syntaktische Mehrdeutigkeiten beim Parsen

Lennart Protte

RWTH Aachen University lennart.protte@rwth-aachen.de

Zusammenfassung. Syntaktische Mehrdeutigkeiten stellen eine zentrale Herausforderung beim Parsen von Programmiersprachen dar. Die vorliegende Arbeit untersucht die Auswirkungen solcher Mehrdeutigkeiten und betrachtet dabei Ansätze zu ihrer Erkennung, Vermeidung und Auflösung. Dabei werden sowohl Vorteile als auch Nachteile von mehrdeutigen Sprachen und Grammatiken diskutiert und die Auswirkungen dieser auf das Parsen beleuchtet. Um diese Problematik zu lösen, wurden Strategien und Algorithmen zur Erkennung und behebung syntaktischer Mehrdeutigkeiten diskutiert. Es zeigt sich, dass die Vermeidung von Mehrdeutigkeiten in Grammatiken ein einfacheres Parsen ermöglicht, allerdings nicht immer sinnvoll ist. Die Arbeit stellt praxistaugliche Methoden zur Erkennung und Auflösung von Mehrdeutigkeiten vor und zeigt wie Parser-Techniken dieser Problematik begegnen.
Die Abwägung zwischen der Einfachheit der Grammatik und der Lesbarkeit der Sprache lässt erkennen, dass eindeutige Sprachkonzepte die Komplexität des Parsens reduzieren können, aber Mehrdeutigkeiten dennoch nicht immer ein Hindernis darstellen müssen.

Schlüsselwörter: Parserbau, syntaktische Mehrdeutigkeiten, Sprachkonzeption

1 Einführung

Bei der Handhabung von Mehrdeutigkeiten, bietet es sich an, die Grammatik der Sprache zu betrachten. Dabei stellt sich die Frage nach der Definition von Mehrdeutigkeit und nach der Bedeutung von Mehrdeutigkeiten für die syntaktische Analyse im Parserbau.

Die Frage nach der Definition einer mehrdeutigen Grammatik ist simpel zu beantworten. Eine Grammatik ist genau dann mehrdeutig, wenn es mindestens eine Eingabe gibt, die mehrere Parse-Bäume besitzt. Daher ist eine Grammatik mehrdeutig, wenn für ein Wort der Grammatik mehrere Linksableitungen existieren[13]. Bei Mehrdeutigkeiten lässt sich zwischen semantischer und syntaktischer Mehrdeutigkeit unterscheiden. Semantische Mehrdeutigkeiten treten auf, wenn ein Token je nach Operation mehrere Bedeutungen haben kann. Ein Beispiel hierfür ist die semantische Mehrdeutigkeit des Tokens „+“ in der Programmiersprache Java.

In Java kann das Token „+“ sowohl als arithmetischer Operator, als auch als Konkatenationsoperator von Strings verstanden werden. Solche Mehrdeutigkeiten werden in der semantischen Analyse behandelt und im Weiteren nicht näher betrachtet.

Syntaktische Mehrdeutigkeiten hingegen entstehen durch die Struktur der Grammatik selbst. Das wohl bekannteste Beispiel dafür ist das „Dangling else“-Problem, welches in vielen Compilern gängiger Programmiersprachen auftritt[1].

Ein beispielhafter Ausschnitt aus den Produktionsregeln einer Grammatik, die den Syntax einer Wenn-Dann-Abfrage beschreiben, ist der folgende:

$$
\begin{aligned}
P = \{ & \\
& \textit{ausdruck} \rightarrow \textbf{if}\ \textit{bedingung}\ \textbf{then}\ \textit{ausdruck} \\
& \textit{ausdruck} \rightarrow \textbf{if}\ \textit{bedingung}\ \textbf{then}\ \textit{ausdruck}\ \textbf{else}\ \textit{ausdruck} \\
\} &
\end{aligned}
$$

Wie zu erkennen ist, ist diese Grammatik mehrdeutig, da unklar ist, welcher „else“-Block zu welchem „if“-Block gehört. Im Folgenden sind zwei unterschiedliche Parse-Bäume des Wortes „**if** *bedingung* **then if** *bedingung* **then** *ausdruck* **else** *ausdruck*“ dargestellt.

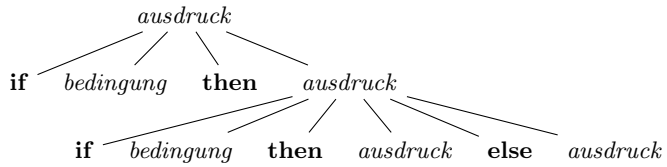

Abb. 1. Hier wird der else-Block der inneren if-Abfrage zugeordnet.

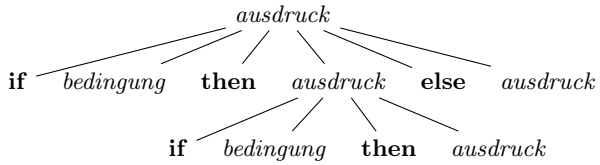

Abb. 2. In diesem Parse-Baum wird der else-Block der äußeren if-Abfrage zugeordnet.

Wie hier zu erkennen ist, können syntaktische Mehrdeutigkeiten zu unterschiedlichen Interpretationen desselben Quellcodes führen. Die Problematik dessen wird im Folgenden behandelt.

2 Problemstellung

Die Problematik von Mehrdeutigkeiten in Grammatiken besteht darin, dass ein Parser keinen eindeutigen Parse-Baum für ein zu parsendes Programm erzeugen kann. Wenn Mehrdeutigkeiten bei der Konzeption einer Programmiersprache nicht vermieden werden können, muss der Parser in der Lage sein, diese zu parsen. Um die korrekte Struktur eines mehrdeutigen Programmes zu ermitteln, benötigt ein Parser entweder zusätzliche Informationen oder eine festgelegte Vorgehensweise, um diese Mehrdeutigkeiten aufzulösen. Mehrdeutigkeiten die nicht aufgelöst werden können, da die Sprache inhärent mehrdeutig ist[5], werden an den Programmierer weitergegeben und erschweren die Entwicklung in der jeweiligen Sprache.

```
class A {public: void func() {}};
class B {public: void func() {}};
class C: public A, public B {};
int main() {
    C c;
    c.func();
    return 0;
}
```

```
class A {public: void func() {}};
class B {public: void func() {}};
class C: public A, public B {};
int main() {
    C c;
    c.A::func();
    return 0;
}
```

Abb. 3. Mehrdeutigkeit in der Mehrfachvererbung in C++

An diesem Beispiel wird deutlich, wieso es nicht sinnvoll ist, Mehrdeutigkeiten an den Entwickler weiterzugeben. Der Code auf der linken Seite ist mehrdeutig und wird auch als fehlerhaft erkannt. Die Mehrdeutigkeit kommt zustande, da die Klasse C von den Klassen A und B erbt, die beide eine Methode mit dem Namen „func“ besitzen. Beim Aufruf der Methode „func“ auf einem Objekt der Klasse C ist unklar, ob die Methode von der Oberklasse A oder B aufgerufen werden soll. Diese Mehrfachvererbung führt dazu, dass sich nun der Entwickler mit dieser Mehrdeutigkeit auseinandersetzen muss. Dies passiert nur, weil die Sprache C++ keine explizite Angabe der Klasse bei einem Methodenaufruf verlangt. In diesem Fall lässt sich die Mehrdeutigkeit durch eine explizite Angabe der Klasse lösen, wie im Code auf der rechten Seite zu sehen ist. Wenn Mehrdeutigkeiten nicht vermieden werden können, kann der Ansatz versucht werden, diese zu erkennen und aufzulösen. Dies kann jedoch Einfluss auf die Laufzeit und Komplexität eines Parsers nehmen und bedarf daher einer guten Abwägung[3]. Das Parsen von mehrdeutige Sprachen ist komplexer und weniger effizienter als bei eindeutigen Sprachen, da für eine korrekte Lösung alle möglichen Ableitungen betrachtet werden müssen[14]. Eine weitere Problematik ist, dass es theoretisch nicht möglich ist zu entscheiden, ob eine gegebene Grammatik mehrdeutig ist oder nicht[12]. Dieses Problem, auch bekannt als Post'sche Korrespondenz-

problem, spielt vor allem in der Konzeption neuer Programmiersprachen eine Rolle[10].

3 Vermeidung von Mehrdeutigkeiten

Da Mehrdeutigkeiten in Grammatiken sowohl in der Entwicklung des Parsers, als auch in der Anwendung der Sprache hinderlich sein können, ist es sinnvoll Mehrdeutigkeiten von vornherein zu vermeiden. Um eine Sprache eindeutig zu gestalten, muss die Grammatik der Sprache so konzipiert werden, dass es für jedes Wort der Sprache nur einen Parse-Baum gibt. So können wir Beispielsweise die obige Grammatik des „Dangling else“-Problems auch wie folgt realisieren:

$$
\begin{aligned}
P = \{ & \\
& ausdruck \rightarrow ausdruck_{auf} | ausdruck_{zu} \\
& ausdruck_{auf} \rightarrow \textbf{if } bedingung \textbf{ then } ausdruck \\
& \qquad\qquad | \textbf{if } bedingung \textbf{ then } ausdruck_{zu} \textbf{ else } offener\_ausdruck \\
& ausdruck_{zu} \rightarrow \textbf{if } bedingung \textbf{ then } ausdruck_{zu} \textbf{ else } ausdruck_{zu} \\
\} &
\end{aligned}
$$

Dadurch, dass nun zwischen offenen und geschlossenen „if“-Blöcken unterschieden wird, kann diese Grammatik eindeutig von einem LR-Parser verarbeitet werden[1]. Allerdings kann es auch Nachteile haben, eine Sprache eindeutig zu gestalten[14]. So ist die Grammatik einer mehrdeutigen Sprache oft einfacher, kürzer und intuitiver als die einer eindeutigen Sprache und damit für den Entwickler der Sprache einfacher zu entwerfen. Auch können die Eindeutigkeitsanforderungen an eine Sprache schnell Boilerplate-Code verursachen.

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Abb. 4. Beispiel für Boilerplate-Code in Java

Dieses Beispiel der Sprache Java zeigt, dass zu große Anforderungen an die Eindeutigkeit, hier in Form von Sichtbarkeitsmodifikatoren und Typenangaben, schnell zu verbosem Code führen können. Es ist daher eine Abwägung zwischen der Einfachheit des Parsers und der Komplexität und Lesbarkeit der Sprache zu treffen.

4 Erkennung von Mehrdeutigkeiten

Um eine eindeutige Grammatik für eine Sprache zu entwerfen, ist es notwendig diese Grammatik auf Mehrdeutigkeiten zu überprüfen. Wie bereits bemerkt, ist zu beachten, dass die Überprüfung, ob eine Grammatik mehrdeutig ist oder nicht, unentscheidbar ist[5].

Daher wird auf Suchalgorithmen zurückgegriffen, die durch die Ableitungen der Grammatik traversieren und dabei auf Mehrdeutigkeiten prüfen. Diese bietet zwar keine sicheren Ergebnisse, aber immerhin eine gute Annäherung. Statistisch gesehen ist es wahrscheinlicher eine Mehrdeutigkeiten bei einer Breitensuche zu finden, als bei einer Tiefensuche. Der „dynamic1"-Algorithmus[12] ist ein nicht-deterministischer Algorithmus, welcher zufällige Ableitungsalternativen wählt und dabei bis zu einer festgelegten Tiefe traversiert. Wenn der Algorithmus diese Tiefe erreicht hat, bevorzugt er Alternativen, welche zuvor noch gar nicht oder verhältnismäßig selten gewählt oder weniger bis keine nicht-Terminalsymbole enthalten. In einem Experiment[12] zeigte sich, dass dieser Algorithmus eine Verbesserung zu ACLA, AMBER und AmbiDexter darstellt. Zwar übersieht er teilweise Mehrdeutigkeiten, die von den anderen Algorithmen gefunden wurden, aber insgesamt findet er zum einen mehr und zum anderen deutlich tiefer verschachtelte Mehrdeutigkeiten.

5 Auflösung von Mehrdeutigkeiten

Sollte eine Grammatik mehrdeutig sein, so muss ein Parser in der Lage sein, diese Mehrdeutigkeiten aufzulösen. Um dies zu erreichen, können zusätzliche Regeln, wie beispielsweise Assoziativität und Vorrangregeln, für Operatoren festgelegt werden, unter welchen die Grammatik dann in eine eindeutige Form gebracht werden kann[2]. Dazu eignet sich ein Algorithmus[1], welcher die gegebenen Regeln auf die Grammatik anwendet und sie so in eine eindeutige Form transformiert. Der Algorithmus arbeitet mit sogenannten „Mustern zur Beseitigung von Mehrdeutigkeiten". Ein Muster ist ein vier-Tupel $(S, \alpha \circ \beta, \gamma)$, mit S als das Startsymbol der Produktion, $\alpha \circ \beta$ sind zwei Nichtterminale aus G mit $\circ$ als Operator und γ ist die gewählte Ableitung, wobei $\bullet$ markiert, ob α oder β abgeleitet wird. Mithilfe von diesen Mustern können Tabellen[2] für unsere zwei binären Operatoren $+$ und $*$ aufgestellt werden:

	$E ::= E\alpha_2 E$
$E ::= E\alpha_1 E$	$(E, \bullet E\alpha_1 E, E\alpha_2 E)$
	$(E, E\alpha_1 \bullet E, E\alpha_2 E)$

Abb. 5. Tabelle für Vorrangsregeln

Diese Tabelle zeigt, dass der Operator α_1 Vorrang vor dem Operator α_2 hat, daher immer zuerst abgeleitet wird.

	$E ::= E\alpha_1 E$	$E ::= E\alpha_2 E$
$E ::= E\alpha_1 E$	$(E, E\alpha_1 \bullet E, E\alpha_1 E)$	$(E, E\alpha_1 \bullet E, E\alpha_2 E)$
$E ::= E\alpha_2 E$	$(E, E\alpha_2 \bullet E, E\alpha_1 E)$	$(E, E\alpha_2 \bullet E, E\alpha_2 E)$

Abb. 6. Tabelle die links-Assotiativität

Hier ist die Assoziativität festgelegt. Wie zu erkennen ist, wird in jeder möglichen Kombination beider Operatoren immer erst das rechte Nichtterminal abgeleitet. Im Weiteren betrachten wir einen beispielhaften Durchlauf des Algorithmus anhand einer gegebenen Grammatik. Angenommen wir haben folgende Grammatik für einfache arithmetische Ausdrücke mit dazu gegebener Assoziativität und Vorrang:

$$
\begin{aligned}
&G(N, T, P, S)\\
&N = \{E\}\\
&T = \{+, *, \mathbf{zahl}\}\\
&S = \{E\}\\
&P = \{\\
&\qquad E \to E + E | E * E | \mathbf{zahl}\\
&\}
\end{aligned}
$$

Operator	Assoziativität	Priorität
$+$	links	1
$*$	links	2

Abb. 7. mehrdeutige Grammatik mit Operator-Assoziativitäten und Vorangregeln

Der Algorithmus ermittelt nun die Muster zur Beseitigung von Mehrdeutigkeiten und kommt dabei zu folgendem Ergebnis:

$$
\begin{aligned}
&(E, E * \bullet E, E * E)\\
&(E, \bullet E * E, E + E)\\
&(E, E + \bullet E, E * E)\\
&(E, \bullet E + E, E + E)
\end{aligned}
$$

$$
\begin{aligned}
&G(N, T, P, S)\\
&N = \{E, E_1, E_2, E_3, E_4\}\\
&T = \{+, *, \mathbf{zahl}\}\\
&S = \{E\}\\
&P = \{\\
&\qquad E \to E_1 * E_2 | E_3 + E_4 | \mathbf{zahl}\\
&\}
\end{aligned}
$$

Abb. 8. Anwenden der Muster auf die gegebene Grammatik

Wie hier zu erkennen ist, wurde im ersten Schritt für jedes der Muster ein neues Nichtterminal hinzugefügt. Der Algorithmus kopiert die Produktion der Grammatik nun für jedes neu eingeführte Nichtterminal und wendet dabei jeweils

eines Muster auf die Produktion an. In einem weiteren Schritt wird der Prozess dann für verschachtelte Fälle wiederholt.

$G(N, T, P, S)$
$N = \{E, E_1, E_2, E_3, E_4\}$
$T = \{+, *, \mathbf{num}\}$
$S = \{E\}$
$P = \{$
$E \to E_1 * E_2 | E_3 + E_4 | \mathbf{num}$
$E_1 \to E_1 * E_2 | \mathbf{num}$
$E_2 \to E_3 + E_4 | \mathbf{num}$
$E_3 \to E_1 * E_2 | \mathbf{num}$
$E_4 \to E_3 + E_4 | \mathbf{num}$
$\}$

$G(N, T, P, S)$
$N = \{E, E_1, E_2, E_3, E_4, E_5\}$
$T = \{+, *, \mathbf{num}\}$
$S = \{E\}$
$P = \{$
$E \to E_1 * E_2 | E_3 + E_4 | \mathbf{num}$
$E_1 \to E_1 * E_5 | \mathbf{num}$
$E_2 \to E_3 + E_4 | \mathbf{num}$
$E_3 \to E_5 * E_2 | \mathbf{num}$
$E_4 \to E_3 + E_4 | \mathbf{num}$
$E_5 \to \mathbf{num}$
$\}$

Abb. 9. Erstellung neuer Produktionen und prüfen von Verschachtelungen

Hier wurde der verschachtelte Fall aufgelöst, wo die Multiplikation über einen Zwischenschritt von rechts aufgebaut werden konnte. Eine solche Grammatik kann dann effizient und in linearer Zeit von einem LR-Parser verarbeitet werden.

Eine weitere Möglichkeit Mehrdeutigkeiten in einer gegebenen Grammatik aufzulösen ist die Umformung der Grammatik in die Chomsky-Normalform. Bei der Umformung in die Chomsky-Normalform können Mehrdeutigkeiten aufgelöst werden, dies muss aber nicht passieren[8]. Nach der Umformung kann daher keine Aussage über die Mehrdeutigkeit der Grammatik getroffen werden. Dennoch kann eine Umformung in die Chomsky-Normalform sinnvoll sein, besonders da die Parse-Bäume, die aus einer Grammatik in der Chomsky-Normalform resultieren, immer Binärbäume sind[13].

[1] Vasudevan, N., Tratt, L. (2013). Safe Specification of Operator Precedence Rules Seite 149 - 150
[2] Vasudevan, N., Tratt, L. (2013). Safe Specification of Operator Precedence Rules. Seite 147

6 Parsen von Mehrdeutigkeiten

Zum Parsen von Mehrdeutigkeiten benötigt es teilweise komplexere Parser-Techniken als die klassischen LL- und LR-Parser[11]. Um die richtige aus den möglichen Ableitungen zu wählen, muss ein Parser in der Lage sein, alle möglichen Ableitungen zu finden. Eine Möglichkeit Mehrdeutigkeiten zu parsen sind Lookahead-Parser. Lookaheads sind eine Technik, bei der der Parser nicht nur das aktuelle Token betrachtet, wie es bei einem LL- oder LR-Parser der Fall ist, sondern auch nachfolgende Tokens. Beispielsweise verwendet der ANTLR-Parser-Generator Lookaheads, für beliebig viele nachfolgende Tokens[9]. Eine weitere Möglichkeit Mehrdeutigkeiten zu parsen sind Chart-Parser. Eine Variante davon ist der CYK-Parser[4], welcher auf dem Cocke-Younger-Kasami-Algorithmus basiert. Dieser benötigt eine Grammatik in der Chomsky-Normalform. Wir betrachten beispielhaft unsere mehrdeutige „Dangline-Else"-Grammatik in der Chomsky-Normalform. Für das Wort „**if** *bedingung* **then if** *bedingung* **then** *ausdruck* **else** *ausdruck*", stellt der CYK-Parser folgende Tabelle auf:

$$
\begin{aligned}
P = \{ & \\
& S \rightarrow ausdruck|CA|DA \\
& A \rightarrow ausdruck|CA|DA \\
& B \rightarrow bedingung \\
& C \rightarrow FE \\
& D \rightarrow HG \\
& E \rightarrow \textbf{then} \\
& F \rightarrow IB \\
& G \rightarrow \textbf{else} \\
& H \rightarrow CA \\
& I \rightarrow \textbf{if} \\
\} &
\end{aligned}
$$

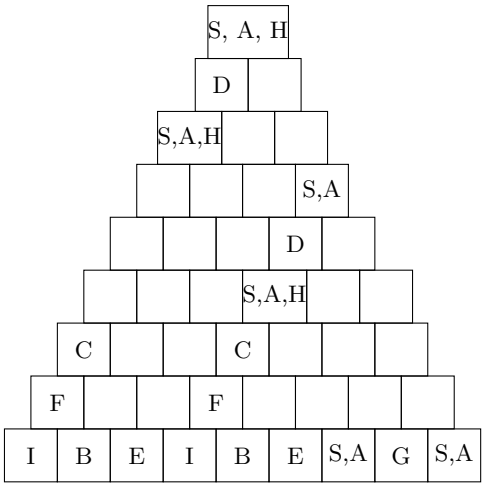

Abb. 10. CYK-Parser für mehrdeutige Grammatik

Eine wichtige Anmerkung zu der hier gezeigten Grammatik ist, dass zur besseren Darstellung der Tabelle, die Nichtterminale *ausdruck* und *bedingung* hier als Terminalsymbole in der Grammatik aufgeführt sind. Wie an diesem Ergebnis zu erkennen ist, findet der CYK-Parser hier alle mögliche Ableitungen für das gegebene Wort. Der Parser kann nun anhand festgelegter Regeln eindeutig entscheiden, welche dieser Ableitungen die richtige ist.

7 Fazit

Abschließend lässt sich festhalten, dass Mehrdeutigkeiten in Grammatiken von Programmiersprachen eine Herausforderung beim Parsen darstellen. Durch die Vermeidung von Mehrdeutigkeiten in der Sprache können einfachere und effizientere Parser verwendet werden[3]. Neuere Programmiersprachen wie Swift[7] oder C# vermeiden Mehrdeutigkeiten und können daher effizienter, beispielsweise mit einfachen Lookahead-Parsern, geparst werden. Wenn zudem vermieden werden kann, dass Mehrdeutigkeiten an den Entwickler weitergegeben werden, erleichtert es zusätzlich die Entwicklung in der jeweiligen Sprache. Auf der anderen Seite kann die Vermeidung von Mehrdeutigkeiten zu Boilerplate-Code führen. Die Tatsache, dass viele gängige Programmiersprachen wie C, Java oder Python Mehrdeutigkeiten aufweisen, zeigt, dass Mehrdeutigkeiten in Programmiersprachen praxistauglich sind. Ein weiteres Beispiel für die Eignung von mehrdeutiger Sprache im Programmieren ist AppleScript[6], welches eine Syntax ähnlich der natürlichen Sprache besitzt und damit besonders einsteigerfreundlich ist. Die Tatsache aber, dass mehrdeutiger Code das Programmieren erschweren kann und dass Parser für mehrdeutige Sprachen weniger effizient sind[3], zeigt, dass bei der Konzeption einer neuen Programmiersprache darauf geachtet werden sollte, Mehrdeutigkeiten grundsätzlich zu vermeiden und nicht an den Entwickler weiterzugeben.

Literatur

1. Abrahams, P.W.: A final solution to the dangling else of algol 60 and related languages. Commun. ACM **9**(9), 679–682 (sep 1966). https://doi.org/10.1145/365813.365821, https://doi.org/10.1145/365813.365821
2. Afroozeh, A., van den Brand, M., Johnstone, A., Scott, E., Vinju, J.: Safe specification of operator precedence rules. In: Erwig, M., Paige, R.F., Van Wyk, E. (eds.) Software Language Engineering. pp. 137–156. Springer International Publishing, Cham (2013)
3. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools (2nd Edition). Addison-Wesley Longman Publishing Co., Inc., USA (2006)
4. Daniel Bruder, L.G.: Chartparsing cky algorithmus (2009), [Online; accessed 10-Juli-2024]
5. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Boston, 3rd edn. (2006)
6. Inc, A.: Applescript language guide (2016), [Online; accessed 10-Juli-2024]
7. Inc, A.: Swift (2024), [Online; accessed 10-Juli-2024]
8. Kemp, R.: Mehrdeutigkeiten kontextfreier grammatiken. In: Loeckx, J. (ed.) Automata, Languages and Programming. pp. 534–546. Springer Berlin Heidelberg, Berlin, Heidelberg (1974)
9. Parr, T.: The definitive antlr reference. https://theswissbay.ch/pdf/Gentoomen(2024), [Online; accessed 12-June-2024]

[3] Aho, Alfred V. and Lam, Monica S. and Sethi, Ravi and Ullman, Jeffrey D. (2006). Compilers: Principles, Techniques, and Tools, Seite 192

10. Simon, H.U.: Post'sches korrespondenzproblem (2014/2015), [Online; accessed 10-Juli-2024]
11. Thorup, M.: Controlled grammatic ambiguity (1994), [Online; accessed 17-April-2024]
12. Vasudevan, N., Tratt, L.: Detecting ambiguity in programming language grammars. In: Erwig, M., Paige, R.F., Van Wyk, E. (eds.) Software Language Engineering. pp. 157–176. Springer International Publishing, Cham (2013)
13. Watrous, J.: Parse trees, ambiguity, and chomsky normal form. https://cs.uwaterloo.ca/ watrous/ToC-notes/ToC-notes.08.pdf (2020), [Online; accessed 17-April-2024]
14. Wharton, R.M.: Resolution of ambiguity in parsing. Acta Informatica **6**(4), 387–395 (1976). https://doi.org/10.1007/BF00268139, https://doi.org/10.1007/BF00268139