UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft
Fakultät für Elektrotechnik, Informatik und Mathematik

Bachelor's Thesis

Ambiguity Detection for Context-Free Grammars in Eli

Michael Kruse
Student Id: 6284674
E-Mail: meinert@uni-paderborn.de

Paderborn, 7th May, 2008

presented to
Dr. Peter Pfahler
Dr. Matthias Fischer

Statement on Plagiarism and Academic Integrity

I declare that all material in this is my own work except where there is clear acknowledgement or reference to the work of others. All material taken from other sources have been declared as such. This thesis as it is or in similar form has not been presented to any examination authority yet.

Paderborn, 7th May, 2008

Michael Kruse

Acknowledgement

I especially whish to thank the following people for their support during the writing of this thesis:

- *Peter Pfahler* for his work as advisor
- *Sylvain Schmitz* for finding some major errors
- *Anders Møller* for some interesting conversations
- *Ulf Schwekendiek* for his help to understand Eli
- *Peter Kling* and *Tobias Müller* for proofreading

Contents

1. Introduction

This thesis is about ambiguities in context-free grammars and how potentially ambiguous grammars can be found. This knowledge is used to write a program that identifies unambiguous grammars programmatically.

A *context-free grammar* (CFG) is a recursive definition of a (usually non-natural) language. One application is the definition of the syntax of a programming language like Pascal or C++. A context-free grammar can generate a string (a word, sentence, text file or sequence of tokens) by applying productions one after the other. To analyze such a sequence it is necessary to rediscover the productions and structure that have been used to generate it. This is called *parsing*, its input is a sequence of terminals and its result is a *parse tree*. See Definition 2.1 for a formal definition of a context-free grammar.

Algorithms exist which parse any sequence of length n for any definition a CFG in $O(n^3)$ asymptotic time. The *Cocke-Younger-Kasami Algorithm* (CYK-Algorithm) is such an algorithm. However, subsets of context-free grammars exist that are parsable in quadratic or linear time (Figure 1.1).

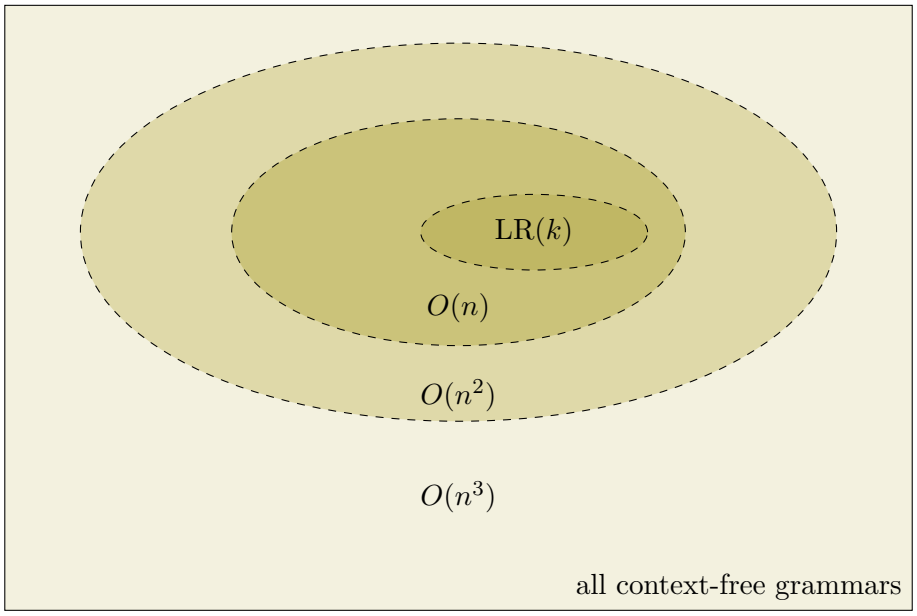

Figure 1.1.: The parsing complexity subsets of context-free grammars

LR(k) is such a subset, which is parsable in linear time. This class of context-free grammars is even defined as the set of grammars that can be parsed using an LR(k)-parser which is an algorithm that needs only one pass over the input sequence. k is the amount of lookahead symbols. The more lookahead symbols are used, the more grammars can be parsed, but the bigger the parsing table becomes. The parsing table determines the working of an LR(k) parser. $k = 1$ is the most often used number of lookahead symbols, because its parsing table normally does not become too big. An LALR(k)-parser reduces the size of the parsing table even further. Grammars which are accepted by this parser are called LALR(k), which form a subset of the class of LR(k) grammars.

All LR(k) and LALR(k) grammars are unambiguous grammars. An unambiguous grammar is a grammar for which only a single parse tree for every input sequence exits. Hence ambiguous grammars have multiple parse trees. See Figure 1.2 for a hierarchy of all grammars mentioned here.

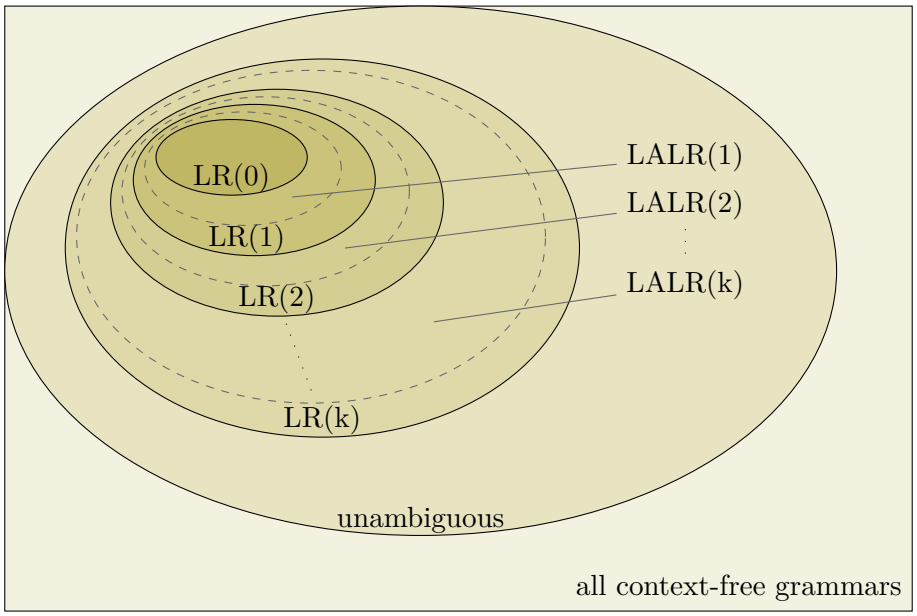

Figure 1.2.: The hierarchy of context-free grammars

The construction of a LR(k) or LALR(k) parser is rather complicated but can be automated by a parser generator. *yacc* [7], *Bison* [5] and *Cola for Eli* [13] are generators for LR(1) and/or LALR(1) grammars.

The Eli system is a toolset, which assists with every step of building a compiler. Generating a parser is one of these steps. Eli also includes tools to create scanners, abstract syntax trees, etc.

If a parser generator tries to generate a parser for a grammar which does not belong to the class of supported grammars, it will stop because of a *conflict*. A conflict means that there is more than one operation the generated parser should

execute while processing the next terminal of the input sequence. However, an LR(k)-parser allows only one operation at a time and as a consequence, a parser for such a grammar cannot be generated.

An approach to handle conflicts is called *Tomita-* or *Generalised LR* (GLR) Parsing. What the parser does is to follow all possibilities that might arise from the conflicts. The parser's internal state is copied and continued separately for every possibility. The derived paths may lead to dead ends or even to another conflict, which will cause another split. However, as long as at least one path continues until the last terminal has been read, the input is parsed successfully.

Bison [5] can also generate GLR parsers and a recent work by Ulf Schwekendiek [19] integrated the Bison GLR parser generator into the Eli system.

If a grammar is ambiguous, a GLR parser may return multiple parse trees. Furthermore, there may be exponentially many parse trees dependent on the input length. Usually, this is unwanted for technical languages and when a grammar is designed, one usually tries to keep it unambiguous.

A practical problem is, that it is undecidable whether a context-free grammar is ambiguous or not: There is no algorithm that is able to decide whether an input exists such that a GLR parser returns more than one syntax tree. This does not mean that no approximations exist. For instance, if an LR(k) parser generator can create a parser, the source grammar is guaranteed to be unambiguous. If this is known the GLR parser will never return more than one parse tree for any input string.

There are two recent articles on other ambiguity detection approximations. The first work by Claus Brabrand, Robert Giegerich and Anders Møller [3] divides the ambiguity problem into horizontal and vertical ambiguity, both are undecidable as well. They can be approximated by using a regular superset of the original grammar. The authors call this technique *Ambiguity Checking with Language Approximations* (ACLA).

The other approximation has been named *Regular/Noncanonical Unambiguity* by Sylvain Schmitz [17]. This approach specialises an LR(k) parsing algorithm to find potential ambiguities. Both approaches can recognise unambiguous grammars which the other cannot and therefore can be used in combination.

In this thesis these ambiguity approximations will be explained closer and added to the Eli system. In more details, the objectives for this thesis are

- to describe the ambiguity problem for context-free grammars
- to present both ambiguity detection approximations
- to implement the ACLA technique
- to integrate both techniques into the Eli system
- to compare and evaluate the results of the two techniques

2. Parsing Techniques

Parsing is the retrieval of the tree structure of a one-dimensional sequence of symbols. The tree structure is described by a context-free grammar. The recovered structure is called a parse tree.

In this chapter context-free grammars are formally defined. Then, we show how an input sequence is parsed. The parsing techniques covered here are *LR-parsing* and *generalized LR-parsing*.

2.1. Context-Free Grammars

Definition 2.1 (Context-free grammar (CFG))
A *Context-Free Grammar* is a 4-tuple $G = (N, \Sigma, P, S)$ where

N is a finite set of *nonterminals*.

Σ is a finite set of *terminals*.

P is a relation $A\mathrm{P}\alpha$ where $A \in N$ and $\alpha \in (N \cup \Sigma)^*$. An element $AP\alpha \in P$ is called a production.

$S \in N$ is the start nonterminal.

The more common notation for $A\mathrm{P}\alpha$ is $A \to \alpha$. A is the left-hand-side and α the right-hand-side of this production. α is allowed to be empty. This is indicated by the ϵ symbol and such a production is called an ϵ-production. The relation P is a set: no combination of left-hand-side nonterminals and right-hand-side sequences appears twice.

Additionally we define $V := N \cup \Sigma$ and call an element of V a symbol (thus a symbol can be a terminal or a nonterminal).

As a convention the characters $a, b, c \cdots \in \Sigma$ are terminals, $A, B, C \cdots \in N$ are nonterminals and $X, Y, Z \in V$ are terminals or nonterminals. In the same manner $u, v, w, x, y, z \in \Sigma^*$ are sequences of terminals and $\alpha, \beta, \gamma \in V^*$ are sequences of symbols. Additionally, S is always the start nonterminal.

Definition 2.2 (The derivation relation $\Rightarrow$)
Let $G = (N, \Sigma, P, S)$ be a context-free grammar and $A \to \gamma$ a production of G. Then $\alpha A \beta \Rightarrow \alpha\gamma\beta$ is called a derivation if G. We say that the production $A \to \gamma$ is applied on $\alpha A \beta$.

Definition 2.3 (The left-/rightmost derivation relations $\Rightarrow_{\mathrm{LM}}$ and $\Rightarrow_{\mathrm{RM}}$)
A derivation $\alpha A\beta \Rightarrow \alpha\gamma\beta$ is a *leftmost derivation* $\alpha A\beta \Rightarrow_{\mathrm{LM}} \alpha\gamma\beta$ iff $\alpha \in \Sigma^*$. In the same way, it is a *rightmost derivation* $\alpha A\beta \Rightarrow_{\mathrm{RM}} \alpha\gamma\beta$ iff $\beta \in \Sigma^*$

$\Rightarrow^*$ is the transitive closure of $\Rightarrow$. If $\alpha \Rightarrow^* \beta$ holds, we say that α can be derived to β. The sequence $\alpha \Rightarrow \cdots \Rightarrow \gamma \Rightarrow \cdots \Rightarrow \beta$ is called the derivation of α to β. In the same sense $\Rightarrow^*_{\mathrm{LM}}$ is the transitive closure of $\Rightarrow_{LM}$. The leftmost derivation is used to avoid that there are multiple derivations only because the right nonterminal can be applied before the left one. Alternatively, the rightmost derivation can be used.

This transitive closure can also be expressed as tree. Whenever a production is applied on a nonterminal, we expand its node in the tree. Every symbol on the right-hand-side of the production becomes a child of this node. The advantage is that the order in which productions are applied does not matter and always result in the same tree. Such a tree is called a derivation tree.

Example 2.4 (Derivation and derivation tree)
Consider the Grammar 2.1. Equation 2.2 shows a (leftmost) derivation where three productions are applied. Figure 2.1 shows the same derivation as a tree.

$$\begin{aligned} &\mathrm{S} \rightarrow \mathrm{a\ b\ A} \\ &\mathrm{A} \rightarrow \mathrm{S\ c\ A} \\ &\mathrm{A} \rightarrow \epsilon \end{aligned} \tag{2.1}$$

$$S \Rightarrow abA \Rightarrow abScA \Rightarrow ababAcA \tag{2.2}$$

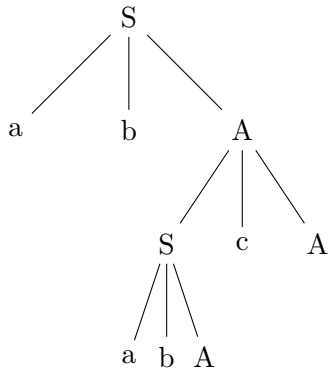

Figure 2.1.: Derivation 2.2 as a derivation tree

Definition 2.5 (The language of a context-free grammar)
The language $\mathcal{L}(G)$ accepted by a context-free grammar $G = (N, \Sigma, P, S)$ is

$$\mathcal{L}(G) := \{w \in \Sigma^* \mid S \Rightarrow^* w\}$$

We say that $w \in \mathcal{L}(G)$ is *recognised* by the grammar G. w is called a *word* of the grammar G/the language $\mathcal{L}(G)$.

The symbol $\mathcal{L}$ can also be used to describe the language of a sequence of terminals and nonterminals $\alpha \in V^*$

$$\mathcal{L}_G(\alpha) := \{w \in \Sigma^* \mid \alpha \Rightarrow^* w\}$$

Hence a word is a sequence of terminals which can be derived from S. We call a derivation tree from the start symbol S to a sequence without any nonterminals a *parse tree*.

To simplify some proofs later, we define that for every nonterminal and production, there is at least one parse tree in which it occurs. Otherwise it can be removed from the grammar without altering its language.

When considering the plain word w, the information which productions were applied is lost. A parser takes a sequence of terminals and returns a parse tree. If it fails to return a parse tree it means that there is no parse tree and thus the input is not a recognised by the grammar.

2.2. LR(k)-Parsing

This section describes the LR(k) parsing technique which was introduced by Donald E. Knuth [8]. An LR parser obtains the structure of any input of its grammar in linear time. The drawback is that it is not possible to create such a parser for every grammar.

LR(k) stands for processing the input sequence from **l**eft to right while producing a rightmost derivation in **r**everse. k is the number of used lookahead symbols and is usually one.

An LR(k) parser consists of an ACTION and a GOTO table. The ACTION table defines actions to execute depending on the next k terminals (the lookahead terminals) in the input sequence. The GOTO table defines the next state when a nonterminal has been recognised. These terminals and nonterminals define the columns of the tables. The rows of the table determine a state of a finite state machine. The transitions of this finite state machine are defined implicitly by the ACTION and GOTO table. Both tables together are called the *parsing table*, which is the static part of the parser and does not change during the parser's execution. This table is specific to a context-free grammar $G = (N, \Sigma, P, S)$ and usually precomputed by a parser generator.

At runtime some more data structures are needed which we summarise as the parser's current configuration.

- A pointer to the next terminal in the input sequence (current position)
- The current state of a finite state machine
- A stack which stores tuples (X, q) of symbols ($X \in V$) and states (q)

The parser is initialised with a pointer to the first input terminal. The initial state is zero and the stack contains a sentinel element $(X, 0)$, where X can be any value because it is never used.

During the execution the parser reads the k terminals beginning at the current position. Then it reads the ACTION table at the column for this sequence of terminals at the current state and executes it. This is done until the input is accepted or an error occurs, which means that the input sequence is not a word of the grammar.

The actions in the ACTION table are chosen from the following list.

- Shift q
  1. Push (q, a) to the stack. a is the terminal at the current position.
  2. Set the current state of the finite state machine to q.
  3. Move the input pointer to the next terminal.

  A parser uses this action as long as more input symbols are needed to complete a nonterminal.

- Reduce $(A \to \alpha) \in P$
  1. Pop $|\alpha|$ tuples off the stack where $|\alpha|$ is the number of items on the right-hand-side of the production.
  2. Get the tuple (p, X) which now on the top of the stack.
  3. Lookup the state q at row p and column A of the GOTO table.
  4. Push the tuple (q, A) to the stack.
  5. Set the current state of the FSM to q.
  6. The input pointer remains unchanged.

  } GOTO q

  Using this action means that all right-hand-side symbols of a production are on the stack. α is replaced by the nonterminal A. The nonterminal acts as a placeholder for α. When a parse tree is built, A becomes the root node for all elements of α.

- Accept
  1. Return. The input has been parsed successfully.

  The parser chooses this when the start symbol S is the only symbol on the stack and there are no more remaining input terminals. If a parse tree was built, then node of S can be returned as the parse tree.

- Error
  1. Abort. The input is not a word of the grammar.

This means that the parser detected that the input is not a word of the grammar's language.

Example 2.6
Consider Grammar 2.3 and a word $w =$"`c a a c b b`". We will see how an LR(1) parser processes this word.

$$\begin{array}{ll} (1) & \text{S} \rightarrow \text{A S} \\ (2) & \text{S} \rightarrow \text{A} \\ (3) & \text{A} \rightarrow \text{a A b} \\ (4) & \text{A} \rightarrow \text{c} \end{array} \tag{2.3}$$

The LR(1) parsing table for this grammar is shown in Table 2.2. Grammar 2.3 has 3 terminals and 2 nonterminals. An additional terminal EOF (**E**nd **O**f **F**ile) is introduced which marks the ending of the input sequence. Thus the word to parse is $w =$"`c a a c b b EOF`". The ACTION table has 4 and the GOTO table 2 columns. 8 rows correspond to 8 states of the finite state machine for this parsing table.

The actions are abbreviated as follows.

- "Shift q" becomes "sq".
- "Reduce ($A \rightarrow \alpha \in P$)" becomes "r$i$", where i is the number of the production.
- "Accept" becomes "acc".
- "Error" becomes an empty cell.

	ACTION				GOTO	
State	a	b	c	EOF	S	A
0	s1		s2		7	5
1	s1		s2			3
2	r4	r4	r4			
3		s4				
4	r3	r3	r3			
5	s1		s2	r2	6	5
6	r1		r1	r1		
7				acc		

Table 2.2.: Parsing table for Grammar 2.3

Now we can run the parser on the input sequence "`c a a c b b EOF`". The parser configuration before every action is shown in Table 2.3. The terminals from the current position up to the end of the input sequence is shown in the column "Remaining Input". The lookahead terminal is highlighted in red. Terminals before the current position are not needed anymore and thus are not displayed.

Step	Stack	State	Remaining Input	Action
1	0	0	c a a c b b EOF	Shift 2
2	0 c2	2	a a c b b EOF	Reduce (4)
	0	0	a a c b b EOF	GOTO 5
3	0 A5	5	a a c b b EOF	Shift 1
4	0 A5 a1	1	a c b b EOF	Shift 1
5	0 A5 a1 a1	1	c b b EOF	Shift 2
6	0 A5 a1 a1 c2	2	b b EOF	Reduce (4)
	0 A5 a1 a1	1	b b EOF	GOTO 3
7	0 A5 a1 a1 A3	3	b b EOF	Shift 4
8	0 A5 a1 a1 A3 b4	4	b EOF	Reduce (3)
	0 A5 a1	1	b EOF	GOTO 3
9	0 A5 a1 A3	3	b EOF	Shift 4
10	0 A5 a1 A3 b4	4	EOF	Reduce (4)
	0 A5	5	EOF	GOTO 5
11	0 A5 A5	5	EOF	Reduce (2)
	0 A5	5	EOF	GOTO 6
12	0 A5 S6	6	EOF	Reduce (1)
	0	0	EOF	GOTO 7
13	0 S7	7	EOF	Accept

Table 2.3.: Parsing the sequence "`c a a c b b EOF`" using the parsing table 2.2

The GOTO action is mentioned explicitly although it is part of the reduce action. The stack before the GOTO action is the stack where the tuples for the reduced production have been removed from the stack. The tuple with the nonterminal of that production is pushed right after.

Step 1 shows the initial configuration. The stack only contains the sentinel state to avoid access to an empty stack. This way the topmost item of the stack always matches the current position. The algorithm ends successfully in step 13. The only symbol left on the stack is the start symbol. The EOF symbol is never pushed on the stack.

A parser does not necessarily create a parse tree. Instead it can execute *semantic actions* whenever a production is reduced. A parser generator can merge user-defined semantic actions into the generated parser.

How such a parsing table is computed is not a subject here. Prevalent books like *Compilers - Principles, Techniques, & Tools* [1] (also known as the *Dragon Book*) cover this topic extensively.

As mentioned earlier it is possible that there is no LR(k) parsing table for a specific grammar. The greater k the more grammars can be processed (and the bigger the parsing table becomes), but there is no k which allows to process all grammars. Hence, the LR parsing method is limited to a subset of all context-free grammars. A context-free grammar, for which an LR(k) parsing table exists, is

called an LR(k)-grammar.

If a parsing table cannot be generated, then because of a conflict: more than one action is possible in a cell of the ACTION table. This means that, given only the k lookahead terminals, it is not possible to decide which action is correct.

There are two classes of conflicts. A shift/reduce conflict means that the next symbol could be either shifted to the stack or cause a reduce action. A reduce/reduce conflict is a clash between two different productions which could be reduced. Both types of conflicts can occur in the same cell of the ACTION table.

Neither an accept nor an error action can be involved in an conflict. The accept action is dominant over all other actions. An error action is just a lack of other possibilities.

2.3. Generalised LR-Parsing

The existence of a conflict prohibits that an LR parser continues because this technique allows only one action per step. Having more than one possible choice is a kind of indeterminism. At the point of the conflict it is unknown which possibility is a path to a successful parse. Multiple paths must be followed in order to find out which one leads to a valid parse tree. One can think of two ways to do this.

If Backtracking is used, only one of those paths is followed. If it does not lead to a valid parse tree, the algorithm goes back to a configuration before the conflict and tries a different path. This repeats until a valid parse is found or all paths have been processed. In the latter case there is no parse tree.

Alternatively, all paths can be processed in parallel using multiple parser configurations. When a conflict is encountered the configuration splits into a new configuration for every possible action. Actions are executed on all configurations independently. An error action causes that no more actions are executed on the affected configuration. The input word is accepted when at least one configuration reaches an accept action.

This idea has been described by Bernhard Lang [9] and elaborated by Masaru Tomita [21]. It is known as *Tomita-* or *generalised LR-parsing* (GLR) by today.

Example 2.7
Consider grammar 2.4. This grammar has just three valid words: "`c c c a`", "`c c c b`" and "`c c c`".

$$\begin{array}{ll} (1) & \mathrm{S} \rightarrow \mathrm{A\ c\ a} \\ (2) & \mathrm{S} \rightarrow \mathrm{B\ c\ b} \\ (3) & \mathrm{S} \rightarrow \mathrm{C\ c\ c} \\ (4) & \mathrm{A} \rightarrow \mathrm{c\ c} \\ (5) & \mathrm{B} \rightarrow \mathrm{c\ c} \\ (6) & \mathrm{C} \rightarrow \mathrm{c} \end{array} \tag{2.4}$$

The LR(1) parsing table (Table 2.4) for this grammar shows two conflicts. There

is a shift/reduce-conflict in state 1 and a reduce/reduce-conflict in state 2. A standard LR(1) parser generator would fail, but when used in a GLR parser, the cells contain sets of actions instead of single actions. Here, an empty cell means an empty set of actions.

	ACTION				GOTO			
State	a	b	c	EOF	S	A	B	C
0			{s1}		11	3	6	9
1			{s2,r6}					
2			{r4,r5}					
3			{s4}					
4	{s5}							
5				{r1}				
6			{s7}					
7		{s8}						
8				{r2}				
9			{s10}					
10				{r3}				
11				{acc}				

Table 2.4.: LR parsing table for grammar 2.4

Figure 2.5 illustrates how a GLR parser processes the input word "`c c c a`". The different parallel configurations are shown in their own boxes. The current state is not shown explicitly because it always corresponds to the topmost item on the stack. Whenever the position pointer moves to the next terminal, a box on the right shows the remaining input and the lookahead terminal highlighted in red.

Both conflicts are encountered during this parse, so there are two splits of configurations. Anyhow, in the further progress two of the three paths stop at dead ends. Just one finishes and accepts the word as part of the grammar's language.

A good implementation of the GLR parser scheme has a worst-case runtime of $O(n^3)$ – the same as a CYK parser. However, if the context-free grammar is LR(k), then a GLR parser has a runtime of $O(n)$ like an LR(k) parser. Hence it combines the advantages of LR(k) and CYK parsing techniques, being able to parse any grammar and to do it in linear time if the grammar permits it.

An open question is what happens if there are multiple paths that accept an input word. Depending on the implementation of the GLR parser, it either returns just the first parse tree it finds or all that exist. In the first case the parser chooses an arbitrary interpretation of the input string without knowing whether this is one meant by the author. In the latter case it may return up to exponentially many parse trees, but the "correct" one is still unknown.

Both approaches are unsatisfying. Therefore, it is normally the best to use grammars which do not have multiple parse trees for any word. This is the topic of the next chapter.

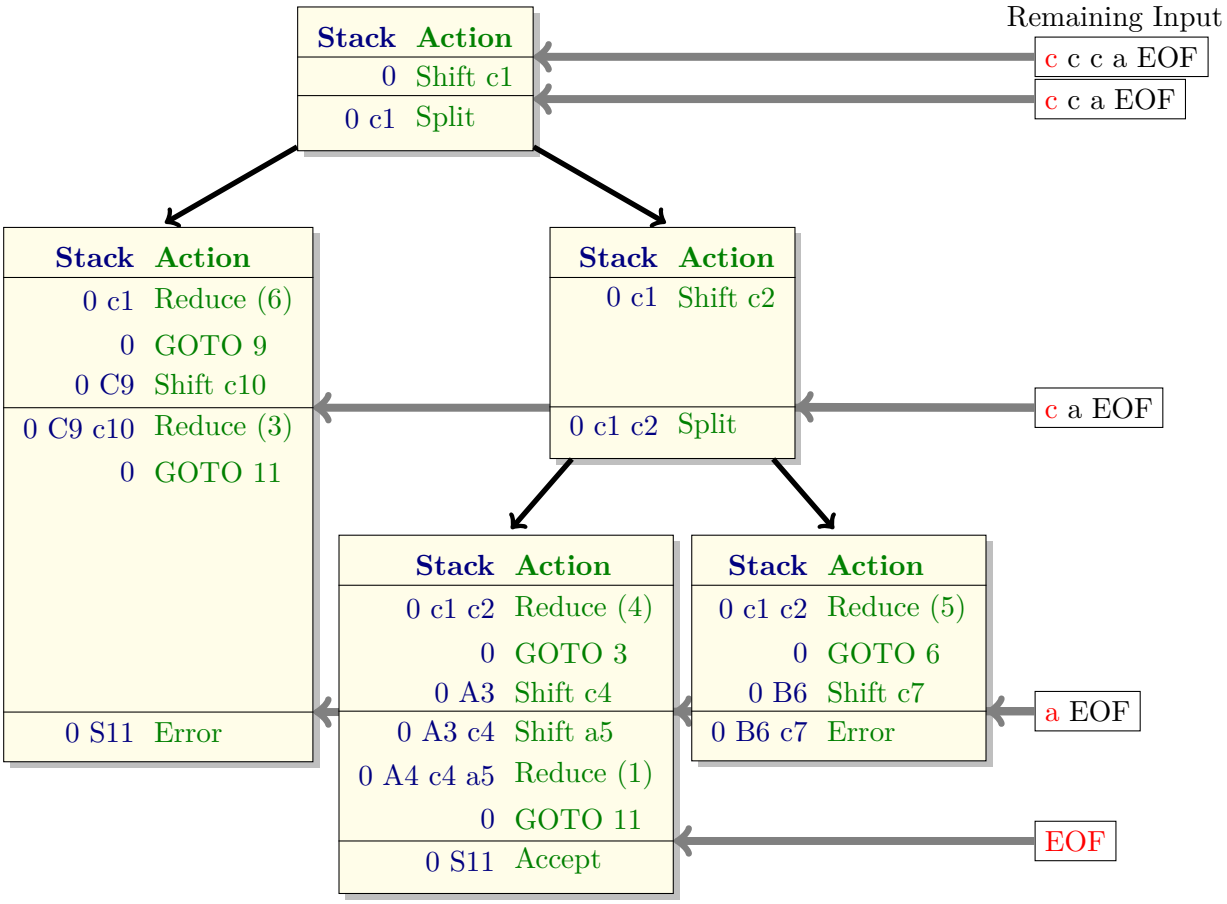

Figure 2.5.: A GLR parser processing the word "`c c c a`" using the parsing table 2.4

3. Ambiguity Detection

For some grammars, there is no bijective relation between input words and parse tree. While every parse tree corresponds to a single word, the reverse does not hold necessarily. A context-free grammar where a word with more than one parse tree exists is called ambiguous.

In this chapter, we will formally define ambiguity of a context-free grammar. Because there is no algorithm that decides whether a grammar is ambiguous, we will present two different techniques to approximate this problem: *Ambiguity Checking with Language Approximations* (ACLA) and *Regular/LR-Regular/Noncanonical Unambiguity*.

3.1. Ambiguous Context-Free Grammars

An example of a grammar with multiple parse trees is Grammar 3.1. It recognises the word "`ab`", which has two different right-/leftmost derivations (Equations 3.2) and parse trees (Figure 3.1).

$$\begin{aligned} &\mathrm{S} \to \mathrm{a\ b} \\ &\mathrm{S} \to \mathrm{A\ b} \\ &\mathrm{A} \to \mathrm{a} \end{aligned} \tag{3.1}$$

$$\begin{aligned} &S \Rightarrow_{\mathrm{LM}} a\ b \\ &S \Rightarrow_{\mathrm{LM}} A\ b \Rightarrow a\ b \end{aligned} \tag{3.2}$$

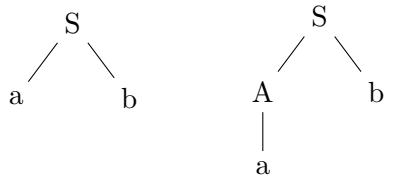

Figure 3.1.: Two parse trees of the word "`ab`" using Grammar 3.1

Definition 3.1 (Ambiguous context-free grammar)
A context-free grammar $G = (V, \Sigma, P, S)$ is *ambiguous*, iff a word $w \in \Sigma^*$ exists, which has multiple leftmost derivations $S \Rightarrow^*_{LM} w$ in G.

In this definition, the leftmost derivation operation is used. Using rightmost derivation or parse trees instead results in equivalent definitions. A grammar is called *unambiguous* if it is not ambiguous.

The *ambiguity problem* for context-free grammars is the question, whether a given context-free grammar is ambiguous. This problem is undecidable (Theorem 3.2): there is no algorithm that solves this problem for every grammar. It is proven in section A in the appendix.

Theorem 3.2 (Undecidability of the ambiguity problem for context-free grammars)
There is no algorithm that decides whether a given context-free grammar is ambiguous or unambiguous.

This result implies practical problems. A technical language should be unambiguous to avoid misinterpretations. If a grammar is unambiguous, there is only one unique interpretation for every word and thus problems with misinterpretation do not occur. But as Theorem 3.2 shows, there are grammars that are unambiguous, but it is not possible to be sure about that.

3.1.1. Ambiguity Approximation

Although no algorithm exists which solves the ambiguity problem for every context-free grammar, algorithms exists which solve the problem for some grammars. We call such algorithms *approximations*. An approximation algorithm can return up to three values for the ambiguity problem: *unambiguous*, *ambiguous* and *n/a*, if it cannot make a decision.

Here, we only consider algorithms that finish in finite time and never return a wrong result (i.e. it does never return "*ambiguous*" when the grammar is unambiguous nor "*unambiguous*" for an ambiguous grammar). Such an algorithm is called a *safe* approximation.

Typically, it is interesting to know whether a grammar is unambiguous so it is safe to use it for a technical language. Thus, we do not distinguish between the results *amb.* and *n/a*. Both mean that ambiguities cannot be ruled out and a GLR could return more than one parse tree. This is a *conservative* assumption.

3.2. Ambiguity Checking with Language Approximations

The ambiguity detection algorithm "Ambiguity Checking with Language Approximations" (or ACLA, for short) has been proposed by Claus Brabrand, Robert Giegerich and Anders Møller [3]. Instead of dealing with the ambiguity problem as a whole, it is separated into two similar problems, namely vertical and horizontal ambiguity. Both problems are undecidable again.

At first, we define horizontal and vertical ambiguity. Then, we show that a superset of the original grammar allows an approximation of the ambiguity problem,

if the intersection and the overlap operators can be computed on the superset. Finally, we show how regular grammars can be used as such supersets.

3.2.1. Horizontal and Vertical Ambiguity

Definition 3.3 (Horizontal ambiguity)
A grammar $G = (N, \Sigma, P, S)$ is horizontally ambiguous iff it has a production $A \to \gamma$ with a partition $\gamma = \alpha\beta$, such that the languages $\mathcal{L}(\alpha)$ and $\mathcal{L}(\beta)$ overlap, i.e. there is a sequence of terminals that could be the ending of $\mathcal{L}(\alpha)$ or the beginning of $\mathcal{L}(\beta)$. As a formula:

$$\exists A \in N, (A \to \alpha\beta) \in P : \mathcal{L}_G(\alpha) \between \mathcal{L}_G(\beta) \neq \varnothing$$

The *overlap* $\between$ of two languages X and Y is defined by

$$X \between Y = \left\{ xvy \mid x, y \in \Sigma^* \wedge v \in \Sigma^+ \wedge x, xv \in X \wedge vy, y \in Y \right\}$$

Definition 3.4 (Vertical ambiguity)
A grammar $G = (N, \Sigma, P, S)$ is vertically ambiguous iff it has two different productions $A \to \alpha$ and $A \to \beta$, which can be derived to the same word. As a formula:

$$\exists A \in N, (A \to \alpha), (A \to \beta) \in P, \alpha \neq \beta : \mathcal{L}_G(\alpha) \cap \mathcal{L}_G(\beta) \neq \varnothing$$

Theorem 3.5 (Equivalence of both ambiguity definitions)
The following statements are equivalent:

1. *Grammar G is unambiguous as in Definition 3.1*
2. *Grammar G is vertically and horizontally unambiguous*

The proof of this theorem is separated into three lemmas. The final proof is at the end of this section.

Lemma 3.6 (Horizontal ambiguity $\Rightarrow$ ambiguity)
If grammar G is horizontally ambiguous then it is also ambiguous as by Definition 3.1.

Proof. Assume that G has a horizontal ambiguity in production $A \to \alpha\beta$ so $\mathcal{L}(\alpha) \between \mathcal{L}(\beta) \neq \varnothing$. Let w be an element of $\mathcal{L}(\alpha) \between \mathcal{L}(\beta)$. By definition of the overlap operator $w = xvy$ with $|v| > 0$, $x, xv \in \mathcal{L}(\alpha)$ and $y, vy \in \mathcal{L}(\beta)$ holds. Since A exists in at least one parse tree two different leftmost derivations can be built as shown here:

$$\begin{aligned} S &\Rightarrow^* \gamma A\delta \Rightarrow^* z_1 A\delta \Rightarrow z_1\alpha\beta\delta \Rightarrow^* z_1 xv\beta\delta \Rightarrow^* z_1 xvy\delta \Rightarrow^* z_1 xvyz_2 = z_1 wz_2 \\ S &\Rightarrow^* \gamma A\delta \Rightarrow^* z_1 A\delta \Rightarrow z_1\alpha\beta\delta \Rightarrow^* z_1 x\beta\delta \Rightarrow^* z_1 xvy\delta \Rightarrow^* z_1 xvyz_2 = z_1 wz_2 \end{aligned} \tag{3.3}$$

The LM subscripts were omitted here to avoid clutter. □

The derivations 3.3 are shown as parse trees in Figure 3.2. Again, two different trees end up with the same word z_1xvyz_2.

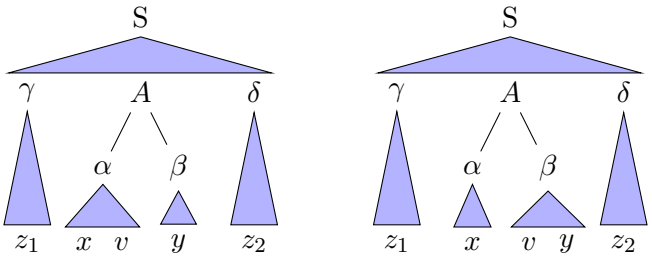

Figure 3.2.: Two parse trees for horizontal ambiguity

Lemma 3.7 (Vertical ambiguity $\Rightarrow$ ambiguity)
If grammar G is vertically ambiguous then it is also ambiguous as by definition 3.1.

Proof. Assume that G has a vertical ambiguity in nonterminal A. Hence, there are two different productions $A \to \alpha$ and $A \to \alpha'$ with $M := \mathcal{L}(\alpha) \cap \mathcal{L}(\alpha') \neq \varnothing$. Let $w \in M$ be an element of this intersection. The word w can be used to build two different leftmost derivations because by definition of a context-free grammar, A exists in at least one derivation.

$$\begin{aligned} S &\Rightarrow^* \beta A\gamma \Rightarrow^* xA\gamma \Rightarrow x\alpha\gamma \Rightarrow^* xw\gamma \Rightarrow^* xwy \\ S &\Rightarrow^* \beta A\gamma \Rightarrow^* xA\gamma \Rightarrow x\alpha'\gamma \Rightarrow^* xw\gamma \Rightarrow^* xwy \end{aligned} \tag{3.4}$$

Again, the LM subscripts were omitted. □

The derivations 3.4 are visualised in the parse trees of Figure 3.3. They are equal except for the symbol α (α' respectively) and their subtrees, but still derive to the same word xwy. Hence, the grammar is ambiguous.

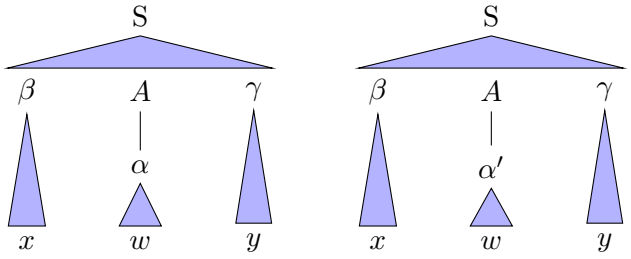

Figure 3.3.: Two parse tress for vertical ambiguity

Lemma 3.8 (Ambiguity $\Rightarrow$ horizontal ambiguity `or` vertical ambiguity)
If grammar G is ambiguous as by definition 3.1 then it is either horizontally or vertically ambiguous (or both).

Proof. Assume that $G = (N, \Sigma, P, S)$ is ambiguous. We show by mathematical induction over the height of the derivation tree that G must be vertically or horizontally ambiguous. The start symbol is always a nonterminal so the height of a parse tree is at least 1.

- Induction Basis: Tree height = 1
  Only one production $S \rightarrow w_1 \dots w_n = w$ has been applied. There can be only one such production, therefore this tree is unambiguous. This does not violate the implication in the induction hypothesis because its antecedent is false (*ex falso sequitur quodlibet*).

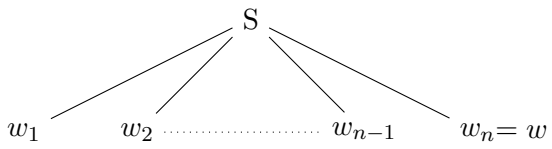

Figure 3.4.: Induction Basis ($h = 1$)

- Induction Hypothesis:
  Any parse tree with height $h-1$ or less, which is ambiguous, is either vertically or horizontally ambiguous.

- Inductive step:
  Without loss of generality, we assume that all productions have n symbols on the right hand side with n being the largest number of symbols in all productions. Productions having less symbols can be filled up using ϵ-symbols.

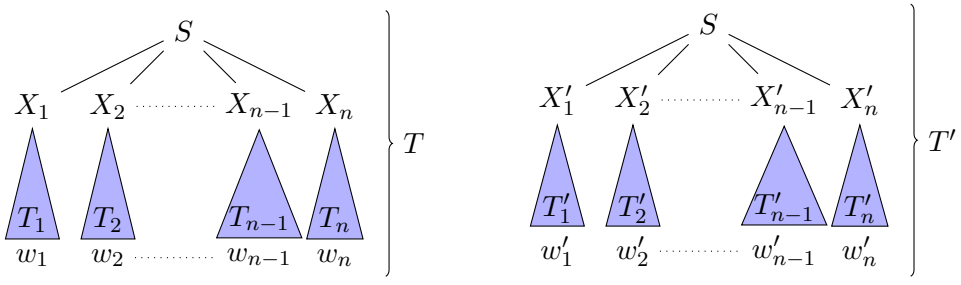

Figure 3.5.: Inductive step of Lemma 3.8

We assumed that grammar G is ambiguous and therefore a word w exists,

which has two different parse trees.

$$T = (S \Rightarrow X_1 \dots X_n \Rightarrow^* w_1 \dots w_n = w)$$
$$T' = (S \Rightarrow X'_1 \dots X'_n \Rightarrow^* w'_1 \dots w'_n = w)$$

with $X_i \Rightarrow^* w_i$ and $X_i \Rightarrow^* w'_i$ for all $i \in \{1, \dots, n\}$. The subtrees $X_i \Rightarrow^* w_i$ and $X'_i \Rightarrow^* w'_i$ are named T_i and T'_i respectively.

Furthermore, let h be the height of the higher tree of T and T'. Now there are three different cases we have to handle.

- Case $\exists i = \{1, \dots, n\} : X_i \neq X'_i$
  (there is a difference in the topmost derivation)

  Two different productions $S \rightarrow X_1 \dots X_n$ and $S \rightarrow X'_1 \dots X'_n$ have been applied. Hence the definition for vertical ambiguity $\mathcal{L}(X_1 \dots X_n) \bigcap \mathcal{L}(X'_1 \dots X'_n) \supseteq \{w\} \neq \varnothing$ shows that the grammar is vertically ambiguous.

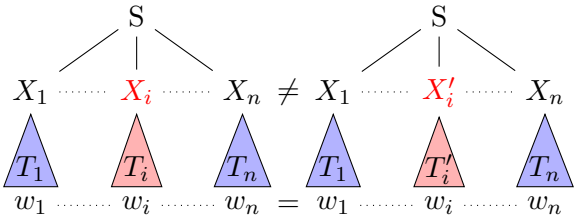

Figure 3.6.: Case 1

- Case $\forall i = \{1, \dots, n\} : X_i = X'_i$ and $\forall i = \{1, \dots, n\} : w_i = w'_i$
  (the topmost derivation is equal and its subtrees derive the same terminal sequences)

  All the subtrees T_i and T'_i start with the same symbol $X_i = X'_i$. However, since we assumed that $T \neq T'$ there is at least one i such that $T_i \neq T'_i$. The height of both trees is less than h and therefore the induction hypothesis can be applied to all of them. When applying the hypothesis to w_i, which has two different parse trees T_i and T'_i with starting symbol $X_i = X'_i$, we get that it is vertically or horizontally ambiguous and hence the grammar, which contains these subtrees, too.

- Case $\forall i = \{1, \dots, n\} : X_i = X'_i$ and $\exists i = \{1, \dots, n\} : w_i \neq w'_i$
  (the topmost derivation is equal, but its subtrees derive different sequences of terminals)

  Let i be the lowest index such that $w_i \neq w'_i$. We know that $w_1 \dots w_i w_{i+1} \dots w_n = w = w'_1 \dots w'_i w'_{i+1} \dots w'_n$, but $w_1 \dots w_i \neq w'_1 \dots w'_i$ and $w_{i+1} \dots w_n \neq w'_{i+1} \dots w'_n$.

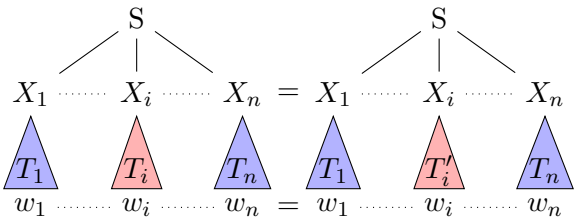

Figure 3.7.: Case 2

Without loss of generality we can assume that $|w_{i+1} \dots w_n| < |w'_1 \dots w'_i|$ (otherwise we switch the roles of T and T'). Then we also know that $|w_{i+1} \dots w_n| > |w'_{i+1} \dots w'_n|$ holds. The lengths cannot be equal, because w_i would not be different from w'_i, even if w_i or w'_i was ϵ.

We define $w_L := w_{i+1} \dots w_n$ and $w_R := w'_{i+1} \dots w'_n$ as the smaller of both parts. w_L matches the first terminals of w whereas w_R matches the last terminals of w. However, because $|w_1 \dots w_i| + |w'_{i+1} \dots w'_n| < |w|$, a "gap" remains between w_L and w_R. We fill this gap with a sequence of terminals $v \in \Sigma^+$ such that $w_L v w_R = w$.

When comparing to the overlap operator definition, we see that it matches $\mathcal{L}(X_1 \dots X_i) \between \mathcal{L}(X_{i+1} \dots X_n)$, because

$$\begin{aligned} X_1 \dots X_i &\Rightarrow^* w_1 \dots w_i = w_L \in \mathcal{L}(X_1 \dots X_i) \\ X_1 \dots X_i = X'_1 \dots X'_i &\Rightarrow^* w'_1 \dots w'_i = w_L v \in \mathcal{L}(X_1 \dots X_i) \\ X_{i+1} \dots X_n = X'_{i+1} \dots X'_n &\Rightarrow^* w'_{i+1} \dots w' = w_R \in \mathcal{L}(X_{i+1} \dots X_n) \\ X_{i+1} \dots X_n &\Rightarrow^* w_{i+1} \dots w = v w_R \in \mathcal{L}(X_{i+1} \dots X_n) \end{aligned}$$

Hence $\mathcal{L}(X_1 \dots X_i) \between \mathcal{L}(X_{i+1} \dots X_n) \supseteq \{w_L v w_R\} \neq \varnothing$, which fulfils the definition of horizontal ambiguity.

The existence of two different parse trees of height $\leq h$ implies that the grammar is vertically or horizontally ambiguous.

The theorem follows by applying mathematical induction. □

Proof of Theorem 3.5.

(1) $\Rightarrow$ (2)
Lemma 3.6 shows that G is ambiguous if G is horizontally ambiguous. Similarly, Lemma 3.7 states that G is ambiguous if G is vertically ambiguous. It follows that G is horizontally or vertically ambiguous, then G is ambiguous. The contraposition is: G is unambiguous $\Rightarrow$ G is vertically and horizontally unambiguous. □

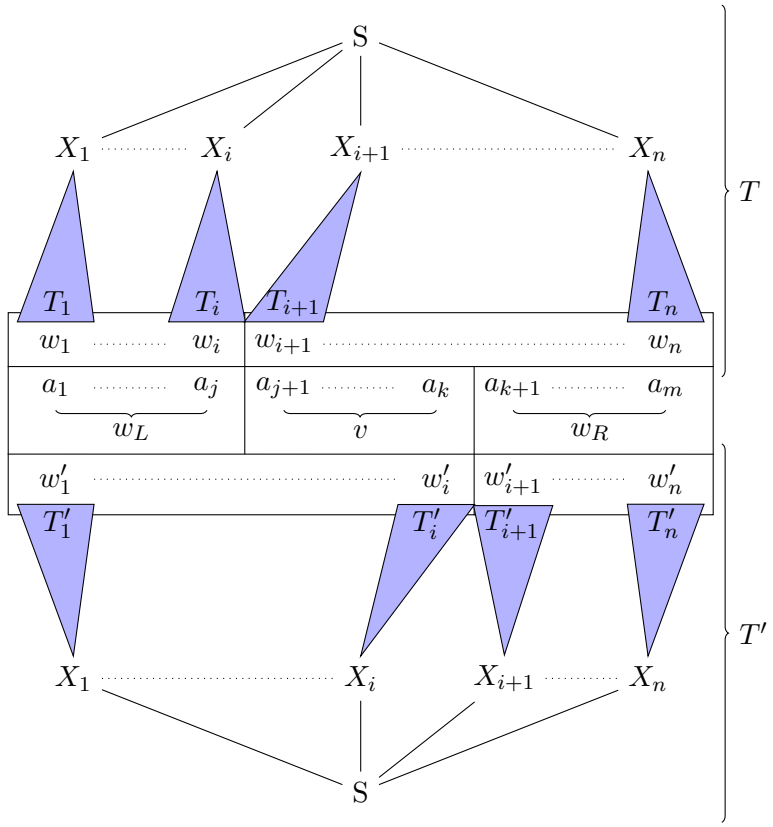

Figure 3.8.: Case 3

(2) $\Rightarrow$ (1)
This is the contraposition of Lemma 3.8. □

3.2.2. Approximation of Horizontal and Vertical Ambiguity

Dividing the ambiguity problem into two other problems does not bypass its undecidability. However, it allows to examine parts of a grammar independently. Horizontal ambiguity requires the overlap operator $\barwedge$ and vertical ambiguity the intersection operator $\cap$. These operators are not closed over the context-free languages.

If we use a superset of the original context-free language over which both operators are closed, the ambiguities remain. The next theorems show the property that causes this behaviour.

Theorem 3.9 (Superset inequations)
Let X and Y be sets of terminal sequences and $X' \supseteq X$ and $Y' \supseteq Y$ supersets.

Then

$$X' \cap Y' \supseteq X \cap Y \tag{3.5}$$
$$X' \between Y' \supseteq X \between Y \tag{3.6}$$

Proof. Equation 3.5 is basic set theory. $X \between Y$ is by definition equivalent to:

$$\Big\{ xvy \mid x, y \in \Sigma^* \wedge v \in \Sigma^+ \wedge x, xv \in X \wedge y, vy \in Y \Big\}$$

Let xvy be an element of $X \between Y$. Then $x, xv \in X$ and $y, vy \in Y$. X' and Y' are supersets of X and Y, so $x, xv \in X'$ and $y, vy \in Y'$, which makes the condition $xvy \mid x, y \in \Sigma^* \wedge v \in \Sigma^+ \wedge x, xv \in X' \wedge y, vy \in Y'$ true. Hence xvy is an element of $X' \between Y'$. □

We now define the language approximation $\mathcal{A}_G(\alpha) \supseteq \mathcal{L}_G(\alpha)$ of a grammar G and a symbol sequence α. By using Theorem 3.9 we can show that all horizontal and vertical ambiguities persist when using $\mathcal{A}_G(\alpha)$ instead of $\mathcal{L}_G(\alpha)$.

Definition 3.10 (Approximated horizontal ambiguity)
Let $G = (N, \Sigma, P, S)$ be a context-free grammar and $\mathcal{L}_G(\alpha) \subseteq \mathcal{A}_G(\alpha)$ for every $\alpha \in V^*$. G has a potential horizontal ambiguity if

$$\exists P \in N, (P \to \alpha\beta) \in P : \mathcal{A}(\alpha) \between \mathcal{A}(\beta) \neq \varnothing$$

Definition 3.11 (Approximated vertical ambiguity)
Let $G = (N, \Sigma, P, S)$ be a context-free grammar and $\mathcal{L}_G(\alpha) \subseteq \mathcal{A}_G(\alpha)$ for every $\alpha \in V^*$. G has a potential vertical ambiguity iff

$$\exists A \in N, (A \to \alpha), (A \to \beta) \in P, \alpha \neq \beta : \mathcal{L}(\alpha) \cap \mathcal{L}(\beta) \neq \varnothing$$

Theorem 3.12 (Horizontal ambiguity $\Rightarrow$ approximated horizontal ambiguity)
Let $G = (N, \Sigma, P, S)$ be horizontally ambiguous. Then G has a potential horizontal ambiguity for any approximation $\mathcal{A}_G$.

Proof. Let $(A \to \alpha\beta) \in P$ be the production that causes a horizontal ambiguity in G, i.e. $\mathcal{L}_G(\alpha) \between \mathcal{L}_G(\beta) \neq \varnothing$. This implicates $\mathcal{A}_G(\alpha) \between \mathcal{A}_G(\beta) \neq \varnothing$ (Theorem 3.9) and therefore $(A \to \alpha\beta)$ matches the definition of a potential horizontal ambiguity (Definition 3.10). □

Theorem 3.13 (Vertical ambiguity $\Rightarrow$ approximated vertical ambiguity)
Let $G = (N, \Sigma, P, S)$ be vertically ambiguous. Then G has a potential vertical ambiguity for any approximation $\mathcal{A}_G$.

Proof. Let $A \in N$ be the nonterminal that causes a vertical ambiguity in G, i.e. there are two productions $(A \to \alpha), (A \to \beta) \in P$ such that there is a $w \in \mathcal{L}_G(\alpha) \cap \mathcal{L}_G(\beta)$. Then, by Theorem 3.9, w is also an element of $\mathcal{A}_G(\alpha) \cap \mathcal{A}_G(\beta)$

and therefore this intersection is not empty. By Definition 3.11, G has a potential vertical ambiguity. □

By using contradiction these theorems show that if a grammar has no potential ambiguity, then it is unambiguous. If it is decidable whether $\mathcal{A}(\alpha) \cap \mathcal{A}(\beta)$ and $\mathcal{A}(\alpha) \between \mathcal{A}(\beta)$ are empty or not, then we have a conservative approximation for the ambiguity problem.

3.2.3. Approximation Using Regular Grammars

Regular grammars allow the computation of the intersection and the overlap and can be used for the superset approximation. Here we show that it is possible to get a regular superset of a context-free grammar and how both operators can be implemented.

Regular grammars are usually represented by *finite state machines* (FSM). A finite state machine is a directed graph. Its nodes and edges are called states and transitions. A transition can consume a symbol from the input string when it is traversed. There is one start state where every traversal through the graph starts and a set of accepting states where it can end.

The intersection of two regular languages can be computed in quadratic runtime depending on the number of states when both regular languages are represented as finite state machines. This is done by simulating the execution of both state machines simultaneously. This standard procedure in the field of automata theory is explained in [6], for instance.

The overlap operation is more complicated. The next theorem gives a hint how to implement it.

Theorem 3.14 (The overlap operator for regular languages)
Let FSM_1 and FSM_2 be two state machines that recognise a regular language and let Σ be the set of symbols used in FSM_1 or FSM_2. We introduce two new terminals <XV> and <VY> which do not occur in Σ. Then $\mathcal{L}(FSM_1) \between \mathcal{L}(FSM_2)$ equals to the intersection of the three sets

$$\{x_1 <VY> y_1 \mid x_1 \in \mathcal{L}(FSM_1 \text{ extended by } <XV>), y_1 \in \mathcal{L}(FSM_2)\} \tag{3.7}$$

$$\{x_2 <XV> y_2 \mid x_2 \in \mathcal{L}(FSM_1), y_2 \in \mathcal{L}(FSM_2 \text{ extended by } <VY>)\} \tag{3.8}$$

$$\left\{x_3 <XV> v_3 <VY> y_3 \mid x_3, y_3 \in \Sigma^*, v_3 \in \Sigma^+\right\} \tag{3.9}$$

when the terminals <XV> and <VY> have been removed from the intersection.

The notation "$\mathcal{L}(FSM$ extended by $a)$" means a modified version of the language recognised by FSM. It includes all strings where an arbitrary number of a-terminals have been inserted at any position into a word $w \in \mathcal{L}(FSM)$.

Proof. This theorem is proven by showing both subset relations.

$\subseteq$ Let xvy be an element of $\mathcal{L}(FSM_1) \between \mathcal{L}(FSM_2)$. We insert the additional terminals such that we get x<XV>v<VY>y. We now show that x<XV>v<VY>y is an element of all three sets. We define

$$\begin{aligned} x_1 &:= x\text{<XV>}v \\ x_2 &:= x \\ x_3 &:= x \\ y_1 &:= y \\ y_2 &:= c\text{<VY>}y \\ y_3 &:= y \\ v_3 &:= v \end{aligned}$$

and get

$$\begin{aligned} xv \in \mathcal{L}(FSM_1) &\Rightarrow x\text{<XV>}v = x_1 \in \mathcal{L}(FSM_1 \text{ extended by <XV>}) \\ y \in \mathcal{L}(FSM_2) &\Rightarrow y = y_1 \in \mathcal{L}(FSM_2) \\ x \in \mathcal{L}(FSM_1) &\Rightarrow x = x_2 \in \mathcal{L}(FSM_1) \\ vy \in \mathcal{L}(FSM_2) &\Rightarrow c\text{<VY>}y = y_2 \in \mathcal{L}(FSM_2 \text{ extended by <VY>}) \\ x \in \Sigma^* &\Rightarrow x = x_3 \in \Sigma^* \\ v \in \Sigma^+ &\Rightarrow v = v_3 \in \Sigma^+ \\ y \in \Sigma^* &\Rightarrow y = y_3 \in \Sigma^* \end{aligned}$$

Hence, the conditions of all three sets are fulfilled.

$\supseteq$ Let $w := x_1\text{<VY>}y_1 = x_2\text{<XV>}y_2 = x_3\text{<XV>}v_3\text{<VY>}y_3$ be an element of all three sets.

<XV> and <VY> do not occur in $y_1 \in \mathcal{L}(FSM_2)$, $x_2 \in \mathcal{L}(FSM_1)$, $x_3, y_3 \in \Sigma^*$ and $v_3 \in \Sigma^+$. And because of equation 3.9, they occur exactly once in w. <XV> and <VY> also occur in equations 3.7 and 3.8. They separate the variables the following way:

$$\overbrace{\underbrace{x_3}_{x_2}\ \text{<XV>}\ v_3}^{x_1}\ \text{<VY>}\ \overbrace{\underbrace{y_3}_{y_2}}^{y_1}$$

Hence

$$
\begin{aligned}
x &:= x_3 = x_1 \in \mathcal{L}(FSM_1) \subseteq \Sigma^* \\
v &:= v_3 \in \Sigma^+ \\
y &:= y_3 = y_3 \in \mathcal{L}(FSM_3) \subseteq \Sigma^* \\
x\text{<XV>}v &= x_3\text{<XV>}v_3 = x_1 \in \mathcal{L}(FSM_1 \text{ extended by <XV>}) \\
v\text{<VY>}y &= v_3\text{<VY>}y_3 = y_2 \in \mathcal{L}(FSM_2 \text{ extended by <VY>})
\end{aligned}
$$

When the terminals <XV> and <VY> are removed from of x_1 and y_2 (they do not occur in x, v and y) we get

$$
\begin{aligned}
xv &= x_3v_3 \in \mathcal{L}(FSM_1) \\
vy &= v_3y_3 \in \mathcal{L}(FSM_2)
\end{aligned}
$$

As a result, $w = xvy$ is an element of $\mathcal{L}(FSM_1) \bowtie \mathcal{L}(FSM_2)$ □

A finite state machine can be extended by adding a transition from every state to itself that consumes the terminal by which the state machine is extended.

The three sets are concatenations of finite state automata, Σ^*, Σ^+ and the terminals <XV> and <VY>. They all can be expressed as finite state machines. Finite state machines are appended after each other by adding transitions from the accepting states of the first to the start state of the second state machine. The intersection between them can be computed as mentioned before.

3.2.4. Regular Supersets

Finally, the regular superset of a context-free grammar itself has to be computed. Clearly it exists, because Σ^* is regular and a superset of every language. The authors of [3] use the Mohri-Nederhof-Transformation [10] to get a superset of a context-free grammar.

Another approach based on the Regular Unambiguity technique in the next section has been implemented for this thesis. It has a closer approximation of the context-free grammar, but its runtime behaviour is worse. A comparison can be found at section 4.1.1.

3.3. Detection Schemes Based on Position Automata

An often used method to determine whether a grammar is unambiguous is to generate an LR(k) parser. If there are no conflicts, then the grammar is unambiguous. This does not necessarily mean that if conflicts were found that the grammar is ambiguous, thus this is a conservative approximation.

An extension has been presented by Sylvain Schmitz in [17]. It has some similarities to the GLR extension for LR parsers: The processing is not aborted when

a conflict is encountered, but continued on best effort basis. Within a GLR parser, this means to run multiple parser instances in parallel. For an ambiguity detector it means to find out whether it is possible for more than one instance to reach an accept action.

The approach is not specific to LR(k) parsers. Instead, it works on grammar positions. A grammar position is the abstraction of the configuration of a parser and denotes its current position within a parse tree.

Definition 3.15 (Grammar item)
Let $G = (N, \Sigma, P, S)$ be a grammar and $A \to \gamma$ one if its productions. $A \to \alpha \cdot \beta$ is an *item* for every partition $\gamma = \alpha\beta$. It identifies a position on the right-hand-side of a production.

Definition 3.16 (Grammar position)
Let $G = (N, \Sigma, P, S)$ be a grammar. A *grammar position* in G is a stack of items of the form

$$\begin{aligned} S &\to \alpha_1 \cdot A_1 \beta_1 \\ A_1 &\to \alpha_2 \cdot A_2 \beta_2 \\ &\vdots \\ A_{n-1} &\to \alpha_{n-1} \cdot A_n \beta_{n-1} \\ A_n &\to \alpha_n \cdot \beta_n \end{aligned}$$

The number of items n is the *level* of the position. We call $A_n \to \alpha_n \cdot \beta_n$ the *topmost item*, and all other items *ancestors*.

For a better overview we introduce a "level 0" which contains $\cdot S$ and $S \cdot$. These mean "before the parse begins" and "after the parse has finished successfully", respectively.

Grammar positions can also be interpreted as states of an automaton (a *position automaton*). Then, "$\cdot S$" is the starting state and "$S \cdot$" the accepting state.

The following list describes the creation of outgoing transitions from a source state, depending on the position it represents (the "current" position). It starts at the state representing "$\cdot S$" and finishes at "$S \cdot$".

- Case 1: The symbol after the dot in the topmost item is a nonterminal A.
  For every production of the from $A \to \alpha$ create a new state by pushing $A \to \cdot\alpha$ on the stack of the current position. Add ϵ-transitions from the source state to all these newly created states.

- Case 2: The symbol following the dot is a terminal a.
  Create a new position by changing the topmost item of the current position from $\alpha \cdot a\beta$ to $\alpha a \cdot \beta$. Connect it by a transition that consumes a.

- Case 3: There is no symbol behind the dot and there is at least one ancestor. Create a new position by removing the topmost item from the stack of the current position. Change the item, which is now the topmost item, from $\alpha{\cdot}B\beta$ to $\alpha B{\cdot}\beta$. Connect it with an ϵ-transition.

- Case 4: There is no ancestor and the topmost item is "$S{\cdot}$".
  There is no successor state, because the end of the grammar has been reached. This position is the accepting state.

Note that every position in an position automaton is unique. If a position is created that already exists, both positions and states are merged.

When compared to an LR parser, case 2 conforms to a shift, case 3 is a reduce action and case 4 is equivalent to an accept action. Case 1 does not exist in an LR parser but is known as *closure* in LR parser generators.

Example 3.17
Consider Grammar 3.10, which accepts only one word ("`a b c`"). Figure 3.9 shows the position automaton for this grammar. Greyed items do not belong to the automaton but have been added for information purposes. The transitions with nonterminals show transitions that could exist, if nonterminals could be found directly in the input sequence. They will be used later in this section.

(1) S → A B
(2) A → a (3.10)
(3) B → b c

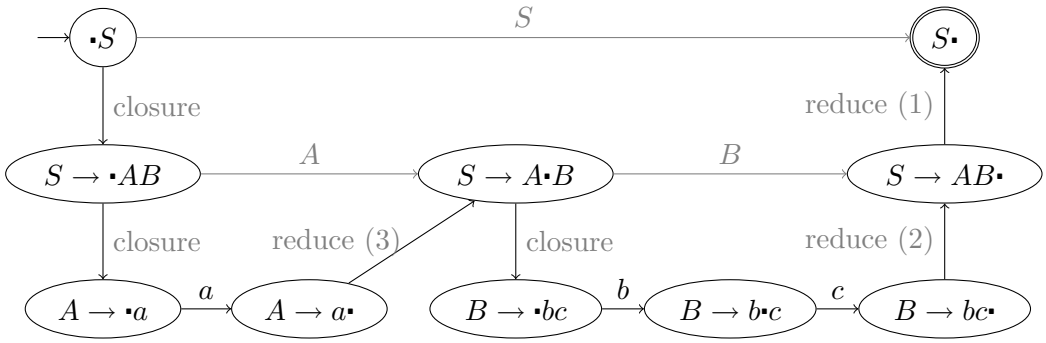

Figure 3.9.: Position automaton for grammar 3.10. The states are labeled using the topmost grammar item only.

Not every automaton built this way is a finite state machine because most context-free grammars allow an infinite recursion level and thus infinite number of states in its position automaton. Additionally, it not deterministic in general because case 1 (closure) allows ϵ-transitions to multiple productions. The other cases do not cause indeterminisms, as they do not allow any choices.

Every distinct path from the start state to the accepting state corresponds to a different parse tree. This is because at every closure there is a choice which production to choose for a nonterminal. The same choice is taken in a parse tree when a node is expanded by a number of child nodes of a chosen production (Figure 3.10). Hence, there is a one-to-one relationship between paths in a position automaton and the parse trees.

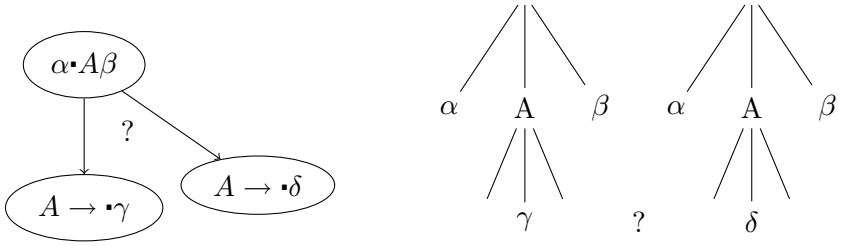

Figure 3.10.: Choices in an position automation and a parse tree.

A finite state machine can be made deterministic by using the powerset construction method ([20] explains this algorithm). However, this does not work with an infinite number of states. A solution is to define an equivalence relation $\equiv$ to map all states to a finite number of equivalence classes.

Example 3.18
Consider Grammar 3.11. Figure 3.11 shows the position automation of this grammar. The grey states adumbrate the infinite number of levels. Instead of repeating the same production at every level, the levels 2 and upwards are declared to be equivalent to the corresponding states at level 1. Note that the transitions to and from the merged states still exist, but have been redirected to the equivalent states.

$$\begin{array}{ll} (1) & \mathrm{S} \to \mathrm{a\ S\ b} \\ (2) & \mathrm{S} \to \mathrm{c} \end{array} \tag{3.11}$$

The original grammar accepts the language $\{a^n c b^n \mid n \geq 0\}$, which is a non-regular language. The new automaton is a finite state machine which accepts the language $\{a^n c b^m \mid n, m \geq 0\}$, a superset of the original language. The level of recursion has been "forgotten". The new language will be a regular superset for any grammar if the equivalence relation creates a finite number of equivalence classes. The finite state machine is called the *approximation automaton* of the context-free grammar. It can be used as the regular superset needed in section 3.2.3.

For later definitions of ambiguity approximations we need the following transformation on position automata:

1. Use an equivalence relation where all grammar positions with the same topmost item (i.e. same production and same placement of the dot) have their distinct equivalence classes.

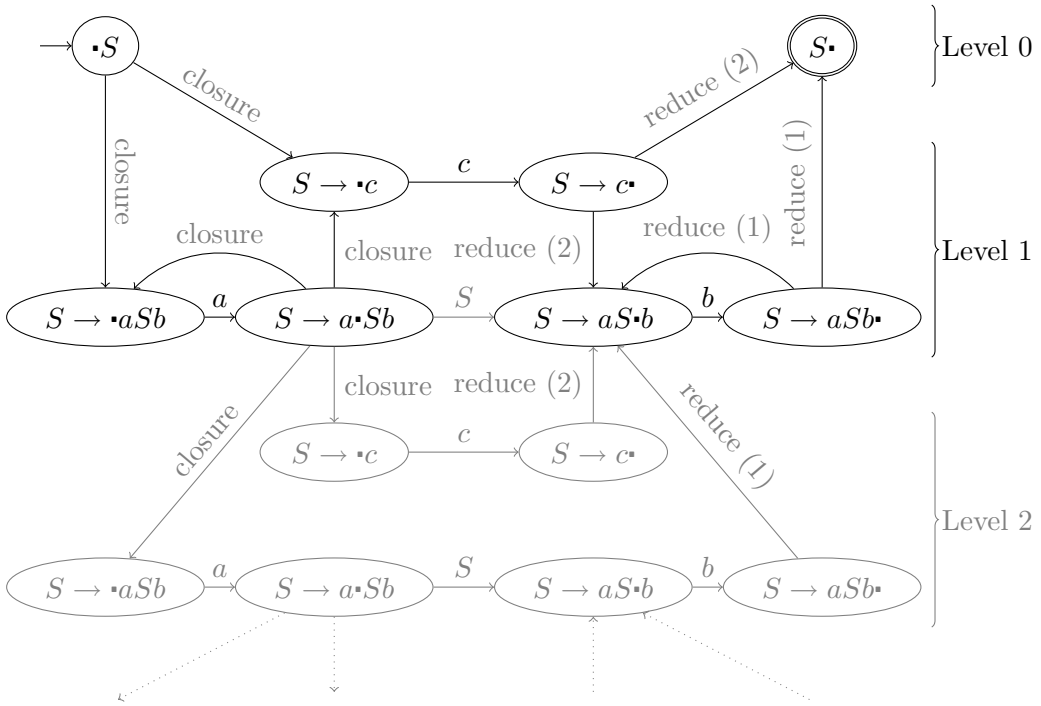

Figure 3.11.: Position automaton with equivalence classes for Grammar 3.11

2. Remove all reduce-transitions, but include all transitions that consume non-terminals, as adumbrated in the figures.

3. Determinize the automaton using the powerset construction method [20].

This automaton is known as the LR(0) automaton, which is used to compute the parsing table as in section 2.2. The reduce-transitions can be added again, such that this automaton recognises the same language as before the removal of the reduce-transitions. We call this finite state machine the *LR(0) approximation automaton*. It can be indeterministic again.

Example 3.19
Figure 3.12 shows the LR(0) automaton of Grammar 3.11.

The same construction is possible with LR(k) for any k. The set of possible lookahead terminals can be extracted from the ancestor items. In the grammar position automaton, the lookahead terminals can be computed by following all paths starting at the reduction state. Every path must be followed until k terminals have been consumed on the path. These are the lookahead terminals.

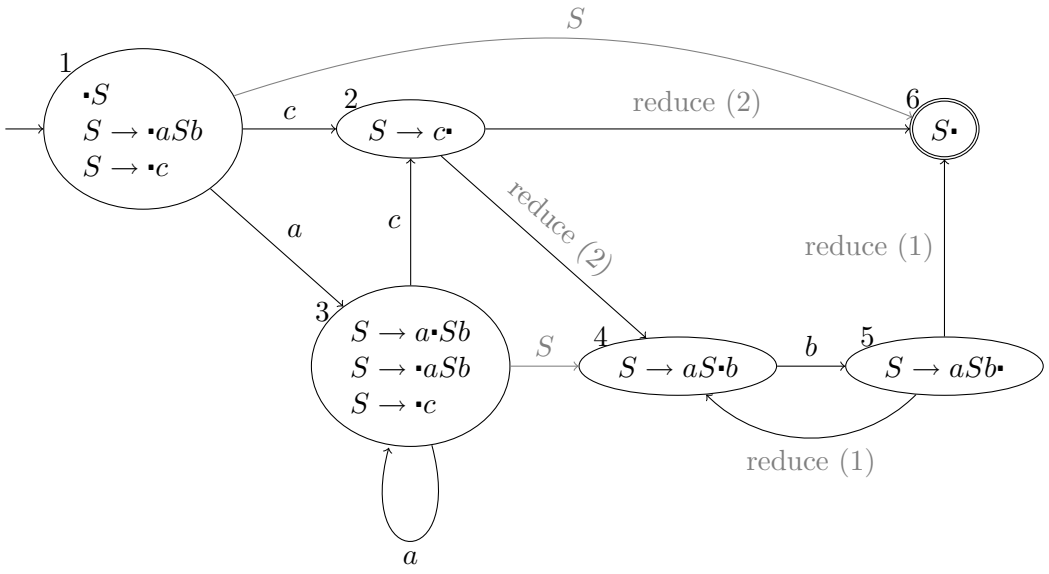

Figure 3.12.: The LR(0) approximation automaton of grammar 3.11. The states are labeled with the set of topmost grammar items which were joined during the automaton determinization.

3.3.1. Regular Unambiguity

The original question was whether a grammar is unambiguous. A conflict in a parsing table does not necessarily mean, that the grammar is ambiguous. Still, an ambiguous word with more than one parse tree is required. Because of this, we have to find such a word.

We also know that there is a one-to-one correspondence between parse trees and paths through the positions automaton (without equivalence classes). Hence, we have to find two different paths through the position automaton for the same word.

Paths in the position automaton for the same input can only split up at closure-transitions. Moreover, after the split-up, the paths must still reach an accepting state in order to map a complete parse tree. Hence, we can characterise ambiguous context-free grammars as in Corollary 3.20.

Corollary 3.20 (Ambiguities in position automata)
Let G be a grammar with an position automaton. G is ambiguous if and only if there are two different paths from the start state to an accepting state in this position automaton.

A problem is, that there can be infinite many positions in a position automaton. Therefore we apply an equivalence relation to the position automaton an get the definition of Regular Unambiguity.

There is a problem that not all equivalence relations are useful for a conservative ambiguity detection. For instance, assume we have a relation with only one equiv-

alence class. It is necessarily the start and the accepting state and there are no two different paths. Thus, every grammar would be detected as "unambiguous".

Therefore, we introduce bracketed grammars. The wanted property of a bracketed grammar is that different productions are kept distinct after applying any equivalence relation.

Definition 3.21 (Bracketed grammar)
A *bracketed grammar* is a modification of a context-free grammar, where every production begins with d_i and ends with r_i. i is a unique number of that production.

A d_i terminal makes closures in the position automaton explicit in the recognised language: Every closure transition is followed by a transition that consumes d_i. In the same way, r_i indicates a reduce: Every reduce-transition is preceded by a unique r_i-transition. A position automaton of any grammar can be modified to recognise the corresponding bracketed grammar by making the ϵ-transitions for closures and reductions consuming the corresponding d_i and r_i terminals.

A bracketed grammar is always unambiguous. This is because the d_i terminals uniquely define the production a nonterminals is expanded to. There is no more choice which production to take. In this sense, the r_i terminals are redundant since they contain the same information. Additionally, because every production has a constant length, the position of the r_i terminals in the recognises language can be derived from the d_i terminals. Therefore, the r_i terminals can be omitted without losing any information.

The same argument applies to the d_i terminals since a context-free grammar can also be analysed from right-to-left. Therefore, either the d_i or the r_i terminals can be omitted.

An ambiguity in a grammar can be observed in its bracketed grammar: There are two words in the bracketed grammar, which differ in the r_i terminals only. The different r_i terminals indicate different paths through the position automaton. We use this observation to define the term of *Regular Unambiguity*.

Definition 3.22 (Regular Unambiguity)
Let $\equiv$ be an equivalence relation on grammar position with a finite number of equivalence classes. Additionally, let G be a context-free grammar and G' its bracketed modification. We generate a finite state machine using the equivalence classes of $\equiv$ by applying it on the position automaton G'. G is *regular unambiguous* in respect of $\equiv$ (RU($\equiv$) for short), iff the recognised language of the finite state machine does not contain any two words that differ in the d_i and r_i symbols only.

We now have to prove that this results in a conservative ambiguity detection.

Theorem 3.23 (Regular Unambiguity $\Rightarrow$ unambiguity)
Every grammar, which is regularly unambiguous in respect of an arbitrary equivalence relation, is unambiguous as by Definition 3.1.

Proof. We show the contraposition "ambiguity $\Rightarrow$ regular ambiguity". That is, if

there are multiple paths for the same word, then the bracketed grammar recognises two words which differ in the r_i terminals only.

Let G be an ambiguous grammar with an ambiguous word w that has two different paths in G's position automaton. The start state of both paths are the same, so they split up somewhere on the path. The only transitions where this is possible are the closure-transitions. Following two different closure-transitions means to select two different productions.

Let G' be the bracketed grammar based on G. In the position automaton of G', the closure-transitions are explicit: they consume a d_i terminal and therefore part of the input sequence. The same applies for the r_i terminals at the end of the production. The two paths in the position automaton of G have corresponding paths in the position automaton of G'. We call the sequences of consumed terminals on these paths w_1' and w_2'. There first split of the paths are necessarily closure-transitions, which consume two different d_i-terminals. Hence, w_1' and w_2' are unequal at that position. All non-d_i and r_i terminals remain equal, so w_1' and w_2' differ in d_i and r_i terminals only.

After applying an equivalence relation $\equiv$ on the position automaton of G', the finite state machine recognises a superset of the original language of G'. In particular, this superset also contains w_1' and w_2' and therefore fulfils the definition of Regular Unambiguity in respect of $\equiv$. □

Without the equivalence relation, Regular Unambiguity would match the general ambiguity definition, but it would be uncomputable. An equivalence relation with a finite number of equivalence classes makes it computable, but may also show up more ambiguities than in the original position automaton. Hence, we call these *potential ambiguities*.

The computation can also operate on the original grammar without explicit brackets. However, the equivalence relation must keep different productions in different paths. In a bracketed grammar, this happens automatically, because the different d_i and r_i transitions cannot be joined to a single transition. The LR(k) approximation automaton does this by keeping the reduce-transitions distinct.

Example 3.24
Again consider the unambiguous Grammar 3.11 and its LR(0) approximation automaton in Figure 3.12. When interpreted as a bracketed grammar (with the reduce-transitions consuming r_i terminals), the automaton recognises the language $\{a^n c r_2 (b r_1)^m \mid n \geq 0, m \geq 0\}$ (the d_i transitions were omitted here). The position and number of the r_i terminals cannot be modified without also changing other terminals. Because of this, no two different words exist, which only differ in the r_i terminals.

When ambiguity is interpreted as different paths through the position automaton, one can see that there are no two different paths in the automaton in Figure 3.12 which consume the same sequence of terminals.

3.3.2. LR-Regular Unambiguity

We known that only shift/reduce conflicts and reduce/reduce conflicts between different productions can cause ambiguities. Hence, we can ignore other indeterminisms in the position automaton. This guarantees that any LR(k) grammar is found unambiguous when using the LR(k) item set (using the same or a greater k). The definition of Regular Unambiguity does not consider this, so we come up with a new definition of ambiguity approximation, *LR-Regular Unambiguity*.

In the LR(k) approximation automaton two grammar positions are equivalent if their topmost item (including the lookahead) is equal. The items have the format $(A \to \alpha{\cdot}\beta, w)$. w is the lookahead, a sequence of k terminals. We define $\mathbb{Q}$ as the set of items specific to a grammar. This set does not need to be determinised, because the mutually accessibility scheme operates on single items. $\mathbb{Q}$ also includes the items $({\cdot}S, EOF)$ and $(S{\cdot}, EOF)$, where EOF is a placeholder that marks the end of the input sequence. They identify the position before and after the start nonterminal.

Then we apply the *mutual accessibility scheme* on the problem. That is, we search for pairs of items that consume the same sequence of terminals. A conflict occurs when one of the items in the tuple executes a different action than the other (shifts or different reduce actions).

To make mutual accessibility work, we define five relations on tuple of items ($\mathbb{Q}^2$). $q \in \mathbb{Q}$ can be a any item.

mas : $((A \to \alpha{\cdot}a\beta, w), (B \to \gamma{\cdot}a\delta, v))$ mas $((A \to \alpha a{\cdot}\beta, w), (B \to \gamma a{\cdot}\delta, v))$

mad : $((A \to \alpha{\cdot}C\beta, w), q)$ mad $((C \to {\cdot}\gamma, w'), q)$
– or –
$(q, (A \to \alpha{\cdot}C\beta, w))$ mad $(q, (C \to {\cdot}\gamma, w'))$
In both cases, w' must be a valid beginning of β (the lookahead condition).

mar : $((C \to \alpha{\cdot}, w'), q)$ mar $((A \to \beta C{\cdot}\gamma, w), q)$
– or –
$(q, (C \to \alpha{\cdot}, w'))$ mar $(q, (A \to \beta C{\cdot}\gamma, w))$
Again, w' must be a valid beginning of γ (the lookahead condition).

maa : $((S{\cdot}, EOF), (S{\cdot}, EOF))$ maa $((S{\cdot}, EOF), (S{\cdot}, EOF))$

mac : $((C \to \alpha{\cdot}, w'), (D \to \beta{\cdot}, w'))$ mac $((A \to \gamma C{\cdot}\delta, w), (B \to \gamma' D{\cdot}\delta', w))$
$C \neq D \vee \alpha \neq \beta$ and w' must be a valid beginning of δ and δ'.
– or –
$((A \to \alpha{\cdot}w_1\beta, v), (C \to \gamma{\cdot}, w'))$ mac $((A \to \alpha{\cdot}w_1\beta, v), (B \to \delta C{\cdot}\delta', w))$
w' must be a valid beginning of $w_1\beta$ and δ', where w_1 is the first symbol of w (or any symbol, if the number of lookahead tokens is $k = 0$).
– or –
$((C \to \alpha{\cdot}, w'), (B \to \beta{\cdot}w_1\gamma, v))$ mac $((A \to \delta C{\cdot}\delta', w), (B \to \beta{\cdot}w_1\gamma, v))$
The same conditions as above apply.

The purpose of these relations can be described as follows.

mas (mutually accessible by shift)
Both items shift the same terminal.

mad (mutually accessible by closure)
This relation allows to enter a production of the next nonterminal. Since no symbols are consumed, this can happen on both elements of the tuple independently without altering the accessibility on the same input string

mar (mutually accessible by reduce)
Maps a reduction of one production. Again, no symbols are consumed so items can be reduced independently.

maa (mutually accepting)
This is a filter which ensures that the final tuple accepts the input.

mac (mutually accessible by conflict)
A relation, which identifies conflicts. The first condition maps a reduce/reduce conflict. The second identifies shift/reduce conflicts. For the sake of symmetry, the third identifies reduce/shift conflicts. Note that mac is a subset of mar * because one (shift/reduce) or two (reduce/reduce) applications of mar have the same effect.

Definition 3.25 (LR-Regular Unambiguity)
A grammar G is *LR-regular unambiguous* with k lookahead terminals iff

$$((\text{mas} \cup \text{mae} \cup \text{mar})^* \circ \text{mac} \circ (\text{mas} \cup \text{mae} \cup \text{mar})^* \circ \text{maa})(\bullet S, EOF; \bullet S, EOF) \neq \varnothing$$

The above equation is either empty or contains $(S\bullet, EOF; S\bullet, EOF)$ as the only element because of the maa filter. However, the information, which relations where needed from $(\bullet S, EOF; \bullet S, EOF)$ to $(S\bullet, EOF; S\bullet, EOF)$, can help the author of the grammar to eliminate the problem.

The definition of LR-Regular Unambiguity exactly matches the grammar class LR(Π) [22] where Π is a set of regular languages. In LR(Π)-parsers, these language are used as lookaheads instead of fixed-length sequences. LR-Regular Unambiguity just tests whether the languages of all actions, which can possibly execute at the same time have, disjoint lookaheads. In our definition of LR-Regular Unambiguity Π is the set of languages of the finite state machines that start after every reduce-transition of the LR(k) approximation automaton.

Example 3.26
Consider the ambiguous Grammar 3.12 and its LR(0) approximation automaton in Figure 3.13. There is one conflict in state 3 and the path

$$(1,1)\ \text{mas}\ (3,3)\ \text{mac}\ (2,4)\ \text{mar}\ (5,4)\ \text{mar}\ (5,5)\ \text{maa}\ (5,5)$$

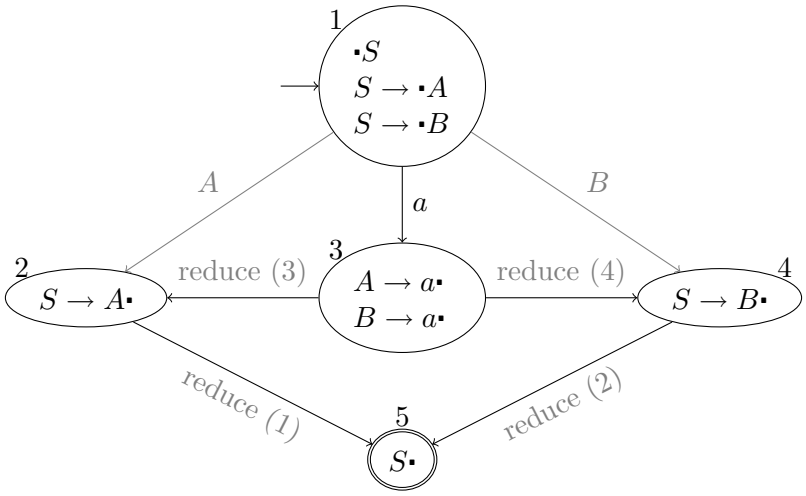

Figure 3.13.: LR(0) approximation automaton of Grammar 3.12

fulfils the definition of a potential ambiguity in the sense of Definition 3.25. Hence, this grammar is not LR-regular unambiguous when using the LR(0) equivalence classes.

(1) S → A
(2) S → B
(3) A → a
(4) B → a (3.12)

Example 3.27
An example of an grammar, which is LR-regular unambiguous with $k = 0$, but not regular unambiguous over LR(0), is Grammar 3.13 (taken from [17]).

(1) S → a A a
(2) S → b A a
(3) A → c (3.13)

It is LR-Regular Unambiguous because these is neither a shift/reduce-conflict nor a reduce/reduce-conflict in its LR(0) approximation automaton (Figure 3.14).

However, the two paths

$$1, 2, 6, 3, 4, 9$$
$$1, 2, 6, 7, 8, 9$$

both consume the sequence "`a c a`". In bracketed notation (again without the d_i terminals) these paths correspond to the sequences "`a c` r_3 `a` r_1" and "`a c` r_3 `a` r_2", which differ only in the last terminals r_1 and r_2. Hence, this grammar is not

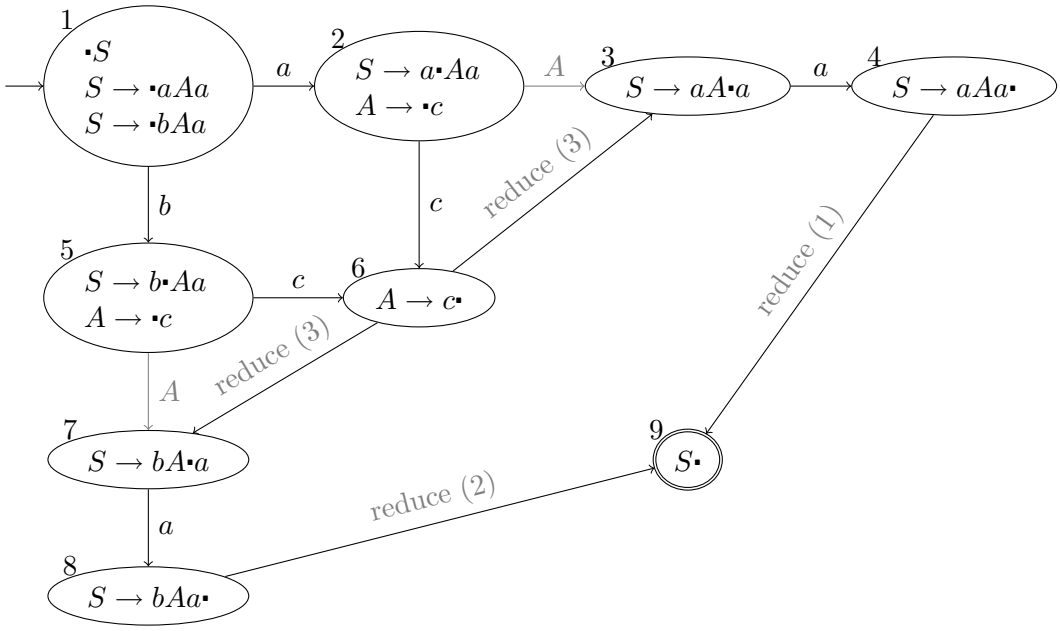

Figure 3.14.: LR(0) approximation automaton of grammar 3.13

regular unambiguous.

Example 3.28
Grammar 3.14 is not LR(k) for any k, but unambiguous and can be identified correctly by regular and LR-Regular Unambiguity. It is not LR(k) because the only difference between production (1) and (2) is the last terminal, but there can be any number of c's between the reduction of nonterminal A or B and the last token. An unbounded lookahead is necessary.

The LR(0) approximation automaton can be seen in Figure 3.15. One can see easily that there are no two different paths to state 11 because the two existing "lanes" on the top and on the right require either a or b to be the last terminal. Once they passed state 6, there is no more interaction between those lanes. There is no single input sequence which follows both lanes. Therefore this grammar is regular unambiguous, which also implies LR-Regular Unambiguity.

(1) S → A C a
(2) S → B D b
(3) A → c
(4) B → c
(5) C → C c
(6) C → ϵ
(7) D → D c
(8) D → ϵ

(3.14)

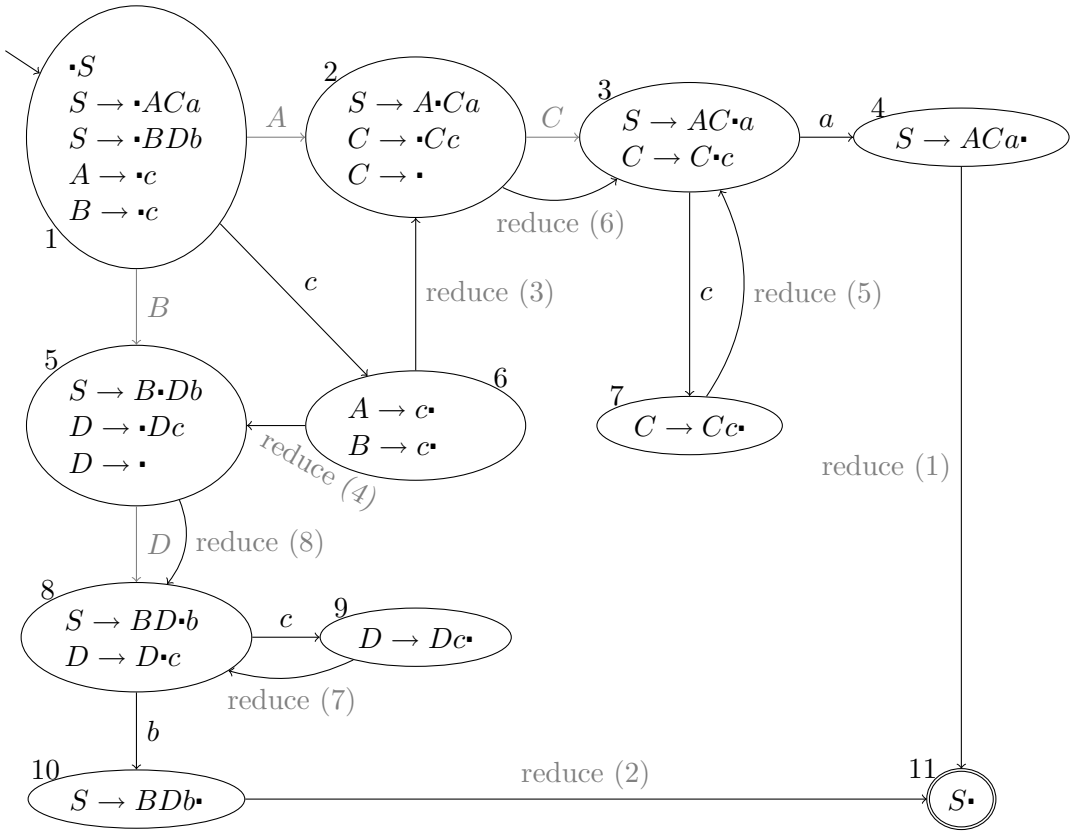

Figure 3.15.: LR(0) approximation automaton of grammar 3.14

3.3.3. Noncanonical Unambiguity

Another extension has been presented by Sylvain Schmitz in the same article [17]. It avoids to follow closure-transitions whenever possible and shifts the expected nonterminal instead. The nonterminal is a placeholder for the language it represents and therefore closer than any approximation can be.

The only reason why to take a detour starting at the closure-transition and ending at the reduce-transition of a production instead the more direct path over the nonterminal is because the path may contain a conflict. If the shortcut over the nonterminal is taken then the conflict is not detected. Giving priority to the nonterminals avoids additional paths that only exist because the finite state machine is a superset of the original language. The technique has been named *Noncanonical Unambiguity* (or NU(≡) for short).

A similar mutual accessibility scheme like the one presented for LR-Regular Unambiguity can be developed for Noncanonical Unambiguity. In order fo find conflicts, the closure-transitions have to be followed, but the corresponding reduce-transition is only valid if a conflict has been found in this or any subordinate pro-

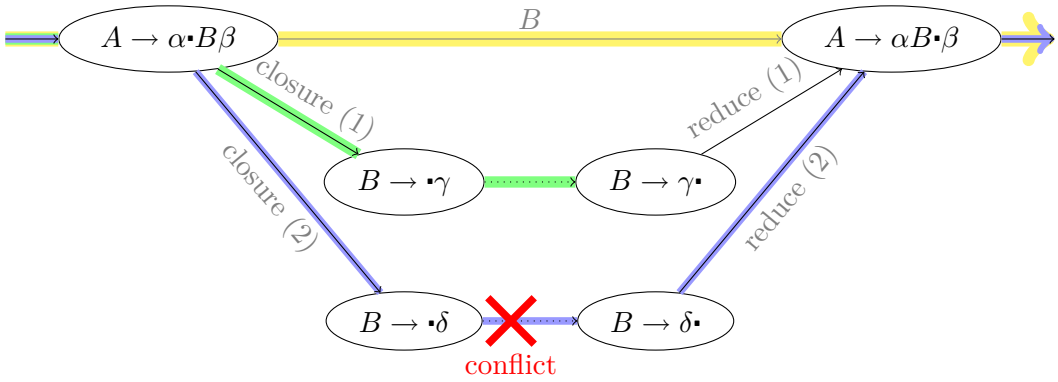

Figure 3.16.: Schema for Noncanonical Unambiguity. The yellow path is the shortcut over the nonterminal. The green path is a dead end because it does not contain any conflict. The blue path encounters a conflict and therefore can continue.

duction. If there is no conflict, then the path leads into a dead end is not considered any further. In contrast, transitions that shift nonterminals are always valid, but the conflict is still considered by the path that follows the closure. Figure 3.16 illustrates the principle.

Noncanonical Unambiguity is a conservative ambiguity detection scheme and detects more unambiguous grammars than regular and LR-Regular Unambiguity do. This is shown in [16].

Example 3.29
Grammar 3.15 illustrates the advantage of Noncanonical Unambiguity over LR-Regular Unambiguity. Its position automaton is shown in Figure 3.17. There is a pair of paths that fulfil the conditions of a potential LR-Regular ambiguity:

$$1, 5, 6, 7, 8, 9, 3, 4, 11$$
$$1, 5, 6, 7, 8, 9, 8, 9, 10, 11$$

For Noncanonical Unambiguity it is necessary that the sub-paths for the nonterminal C must traverse a conflict. C is not directly involved in any conflict so any path has to go over the C-transitions $(6, 10)$ or $(2, 3)$. There are no two different paths that consume the same word under this condition.

Note that Grammar 3.15 is a simplification of the grammar file `s5.con` mentioned in the evaluation chapter. In the simplified grammar there are two additional shift/reduce-conflicts in state 2 and 6 when using the LR(0) item set. The LR(1) item set is necessary to make it detected as Noncanonically Unambiguous by our implementation in the next chapter.

(1) S → A C b
(2) S → B C
(3) A → a
(4) B → a
(5) C → c C b
(6) C → ϵ

(3.15)

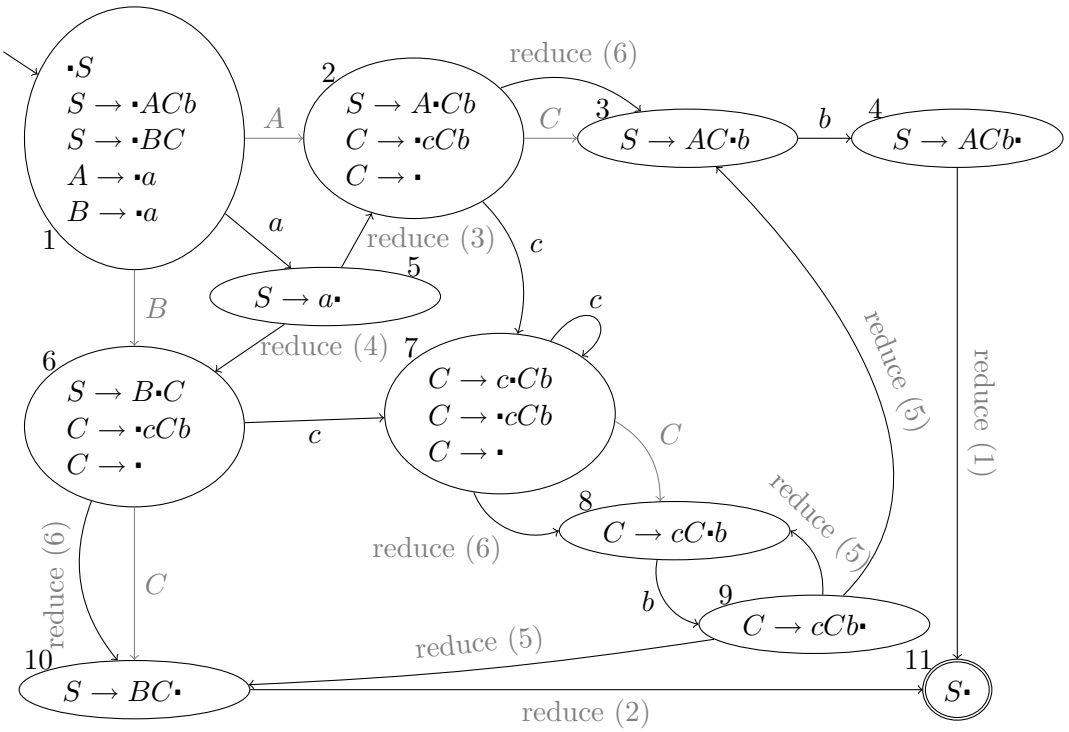

Figure 3.17.: The LR(0) approximation automaton of Grammar 3.15.

4. Design, Implementation & Integration

The theory of ambiguity detection has been explained in the previous chapters. We can now apply this knowledge practically in an implementation for the Eli system. We will call the implementation of the ACLA technique "*grambiguity*" and the Noncanonical Unambiguity package "*bisonamb*".

We will start by thinking about a design for the *grambiguity* application followed by a discussion of its implementation. Instead of also implementing the LR-regular and Noncanonical Unambiguity technique, we will use the implementation by Sylvain Schmitz [15]. Then we integrate both into the Eli system.

4.1. Design

A reference implementation by Anders Møller for the ACLA method exists [11], but it is written Java. The Eli system does not support Java so we need to write a new implementation to integrate it into the Eli system.

A software design is needed for the implementation. It must embrace a framework for finite state machines, context-free grammars and several algorithms.

Terms like "state", "nonterminal", "terminal", etc. lend themselves to be modelled in an object-oriented way. They can be separated into those related to automata (like states) and related to grammars (like nonterminals).

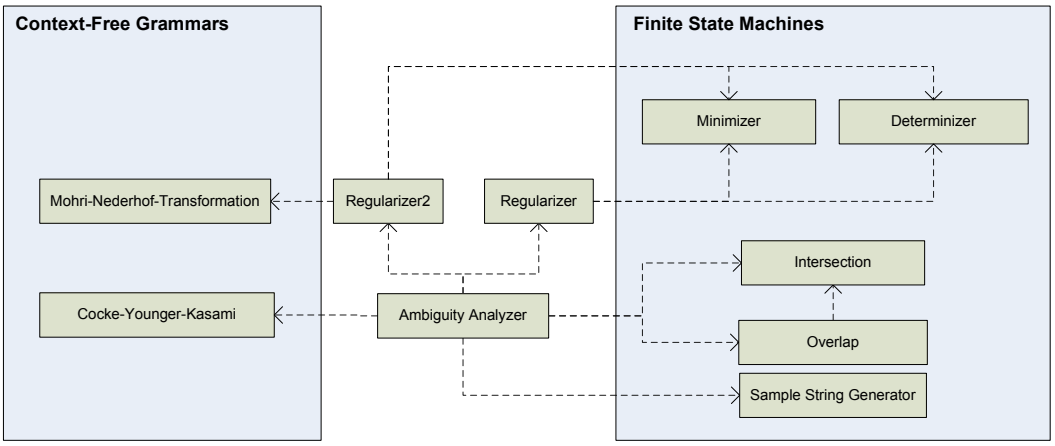

Figure 4.1.: Algorithms and their dependencies in *grambiguity*

The algorithms and their relationship to each other are shown in Figure 4.1. Algorithms that operate on context-free grammars are shown on the left, whereas algorithms on automata are on the right. Algorithms that work with both are in the middle of both boxes.

The class structure of the most important classes can be seen in Figure 4.2. Associations (except derivations) from or to the Automata or Grammar domain are not shown in the figure because of their numerousness.

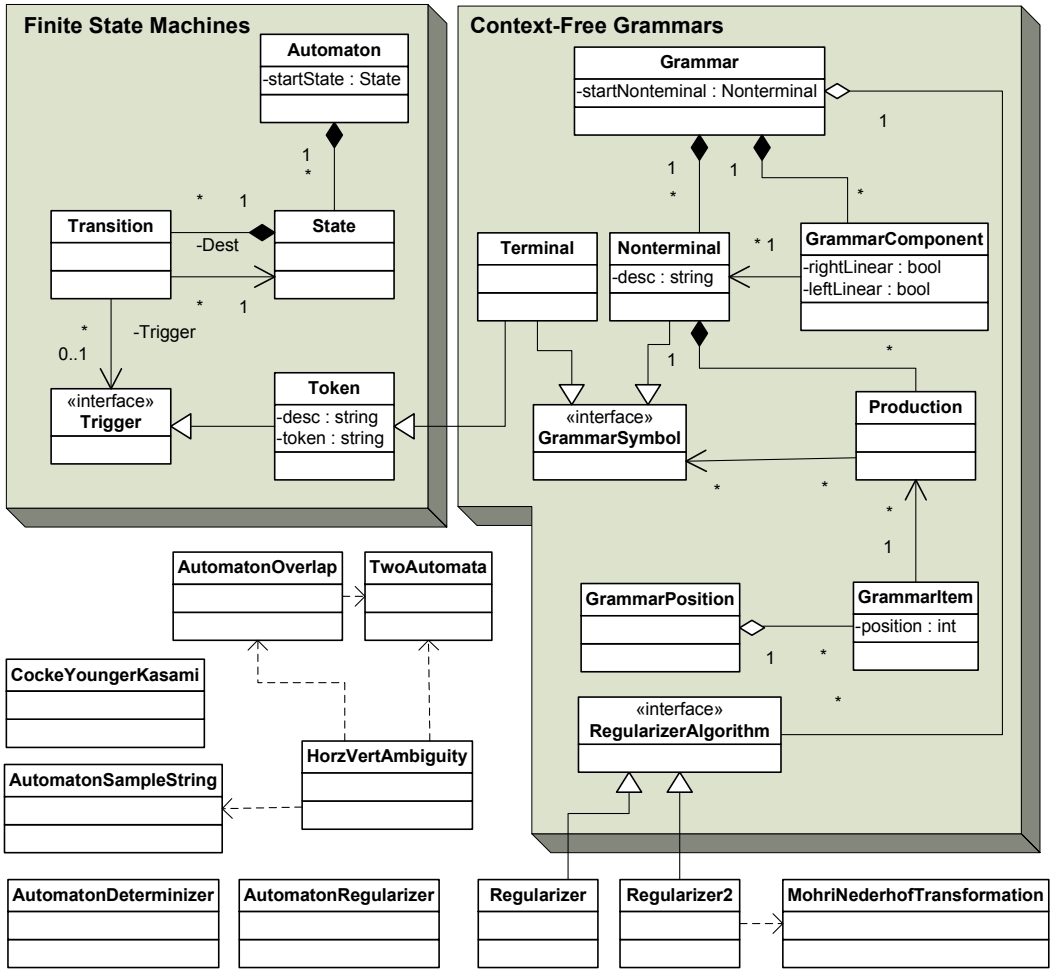

Figure 4.2.: UML class diagram for *grambiguity*

A *Trigger* is the condition that makes a transition traversable. In this context the conditions are the input symbol consumed when traversing the condition. The symbol is often called a *Token*. Token classes are global instances, which are instantiated only once per symbol. A different instance means a different Token, although their description may be equal.

To simplify the interaction with automata and grammars, *Terminal* is derived

from *Token*. This allows using Terminal classes as triggers in automata. The drawback is that this makes the domain of context-free grammar dependent on the domain of automata although these are independent concepts.

RegularizerAlgorithm is the abstract base class for both grammar approximation algorithms. "Regularizer" uses grammar positions and equivalence classes while "Regularizer2" processes the output of the Mohri-Nederhof Transformation. Which one is used is determined by assigning an instance of one of both to the Grammar class.

The algorithms *AutomatonMinimizer* and *AutomatonDeterminizer* are not used directly, but indirectly over the grammar class. Whenever a grammar is approximated by an automaton, it gets optimised by both algorithms.

4.1.1. The Algorithms

Here the algorithms used for the Ambiguity Checking by Language Approximation technique are described briefly. We start at the algorithms on automata, continue with the regular approximation algorithms and finish with the main ambiguity analyzer.

Automaton Determinizer

The determinizer converts a nondeterministic finite automaton to a deterministic finite automaton. It uses the powerset construction algorithm. The book [20] contains a description of the powerset construction.

Automaton Minimizer

This algorithm, also known as table-filling algorithm, identifies equivalent states of an automaton and merges them. If the automaton is deterministic, the output automaton has a minimal number of states. It also works on nondeterministic automata, but the result is not guaranteed to be minimal. A more detailed description can be found in [6].

Intersection Operator

The intersection of the language of two automata is computed similarly to the mutual accessibility scheme. The main difference is, that the tuples contain state from two different automata. Again, tuples are discovered, which are reachable by the same input string. A state becomes an accepting state, iff both states are accepting states. The output of this algorithm is the whole automaton of tuples as states and possible transitions between them.

Overlap Operator

This computes the set of overlapping words $(X \between Y)$ as in Definition 3.3. This set is a regular language; consequently it returns a finite state machine. The computation of this algorithm has been described in section 3.2.3.

First, it builds the automaton of equitations 3.7, 3.8 and 3.9. Their intersection is computed by executing the previous algorithm two times.

Sample String Generator

The sample string generator finds the shortest word which is recognised by an automaton, if any.

This is a breath-first search through the automaton: New states are explored beginning at the start state until an accepting state is found. Besides, the sequence of consumed terminals is stored and returned.

Regularizer

The basics of this algorithm have already been described in section 3.3. We can think of a context-free grammar with infinite recursion as an automaton with an infinite number of states. What has to be found is an equivalence relation that maps states which are "too deep" in recursion to states of a lower level.

In addition to the grammar items in productions, we also add items before ($\blacksquare A$) and behind ($A\blacksquare$) a nonterminal. We refer to such items as *nonterminal items* (All others are production items). This simplifies equivalence rule number 2 (described later in this section).

The rules of section 3.3 are modified for this change. The first grammar position is an item $\blacksquare S$ without ancestors. Note that $B \to \blacksquare A$ is a position within a production of B whereas $\blacksquare A$ is the position before a nonterminal. These are different grammar positions. The same applies for $B \to A\blacksquare$ and $A\blacksquare$.

1. Case 1a (Enter): On a topmost item $B \to \alpha\blacksquare A\beta$
   Push $\blacksquare A$ to the grammar position stack. Add an ϵ-transition to this state.

2. Case 1b (Closure): On a topmost item $\blacksquare A$
   Push an item $A \to \blacksquare\alpha$, where $A \to \blacksquare\alpha$ is a production of A on the grammar position stack. Do this for every production with A on the left-hand-side. Connect all by ϵ-transitions to these states.

3. Case 2 (Shift): On a topmost item $B \to \alpha\blacksquare a\beta$
   Replace the topmost item by $B \to \alpha a\blacksquare\beta$. Add a transition consuming a to this new position.

4. Case 3a (Finish): On a topmost item $A \to \alpha\blacksquare$
   Remove this topmost item from the grammar position. Add an ϵ-transition to the new state.

5. Case 3b (Reduce): On a topmost item $A\bullet$, which is not the only item in the grammar position
   Pop this item off the stack. Now the new topmost item has the format $B \to \alpha\bullet A\beta$. Replace it by $B \to \alpha A\bullet\beta$.

6. Case 4 (Accept): On item $S\bullet$, which is the only item in the grammar position
   Set this state as the accepting state of the automaton.

These changes do not alter the language which is recognised by the automaton since it only adds states with ϵ-transitions. It adds more structure to the automaton and makes it easier to apply an equivalence relation. We consider positions equivalent if they are nested "too deep".

The recursion can be considered as too deep if an item appears multiple times in the ancestor list. Whenever a new grammar position is created, it is compared to all grammar positions created before. If one is found to be equivalent, the former is reused instead.

There are several choices for equivalence relations. Here is a selection of possibilities:

1. Both grammar positions are normalised by removing topmost items until every production item appears only once. The positions are equivalent, iff the normalised positions are equal.

2. Both grammar positions are normalised by removing topmost items until every nonterminal item appears only once. The positions are equivalent, iff the normalised positions are equal.

3. Two grammar positions are equivalent, if their topmost items are equal.

There is only a finite number of items so it will always result in a finite state automaton. Choice 2 is more inaccurate than choice 1 but will result in less states. The same applies when comparing choice 2 and 3. Choice 3 results in the LR(0) approximation automaton. For this implementation we choose possibility 2 as a trade-off between number of states and accuracy.

Transitions, which start or end at a grammar positions that do not appear as states in the finite state machine, must be redirected to the equivalent state.

The grammar positions are stored in a dictionary that maps to the automaton states they correspond to and in a list of positions that have not been processed yet. When a new grammar position is referenced by a transition, its state is created and the position is added to the work-list. The algorithm continues until the work-list is empty.

Example 4.1
The automaton in Figure 4.3 is the direct output of the implementation of this algorithm when using Grammar 4.1 as input. The output uses a different convention than the other finite state machine in this thesis. The start state is displayed as a

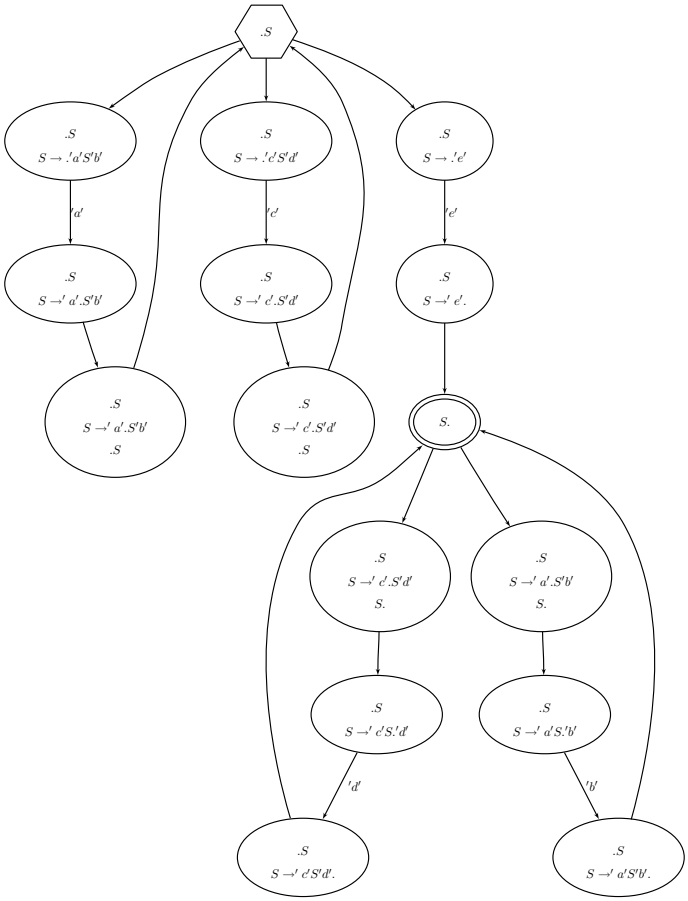

Figure 4.3.: Approximation automation of Grammar 4.1

hexagon and the states are labelled using the complete grammar position, not only its topmost item.

$$\begin{aligned} &S \rightarrow a\ S\ b \\ &S \rightarrow c\ S\ d \\ &S \rightarrow e \end{aligned} \tag{4.1}$$

After determinizing and minimizing the automaton 4.3, the automaton looks like in figure 4.4.

Mohri-Nederhof Transformation

This is the algorithm as described in [10]. It converts any context-free grammar into a strongly regular context-free grammar, i.e. a regular grammar described using nonterminals and productions as in Definition 2.1. It can be converted to an equivalent finite state machine without changing the recognised language. The

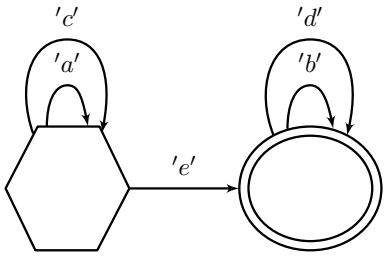

Figure 4.4.: Equivalent minimal automaton of the automata in Figure 4.3 and 4.5

converted grammar is a superset of the original grammar.

Let $G = (N, \Sigma, P, S)$ be the grammar to transform. The transformation starts by identifying the grammar's components. This is done by first determining which nonterminal can be the child of another nonterminal. The grammar's components are the strongly connected nonterminals of this relation. They are also called mutually recursive nonterminals.

If all nonterminals of such a component have left-linear productions, then these do not need to be processed any further, because this part of the grammar is already strongly regular. The same applies if all productions are right-linear.

For all other components the productions must be replaced. Let $M \subseteq N$ be a set of mutually recursive nonterminals. For all nonterminals $A \in M$ we add a new nonterminals A' to the grammar and remove all productions of the form $A \to \alpha_0 B_1 \alpha_1 \dots B_n \alpha_n$, where $n \geq 0$, $A, B_1, \dots, B_n \in M$, $\alpha_0, \dots, \alpha_n \in (V \setminus M)^*$. As a replacement, the productions

$$A' \to \epsilon$$

$$A \to \alpha_0 B_1$$

$$B'_1 \to \alpha_1 B_2$$

$$\vdots$$

$$B'_{n-1} \to \alpha_{n-1} B_n$$

$$B'_n \to \alpha_n A'$$

are added to the grammar.

After this transformation all productions belonging to this component are right-linear. This property guarantees that the grammar is strongly-regular. Note that we allow nonterminals $\in (V \setminus N)$ to occur in $\alpha_1, \dots, \alpha_n$. These do not compromise the property of being strongly-regular because these themselves are placeholders for regular grammars (as long as their components are themselves strongly-regular).

Example 4.2
Grammar 4.2 is the transformed version of Grammar 4.1.

$$
\begin{aligned}
&\text{S'} \rightarrow \epsilon \\
&\text{S} \ \rightarrow \text{a S} \\
&\text{S'} \rightarrow \text{b S'} \\
&\text{S} \ \rightarrow \text{c S} \\
&\text{S'} \rightarrow \text{d S'} \\
&\text{S} \ \rightarrow \text{e S'}
\end{aligned}
\tag{4.2}
$$

Regularizer2

The second algorithm for approximating context-free grammars only works with grammars with only left- or right linear components. Hence, an algorithm like the Mohri-Nederhof Transformation must transform it into such a grammar if it contains other non-linear components.

The automaton is created without the information which nonterminal is the start nonterminal. When the creation has been finished, a start and an accepting state is chosen, depending on which nonterminal is the start nonterminal of the grammar.

First, a start state ("Start A" for a nonterminal named A), a "component" state ("A") and an finish state ("Finish A") is created for every nonterminal. Then, every component is processed independently.

For every right-linear component, create a common finish state for M ("Exit M") and apply the following rules.

- For every production $A \rightarrow \alpha$, $\alpha \in (V \setminus M)^*$
  Add a path from "A" to "Exit M" that consumes α.

- For every production $A \rightarrow \alpha B$, $\alpha \in (V \setminus M)^*$, $B \in M$
  Add a path that consumes α from "A" to "B".

- For every nonterminal $A \in M$
  Add an ϵ-transition from "Start A" to "A" and another ϵ-transition from "Exit M" to "Finish A"

Similar, for every left-linear component, create a common start state for M ("Begin M") and apply these rules.

- For every production $A \rightarrow \alpha$, $\alpha \in (V \setminus M)^*$
  Add a path that consumes α from "Begin M" to "A".

- For every production $A \rightarrow B\alpha$, $\alpha \in (V \setminus M)^*$, $B \in M$
  Add a path from "B" to "A" that consumes α.

- For every nonterminal $A \in M$
  Add an ϵ-transition from "Start A" to "Begin M" and another ϵ-transition from "A" to "Finish A"

If the path that consumes an $\alpha \in (V \setminus M)^*$ contains a nonterminal $C \in N \setminus M$, the path is redirected to C's start state ("Start C") and continued from C's finish state ("Finish C") by appropriate ϵ-transitions. This way this nonterminal is included into the path by using the existing definition that already exists in this automaton.

The disadvantage of this method is that the automaton recognises a superset of the language recognised by the strongly linear grammar. When every component is put into a graph and a directed transition is added from a component to every component it references, then we get a directed acyclic graph. But to create an automaton, which exactly matches the strongly regular grammar, the reference graph must be a tree. One can copy components, which are referenced multiple times, until the graph is a tree. This was avoided because it can result in an exponential number of states.

When the actual start nonterminal is known, it can be extracted from this automaton. The start state is set "Start S" and "Finish S" becomes the only accepting state. All states that belong to an unreachable component should be removed from the automaton. This avoids paths through productions that shouldn't be accessible.

Example 4.3
Grammar 4.1, which was introduced and discussed for the other grammar approximation algorithm in Example 4.1, can also be approximated by applying a Mohri-Nederhof-Transformation and Regularizer2. The transformed Grammar 4.2 is converted to the finite state machine in Figure 4.5.

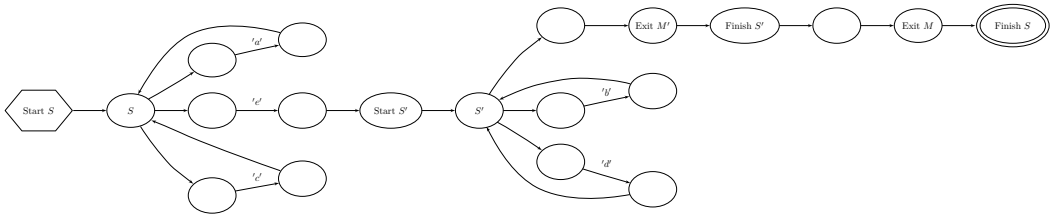

Figure 4.5.: Approximation automaton of Grammar 4.1 by Regularizer2

Cocker-Younger-Kasami

In the context of the ambiguity analyzer, this algorithm is not used to find a parse tree of a sequence, but how many parse trees a sequence has.

As the traditional Cocker-Younger-Kasami algorithm it uses dynamic programming. The difference is that it stores the amount of possible parse trees of subsequences for every nonterminal. These values are used to compute the number of possibilities for longer substrings. The algorithm finishes if the number of possibilities of the start nonterminal for the whole input string has been computed.

Ambiguity Analyzer

The main ambiguity detection works in a brute-force manner: Check every position within every production for horizontal ambiguity and check every combination of two productions of the same nonterminal for vertical ambiguity.

Horizontal ambiguity is checked by generating finite state automata for both sides of the production. If the overlap operator returns an automaton which recognises a non-empty language, then the sample string generator will find a sequence. In this case a potential horizontal ambiguity has been found.

Vertical ambiguity is checked similarly. Finite state machines are generated for both productions. If the sample string generator finds a sequence in their intersection, a potential vertical ambiguity has been found.

Both, vertical and horizontal ambiguities, compute sample sequences that are not necessarily words of the grammar because the check operates on a language superset. The modified Cocker-Younger-Kasami algorithm tries to find how many derivations these sample strings really have. If it has at least two, then the potential ambiguity becomes a confirmed ambiguity.

4.2. Implementation

The implementation is written in C++ and can be compiled using the GNU C++ Compiler or Microsoft Visual Studio 2008. Table 4.6 contains an overview on all files of this implementation.

File	Purpose
`Automaton.h`, `Automaton.cxx`	Automaton data structures
`Grammar.h`, `Grammar.cxx`	Grammar data structures
`AbstractGrammarSyntaxTree.h`, `AbstractGrammarSyntaxTree.c`	Building an abstract syntax tree of a grammar file
`Ambiguity.h`, `Ambiguity.cxx`	Invocation and display of results
`AutomatonDeteminizer.h`, `AutomatonDeterminizer.cxx`	Determinization of nondeterministic automata using the powerset construction method
`AutomatonMinimizer.h`, `AutomatonMinimizer.cxx`	Minimizes the number of states of an automaton
`AutomatonOverlap.h`, `AutomatonOverlap.cxx`	Implementation of the overlap operator
`AutomatonSampleString.h`, `AutomatonSampleString.cxx`	Finding the shortest accepted sequence of an automaton
`CockeYoungerKasami.h`, `CockeYoungerKasami.cxx`	Implementation of the variant of the Cocker-Younger-Kasami algorithm
`BasicCollections.h`, `BasicVector.h`, `BasicSet.h`, `BasicDictionary.h`	Supporting data structures

File	Purpose
`HorzVertAmbiguity.h`, `HorzVertAmbiguity.cxx`	Implementation of the ACLA algorithm
`Regularizer.h`, `Regularizer.cxx`	Approximation of context-free grammars with position automata
`GrammarUnfolding.h`, `GrammarUnfolding.cxx`	Creates clones of grammar structures
`MohriNederhofTransformation.h`, `MohriNederhofTransformation.cxx`	The algorithm for the Mohri-Nederhof-Transformation
`Regularizer2.h`, `Regularizer2.cxx`	Approximation of left- and right-linear context-free grammars
`SmartPtr.h`	Additional smart pointers
`Tests.h`, `Tests.cxx`	Unit tests
`TwoAutomata.h`, `TwoAutomata.cxx`	Computation of the intersection and union of two regular grammars

Table 4.6.: Source files for *grambiguity*

The files `Automaton.h` and `Automaton.cxx` contain the implementation of the automaton-related classes. Similarly `Grammar.h` and `Grammar.cxx` implement the domain of context-free grammars. Most algorithms are encapsulated into separate files. This was done because these mostly declare object fields that should not appear in algorithm-independent data structures like "Automaton" and "Grammar".

The files `BasicCollections.h`, `BasicVector.h`, `BasicSet.h`, `BasicDictionary.h` and `SmartPtr.h` are general helpers. For instance, `BasicCollections.h` (and its included files) include wrappers for STL template classes. Although the STL is itself a high level library, its usage requires much boiler plate code (code for standard situations that cannot be avoided). This implementation makes heavy use of a `foreach` macro and thus makes it unnecessary to handle with STL iterators directly. The macro is similar to the `BOOST_FOREACH` macro found in the BOOST C++ library. However, use of a heavyweight library like BOOST was discouraged.

The standard template library (STL) only offers binary tree-based sets and dictionaries. Hash based version only exist as proprietary extensions like in Microsoft Visual Studio. Therefore only the tree based versions can be used by the collections wrappers. An alternative is the *IdObjDictionary* class. It uses numeric identifiers to find a unique location in an array. For instance this is possible foe *State* object, whose identifier is assigned and managed by its parent automaton.

`SmartPtr.h` implements a smart pointer with intrusive reference counting (refcount_ptr). It is faster than non-intrusive reference counting smart pointers, but the objects need to be prepared for this. This can be done by deriving from "refcount_obj".

Additional helper classes are a smart pointer for fixed-size arrays (array_ptr) and a fixed-size two dimensional array called *Matrix*.

4.3. Integration

Eli is built on top of *Odin*. Odin describes itself as a replacement for *Make* [12]. *Make* reads its configuration from a file named `Makefile` whereas Odin uses an `Odinfile`. A `Makefile` describes the sequence of shell commands which create a file. In contrast, an `Odinfile` describes the source files and the destination format. The steps needed (preprocessor, compiler, linker, ...) are determined automatically. Make cannot do this by itself but additional tools like *autotools* help to generate `Makefile`s which describe the toolchain.

The core of Eli is an own collection of components (called packages) for the Odin system. Therefore Eli uses the same commands, syntax and help system as Odin does. The Odin system is described in [4]. See Figure 4.7 for the components Eli consists of.

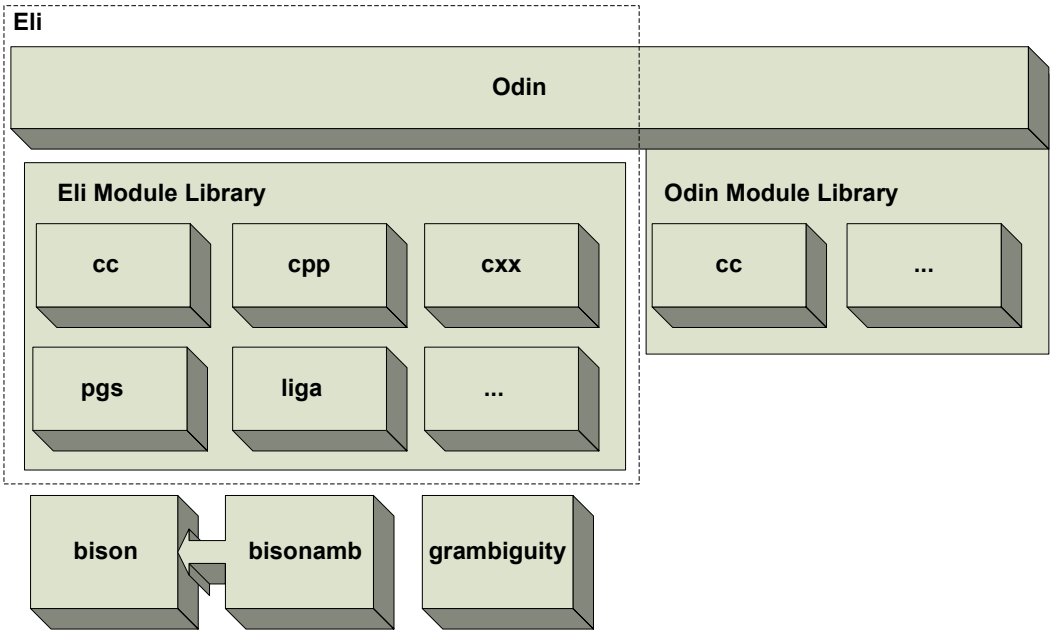

Figure 4.7.: Eli's architecture

We will extend the system by adding two packages "bisonamb" and "grambiguity". The "bisonamb" additionally requires the "bison" package.

4.3.1. Installation of Eli

The Eli system can be downloaded at `http://sourceforge.net/projects/eli-project` or checked out from the CVS repository at `eli-project.cvs.sourceforge.net:/cvsroot/eli-project`. Its file system looks like below. Only the most important files and directories are shown.

```
/ (Eli root directory)
   configure.in, configure, Makefile.in
   Odin/
      pkg/
         cc/
         cxx/
         ...
   Eli/
      configure.in, configure, Makefile.in
      pkg/
         Makefile.in, PKGLST
         cc/
         cxx/
         pgs/
         bison/
         bisonamb/
            Makefile.in, bisonamb.dg, version, bisonAmbiguity.sh
            schmitz/
               ...
         grambiguity/
            Makefile.in, grambiguity.dg, version, grambiguity.sh,
            Odinfile, grambiguity.specs, grambiguity.clp,
            pgram.gla, pgram.con, pgram.lido,
            ...
```

Odin is included in the `Odin/` directory and uses itself as build system. Eli is included in the `Eli/` directory and uses GNU Make as build system. Some parts also require Odin. The standard GNU Make process consists of 3 phases. Before starting this, `autoconf` should be executed if `configure.in` has been changed to generate the configure-scripts. The command `./symlinks` does this for all directories where it is necessary.

At first, the user executes `./configure` in the root directory. This generates `Makefile` from `Makefile.in` in all necessary subdirectories. Additionally. it will call `./configure` for included third-party components.

Second, the user calls `make` (equivalent to `make all`) in the root directory. Because Eli needs Odin to execute, it will install Odin first. Then it compiles the source files by executing the `Makefile`s. Each package is built one after each other and may use the functionality of the packages built before them. The order is determined by `Eli/pkg/Makefile.in`.

Third, `make install` copies the executables into the `bin/` directory of the system and the packages of Odin and Eli into the system's `lib/` directory.

Then `eli -R` needs to be executed to register all installed packages into a cache in ∼`.ODIN/`. This is done automatically if the cache does not exist yet. The packages to be loaded are stored in `PKGLST`. This is not a part of the Make build process, but is the preparation of the start of Eli.

`eli -R` also opens a command prompt where the full functionality can be used, for example the ":ambiguity" and ":grambiguity" derivations.

The build process has been tested on Ubuntu Linux 7.10 with automake 1.10.1, autoconf 2.61, GNU Make 3.81 and gcc 4.1. Additionally, it works using Windows XP with Cygwin 1.5.25-11, automake 1.10.1-1, autoconf 2.61, make 3.81, gcc 3.4.4, GNU Awk 3.1.6, file 4.21 and GNU diffutils 2.8.7.

4.3.2. Introduction to Odin/Eli Packages

Odin is organised in packages. There are packages that call the preprocessor (package `cpp`), compiler and the linker for C and C++ files (packages `cc` and `cxx`). Basically the Eli system is a new set of packages.

Every package has its own directory in the `Eli/pkg/` folder. Each contains at least three files:

- `version` with a single text line which describes the version of the package.
- A *derivation graph* with extension `.dg`. This file defines the extensions this package adds to the system. The syntax is described in [4]. Multiple `.dg`-files are possible.
- A `Makefile.in` which builds the package in the second phase. It also defines which files have to be copied to the `lib` directory in the third phase.

4.3.3. Grammar Files

Eli's description format for concrete grammars is `.con`. Its syntax is described in the Eli manual [14]. Another format is *pgram*, which contains complete grammar descriptions as used by parser generators. This is the format we are interested in.

Pgram grammars usually do not exist as files but as internal derivations which are converted from con files and other sources by the package *maptool*. `.con` files support EBNF syntax whereas pgram is plain BNF. The conversion is also done by the maptool package.

Unfortunately, the pgram syntax is undocumented, but Ulf Schwekendiek [19] created a pgram scanner and parser by reverse engineering for his *bison* package. We will reuse his "grammar-grammar" `pgram.con` for our implementation.

4.3.4. The *grambiguity* Package

This package adds support for the ACLA ambiguity detection scheme. It consists of the files shown in Table 4.8.

File	Functionality
`code.HEAD.phi`	Import of declarations needed in `pgram.lido`
`grambiguity.clp`	Command line arguments parser
`grambiguity.dg`	Derivation graph of the grambiguity package
`grambiguity.sh`	Invocation shell script
`grambiguity.specs`	Executable specifications
`Makefile.in`	Build script
`mkCustomStr1.c`	Postprocessor for semantic action tokens (copied from bison package)
`mkCustomStr.c`	Postprocessor for literal tokens (copied from bison package)
`Odinfile`	Instructions for Odin how to build the executable
`pgram.con`	Grammar-grammar syntax (copied from bison package)
`pgram.gla`	Grammar-grammar token definitions (copied from bison package)
`pgram.lido`	Builds an abstract syntax tree of a grammar
`SpecSemantic.c`	Token recognition of semantic actions in C syntax (copied from bison package)
`version`	Package version (1.0.0)
Other `.h` and `.cxx` files	Source code (described in table 4.6)

Table 4.8.: Files of the grambiguity package

The package adds a new derivation to the Eli system. It takes a grammar as input, invokes the executable and redirects its analysis to the output. See Table 4.9 and 4.10 for the additions to the Eli system.

Derivation	Meaning
:grambiguity	Computes the ACLA of a grammar given by a `.con` or `.specs` file

Table 4.9.: Derivations defined in `grambiguity.dg`

The package is built together with the Eli system. When "`./configure`" is called in the root directory, it also processes this package to generate a `Makefile` from `Makefile.in`.

A definition of how to build the executable `grambiguity.exe` is found during the "`make`" phase. It is built by calling Eli to process the `Odinfile`. This is possible because Eli and all necessary packages have been built before the grambiguity package. The `Odinfile` defines `grambiguity.exe` as a derivation of `grambiguity.specs` using "`g++`" as the compiler for C. Specifying the compiler is necessary because it needs to compile and link C++ and C sources. The C sources also compile with a C++ compiler. Eli will generate the application skeleton and pgram parser, include the

Option	Meaning
+positionautomaton	Use the position automaton regularizer (default).
+mohrinederhof	Use Mohri-Nederhof Transformation and Regularizer2.

Table 4.10.: Options defined in `grambiguity.dg`

other source files and compile it.

During "`make install`" the files `version`, `grambiguity.exe`, `grambiguity.dg` and `grambiguity.sh` are copied to their intended location in the file system.

When the :grambiguity derivation is used, the Eli system will pass a pgram grammar to `grambiguity.sh`. This script will call the executable `grambiguity.exe`. Its output is the result of the :grambiguity derivation.

4.3.5. The *bisonamb* Package

This package adds support for the LR-regular and noncanonical ambiguity detection scheme to the Eli system.

Both algorithms are not implemented in this package. Instead, it uses the implementation by Schmidt, which is integrated into a modified version of *bison*. It is available at [15] and [18] provides a description. The *bison* package by Ulf Schwekendiek [19] needs to be installed in order to run this package. This package can convert pgram grammars to the grammar format used by GNU bison.

The package's files are described in Table 4.11.

File or directory	Purpose
`schmitz/`	This directory contains the bison source including the ambiguity extension by Sylvain Schmitz [15]
`bisonamb.dg`	Derivation graph of the bisonamb package
`bisonAmbiguity.sh`	Invokes bison with the ambiguity detection option
`Makefile.in`	Build script
`version`	Package version (1.0.0)

Table 4.11.: Files of the bisonamb package

As the grambiguity package, this package has been integrated into the Eli build system. In the first phase, `./configure` will also call `configure` in the `schmitz` directory, so bison prepares itself for the second phase.

In the same manner the second phase will call "`make`" in the `schmitz` directory as instructed in `Makefile.in`. Then, it copies the executable `bison` into the root directory of the package.

In the "`make install`" phase the files `version`, `bison`, `bisonAmbiguity.sh` and `bisonamb.dg` are copied to the install directory.

When "`eli -R`" is called the next time, the package is loaded into the cache and `bisonamb.dg` is integrated into the system. See Table 4.12 for the extension that the bisonamb package adds to the system. The options in Table 4.13 are the same as supported by the bison executable. A more detailed description of their meaning can be found in its documentation.

Derivation	Meaning
:ambiguity	Invokes bison to analyse the ambiguity of a grammar given by a `.con` or `.specs` file

Table 4.12.: Derivations defined in "bisonamb.dg"

Option	possible values	Meaning
+test	"clr", "lrr", "ambig" (default)	Type of test. "clr" tests whether the grammar is of the type specified by +precision, "lrr" tests LR-Regular Unambiguity and "ambig" tests for Non-canonical Unambiguity.
+precision	"lr0", "slr", "lalr1", "lr1" (default)	Language class to use for the ambiguity detection. "lalr1" is only supported by "+test=clr".
+trace	"ma", "metrics"	Verbosity of the output. "ma" shows the conflicting mutual accessible states and "metrics" displays some general information about the grammar.
+conflicts	no value	Do not display potential ambiguities and LRR-intersections, but the LR(0)-conflicts that caused them.

Table 4.13.: Options defined in "bisonamb.dg". The effects are explained in [18].

When the :ambiguity derivation is used, Eli will first try to convert the source file to a bison grammar. When the bison package was installed, Eli will first derive a pgram grammar and then convert it to a bison grammar by using the bison package. Without the bison package, Eli is unable to do this conversion.

Then, Eli executes `bisonAmbiguity.sh` on the bison grammar. This shell script will invoke the GNU Bison executable from the `schmitz` directory. Its output is also the result of the :ambiguity derivation.

5. Evaluation

In the previous chapters two main ambiguity detection schemes have been described, but their usefulness has not been discussed yet. This is the subject of this chapter.

The first section will show the worst-case runtime of $O(n^5)$ for the ACLA method and $O(n^2)$ for the Regular Unambiguity method. Then, the results of both implementations on a collection of test grammars are presented.

5.1. Asymptotic Worst-Case Runtime

We define the size of a grammar by a 3-tuple (u, v, h), where u is the number of nonterminals, v the maximum number of productions a nonterminals has and h is the maximum number of right-hand-side symbols of a production. Alternatively, a single value $n := u \cdot v \cdot h$ is used.

5.1.1. Ambiguity Checking with Language Approximation

An important factor of the asymptotic runtime is the size of the approximation automaton. The position automaton method used in the implementation can return an exponentially growing number of states. An example showing this is Grammar 5.1, which is of size $(m, m, 2)$ or $2m^2$. Its language consists of only one word with 2^m a's. Hence, a finite state machine that accepts this grammar has at least 2^m transitions and $2^m + 1$ states.

$$\begin{aligned} \mathrm{S} &\to A_1\ A_1 \\ A_1 &\to A_2\ A_2 \\ &\vdots \\ A_{m-2} &\to A_{m-1}\ A_{m-1} \\ A_{m-1} &\to \mathrm{a}\ \mathrm{a} \end{aligned} \tag{5.1}$$

A Mohri-Nederhof transformed grammar has up to twice as much nonterminals as the original grammar. Which nonterminals reference which others can be determined in $O(uvh)$. The transitive closure to determine the reachability between nonterminals requires quadratic runtime in terms of number of nonterminals. The resulting runtime for the Mohri-Nederhof Transformation is $O(u^2{+}uvh) \subseteq O(u^2vh) \subseteq O(n^2)$.

Regularizer2 was designed to avoid the exponential blowup of the other method. It has only a linear number of states compared to the number of nonterminals.

Additionally, there is a linear number of states for each of the $2uv$ right-hand-side production sequences of a length up to h. Accordingly, the maximal number of states is $O(2u + 2uvh) \subseteq O(uvh) = O(n)$. The price is that, for instance with Grammar 5.1, this results in a less close approximation.

Regularizer2 has to be executed for every nonterminal that needs to be approximated. Therefore, the combined runtime of the Mohri-Nederhof Transformation and Regularizer2 on $2u$ nonterminals is $O(u^2 + uvh + u(2u + 2uvh)) \subseteq O(u^3vh) \subseteq O(n^3)$.

Because of the potential exponential runtime of the position automaton method, which would cover all other worst-case estimations, we will use Regularizer2 for any further discussions. We define $s := O(2u + 2uvh)$ as the number of states of the approximation automaton. The approximation automaton of the right-hand-side of a production has $O(hs)$ states, because this is the concatenation of up to h automata for nonterminals with $O(s)$ states each. We assume that the number of transitions is linear dependent on the number of states. This is a realistic assumption if the number of terminals is treated as a constant.

A single check of a vertical ambiguity between two productions needs up to $O((hs)^2 + (hs)^2)$ processing steps. This includes the quadratic runtime of the intersection and the breadth-first search to find a sample sequence. The breath-first search algorithm has a runtime of $O((hs)^2)$, the number of states in the automaton.

The vertical ambiguity check has to be done for every combination of two productions of the same nonterminal, thus up to uv^2 times. This results in a total runtime of $O(uv^2((hs)^2 + (hs)^2)) \subseteq O(u^3v^4h^4) \supseteq O(n^4)$.

The overlap operator requires a runtime of $O((hs)^2)$. This is the quadratic runtime of intersection between the regular expressions 3.7 and 3.8. The automaton representations have $O(hs)$ states each because these are variations of a single production. The third regular expression 3.9 is constant and thus the intersection requires only constant time and space. Again, the search for a sample string using breadth-first search requires linear time in respect of the number of states. The total runtime for a single horizontal ambiguity check is $O((hs)^2 + (hs)^2)$.

There are $h - 1$ possible divisions in up to uv productions. This makes $O(uvh)$ combinations and a total runtime of $O(uvh(hs)^2) \subseteq O(u^3v^3h^5) \subseteq O(n^5)$ for all horizonal ambiguity checks.

Together horizontal and vertical ambiguity requires a total runtime of

$$O(\underbrace{u^2 + uvh}_{\text{Mohri-Nederhof}} + \underbrace{2u^2 + 2uvh}_{\text{Regularizer2}} + \underbrace{u^3v^4h^4}_{\text{vertical a.}} + \underbrace{u^3v^3h^5}_{\text{horizontal a.}}) \subseteq O(u^3v^4h^5) \subseteq O(n^5)$$

The approximation automata can be optimized by determinizing and minimizing them. In theory, the determinization can blow up the automaton exponentially again and hence it is not considered here. Nevertheless, in practice this optimization decreases the number of states and therefore this is done with every automaton in the implementation.

5.1.2. Regular/LR-Regular/Noncanonical Unambiguity

For Regular, LR-Regular and Noncanonical Unambiguity based on LR(0), all items of the grammar have to be found. There are up to $uv(h+1)$ different positions for a grammar, in particular $h+1$ positions for a single production with h right-hand-side symbols for all of the $O(uv)$ productions.

The mutual accessibility algorithm evaluates tuples. When $uv(h+1)$ is the number of positions, then there are $O((uvh)^2)$ tuples. This results in a asymptotic runtime of $O(n^2)$.

Choosing a k different from 0 results in a greater number of positions, up to t^k times more than for an LR(0) automaton (This is the number of combinations of a sequence of k lookahead symbols out of a set of t terminals). The total runtime would be $O((uvht^k)^2) = O(n^2t^{2k})$. This has exponential growth in respect of k, but k is rarely chosen to be greater than 1.

5.2. Results

To test the power and usefulness, a collection of grammars is tested on the ACLA and the Noncanonical Unambiguity algorithms. This also allows us to compare both algorithms.

An example output of the ambiguity package is the following.

```
1 potential ambiguities with LR(1) precision detected:
(S -> "a" "b" . , S -> A "b" . )
```

This if the output of the command

```
eli "grammar3.1.con :ambiguity >"
```

when the Eli system, including the `bisonamb` package, have been installed successfully and the file `grammar3.1.con` (the file of Grammar 3.1) is in the current directory. The output shows the position of a conflict between the items $S \to ab\bullet$ and $S \to Ab\bullet$. This is a reduce/reduce-conflict because both positions are at the end of the productions.

An example output produced by the `grambiguity` package is

```
Total number of nonterminals: 2
Total number of terminals: 2
Total number of productions: 3
Number of strongly connected components: 2
Dimension sizes: (2, 2, 2)

S: 12/3/3
A: 4/2/2

Confirmed vertical ambiguity in nonterminal S
  between productions S -> 'a' 'b' and S -> A 'b'
  shortest sample string: "ab"
```

```
  found 2 derivation trees for nonterminal S

Found 0 horizontal (0 confirmed) and 1 vertical (1 confirmed) ambiguities.

This grammar is ambiguous
```

This is printed on the screen when the command

```
eli "grammar3.1.con :grambiguity >"
```

is executed. At first, it displays some basic properties of the grammar. These can be different from the data on the `.con` file because package `maptool` pre-processes it. Then, the approximation automaton size of every nonterminal is printed. The first number is the number of states produced by the regularizer. The second is the number of states after determinisation and the third is the number of states after this automaton has been minimized. Finally, it prints all found potential and confirmed ambiguities. An ambiguity becomes a confirmed ambiguity if at least two derivation trees have been found for the sample string.

Most test grammars have been taken from the grammar collection of Anders Møller [11]. These include the grammars that were used in the articles [2], [17], [18] and [3].

Other grammars tested are `trivial.con` (which consists of only one production $S \rightarrow \epsilon$) and all grammars mentioned in this document. The grammar `pgram.con`, which is used to generate a parser for grammar descriptions for the `grambiguity` package, is also included in this test.

Moreover, the "real world" grammars `ALGOL60.con`, `F77Phrase.con` and `F90Phrase.con` (both Fortran), `Java.con`, `cminus.con` and `pascal-.con` (shrinked versions of C and Pascal) where used for the testings. These are included in the Eli system distribution.

The computation time was measured using the real execution time printed by the command "`time -p`". Only the execution time of the executables `grambiguity.exe` and `bison` without the overhead of the Eli system was measured. The execution environment was "Ubuntu 7.10" running on a Pentium M (1600 Mhz) processor with 1.25 GB of physical memory.

Some computations did not finish in reasonable time and were aborted. Other computations required more memory than available in the virtual address space (2 GB on Windows without /3GB option, 3 GB on Linux, both 32 bit systems). Values that are unknown because of these limits are replaced by question marks.

The implementation used for the `bisonamb` package still has some issues. First, the computation of the LR-Regular condition for some unambiguous grammars fails because of invalid memory accesses. Second, it reports two ambiguous grammars as unambiguous when (and only then) using the "+conflicts" option. This violates the conservativeness of the algorithm. Both problems have been reported to the author, Sylvain Schmitz.

In total 67 context-free grammars have been tested of which 50 are unambiguous. 30 of them are horizontally and vertically unambiguous and 39 are Noncanonically

Unambiguous in regard of the LR(1) item set. More detailed statistics can be found in Figure 5.1.

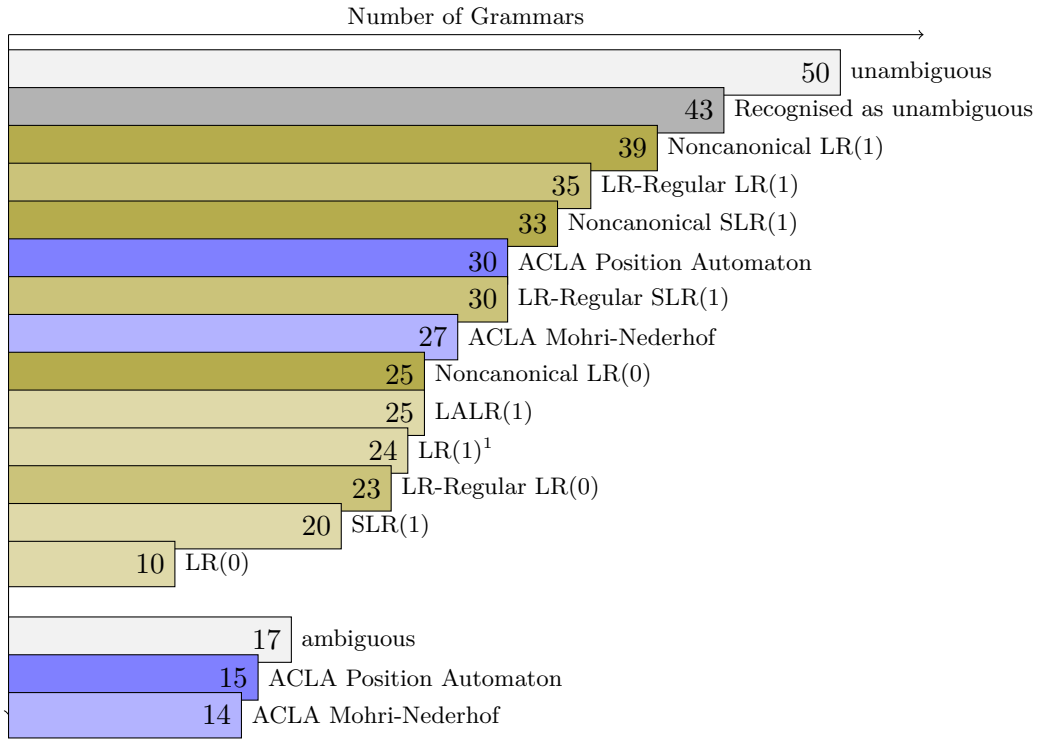

Figure 5.1.: Results of all tested unambiguity definitions over 66 grammars. The displayed numbers are the numbers of grammars which have been found as unambiguous by the given methods. "Recognised as unambiguous" means the number of grammars which has been identified correctly by at least one method. Only the `grambiguity` package (blue) is also able to detect ambiguous grammars.

5.2.1. Unambiguous Grammars

This section presents the results of the detection approximations on grammars, which are known to be unambiguous. The first table (Table 5.1) contains some basic properties of the grammars and the results return by the `bisonamb` package. These are the number LR-conflicts, the LR-Regular test and the test for Noncanonical Unambiguity. For better comparability the "+conflicts" has been used for the latter two. Thus, the tables shows the number of LR(0) conflicts that are involved in a potential noncanonical ambiguity or lookahead-intersection respectively. The precision column shows the lowest precision that finds as many potential noncanonical ambiguities (without "+conflicts" option) as LR(1) does.

[1] LR(1) is less than LALR(1) because the LR(1) item set could not be computed for some grammars

The next two Tables 5.2 and 5.3 show the results of the `grambiguity` package. The automaton column contains the number of states of the approximation automaton of the whole grammar. This automaton is not always used by the horizontal and vertical ambiguity checking, but normally it is greater than all automata used there and therefore a useful estimation of the overall size.

	Grammar properties				Results of bisonamb					
Grammar	$\|P\|$	$\|\Sigma\|$	$\|C\|$[1]	(u,v,h)[2]	Prec.[3]	C[4]	I[5]	A[6]	Time	D[7]
`trivial.con`	1	0	1	(1,1,0)	LR(0)	0	0	0	0.00s	una.
Grammar 2.1	4	3	2	(3,2,3)	SLR(1)	0	0	0	0.00s	una.
Grammar 2.3	3	3	2	(3,1,3)	LR(0)	2	0	0	0.00s	una.
Grammar 2.4	6	3	4	(4,3,3)	LR(0)	3	0	0	0.00s	una.
Grammar 3.10	3	3	3	(3,1,2)	LR(0)	0	0	0	0.00s	una.
Grammar 3.11	3	3	2	(2,2,3)	LR(0)	0	0	0	0.00s	una.
Grammar 3.13	3	3	2	(2,2,3)	LR(0)	0	0	0	0.12s	una.
Grammar 3.14	5	4	3	(3,2,3)	LR(0)	0	0	0	0.00s	una.
Grammar 3.15	6	3	4	(4,2,3)	LR(1)	1	1	0	0.15s	una.
Grammar 4.1	4	5	2	(2,3,3)	LR(0)	0	0	0	0.00s	una.
Grammar 5.1 ($m = 10$)	10	1	10	(10,1,2)	LR(0)	0	0	0	0.00s	una.
`pgram.con`	24	11	15	(15,3,4)	SLR(1)	0	0	0	0.00s	una.
`03_02_124.con`	6	3	5	(5,2,4)	LR(0)	1	0	0	0.00s	una.
`03_09_027.con`	8	6	6	(6,2,3)	LR(0)	2	0	0	0.00s	una.
`03_09_081.con`	10	3	6	(6,2,3)	LR(0)	4	1	0	0.00s	una.
`04_02_041.con`	4	6	3	(3,2,4)	LR(0)	0	0	0	0.00s	una.
`05_03_092.con`	12	9	5	(5,4,4)	LR(1)	0	0	0	0.00s	una.
`05_03_114.con`	7	3	4	(4,2,3)	LR(0)	2	0	0	0.00s	una.
`90_10_042.con`	8	3	5	(5,3,3)	LR(1)	3	?[8]	1	0.00s	n/a
`98_05_030.con`	7	6	2	(3,3,6)	LR(0)	4	1	0	0.00s	una.
`98_08_215.con`	7	6	5	(5,2,4)	LR(0)	2	0	0	0.00s	una.
`basepairs.con`	8	4	2	(2,7,3)	LR(1)	12	6	6	0.00s	n/a
`exp.con`	5	4	2	(3,2,3)	LR(0)	1	0	0	0.00s	una.
`g3.con`	9	3	3	(4,4,3)	LR(1)	4	2	2	0.00s	n/a
`g4.con`	7	3	2	(3,3,5)	SLR(1)	0	0	0	0.00s	una.

[1] Number of strongly connected components

[2] Grammar dimension: u = number of nonterminals, v = max. number of productions per nonterminal, h = max. number of right-hand-side symbols

[3] Precision; smallest item set which finds the same number of potential noncanonical ambiguities as LR(1)

[4] Conflicts; Number of LR conflicts with the given precision

[5] Intersections; Number of LR-regular conflicts using the same precision

[6] Ambiguities; Number of conflicts that are potential noncanonical ambiguities with the same precision

[7] Detection result; "una." means that the grammar has been correctly identified as unambiguous, i.e. no potential ambiguities were found

Grammar	Grammar properties				Results of bisonamb					
	$\|P\|$	$\|\Sigma\|$	$\|C\|$[1]	(u,v,h)[2]	Prec.[3]	C[4]	I[5]	A[6]	Time	D[7]
`g5.con`	4	3	2	(2,3,4)	SLR(1)	0	0	0	0.00s	una.
`g6.con`	7	3	2	(4,2,3)	LR(1)	0	0	0	0.00s	una.
`g7.con`	14	3	3	(6,4,3)	LR(1)	5	2	2	0.00s	n/a
`g8.con`	12	3	2	(5,3,4)	LR(1)	0	0	0	0.00s	una.
`java_arrays.con`	12	8	8	(10,2,4)	LR(0)	1	0	0	0.00s	una.
`java_casts.con`	10	3	3	(8,2,4)	LR(0)	1	0	0	0.00s	una.
`java-exp.con`	27	18	3	(11,5,3)	LR(0)	36	0	0	0.01s	una.
`java_modifiers.con`	48	23	22	(22,8,4)	SLR(1)	12	0	0	0.00s	una.
`java_names.con`	20	11	12	(12,2,4)	LR(0)	10	0	0	0.00s	una.
`odd-even.con`	6	2	3	(3,2,5)	SLR(1)	0	0	0	0.00s	una.
`palindromes.con`	6	2	2	(2,5,3)	LR(1)	14	14	14	0.00s	n/a
`R.con`	8	4	2	(2,7,3)	LR(1)	12	6	6	0.03s	n/a
`reverse.con`	6	2	2	(2,5,3)	LR(1)	14	14	14	0.00s	n/a
`s2.con`	3	3	2	(2,2,3)	LR(0)	0	0	0	0.00s	una.
`s3.con`	9	1	7	(7,2,3)	LR(0)	2	0	0	0.00s	una.
`s5.con`	6	3	4	(4,2,3)	LR(0)	1	1	0	0.00s	una.
`s7.con`	3	2	2	(2,2,3)	LR(0)	0	0	0	0.00s	una.
`voss-light.con`	15	3	6	(7,4,3)	LR(1)	1	1	0	0.00s	una.
`voss.con`	64	3	5	(27,9,7)	LR(1)	68	19	18	0.16s	n/a
`sets.con`	9	3	7	(7,2,4)	LR(0)	2	0	0	0.04s	una.
`ALGOL60.specs`	161	55	48	(81,8,5)	LR(1)	0	0	0	3.85s	una.
`ebnf.specs`	23	15	6	(10,4,4)	SLR(1)	0	0	0	0.00s	una.
`F77Phrase.specs`	532	102	189	(225,22,11)	SLR(1)[9]	11	9	7	0.94s	n/a
`F90Phrase.specs`	958	184	292	(352,29,11)	SLR(1)[9]	12	10	10	4.21s	n/a
`Java.specs`	361	103	89	(150,12,9)	SLR(1)[9]	12	11	11	0.61s	n/a

Table 5.1.: The properties of unambiguous test grammars and the results of the `bisonamb` package.

Grammar	Position Automaton			
	Automaton size[10]	Ambiguities[11]	Time	D[7]
`trivial.con`	3/1/1	0/0 (0/0)	0.53s	una.
Grammar 2.1	21/4/3	2/0 (0/0)	0.00s	n/a
Grammar 2.3	33/4/2	0/0 (0/0)	0.01s	una.

[8] The computation failed because of a segmentation fault

[9] The computation using LR(1) did not finish in reasonable time or required more than the available amount of memory

[10] Number of states in the approximation automaton; without optimization/after determinisation/after minimization

[11] Number of potential ambiguities; horizontal/vertical (confirmed horizontal/confirmed vertical)

Grammar	Position Automaton			
	Automaton size[10]	Ambiguities[11]	Time	D[7]
Grammar 2.4	28/6/5	0/0 (0/0)	0.01s	una.
Grammar 3.10	14/4/4	0/0 (0/0)	0.00s	una.
Grammar 3.11	14/4/2	0/0 (0/0)	0.00s	una.
Grammar 3.13	18/7/4	0/0 (0/0)	0.00s	una.
Grammar 3.14	30/7/4	0/0 (0/0)	0.01s	una.
Grammar 3.15	35/4/3	0/1 (0/0)	0.12s	n/a
Grammar 4.1	20/6/2	0/0 (0/0)	0.01s	una.
Grammar 5.1 ($m = 10$)	5115/1025/1025	0/0 (0/0)	7.29s	una.
`pgram.con`	196/42/13	0/0 (0/0)	0.17s	una.
`03_02_124.con`	27/5/5	0/0 (0/0)	0.01s	una.
`03_09_027.con`	67/19/9	0/0 (0/0)	0.03s	una.
`03_09_081.con`	61/10/9	0/0 (0/0)	0.02s	una.
`04_02_041.con`	19/7/6	0/0 (0/0)	0.01s	una.
`05_03_092.con`	89/25/15	0/0 (0/0)	0.04s	una.
`05_03_114.con`	32/5/4	0/0 (0/0)	0.01s	una.
`90_10_042.con`	62/5/4	0/0 (0/0)	0.06s	una.
`98_05_030.con`	63/9/3	7/3 (0/0)	0.06s	n/a
`98_08_215.con`	33/9/8	0/0 (0/0)	0.02s	una.
`basepairs.con`	43/5/1	0/0 (0/0)	0.02s	una.
`exp.con`	34/7/2	2/1 (0/0)	0.01s	n/a
`g3.con`	57/6/1	2/1 (0/0)	0.02s	n/a
`g4.con`	36/6/1	2/1 (0/0)	0.02s	n/a
`g5.con`	21/4/1	2/0 (0/0)	0.00s	n/a
`g6.con`	59/5/2	2/2 (0/0)	0.03s	n/a
`g7.con`	184/23/7	2/4 (0/0)	0.10s	n/a
`g8.con`	98/14/5	4/2 (0/0)	0.08s	n/a
`java_arrays.con`	55/11/10	0/0 (0/0)	0.01s	una.
`java_casts.con`	45/6/5	0/0 (0/0)	0.01s	una.
`java-exp.con`	35554/4047/2	26/22 (0/0)	2652.29s	n/a
`java_modifiers.con`	230/59/22	0/0 (0/0)	1.05	una.
`java_names.con`	129/19/10	0/0 (0/0)	0.04s	una.
`odd-even.con`	30/8/4	0/0 (0/0)	0.02s	una.
`palindromes.con`	23/3/1	0/0 (0/0)	0.00s	una.
`R.con`	43/5/1	0/0 (0/0)	0.05s	una.
`reverse.con`	23/3/1	0/0 (0/0)	0.00s	una.
`s2.con`	18/7/4	0/0 (0/0)	0.00s	una.
`s3.con`	171/35/35	0/0 (0/0)	0.10s	una.
`s5.con`	39/5/4	0/1 (0/0)	0.01s	n/a
`s7.con`	25/6/3	1/0 (0/0)	0.00s	n/a
`voss-light.con`	83/8/7	0/1 (0/0)	0.02s	n/a

Grammar	Position Automaton			
	Automaton size[10]	Ambiguities[11]	Time	D[7]
`voss.con`[7]	30676/98/7	?/? (?/?)	?	?
`sets.con`	77/11/5	0/0 (0/0)	0.28s	una.
`ALGOL60.specs`[7]	?/?/?	?/? (?/?)	?	?
`ebnf.specs`	1447/93/8	3/2 (0/0)	5.85s	n/a
`F77Phrase.specs`[9]	?/?/?	?/? (?/?)	?	?
`F90Phrase.specs`[9]	?/?/?	?/? (?/?)	?	?
`Java.specs`[9]	?/?/?	?/? (?/?)	?	?

Table 5.2.: Results of the `grambiguity` package on unambiguous grammars using the position automaton regularizer

Grammar	Mohri-Nederhof Transformation			
	Automaton size[10]	Ambiguities[11]	Time	D[7]
`trivial.con`	5/1/1	0/0 (0/0)	0.54s	una.
Grammar 2.1	30/4/3	2/0 (0/0)	0.00s	n/a
Grammar 2.3	30/4/2	0/0 (0/0)	0.00s	una.
Grammar 2.4	36/6/5	0/0 (0/0)	0.01s	una.
Grammar 3.10	20/4/4	0/0 (0/0)	0.00s	una.
Grammar 3.11	22/4/2	0/0 (0/0)	0.00s	una.
Grammar 3.13	18/5/4	0/0 (0/0)	0.17s	una.
Grammar 3.14	25/6/3	0/0 (0/0)	0.01s	una.
Grammar 3.15	38/4/3	0/1 (0/0)	0.07s	n/a
Grammar 4.1	26/6/2	0/0 (0/0)	0.01s	una.
Grammar 5.1 ($m = 10$)	70/3/3	8/0 (0/0)	7.29s	n/a
`pgram.con`	121/21/13	0/0 (0/0)	0.06s	una.
`03_02_124.con`	37/5/5	0/0 (0/0)	0.01s	una.
`03_09_027.con`	48/9/9	0/0 (0/0)	0.01s	una.
`03_09_081.con`	50/8/8	0/0 (0/0)	0.01s	una.
`04_02_041.con`	25/7/6	0/0 (0/0)	0.01s	una.
`05_03_092.con`	57/18/11	0/0 (0/0)	0.01s	una.
`05_03_114.con`	34/5/4	0/0 (0/0)	0.01s	una.
`90_10_042.con`	37/4/3	1/1 (0/0)	0.00s	n/a
`98_05_030.con`	45/9/3	7/3 (0/0)	0.21s	n/a
`98_08_215.con`	40/9/8	0/0 (0/0)	0.21s	una.
`basepairs.con`	41/5/1	0/0 (0/0)	0.03s	una.
`exp.con`	33/5/2	2/1 (0/0)	0.02s	n/a
`g3.con`	58/6/2	1/2 (0/0)	0.01s	n/a
`g4.con`	39/6/1	2/1 (0/0)	0.02s	n/a

[9]The computation did not finish in reasonable time or required more than the available amount of memory

Grammar	Mohri-Nederhof Transformation			
	Automaton size[10]	Ambiguities[11]	Time	D[7]
g5.con	23/4/1	2/0 (0/0)	0.01s	n/a
g6.con	46/5/2	2/2 (0/0)	0.01s	n/a
g7.con	79/11/6	2/4 (0/0)	0.04s	n/a
g8.con	72/11/5	4/2 (0/0)	0.05s	n/a
java_arrays.con	68/11/10	0/0 (0/0)	0.01s	una.
java_casts.con	78/6/5	0/0 (0/0)	0.03s	una.
java-exp.con	155/19/2	26/22 (0/0)	0.41s	n/a
java_modifiers	200/46/22	0/0 (0/0)	0.54s	una.
java_names.con	103/14/10	0/0 (0/0)	0.02s	una.
odd-even.con	44/8/4	0/0 (0/0)	0.00s	una.
palindromes.con	31/3/1	0/0 (0/0)	0.00s	una.
R.con	41/5/1	0/0 (0/0)	0.01s	una.
reverse.con	31/3/1	0/0 (0/0)	0.00s	una.
s2.con	18/5/4	0/0 (0/0)	0.00s	una.
s3.con	52/4/2	3/0 (0/0)	0.01s	n/a
s5.con	40/5/4	0/1 (0/0)	0.00s	n/a
s7.con	23/4/2	1/0 (0/0)	0.00s	n/a
voss-light.con	77/8/4	2/2 (0/0)	0.02s	n/a
voss.con	390/15/6	27/42 (0/0)	0.98s	n/a
sets.con	49/5/5	0/0 (0/0)	0.06s	una.
ALGOL60.specs	869/114/50	41/31 (0/0)	30.32s	n/a
ebnf.specs	117/17/6	3/5 (0/0)	0.30s	n/a
F77Phrase.specs	2742/406/160	158/254 (0/0)	657.47s	n/a
F90Phrase.specs	5080/815/269	322/403 (0/0)	6781.21s	n/a
Java.specs	1944/246/93	105/105 (0/0)	820.82s	n/a

Table 5.3.: Results of the `grambiguity` package on unambiguous grammars using the Mohri-Nederhof Transformation

Nearly all unambiguous grammars have been detected as such by one of the techniques. Exceptions are `g3.con` and `g7.con`. The grammars `voss.con`, `ALGOL60.con`, `F77Phrase.specs`, `F90Phrase.specs` and `Java.specs` are too big and could not be processed by all algorithms.

Some grammars, like `palindromes.con` were detected by the `grambiguity` package but not by the `bisonamb` package. Others, like `exp.con`, have been detected by `bisonamb`, but not by `grambiguity`. Hence, it makes sense to use both algorithms in conjunction.

`s7.con` has not been recognised as unambiguous by the `grambiguity` package, although it is LR(0), but the Noncanonical Unambiguity necessarily detects any LR(1) (and LR(0)) grammar correctly. This is a general rule caused by the different techniques. Furthermore, as shown in [17], ACLA detects every unambiguous

grammar which is also detected by Regular Unambiguity. Hence, Regular Unambiguity can be considered as the weakest definition. ACLA and Noncanonical Unambiguity can be used in combination to detect more grammars.

There are just four grammars (Grammar 3.15, `03_09_081.con`, `98_05_030.con` and `voss-light.con`) that are correctly found as unambiguous by Noncanonical Unambiguity, but not by LR-Regular Unambiguity. However, `03_09_081.con` and `98_05_030.con` are LR-Regular if the LR(1) item set is used.

Another observation is that the position automaton regularizer seems to be more accurate than Regularizer2. Grammar 5.1 illustrates the difference between them. The position automaton method produces a huge finite state machine, Regularizer2 creates a smaller one, but it does not detect the grammar as unambiguous. However, Regularizer2 can analyze grammars where the other methods exhaust the limits.

Note that this does not mean that the Mohri-Nederhof Transformation is necessarily less accurate. This weakness can be caused by the optimization of Regularizer2 to have polynomial runtime. The reference implementation by Andres Møller does not show this difference, but is unable to check Grammar 5.1 for instance (using the default Java VM settings), due to exhausted resources.

In general, the ACLA scheme seems to have a weakness with expression/tree grammars, which can be seen on `exp.con` and `java-exp.con`. The latter one has a huge approximation automaton, which is optimized to an automaton with only 2 states (the same size as Regularizer2). The reason is probably that expression grammars usually consist of only one big strongly connected component, but the position automaton regularizer processed every nonterminal separately.

Moreover, even the relatively simple expression grammar `exp.con` is not detected as unambiguous. Anyhow, a transformation called Unfolding developed by Braband and Møller [3] modifies grammars, such that it can be detected as unambiguous by the ACLA technique. The grammar `voss.con` can also be detected correctly after this transformation.

The “real world” grammars ALGOL60, Fortran and Java are too big to be processed by the position automaton regularizer. Additionally, programming languages like these usually contain expressions that we identified as a weakness of the ACLA technique. The Noncanonical Unambiguity technique should work well on these grammars, because they have been created to be processed by an LALR(1) parser. However, LR(1) can have a lot more items and causes some algorithms to exceed the limits.

Although these grammars may be too big to be analyzed as a whole, they can be analyzed by parts. For instance, the grammars `java_arrays.con`, `java_casts.con`, `java-exp.con`, `java_modifiers` and `java_names.con` are such sub-grammars. All of them have been proven unambiguous by the Noncanonical Unambiguity algorithm.

5.2.2. Ambiguous Grammars

While the previous section only considered unambiguous grammars, this section is about grammars which are known to be ambiguous. The three tables are organised like in the previous section.

The Noncanonical Unambiguity implementation just finds potential ambiguities, but does not try to verify them. Therefore, it cannot confirm that a grammar is ambiguous. If it works correctly, it finds at least one potential ambiguity for all of these grammars.

In contrast, the ACLA method produces sample strings which can be checked whether more than one derivation tree exists for them. If this is the case for at least one of the sample strings, the grammar is known to be ambiguous.

	Grammar properties				Results of `bisonamb`				
Grammar	$\|P\|$	$\|\Sigma\|$	$\|C\|$[1]	(u,v,h)[2]	Prec.[3]	C[4]	I[5]	A[6]	Time
Grammar 3.1	3	2	2	(2,2,2)	LR(0)	1	1	1	0.00s
Grammar 3.12	4	1	3	(3,2,1)	LR(0)	1	1	1	0.00s
`01_05_076.con`	12	5	6	(7,2,3)	SLR(1)	3	2	0[12]	0.00s
`03_01_011.con`	9	4	9	(6,2,2)	LR(0)	2	1	0[12]	0.00s
`03_05_170.con`	11	8	7	(7,2,7)	LR(0)	1	1	1	0.00s
`04_11_047.con`	27	14	6	(11,7,7)	LR(1)	23	?[8]	9	0.04s
`05_06_028.con`	9	8	4	(4,4,8)	LR(0)	4	2	2	0.00s
`06_10_036.con`	33	18	3	(13,5,4)	LR(0)	15	2	2	0.09s
`91_08_014.con`	16	12	3	(6,6,4)	LR(0)	15	8	8	0.00s
`g1.con`	6	3	2	(2,5,3)	SLR(1)	16	16	16	0.00s
`g2.con`	8	3	2	(3,5,3)	SLR(1)	18	18	18	0.00s
`h-amb.con`	5	3	3	(3,2,2)	LR(0)	1	1	1	0.00s
`s1.con`	7	5	3	(4,3,2)	LR(0)	3	2	2	0.00s
`s4.con`	9	1	7	(7,2,4)	LR(0)	2	1	1	0.01s
`v-amb.con`	4	3	3	(3,2,3)	LR(0)	1	1	1	0.00s
`cminus.specs`	68	31	22	(34,9,7)	SLR(1)	15	15	11	0.01s
`pascal-.specs`	113	41	53	(64,8,6)	LR(1)	3	1	1	0.56s

Table 5.4.: The properties of some ambiguous test grammars and the results of the `bisonamb` package.

[1] Number of strongly connected components

[2] Grammar dimension: u = number of nonterminals, v = max. number of productions per nonterminal, h = max. number of right-hand-side symbols

[3] Precision; smallest item set which finds the same number of potential noncanonical ambiguities as LR(1)

[4] Conflicts; Number of LR conflicts with the given precision

[5] Intersections; Number of LR-regular conflicts using the same precision

[6] Ambiguities; Number of conflicts that are potential noncanonical ambiguities with the same precision

Grammar	Position Automaton			
	Automaton size[10]	Ambiguities[11]	Time	D[7]
Grammar 3.1	12/3/3	0/1 (0/1)	0.00s	amb.
Grammar 3.12	14/2/2	0/1 (0/1)	0.00s	amb.
`01_05_076.con`	65/10/4	2/0 (1/0)	0.03s	amb.
`03_01_011.con`	42/7/4	1/0 (1/0)	0.01s	amb.
`03_05_170.con`	88/23/8	2/1 (1/1)	0.04s	amb.
`04_11_047.con`	438/71/10	6/2 (0/1)	0.85s	amb.
`05_06_028.con`	142/44/12	10/3 (2/0)	0.11s	amb.
`06_10_036.con`	2120/732/25	18/9 (0/1)	37.64s	amb.
`91_08_014.con`	169/52/10	5/5 (0/3)	0.50s	amb.
`g1.con`	30/4/1	1/5 (1/5)	0.00s	amb.
`g2.con`	40/5/1	1/6 (0/6)	0.08s	amb.
`h-amb.con`	19/7/5	1/0 (1/0)	0.00s	amb.
`s1.con`	72/13/6	2/1 (2/1)	0.02s	amb.
`s4.con`	172/36/2	0/1 (0/1)	0.07s	amb.
`v-amb.con`	18/4/4	0/1 (0/1)	0.00s	amb.
`cminus.specs`[6]	?/?/?	?/? (?/?)	?	?
`pascal-.specs`[6]	?/?/?	?/? (?/?)	?	?

Table 5.5.: Results of the `grambiguity` package on ambiguous grammars using the position automaton regularizer

Grammar	Mohri-Nederhof Transformation			
	Automaton size[10]	Ambiguities[11]	Time	D[7]
Grammar 3.1	16/3/3	0/1 (0/1)	0.00s	amb.
Grammar 3.12	20/2/2	0/1 (0/1)	0.00s	amb.
`01_05_076.con`	61/6/2	2/0 (0/0)	0.01s	n/a
`03_01_011.con`	41/5/4	1/0 (1/0)	0.00s	amb.
`03_05_170.con`	58/12/7	2/1 (2/1)	0.03s	amb.
`04_11_047.con`	134/22/10	6/2 (0/1)	0.23s	amb.
`05_06_028.con`	62/12/3	10/4 (2/0)	0.07s	amb.
`06_10_036.con`	182/32/13	18/9 (0/1)	0.49s	amb.

[7] Detection result; "amb." means that the grammar has been correctly identified as ambiguous, i.e. at least one confirmed ambiguities was found

[8] The computation failed because of a segmentation fault

[10] Number of states in the approximation automaton; without optimization/after determinisation/after minimization

[11] Number of potential ambiguities; horizontal/vertical (confirmed horizontal/confirmed vertical)

[12] Obviously, this is wrong. However, there are potential ambiguities when the conflicts option is not used.

[6] The computation did not finish in reasonable time or required more than the available amount of memory

Grammar	Mohri-Nederhof Transformation			
	Automaton size[10]	Ambiguities[11]	Time	D[7]
`91_08_014.con`	93/19/10	5/3 (0/2)	0.10s	amb.
`g1.con`	28/4/1	1/5 (1/5)	0.00s	amb.
`g2.con`	41/5/1	1/6 (0/6)	0.03s	amb.
`h-amb.con`	25/7/5	1/0 (1/0)	0.00s	amb.
`s1.con`	42/6/3	2/1 (2/1)	0.00s	amb.
`s4.con`	53/4/2	3/1 (0/0)	0.01s	n/a
`v-amb.con`	24/4/4	0/1 (0/1)	0.00s	amb.
`cminus.specs`	378/51/17	25/10 (0/0)	3.78s	n/a
`pascal-.specs`	625/84/35	22/6 (0/1)	4.58s	amb.

Table 5.6.: Results of the `grambiguity` package on ambiguous grammars using the Mohri-Nederhof Transformation

The ACLA algorithm performs well on these grammars. All grammars (except `cminus.specs`, which could not be processed with the position automaton regularizer) have been successfully detected as ambiguous. This is possibly a coincidence since just one sample string is checked for every potential ambiguity.

Again, the position automaton regularizer seems to be more accurate than Regularizer2 as more sample strings happen to be true ambiguities of the original grammars.

The "real world" grammars `cminus.specs` and `pascal-.specs` are ambiguous because of the dangling else problem. Both require external disambiguation techniques which neither the `bisonamb` nor the `grambiguity` package support so they are classified as ambiguous.

6. Summary

The problem to decide whether a given context-free grammar is ambiguous is undecidable. However, there are algorithms that can confirm the unambiguity (or ambiguity) of a nontrivial subset of all context-free grammars.

Two families of algorithms were presented in this thesis: Ambiguity Checking with Language Approximations (ACLA) and regular, LR-regular and Noncanonical Unambiguity.

The ACLA method checks possible horizontal and vertical ambiguities between the grammar's nonterminals and productions. This is possible on a superset of the original grammar, but may find more ambiguities than the grammar actually has. Hence, this is a conservative approximation. The worst-case runtime is $O(n^5)$ (with n being the size of the grammar). A noticeable advantage of this method is that it provides a less-abstract output, which includes which parts of the grammar are ambiguous and ideally a sequence with multiple parse trees.

The regular, LR-regular and Noncanonical Unambiguity scheme essentially tries

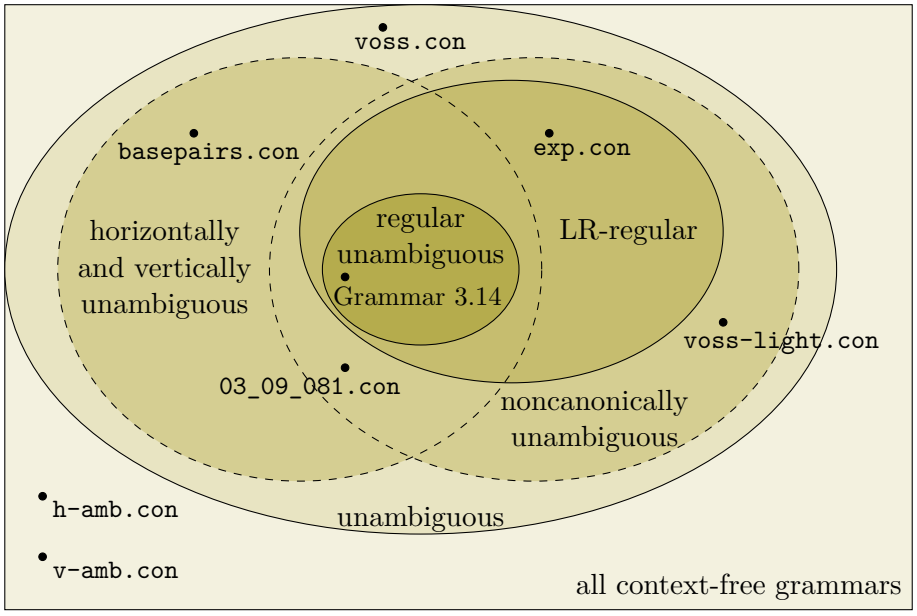

Figure 6.1.: Detection precision inclusions when using the same equivalence classes. The more an area covers the area of all unambiguous grammars the more precise the approximation is. Example grammars for all areas exist.

to find two different paths through a GLR parser. The existence of such two paths means two possible interpretations of the input. The actual check is done on the item set the GLR parser is based on, which is also is a regular approximation of the original grammar. Its worst-case runtime is $O(n^2)$ and therefore considerably better that the ACLA algorithm. Many existing context-free grammars are designed to work in conjunction with LR(k) parsers and this method works particularly well with LR(k) grammars.

Both algorithms are able to recognise unambiguous grammars, which the other cannot. Therefore, one can use both approaches to detect as many unambiguous grammars as possible. An overview which classes of detection precision include which others can be found in Figure 6.1.

Within the context of this thesis the Ambiguity Checking with Language Approximations technique has been implemented and, together with the implementation of Noncanonical Unambiguity by Sylvain Schmitz, integrated into the Eli Framework. Eli is an open source toolset that allows to generate compilers from its specifications. Since the integration of the GNU Bison GLR parser generator by Ulf Schwekendiek, it allows to generate compilers for arbitrary context-free grammars. With the integration of these two ambiguity approximations, these grammars can be checked for potential ambiguity within the same environment.

6.1. Future Work

The execution time of the grambiguity can be optimised. Using a hash set instead of binary trees may reduce memory usage. A different equivalence relation for the position automaton regularizer can reduce the size of the approximation automaton without losing accuracy. These are only some ideas to improve the execution time.

Also the accuracy of the ambiguity approximation can be improved. Unfolding, as presented in [3], can improve the accuracy for expression grammars.

The current implementation of LR-regular and Noncanonical Unambiguity supports precisions up to LR(1). A new implementation could handle any k or even non-LR(k) equivalence relations. Anyhow, one has to consider whether the effort and execution time (exponentially in respect to k) is worth the increased precision. Also, it is possible to generate sample strings for any path through the position automaton. The same modified Cocke-Younger-Kasami algorithm can be used to confirm such ambiguities.

Both detection schemes could be combined for increased accuracy. For instance, the nonterminals and productions involved in a potential horizontal and vertical ambiguity form a sub-grammar. This sub-grammar can be checked using the Noncanonical Unambiguity scheme. The potential horizontal or vertical ambiguity is not a “real” one if this second test does not find any potential ambiguities.

Furthermore, one can think about the relationship between potential horizontal and vertical ambiguities and potential ambiguities found during the noncanonical ambiguity check. This might eliminate even more potential ambiguities. For in-

stance, a horizontal ambiguity in the production $A \rightarrow \alpha\beta$ between the sequences α and β always results in a shift/reduce-conflict in one of the productions that can be derived from α.

A. Appendix

Proof of the Undecidability of the Vertical Ambiguity Problem

Proof of Theorem 3.2. Let $G = (N, \Sigma, P, S)$ be a arbitrary grammar. We reduce the *Post Correspondence Problem* (PCP) to the ambiguity problem. The first step is to convert an instance of the PCP to an instance of the ambiguity problem. A solution of the ambiguity problem is a string of terminals that has two or more leftmost derivations for the converted grammar. Then we have to show that if there is such a solution, then the source PCP also has a solution. Therefore if there is an algorithm that decides the ambiguity problem it also decides the PCP, which is known to be undecidable. This is a contradiction.

At first we convert an instance of the PCP to an instance of the ambiguity problem. For a definition of the PCP, see [20] chapter 5.2 for example. Let $(v_1, w_1), \ldots, (v_n, w_n)$ be an instance of the PCP over the alphabet Σ. Define a context-free grammar $G = (\{S, A, B\}, \Sigma \cup \{a_1, \ldots, a_n\}, P, S)$. The productions in P are constructed as follows. For every pair (v_i, w_i) of the PCP add

$$\begin{aligned} A &\rightarrow v_i A a_i \\ A &\rightarrow v_i a_i \\ B &\rightarrow w_i B a_i \\ B &\rightarrow w_i a_i \end{aligned}$$

and additionally

$$\begin{aligned} S &\rightarrow A \\ S &\rightarrow B \end{aligned}$$

to the set of productions.

Now we have to show that both problems either have a solution or none of them. We start with the assumption that there is a solution of the PCP. Then one word x and a sequence $i_1, \ldots, i_m$ exist such that $v_{i_1} v_{i_2} \ldots v_{i_m} = x = w_{i_1} w_{i_2} \ldots w_{i_m}$. Using this information we can construct a string for the converted grammar.

$$v_{i_1} v_{i_2} \ldots v_{i_m} a_{i_m} \ldots a_{i_2} a_{i_1} = x a_{i_m} \ldots a_{i_2} a_{i_1} = w_{i_1} w_{i_2} \ldots w_{i_m} a_{i_m} \ldots a_{i_2} a_{i_1}$$

This string has two leftmost derivations (all derivations $\Rightarrow$ are leftmost derivations here):

$$
\begin{aligned}
S \Rightarrow & A \\
\Rightarrow & v_{i_1} A a_{i_1} \\
\Rightarrow & v_{i_1} v_{i_2} A a_{i_2} a_{i_1} \\
& \vdots \\
\Rightarrow & v_{i_1} v_{i_2} \dots v_{i_{m-1}} A a_{i_{m-1}} \dots a_{i_2} a_{i_1} \\
\Rightarrow & \underbrace{v_{i_1} v_{i_2} \dots v_{i_{m-1}} v_{i_m}}_{x} a_{i_{m-1}} a_{i_m} \dots a_{i_2} a_{i_1}
\end{aligned}
$$

$$
\begin{aligned}
S \Rightarrow & B \\
\Rightarrow & w_{i_1} B a_{i_1} \\
\Rightarrow & w_{i_1} w_{i_2} B a_{i_2} a_{i_1} \\
& \vdots \\
\Rightarrow & w_{i_1} w_{i_2} \dots v_{i_{m-1}} B a_{i_{m-1}} \dots a_{i_2} a_{i_1} \\
\Rightarrow & \underbrace{w_{i_1} w_{i_2} \dots w_{i_{m-1}} w_{i_m}}_{x} a_{i_{m-1}} a_{i_m} \dots a_{i_2} a_{i_1}
\end{aligned}
$$

Therefore $x a_{i_m} \dots a_{i_2} a_{i_1}$ is a solution of the ambiguity problem.

Now we assume that there is a solution of the ambiguity problem for G. At first, we have a look at G_A, the grammar which has A instead of S as start nonterminal. Every word in $\mathcal{L}(G_A)$ has a trailing sequence of a_i terminals. There are only two productions $A \to v_i A a_i$ and $A \to v_i a_i$, which contain a_i for a given i. Because the second production can only be applied as the last one, the terminals a_i uniquely define the sequence of derivations (which are also leftmost derivations since there is at most one nonterminal symbol in the sequence) to be applied. Therefore, there cannot be any ambiguity within G_A. The same applies for G_B, the grammar with B as start symbol.

In consequence, there is just one potential ambiguous choice left in grammar G: derive S to A or B. In both cases, G_A and G_B, the trailing a_i uniquely define the sequence of productions. If these were different, the resulting word would also be different.

Additionally, if there is an ambiguous word $x a_{i_m} \dots a_{i_1}$, $x \in \Sigma^*$ that can be derived in G_A and G_B, the x-part is a sequence of v_i and w_i. This sequence is defined by the sequence of a_i symbols in reverse order. Accordingly, the sequence $i_1 \dots i_m$ must be a solution for the PCP, because $v_1 \dots v_m = x = w_1 \dots w_m$.

We have shown that the PCP can be reduced to the ambiguity problem. If there was an algorithm that would solve the ambiguity problem, it would also solve the PCP. This is a contradiction because it is known that the PCP is undecidable. Hence there can be no algorithm that solves the ambiguity problem. In other words:

the ambiguity problem is undecidable. □

More precisely, this proves the undecidability of the vertical ambiguity problem since the ambiguity between the productions $S \rightarrow A$ and $S \rightarrow B$ matches the definition of a vertical ambiguity (Definition 3.4).

Bibliography

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principples, Techniques, and Tools*. Pearson Education, Inc, Boston, USA, 2nd edition, 2006.

[2] H. J. S. Basten. The usability of ambiguity detection methods for context-free grammars. In *LDTA'08: 8th Workshop on Language Descriptions, Tools and Applications*, 2008.

[3] Claus Brabrand, Robert Giegerich, and Anders Møller. Analyzing ambiguity of context-free grammars. In *Proc. 12th International Conference on Implementation and Application of Automata, CIAA '07*, Prague, Czech Republic, July 2007. `http://www.brics.dk/~amoeller/papers/ambiguity/journal.pdf`.

[4] Geoffrey Clemm. The odin system. In *Selected papers from the ICSE SCM-4 and SCM-5 Workshops, on Software Configuration Management*, pages 241–262. Springer-Verlag, London, UK, 1995.

[5] Free Software Foundation. Bison - GNU Parser Generator. `http://www.gnu.org/software/bison/`.

[6] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 3rd edition, 2006.

[7] Steve C. Johnson. YACC - Yet Another Compiler-Compiler.

[8] Donald Ervin Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965.

[9] Bernard Lang. Parallel non-deterministic bottom-up parsing. In *Proceedings of the international symposium on Extensible languages*, pages 56–57, New York, USA, 1971. ACM.

[10] Mehryar Mohri and Mark-Jan Nederhof. *Regular Approximation of Context-Free Grammars through Transformation*, chapter 9, pages 251–261. Kluwer Academic Publishers, 2000. `http://www.research.att.com/~fsmtools/grm/postscript/approx.ps`.

[11] Anders Møller. dk.brics.grammar. `http://www.brics.dk/grammar/`.

[12] University of Colorado at Boulder. The odin software build system. `ftp://ftp.cs.colorado.edu/pub/cs/distribs/odin/odin-1.17.4.tar.gz`, 2002.

[13] University of Colorado at Boulder, University of Paderborn, and Macquarie University. Eli: Translator construction made easy. `http://eli-project.sourceforge.net/`.

[14] University of Colorado at Boulder, University of Paderborn, and Macquarie University. *Eli Documentation Version 4.4*, May 2007. `http://eli-project.sourceforge.net/elionline4.4/index.html`.

[15] Sylvain Schmitz. Bison 2.3a + ambiguity. `http://www.loria.fr/~schmitsy/papers.html#expamb`.

[16] Sylvain Schmitz. *Approximating Context-Free Grammars for Parsing and Verification*. PhD thesis, Nice Sophia Antipolis University, Nice, France, 2007. `http://www.loria.fr/~schmitsy/pub/phd.pdf`.

[17] Sylvain Schmitz. Conservative ambiguity detection in context-free grammars. volume 4596 of *Lecture Notes in Computer Science*, pages 692–703, Nice, France, 2007. Laboratoire I3S, Nice Sophia Antipolis University & CNRS, Springer. `http://www.loria.fr/~schmitsy/papers.html#ambiguity`.

[18] Sylvain Schmitz. An experimental ambiguity detection tool. *Electronic Notes in Theoretical Computer Science*, 203(2):69–84, April 2008. `http://www.loria.fr/~schmitsy/pub/expamb.pdf`.

[19] Ulf Schwekendiek. Integrating a GLR Parser Generator in Eli. Bachelor's Thesis, August 2007. `http://ag-kastens.uni-paderborn.de/paper/Bachelor_Schwekendiek.pdf`.

[20] Michael Sipser. *Introduction to the Theory of Computation*. Course Technology, 2nd edition, 2005.

[21] Masaru Tomita. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, Norwell, USA, 1985.

[22] Karel Čulik II and Rina Cohen. Lr-regular grammars – an extension of lr(k) grammars. *Journal of Computer and System Sciences*, 7(1):66–96, 1973.