)$ zum Knoten `node` ist; Maximale Anzahl k an Wiederholungen von Packed Nodes auf dem Pfad vom Wurzelknoten $(S, 0, n)$ zum Knoten `node`.

Ausgabe: Eine Menge `nextNodes` an Tupeln (`nextNode, path`). Dabei ist `nextNode` der Packed Node, der eine nächste mögliche Edit Operation als rechtes Kind besitzt. Der Pfad `path` enthält den eindeutigen Pfad von $(S, 0, n)$ zum Knoten `nextNode`

```
 1: nextNodes, traverseDown ← ∅
 2: for all (node, path) ∈ nodes do
 3:     if path = ε then
 4:         rootReached ← 1
 5:     else
 6:         u_1, ..., u_m, node ← path                              ▷ für m > 0
 7:         i ← m
 8:
 9:         while i > 0 do
10:             if LEFT(u_i) ≠ null and RIGHT(u_i) wurde noch nicht traversiert then
11:                 break
12:             i ← i − 1
13:
14:         if i > 1 then
15:             if i = 1 then
16:                 path ← ε
17:             else
18:                 path ← u_1 · ... · u_i
19:
20:             if SUCC(RIGHT(u_i)) = ∅ then                  ▷ RIGHT(u_i) ist Blatt.
21:                 nextNodes ← nextNodes ∪ {(u_i, path)}
22:             else                                    ▷ RIGHT(u_i) ist ein Symbol Node
23:                 traverseDown ← traverseDown ∪ {(RIGHT(u_i), path)}
24:         else
25:             rootReached ← 1
26:
27: for all (u, path) ∈ traverseDown do
28:     _nextNodes ← TRAVERSEDOWN(u, path, k)
29:     nextNodes ← nextNodes ∪ _nextNodes
30:
31: return nextNodes
```

3.3 Korrektheit und Vollständigkeit

Es wird nun erläutert, wie sich die Korrektheit und die Vollständigkeit des Verfahrens zur nutzergesteuerten Traversierung des SPPF (Algorithmus 3) beweisen lassen. Die Beweise werden hier nur grob geführt, da dieser Algorihmus lediglich als Vorbereitung für die im nächsten Kapitel 4 zu betrachtende Optimierung dient. Zudem lassen sich dort die Korrektheit und die Vollständigkeit leichter für das entwickelte Verfahren zeigen. Außerdem werden wir die Abwendung von der Datenstruktur des SPPF motivieren.

Theorem 2. (Korrektheit des Verfahrens aus 3) Sei $G = (N, \Sigma, P, S)$ eine kontextfreie Grammatik, $w = a_0...a_{n-1} \in \Sigma^*$ ein Wort und $\rho \in C_w(L(G))$ eine Correction für w. Sei zudem F der berechnete SPPF. Dann existiert eine Ausgabe des Algorithmus, welche genau der Correction ρ entspricht.

Beweis. Sei c die Länge der Correction und τ_k für $1 \leq k \leq c$ die k-te Edit Operation in ρ. Wir wissen aus Lemma 1, dass die Correction ρ im SPPF enthalten ist und aus einem Syntaxbaum im SPPF bestimmt werden kann. Wir können den Beweis nun per Induktion über die k-te Edit Operation führen.

Für $k = 1$ ist das Blatt ganz links in der Blätterfront des Syntaxbaums mit τ_1 beschriftet. Nach dem Verfahren TraverseDown aus Algorithmus 4 und den darin angewandten Regeln aus Defintion 21, befindet sich in `currentNodes` ein Eintrag (u, π), wobei Right(u) mit τ_k beschriftet ist. Folglich kann der Nutzer also die Edit Operation τ_1 auswählen.

Für ein beliebiges k lässt sich die Induktionshypothese formulieren, dass die ersten $k - 1$ Nutzereingaben genau den ersten $k - 1$ Edit Operationen $\tau_1, ..., \tau_{k-1}$ entsprechen. Dabei sei ein Eintrag (u, π) in der Menge `filteredNodes` gegeben, nachdem der Nutzer die Auswahl τ_{k-1} getroffen hat. Hier ist also Right(u) mit τ_{k-1} beschriftet. Aus den Regeln der Pfadverfolgung aus Definition 22 lässt sich nun folgern, dass in der Menge `currentNodes` für die nächste Iteration ein Eintrag (u', π') vorhanden ist, wobei Right(u') der Edit Operation τ_k entspricht. Folglich kann der Nutzer auch hier die Edit Operatino τ_k wählen.

Für $k = c$ kann nun argumentiert werden, dass u' den Wurzelknoten $(S, 0, n)$ als Vorgänger besitzt und somit das Verfahren mit der Ausgabe ρ durch den Nutzer terminiert werden kann. □

Wir werden nun kurz argumentieren, warum jede Ausgabe des Verfahrens auch einer wohldefinierten Correction entspricht.

Theorem 3. (Vollständigkeit des Verfahrens aus 3) Sei $G = (N, \Sigma, P, S)$ eine kontextfreie Grammatik, $w = a_0...a_{n-1} \in \Sigma^*$ ein Wort und $\rho' = \langle \tau_1, ..., \tau_c \rangle$ die Ausgabe des Algorithmus. Sei zudem F der berechnete SPPF. Dann existiert eine Correction $\rho \in C_w(L(G))$, die genau der Ausgabe ρ' entspricht.

Beweis. Sei $\rho' = \langle \tau_1, ..., \tau_c \rangle$ die Ausgabe des Algorithmus. Wir können ähnlich wie im Beweis der Korrektheit argumentieren, dass $\rho' \in C_w(L(G))$ gilt. Der Unterschied ist hier lediglich, dass der Beweis per Induktion über die k-te Nutzereingabe geführt wird. Der Induktionsanfang kann wieder anhand der TraverseDown Regeln aus Definiton 21 gezeigt werden. Für den Induktionsschritt können erneut die Regeln zur Pfadverfolgung aus Definiton 22 in der Argumentation verwendet werden. □

Im Anschluss an die Beweisführung wird nun das Verfahren hinsichtlich seiner Probleme analysiert.

3.4 Analyse und Probleme

Nachdem wir das Verfahren zur Traversierung des SPPF besprochen haben, wollen wir einige Probleme des bisherigen Ansatzes herausstellen. Sei dazu $G = (N, \Sigma, P, S)$ eine kontextfreie Grammatik und $w = a_0...a_{n-1}$ ein Wort. Sei zudem F der SPPF bezüglich G und w. Nach [4] und [7] ist die Anzahl der Knoten und Kanten von F durch $O(n^3)$ beschränkt. Diese Schranke gilt jedoch nur unter der Annahme, dass die Grammatik G fest ist und nicht als Teil der Eingabe betrachtet wird.

Das zentrale Problem besteht darin, dass der vollständige SPPF bereits vor Beginn der eigentlichen Traversierung berechnet werden muss. Aufgrund der kubischen Laufzeit in Abhängigkeit von der Wortlänge ist dies für große Eingaben problematisch. Zudem wird redundanter Rechenaufwand betrieben, indem Teilgraphen des SPPF berechnet werden, die später durch den interaktiven Filterprozess verworfen werden.

Die Laufzeit einer möglichen Implementierung kann ebenfalls ein Problem darstellen. Zwar ist es möglich, für die Traversierung eine Worst-Case Laufzeit von $\mathcal{O}(n^3)$ zu erreichem, jedoch nur, wenn folgende Aspekte beachtet werden.

1. Auf einem traversierten Pfad im SPPF darf nur eine konstante Anzahl m an Packed Nodes zugelassen werden, da sonst die Abrollung von zu vielen Zyklen zu einer schlechteren Worst-Case Laufzeit führt.

2. Bereits traversierte Teilgraphen und die darin enthaltenen Edit Operationen müssen sich zum Beispiel mit Memoisation gemerkt werden, da sonst exponentielle Laufzeit resultiert.

An dieser Stelle wird nicht weiter auf Details zur Laufzeitanalyse eingegangen, da wir uns im nächsten Kapitel von dem besprochenen Algorithmus lösen.

4 Nutzergesteuerte Fehlerkorrektur mit dem Earley Parser

Aufgrund der in Kapitel 3.4 genannten Probleme beschäftigen wir uns nun damit, wie die Methode aus dem vorherigen Kapitel zur nutzergesteuerten Fehlerkorrektur optimiert werden kann. Anschließend werden wir uns von der Datenstruktur des SPPF lösen und zum Earley Parser zurückkehren. Die Motivation dazu ergab sich während der Formalisierung der Ideen aus Kapitel 4.1. Gestützt wurde dies durch die Arbeit von Klint und Visser [6], in welcher aus Effizienzgründen empfohlen wird, die Eindeutigkeitsregeln möglichst früh im Parsingprozess einzubauen. Dazu erinnern wir uns, wie sich dies auf unseren Anwendungsfall übertragen ließ (siehe dazu auch das angepasste Schaubild aus Abbildung 1 in Kapitel 1). In unserem Fall entsprechen die Eindeutigkeitsregeln den Nutzereingaben zur Berechnung einer konkreten Correction. Bis jetzt wurden diese als Filter auf Ebene des SPPF angewandt, jetzt wollen wir die Nutzereingaben direkt auf Ebene des Earley Parsers einbauen.

Wir werden in Kapitel 4.1 zunächst erläutern wie sich die bisherige Traversierung im SPPF optimieren lässt. Hier wird nur die Intuition dahinter erläutert, da wir uns im folgenden Kapitel 4.2 vom SPPF lösen und die nutzergesteuerte Fehlerkorrektur auf Basis des Earley Parsers motivieren. Daran anschließend folgt in Kapitel 4.3 die Entwicklung dieses Verfahrens. Dieses Kapitel formuliert das zentrale Resultat dieser Arbeit. Zuletzt wird in Kapitel 4.4 die Korrektheit und die Vollständigkeit des Verfahrens bewiesen und in Kapitel 4.5 die Laufzeit analysiert.

4.1 Optimierte Traversierung im SPPF

Eine bereits erwähnte Optimierung des bisherigen Verfahrens besteht darin, während der Traversierung des SPPF bereits besuchte Teilgraphen zu speichern, sodass diese nur einmal traversiert werden müssen. Obwohl sich dadurch der Aufwand reduzieren lässt, bleibt ein zentrales Problem bestehen: Der SPPF muss weiterhin vollständig berechnet werden, bevor die erste Nutzereingabe stattfinden kann.

Die Idee ist es also zunächst, den SPPF nicht mehr vorab zu berechnen, sondern ihn parallel zur Berechnung der Correction aufzubauen. Dazu kann in jeder Iteration nur der Teil des SPPF lokal berechnet werden, der für die Fortsetzung der Correction relevant ist.

Im Rahmen der Formalisierung dazu zeigt sich jedoch, dass die Berechnungen der benötigten Knoten und Kanten vollständig auf Basis der Regeln des Earley Parsers stattfinden und dadurch der Aufbau des SPPF redundant wird. Dies wird im folgenden Kapitel deutlich, welches die Intuition der lokalen Berechnung des SPPF erklärt.

4.1.1 Lokale Berechnung des SPPF

Sei $G = (N, \Sigma, P, S)$ eine kontextfreie Grammatik und $w = a_0...a_{n-1} \in \Sigma^*$ ein Wort. Der erste Schritt in der optimierten Traversierung ist die Berechnung der Startknoten. Wir erinnern uns, dass ein Startknoten einem Packed Node u entspricht, für den $\textsc{Right}(u)$ mit

einer möglichen ersten Edit Operation einer Correction beschriftet ist. Für unseren optimierten Algorithmus müssen die Startknoten vor dem eigentlichen Verfahren bestimmt werden. Dazu gibt es mehrere Ansätze. Eine Möglichkeit ist es, die Startknoten aus der Berechnung der Earley Sets Q_0 und Q_1 aus Algorithmus 8 zu entnehmen. Hierfür kann man sich überlegen, dass die Vorgänger dieser Packed Nodes u mit Earley Items der Form $[A \to b \bullet \alpha, 0, v]$ korrespondieren, da diese den Parsingzustand nach Verarbeitung des ersten Zeichens a_0 in w beschreiben. Bereits hier fällt auf, dass die Berechnung der ersten möglichen Edit Operationen auf Basis des Earley Parsers stattfinden kann.

Im nächsten Schritt wählt der Nutzer eine der ihm vorgeschlagenen Edit Operationen. Nun können wieder die Pfade verworfen werden, die nicht mit der Nutzereingabe übereinstimmen. Anschließend müssen die nächsten möglichen Edit Operationen berechnet werden.

Dazu wurde in Algorithmus 3 der SPPF nach oben traversiert, bis ein Packed Node u erreicht wurde, für den $\textsc{Right}(u)$ mit einer nächsten Edit Operation beschriftet ist. Analog dazu können die Regeln des AllCep Parsers aus Kapitel 2.6.1 genutzt werden, um diese Knoten zu finden. Wir erinnern uns, dass diese Regeln in Abbildung 7 und 8 bezüglich der Kantenberechnung im SPPF visualisiert sind. Es ist noch zu beachten, dass nach Anwendung der Completer Regel ein Teilgraph mit Wurzelknoten (A, i, j) nach unten traversiert werden muss. Die lokale Kantenberechnung erfolgt hier aber nach dem gleichen Prinzip.

Erneut fällt auf, dass bei der optimierten Traversierung nur anhand der Regeln zur Berechnung der Earley Sets stattfindet. In folgendem Kapitel motivieren wir deshalb die nutzergesteuerte Fehlerkorrektur auf Basis des Earley Parsers.

4.2 Vom SPPF zurück zum Earley Parser

Die bisherigen Überlegungen zeigen, dass der explizite Aufbau des SPPF während der interaktiven Berechnung einer Correction redundant ist. Insbesondere wird klar, dass die Berechnung nächster möglicher Edit Operationen vollständig auf Basis der Regeln des AllCep Parsers aus Kapitel 2.6.1 erfolgen kann.

Dies motiviert die Idee für ein optimiertes Verfahren auf Basis des klassischen Earley Parsers [2]. Dazu ist die Intuition, dass die Traversierung im SPPF der Berechnung der Earley Sets entspricht. Jedes Mal, wenn dabei ein Zeichen im Eingabewort w durch die Scanner Regel (siehe Definition 6) verarbeitet werden soll, kann der Nutzer durch Auswahl einer der Regeln des AllCep Parsers eine Edit Operation auf dieses Zeichen anwenden.

Auf diese Weise werden weniger Berechnungsschritte notwendig und das Verfahren wird zudem konzeptionell vereinfacht. Der zugehörige Algorithmus wird im folgenden Abschnitt detailliert erläutert.

4.3 Nutzergesteuerter Earley Parser

Wir werden jetzt den Algorithmus zur nutzergesteuerten Fehlerkorrektur auf Basis des klassischen Earley Parsers aus [2] erarbeiten. Dieser Algorithmus stellt den Kern der vorliegenden Arbeit dar. Wir beginnen zunächst damit, eine Intuition für das Verfahren zu entwickeln. Sei dazu $G = (N, \Sigma, P, S)$ eine kontextfreie Grammatik und $w = a_0...a_{n-1} \in \Sigma^*$ ein Wort.

Aus Theorem 1 wissen wir, dass sich $C_w(L(G))$ durch eine kontextfreie Grammatik darstellen lässt. Damit lässt sich mit Hilfe des Earley Parsers überprüfen, ob eine wohldefinierte Correction ρ bezüglich w vorliegt. Wir erinnern uns, dass für eine Correction der Länge c insgesamt $c + 1$ Earley Sets berechnet werden.

In unserem Anwendungsfall wollen wir aber weder eine Grammatik für $C_w(L(G))$ bestimmen, noch überprüfen, ob ρ in $C_w(L(G))$ enthalten ist. Wir wollen eine konkrete Correction anhand von Nutzereingaben aus der Menge aller Corrections bestimmen. Die Idee hierzu ist nun, dass wir anstatt der Scanner Regel gewisse Auswahlregeln definieren, die dem Nutzer alle möglichen Edit Operationen vorschlagen. Anhand der Nutzereingabe wird dann genau eine Edit Operation ausgewählt und anschließend mit den Earley Items, die zu dieser Edit Operation passen, wie mit der Scanner Regel verfahren. Es ist leicht zu erkennen, dass sich dadurch alle Corrections der Länge c auswählen lassen.

Im nächsten Schritt überlegen wir uns, wie wir auch alle Corrections beliebiger Länge auswählen können. Dazu erlauben wir zunächst eine unbegrenzte Anzahl an Earley Sets und erstellen für jede ausgewählte Edit Operation ein neues Earley Set. Wir benötigen jetzt noch eine neue Bedingung, wann wir das Verfahren terminieren können.

Hierfür erinnern wir uns, dass im klassischen Earley Parser ein finales Earley Item der Form $[S \to \alpha\bullet, 0]$ in Q_{c+1} enthalten sein muss, wenn wir gegen ρ parsen. Da wir die Länge der Correction nicht vorher kennen, legen wir fest, dass jedes Zeichen im Eingabewort verarbeitet werden muss. Erst dann kann die Correction abgeschlossen werden, falls $[S \to \alpha\bullet, 0]$ im aktuellen Earley Set enthalten ist. Unter einer Verarbeitung verstehen wir hier die Anwendung der Scanner Regel, der Replacement Regel oder der Deletion Regel.

4.3.1 Auswahlregeln des nutzergesteuerten Earley Parser

Als nächstes definieren wir die angesprochenen Auswahlregeln formal. Wie zuvor verstehen wir unter der Verarbeitung eines Zeichens in einem Wort die Anwendung einer Read Operation, einer Replacement Operation oder einer Deletion Operation auf das Zeichen.

Definition 23. (Auswahlregeln) Sei $G = (N, \Sigma, P, S)$ eine kontextfreie Grammatik und $w = a_0...a_{n-1} \in \Sigma^*$ ein Wort. Sei zudem Q_j ein Earley Set für $j \geq 0$, $A \in N, \alpha\beta \in (N \cup \Sigma)^*, b \in \Sigma$ und $0 \leq i < n$. Wir definieren außerdem noch k als die Anzahl der verarbeiteten Zeichen in w. Dann sind die Auswahlregeln des nutzergesteuerten Earley Parser wie folgt definiert.

1. Für jedes Earley Item $[A \to \alpha \bullet b\beta, i] \in Q_j$, schlage dem Nutzer die Insert Operation $b^{\uparrow}$ vor. Falls der Nutzer diese Insert Operation wählt, füge $[A \to \alpha b \bullet \beta, i]$ zu Q_{j+1} hinzu

2. Für jedes Earley Item $[A \to \alpha \bullet b\beta, i] \in Q_j$ mit $a_j = b$ und $k < n$, schlage dem Nutzer die Read Operation $b^{\leftarrow}$ vor. Falls der Nutzer diese Read Operation wählt, füge $[A \to \alpha b \bullet \beta, i]$ zu Q_{j+1} hinzu.

3. Für jedes Earley Item $[A \to \alpha \bullet b\beta, i] \in Q_j$ mit $a_j \neq b$ und $k < n$, schlage dem Nutzer die Replacement Operation $b^{/a_j}$ vor. Falls der Nutzer diese Replacement Operation wählt, füge $[A \to \alpha b \bullet \beta, i]$ zu Q_{j+1} hinzu.

4. Für jedes Earley Item $[A \to \alpha \bullet b\beta, i] \in Q_j$ und $k < n$, schlage dem Nutzer die Deletion Operation $a_j^{\downarrow}$ vor. Falls der Nutzer diese Deletion Operation wählt, setze $Q_{j+1} = Q_j$.

Diese Regeln sammeln anhand von Earley Items der Form $[A \to \alpha \bullet b\beta, i] \in Q_j$ alle möglichen Terminale b, die durch den aktuellen Parsingzustand im Eingabewort erwartet werden. Durch die erste Regel hat der Nutzer nun die Möglichkeit das Zeichen b im Wort einzufügen. Wurden noch nicht alle Zeichen in w verarbeitet, so kann der Nutzer die Read Operation, die Replacement Operation oder die Deletion Operation auf das Zeichen a_j anwenden.

Falls $a_j = b$ gilt, wird dem Nutzer durch die zweite Regel vorgeschlagen, das Zeichen b an Position j im Eingabewort w zu lesen. Ansonsten wird ihm durch die dritte Regel vorgeschlagen, das Zeichen a_j durch das erwartete Terminal b zu ersetzen. Schließlich wird dem Nutzer durch die vierte Regel vorgeschlagen, das Zeichen a_j zu löschen. Dadurch wird die Verarbeitung des Terminals b auf das nächste Zeichen a_{j+1} in w verschoben, weshalb das Earley Set Q_j nach Q_{j+1} kopiert wird.

Im nächsten Schritt soll ein konkreter Algorithmus für das beschriebene Verfahren angegeben werden.

4.3.2 Algorithmus zum Verfahren

Der Earley Parser zur nutzergesteuerten Fehlerkorrektur ist in Algorithmus 7 angegeben. Eine Implementierung, die auf der Lark Bibliothek[1] in Python3 basiert, ist öffentlich zugänglich[2].

Als Eingabe erhält der Algorithmus eine kontextfreie Grammatik $G = (N, \Sigma, P, S)$ und ein Wort $w = a_0 ... a_{n-1} \in \Sigma^*$. Ausgabe ist eine konkrete Correction $\rho \in C_w(L(G))$. Zu Beginn wird in Zeile 1 die leere Correction $\rho = \langle\rangle$ initalisiert. Anschließnd wird das Earley Set Q_0 in Zeile 2 initialisiert. Für die erste Iteration muss nun die Menge `currentItems`

[1] `https://github.com/lark-parser/lark`
[2] `https://github.com/TobiasStd/ugep`

in Zeile 3 berechnet werden. Diese Menge enthält genau die Earley Items, aus denen alle möglichen ersten Edit Operationen einer Correction abgeleitet werden können.

Die Berechnung erfolgt durch die Parse Hilfsfunktion, für die ein Verfahren in Algorithmus 8 angegeben ist. Hier wird die Berechnung der Earley Sets durch die Predictor Regel und die Completer Regel durchgeführt. Wie beschrieben werden zudem alle Earley Items der Form $[A \to \alpha \bullet b\beta, i]$ in `currentItems` gesammelt.

Anschließend müssen die Auswahlregeln aus Definition 23 angewandt werden. Dazu ist ein Verfahren ChooseNextOperation in Algorithmus 9 angegeben. Die Items, die mit der gewählten Edit Operation des Nutzers übereinstimmen, werden dabei in der menge `filteredItems` gesammelt (siehe Zeile 6).Im nächsten Schritt wird in Zeile 14 auf Basis der gewählten Edit Operation und der Menge `filteredItems` das Earley Set Q_{j+1} und die Menge `currentItems` für die nächste Iteration bestimmt. Ein Verfahren dazu ist in der Hilfsfunktion GetNextItems in Algorithmus 10 angegeben.

Der beschriebene Prozess wird iterativ fortgesetzt, bis der Nutzer die Correction ρ abschließt. Voraussetzung hierfür ist, dass es sich bei ρ um eine wohldefinierte Correction handelt. Wir erinnern uns, dass dies der Fall ist, wenn ein finales Earley Item der From $[S \to \alpha\bullet, 0]$ in Q_j enthalten ist und alle Zeichen in w verarbeitet wurden. Dazu nutzen wir den Index k in Zeile 8. Ist diese Bedingung erfüllt, wird dem Nutzer zusätzlich vorgeschlagen, das Verfahren zu beenden (siehe Zeile 15 in Algorithmus 9).

Algorithm 7 Nutzergesteuerter Earley Parser

Eingabe: Kontextfreie Grammatik $G = (N, \Sigma, P, S)$, $w = a_0 a_1 ... a_{n-1} \in \Sigma^*$
Ausgabe: Correction $\rho \in C_w(L(G))$

1: $\rho = \langle\rangle$
2: $Q_0 \leftarrow \{[S \to \bullet\alpha, 0] \mid S \to \alpha \in P\}$
3: `currentItems` $\leftarrow$ Parse(Q_0)
4: $j \leftarrow 0$
5: **while** `True` **do**
6:     `editOp, filteredItems` $\leftarrow$ ChooseNextOperation(j, `currentItems`)
7:     **if** `editOp` $\in \{a_j^{\leftarrow}, a_j^{\downarrow}, b^{/a_j} \mid b \in \Sigma\}$ **then**
8:         $k \leftarrow k + 1$ ▷ Anzahl k der verarbeiteten Zeichen in w
9:     **if** `editOp` = `STOP` **then**
10:         `break`
11:     ρ.append(`editOp`)
12:     $j \leftarrow j + 1$ ▷ Anzahl j der bisherigen Edit Operationen
13:     $Q_j \leftarrow \emptyset$
14:     `currentItems` $\leftarrow$ GetNextItems($Q_0, ..., Q_j$, `editOp, filteredItems`)
15: `return` ρ

Algorithm 8 PARSE

Eingabe: Earley Sets $Q_0, ..., Q_j$ mit $j \geq 0$

Ausgabe: Menge an Earley Items `currentItems` von der Form $[A \rightarrow \alpha \bullet b\gamma, i]$

```
 1: currentItems← ∅
 2: Q'_j ← ∅
 3:
 4: while Q'_j ≠ Q_j do
 5:     Q'_j ← Q_j
 6:
 7:     if [A → α • Bβ, i] ∈ Q_j then
 8:         for all B → γ ∈ P do
 9:             Füge [B → •γ, j] zu Q_j hinzu.                 ▷ Predictor Regel
10:             if γ = bδ für b ∈ Σ then
11:                 Füge [B → •bδ, j] zu currentItems hinzu.
12:
13:     if Q_j enthält finales Item [A → α•, i] then
14:         for all [B → β • Aγ, k] ∈ Q_i do
15:             Füge [B → βA • γ, k] zu Q_j hinzu              ▷ Completer Regel
16:             if γ = bδ für b ∈ Σ then
17:                 Füge [B → βA • bδ, k] zu currentItems hinzu
18:
19: return currentItems
```

Algorithm 9 ChooseNextOperation

Eingabe: Ein Index $j \geq 0$; Menge `currentItems` an Earley Items $[A \rightarrow \alpha \bullet b\beta, i]$, die das Terminalsymbol b für die nächste edit Operation vorgeben.

Ausgabe: Die gewählte nächste Edit Operation `nextOperation`; Menge `filteredItems`, die die anhand der Nutzereingabe gefilterten Einträge aus `currentItems` enthält.

1: $\mathcal{E} \leftarrow \emptyset$ ▷ Menge möglicher nächster Edit-Operationen
2:
3: **if** $j < n$ **then** ▷ Deletion Regel
4: $\quad \mathcal{E} \leftarrow \mathcal{E} \cup \{a_j^{\downarrow}\}$
5:
6: **for all** $[A \rightarrow \alpha \bullet b\beta, i] \in$ `currentItems` **do**
7: $\quad \mathcal{E} \leftarrow \mathcal{E} \cup \{b^{\uparrow}\}$ ▷ Insertion Regel
8:
9: $\quad$ **if** $a_j = b$ und $j < n$ **then** ▷ Scanner Regel
10: $\quad\quad \mathcal{E} \leftarrow \mathcal{E} \cup \{b^{\leftarrow}\}$
11: $\quad$ **else if** $j < n$ **then** ▷ Replacement Regel
12: $\quad\quad \mathcal{E} \leftarrow \mathcal{E} \cup \{b^{/a_j}\}$
13:
14: **if** $[S \rightarrow \alpha\bullet, 0] \in Q_j$ und $j \geq n$ für $S \rightarrow \alpha \in P$ **then**
15: $\quad \mathcal{E} \leftarrow \mathcal{E} \cup \{\texttt{STOP}\}$ ▷ Nutzer soll Correction abschließen können
16:
17: `nextOp` $\leftarrow$ UserChoose($\mathcal{E}$) ▷ Tatsächliche Nutzereingabe
18:
19: **if** `nextOp` $= a_j^{\downarrow}$ **then** ▷ Filtern (Für Deletion wird Q_{j-1} kopiert)
20: $\quad$ `filteredItems` $\leftarrow$ `currentItems`
21: **else**
22: $\quad$ `filteredItems` $\leftarrow \{[A \rightarrow \alpha \bullet b\beta] \in$ `currentItems` $\mid$ `nextOp` $\in \{b^{\uparrow}, b^{\leftarrow}, b^{/a_j}\}\}$
23:
24: **return** `nextOp`, `filteredItems`

Algorithm 10 GetNextItems

Eingabe: Earley Sets $Q_0, ...Q_j$ für ein $j > 0$; Die gewählte Edit Operation `editOp`; Die Menge `filteredItems`, die die anhand der Nutzereinagbe gefilterten Items enthält.
Ausgabe: Menge an Earley Items `nextItems` von der Form $[A \to \alpha \bullet b\beta, i]$ für die nächste Iteration.

1: **if** `editOp` $= a^{\downarrow}_{j-1}$ **then** ▷ Kopie des Earley Sets Q_{j-1} für Deletion Regel
2: $Q_j \leftarrow Q_{j-1}$
3: **else** ▷ Füge alle Items zu Q_j, die der Nutzereingabe entsprechen
4:
5: **for all** $[A \to \alpha \bullet b\beta, i] \in$ `filteredItems` **do**
6: Füge $[A \to \alpha b \bullet \beta, i]$ zu Q_j hinzu
7:
8: `nextItems` $\leftarrow$ Parse$(Q_0, ..., Q_j)$
9: `return nextItems`

4.3.3 Beispiel zum Verfahren

Im diesem Abschnitt betrachten wir ein konkretes Beispiel zum Verfahren aus Algorithmus 7. Sei dazu die kontextfreie Grammatik G_{reg} aus Beispiel 1 und das Wort $w = ++$ gegeben. In Abbildung 12 sind die berechneten Earley Sets nach jeder Iteration visualisiert. Die in der Abbildung rot und grün markierten Earley Items entsprechend jeweils der Menge `currentItems` der entsprechenden Iteration. Das grün markierte Earley Item stimmt dabei mit der vom Nutzer gewählten Edit Operation überein und ist folglich in der Menge `filteredItems` enthalten.

Die vom Nutzer gewählte Edit Operation ist hier durch einen blauen Pfeil dargestellt. Die Anwendung dieser Edit Operation bewirkt, dass im nächsten Earley Set das blau markierte Earley Item übernommen wird. Dies entspricht im klassischen Earley Parser der Anwendung der Scanner Regel. Nachdem im Earley Set Q_5 das finale Item $[S \to S{+}C\bullet, 0]$ enthalten ist, schließt der Nutzer die Correction $\rho = \langle a^{/+}, +^{\leftarrow}, (^{\uparrow}b^{\uparrow})^{\uparrow} \rangle$ ab, wobei durch die Anwendung das Wort $\rho(w) = a + (b) \in L(G_{reg})$ resultiert.

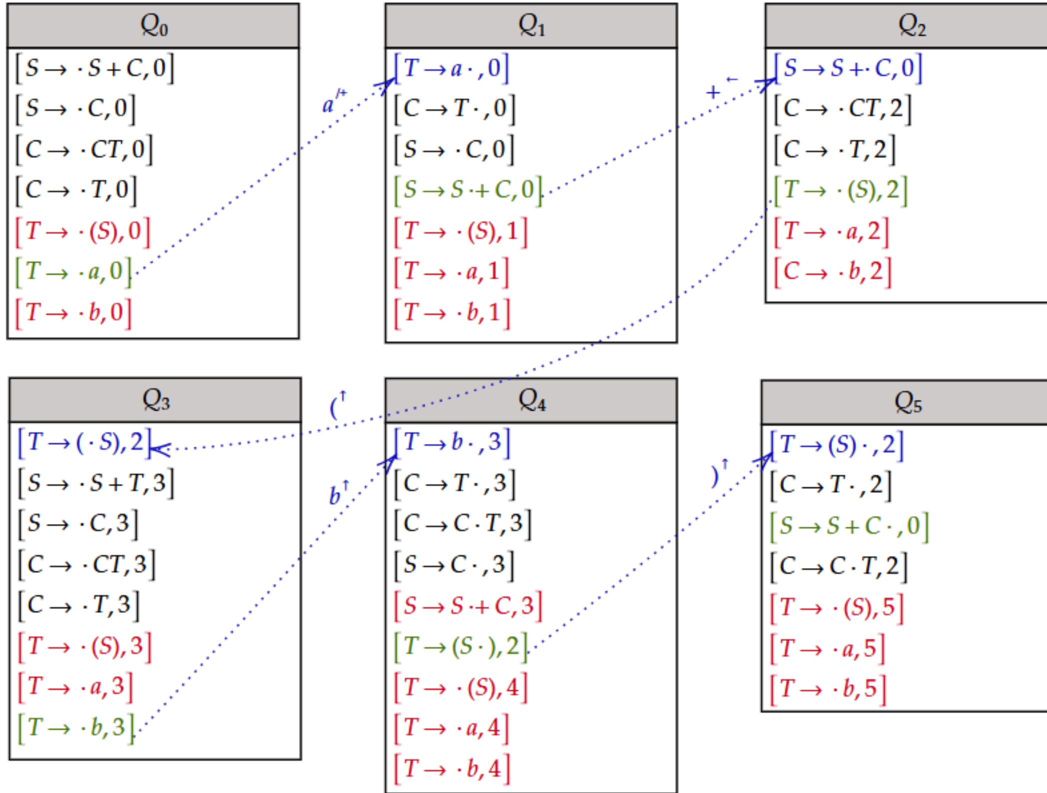

Abbildung 12: Berechnete Earley Sets aus Algorithmus 7 für die Grammatik G_{reg} aus Beispiel 1 und das Wort $w = ++$. Es resultiert die Correction $\rho = \langle a^{/+}, +^{\leftarrow}, (^{\uparrow} b^{\uparrow})^{\uparrow} \rangle$ und somit $\rho(w) = a + (b) \in L(G_{reg})$.

4.4 Korrektheit und Vollständigkeit

Wir beweisen jetzt, dass der entwickelte Algorithmus 7 zur nutzergesteuerten Fehlerkorrektur auf Basis des Earley Parsers korrekt arbeitet. Dazu nutzen wir, dass die Earley Sets wie vom klassischen Earley Parser [2] berechnet werden, wenn dieser gegen die Anwendung $\rho(w)$ einer Correction parsed.

Theorem 4. (Korrektheit des Verfahrens aus 7) Sei $G = (N, \Sigma, P, S)$ eine kontextfreie Grammatik, $w = a_0...a_{n-1} \in \Sigma^*$ ein Wort und $\rho \in C_w(L(G))$ eine Correction für w. Dann existiert eine Ausgabe des Algorithmus, welche genau der Correction ρ entspricht.

Beweis. Sei zunächst $c \geq n$ die Länge der Correction ρ. Wir bezeichnen dann τ_k als die k-te Edit Operation in ρ für $1 \leq k \leq c$. Seien zudem $A \in N$ und $\alpha\beta \in (N \cup \Sigma)^*$. Wir werden das Theorem per Induktion über k beweisen.

Induktionsanfang für $k = 1$:
Wir betrachten eine Fallunterscheidung über die Art der Edit Operation:

1. Es gilt $\tau_1 = a_0^{\downarrow}$: Nach Anwendung der vierten Auswahlregel aus Definition 23 können wir τ_1 als erste Nutzereingabe wählen.
2. Es gilt $\tau_1 \in \{b^{\leftarrow}, b^{\uparrow}, b^{/a_0} \mid b \in \Sigma\}$: Da ρ eine Correction für w darstellt, ist nach Anwendung der Predictor Regel und der Completer Regel, b ein Terminal, welches der Earley Parser als erstes Zeichen im Wort erwartet. Es folgt, dass ein Earley Item der Form $[A \to \alpha \bullet b\beta, 0]$ in Q_0 existiert. Nach Anwendung von einer der ersten drei Auswahlregeln aus Definition 23, können wir auch hier τ_1 als erste Nutzereingabe wählen.

Sei nun $1 < k \leq c$ beliebig.

Induktionsvoraussetzung für $k-1$: Die ersten $k-1$ Nutzereingaben entsprechen genau den ersten $k-1$ Edit Operationen $\tau_1, ..., \tau_{k-1}$ von ρ.

Induktionsschritt für k:
Für $\tau_k = a_j^{\downarrow}$ mit $0 \leq j \leq n$ können wir nach Anwendung der vierten Auswahlregel aus Definition 23 wieder τ_k als Nutzereingabe wählen. Wir betrachten also den Fall $\tau_k \in \{b^{\leftarrow}, b^{\uparrow}, b^{/a_j} \mid b \in \Sigma\}$.

Sei dazu $\rho(w) = b_0...b_i...b_m \in L(G)$ die Anwendung der Correction ρ auf w mit $0 \leq m \leq c$ und $0 \leq i \leq m$. Weiterhin sei b_i das Terminal, welches durch Anwendung der k-ten Edit Operation τ_k entsteht. Wir betrachten jetzt das Earley Set Q'_i, welches vom klassischen Earley Parser bei der Eingabe $\rho(w)$ bezüglich G berechnet wird.

Sei jetzt Q_k das k-te Earley Set, welches durch den Algorithmus 7 berechnet wird. Da die Nutzereingaben $\tau_1, ..., \tau_{k-1}$ nach Induktionsvoraussetzung den ersten $k-1$ Edit Operation von ρ entsprechen, gilt nun $Q_k = Q'_i$. Dies hat den Grund, dass sich die Auswahlregel aus Definition 23 analog zu der Scanner Regel des klassischen Earley Parser bei Eingabe $\rho(w)$ verhält. Die Predictor Regel und die Completer Regel sind zudem identisch.

Folglich ist in Q_k also auch ein Earley Item der Form $[A \to \alpha \bullet b\beta, l]$ für ein $0 \leq l \leq k$ enthalten. Nach Anwendung der ersten drei Auswahlregeln aus Definition 23 kann also τ_k als Nutzereingabe gewählt werden.

Gilt nun $k = c$, so folgt wie zuvor auch die Gleichheit $Q_k = Q'_m$. Somit ist ein finales Earley Item $[S \to \alpha\bullet, 0]$ in Q_k enthalten. Da die Folge der gewählten Nutzereingaben nun $\tau_0...\tau_k$ entspricht, kann das Verfahren nun terminiert werden und die Ausgabe entspricht genau der Correction ρ □

Nach Abschluss des Beweises zur Korrektheit wollen wir nun zeigen, dass jede Ausgabe des Algorithmus tatsächlich einer wohldefinierten Correction entspricht.

Theorem 5. (Vollständigkeit des Verfahrens aus 7) Sei $G = (N, \Sigma, P, S)$ eine kontextfreie Grammatik, $w = a_0...a_{n-1} \in \Sigma^*$ ein Wort und $\rho' = \langle \tau_1, ..., \tau_c \rangle$ die Ausgabe des Algorithmus. Dann existiert eine Correction $\rho \in C_w(L(G))$, die genau ρ' entspricht.

Beweis. Wir können dies nach demselben Prinzip wie aus dem Beweis zu Theorem 4 für die Korrektheit zeigen. Dazu kann erneut ein Induktionsbeweis geführt werden, diesmal über die k-te Nutzereingabe. Analog kann argumentiert werden, dass τ_1 der ersten Edit Operation einer Correction ρ entspricht, und dass $\tau_1, ...\tau_{k-1}$ zu einer Correction erweitern werden muss, wenn die ersten $k-1$ Nutzereingaben bereits den ersten $k-1$ Edit Operationen einer Correction entsprechen. □

4.5 Analyse

Im nächsten Schritt wird die Worst-Case Laufzeit des Algorithmus 7 zur nutzergesteuerten Fehlerkorrektur analysiert.

Sei dazu $G = (N, \Sigma, P, S)$ eine kontextfreie Grammatik, $w = a_0...a_{n-1} \in \Sigma^*$ ein Wort und $\rho = \langle x_0, v_0, ..., x_{n-1}, v_{n-1}, x_n \rangle \in C_w(L(G))$ eine Correction mit
$v_i \in \{a_i^{\leftarrow}, a_i^{\downarrow}, a_i^{/b} \mid a, b \in \Sigma\}$ und $x_i \in \{a^{\uparrow} \mid a \in \Sigma\}^*$ für $0 \leq i \leq n$. Sei zudem Q_j das berechnete Earley Set für den Earley Parsers aus [2] mit $0 \leq j \leq n$.

Theorem 6. Wenn die Correction ρ der Ausgabe entspricht und k die Anzahl der Insertion Operation in ρ ist, dann liegt die Worst Case Laufzeit für das Verfahren aus Algorithmus 7 in $\mathcal{O}((n+k) \cdot n^2)$.

Beweis. Sei also ρ die Ausgabe des Algorithmus und k die Anzahl der Insert Operationen in der Correction. Wir werden zunächst die Laufzeit für eine Iteration der `while`-Schleife in Zeile 5 bestimmen.

Nach [2] wird für die Berechnung eines Earley Sets Q_j für ein $0 \leq j \leq n$ eine Worst-Case Laufzeit von $\mathcal{O}(n^2)$ benötigt. Diese wird maßgeblich durch die Completer Regel bestimmt. Daraus folgt direkt, dass für das PARSE Verfahren aus Algorithmus 8 ebenfalls eine Laufzeit von $\mathcal{O}(n^2)$ anfällt, da lediglich die Completer Regel und die Predictor Regel des klassischen Earley Parsers angewandt werden. Es ist leicht zu sehen, dass sich die hinzugefügten Regeln aus Definiton 23 in konstanter Zeit berechnen lassen. Folglich ist also auch die Laufzeit für eine Iteration durch $\mathcal{O}(n^2)$ abzuschätzen.

In jeder Iteration wird ρ genau eine Edit Operation hinzugefügt. Die Anzahl der Iterationen wird also durch die Anzahl der Edit Operationen bestimmt. Diese ergibt sich aus den n nötigen Verarbeitungen der Zeichen in w (Read Operation, Replacement Operation oder Deletion Operation), um die Corrections abzuschließen und aus den k Insert Operationen. Folglich gibt es also $(n+k)$ Iterationen, weshalb insgesamt eine Worst-Case Laufzeit von $O((n+k) \cdot n^2)$ resultiert. □

5 Fazit und Ausblick

Zum Abschluss möchten wir noch ein Fazit der vorliegenden Arbeit ziehen und in Kapitel 5.1 darauf beschreiben, welche Resultate sich im Rahmen dieser Bachelorarbeit ergeben haben. Anschließend werden wir in Kapitel 5.2 die erste Forschungsfrage aus Kapitel 1 beantworten. Weiterhin wird darauf eingegangen, warum sich eine zweite Forschungsfrage ergeben hat und das entwickelte Verfahren aus Kapitel 3 verworfen wurde. Abschließend gehen wir in Kapitel 5.3 auf mögliche Erweiterungen des entwickelten Verfahrens aus Kapitel 4 ein.

5.1 Resultate

Aus Kapitel 2 wissen wir, dass sich für eine kontextfreie Sprache L und ein (möglicherweise fehlerhaftes) Wort w alle Corrections effizient in der Datenstruktur des SPPF speichern lassen. Darauf aufbauend wurde betrachtet, wie aus der Menge aller Corrections in einem interaktiven Verfahren eine konkrete Correction bestimmt werden kann. Hierzu wurde in Kapitel 3 ein Verfahren entwickelt, das eine nutzergesteuerte Traversierung des SPPF ermöglicht und anhand von Nutzereingaben genau eine Correction berechnet.

Das zentrale Resultat dieser Arbeit wurde schließlich in Kapitel 4 vorgestellt. Dort konnte gezeigt werden, dass die nutzergesteuerte Fehlerkorrektur ohne die Konstruktion des SPPF realisiert werden kann. Stattdessen wurde ein optimiertes Verfahren auf Basis des klassischen Earley Parsers aus [2] entwickelt. Hierzu wurden anstelle der Scanner Regel Auswahlregeln bezüglich der erwarteten Zeichen im Eingabwort verwendet. Anhand dieser Regeln ist es möglich, schon während des Parsings interaktiv anhand von Nutzereingaben eine Correction zu bestimmen.

5.2 Fazit

Die erste Forschungsfrage dieser Arbeit lautete: *Wie lässt sich aus der Menge aller Corrections in einem interaktiven Verfahren eine konkrete Correction bestimmen?*

Durch das Verfahren zur nutzergesteuerten Traversierung im SPPF konnten wir zeigen, dass sich durch Nutzereingaben aus der Menge aller Corrections eine konkrete Correction bestimmen lässt. Da in diesem Verfahren der SPPF vor der ersten Nutzereingabe berechnet werden muss, wurde versucht mögliche Optimierungen zu erarbeiten. Im Zuge der Formalisierungen diesbezüglich zeigte sich, dass die nutzergesteuerte Fehlerkorrektur vollständig auf Basis des Earley Parsers realisiert werden kann. Insbesondere ist also auch die erarbeitete Berechnung und Traversierung des SPPF redundant. Diese Beobachtung wird durch die Arbeit von Klint und Visser [6] gestützt, aus der die Motivation stammt, die Menge möglicher Corrections möglichst früh im Parsingprozess zu reduzieren. Folglich ergab sich eine zweite Forschungsfrage:

Wie kann das Verfahren zur nutzergesteuerten Fehlerkorrektur optimiert werden und möglichst früh in den Parsingprozess integriert werden?

Die Beantwortung dieser Frage erfolgte durch das in Kapitel 4 vorgestellte Verfahren zur nutzergesteuerten Fehlerkorrektur mit dem Earley Parser. Hier konnte gezeigt werden, dass sich sowohl der konzeptionelle Aufwand reduzieren lässt, als auch die Anzahl der benötigten Berechnungsschritte.

5.3 Ausblick

Insgesamt bietet die Entwicklung des vorgestellten Verfahrens aus Kapitel 4 eine Grundlage, um Lösungsansätze für verschiedene praktische Anwendungen der Fehlerkorrektur für kontextfreie Sprachen aufzugreifen.

Zum Beispiel könnten interaktive Lernsysteme wie [1] durch die Implementierung des Algorithmus um eine Nutzergesteuerte Fehlerkorrektur erweitert werden.

Eine weitere interessante Fortführung der Arbeit wäre die Ersetzung der Nutzereingaben durch ein vortrainiertes Large Language Model. Ein solches Modell könnte dazu genutzt werden, automatisch sinnvolle Auswahlentscheidungen zu treffen und damit für einen konkreten Anwendungsfall geeignete Corrections vorzuschlagen.

Darüber hinaus wäre es denkbar eine empirische Evaluation des Verfahrens hinsichtlich der Benutzerfreundlichkeit oder der Qualität der Ergebnisse für verschiedene praktische Anwendungen durchzuführen.

Literatur

[1] Marit Kastaun, Monique Meier, Norbert Hundeshagen, Martin Lange: *ProfiLL–Professionalisierung durch intelligente Lehr-Lernsysteme.* In: Bildung, Schule, Digitalisierung, 2020.

[2] Jay Earley: *An efficient context-free parsing algorithm.* In: *Communications of the ACM* 13.2, S. 94-102, 1970, ACM New York, NY, USA. `https://dl.acm.org/doi/abs/10.1145/362007.362035`

[3] Alfred V. Aho, Thomas G. Peterson: *A minimum distance error-correcting parser for context-free languages.* In: *SIAM Journal on Computing* 1.4, S. 305–312, 1972. `https://epubs.siam.org/doi/abs/10.1137/0201022`

[4] Maurice Herwig, Norbert Hundeshagen, Martin Lange: *An Earley-Based Universal Error-Correcting Parser.* In: International Conference on Implementation and Application of Automata, S.208–222, 2025, Springer. `https://link.springer.com/chapter/10.1007/978-3-032-02602-6_15`

[5] Bram van der Sanden: *Parse Forest Disambiguation.* In: Master Thesis, 2014, Eindhoven University of Technology. `https://pure.tue.nl/ws/portalfiles/portal/46998704/784691-1.pdf`

[6] Paul Klint, Eelco Visser: *Using filters for the disambiguation of context-free grammars* In: *Proc. ASMICS Workshop on Parsing Theory*, S. 1–20, 1994.

[7] Maurice Herwig: *Using Shared Packed Parse Forests to compute all Minimal Corrections.*, In: Master Thesis, Universität Kassel, Theoretische Informatik / Formale Methoden, 2024.

[8] Martin Lange, Norbert Hundeshagen, Marco Sälzer: In: *Formale Sprachen und Logik*, 2024.

[9] Christoph Meinel, Martin Mundhenk: In: *Mathematische Grundlagen der Informatik.* Springer, 2011.

[10] John Aycock und Nigel Horspool: *Directly-executable Earley parsing.* In: *International Conference on Compiler Construction*, S. 229–243, Springer, 2001 `https://link.springer.com/chapter/10.1007/3-540-45306-7_16`

[11] Alfred Aho und Thomas Peterson: *A minimum distance error-correcting parser for context-free languages.* In: *SIAM Journal on Computing*, V. 1, N. 4, S. 305–312, SIAM, 1972 `https://epubs.siam.org/doi/10.1137/0201022`

[12] Florian Bruse, Martin Lange: *Computing all minimal ways to reach a context-free language.* In: International Conference on Reachability Problems, S.38–53, Springer, 2024. `https://link.springer.com/chapter/10.1007/978-3-031-72621-7_4`