Diploma Thesis

Incremental Greibach Normal Form

August 2011

Chair for Software Modeling and Verification
Prof. Dr. Joost-Pieter Katoen
RWTH Aachen University

Author:	Markus Bals
Matriculation Number:	248876
First Revisor:	PD Dr. Thomas Noll
Second Revisor:	Prof. Dr. Joost-Pieter Katoen
Supervisor:	Dipl. Inf. Christina Jansen

Eidesstattliche Erklärung

Ich versichere hiermit eidesstattlich, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen oder anderen Quellen entnommen sind, sind als solche eindeutig kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form noch nicht veröffentlicht und noch keiner Prüfungsbehörde vorgelegt worden.

Aachen, den 26.08.2011

Markus Bals

Contents

1 Introduction **1**

2 The Greibach Normal Form **3**
2.1 An Algorithm for Constructing the Greibach Normal Form 4

3 An Incremental Approach **11**
3.1 Basic Concept and Data Structures 11
3.2 The Incremental Algorithm 20
3.3 Correctness 33
3.4 Complexity 43

4 Optimization **47**
4.1 Adding Multiple Productions 47
4.2 Treating Recursive Nodes Like Non-Ordered Nodes 47
4.3 Managing Worklists 49
4.4 Checking for Usage Pointer and Worklist 52
4.5 Inserting New Nonterminals 52
4.6 Finding Represented Productions 52

5 Testing an Implementation of the Incremental Algorithm **55**

6 Conclusion **61**
6.1 Summary 61
6.2 Future Work 61

1 Introduction

The Greibach Normal Form (GNF) that was devised by Sheila A. Greibach in 1965 [3] is defined for context-free string grammars (CFG).

The absence of left recursions and the characteristic to build terminal words from left to right while increasing the length of the terminal part in each derivation step are its main properties. They allow to use GNF normalized grammars for various proof techniques as well as for compiler construction.

There exists a well-known algorithm to obtain this normal form which is described in many beginners textbooks on theoretical computer science and is known being correct.

So why review such a well known concept? There are many fields in todays computer sience where a short runtime is required, but establishing the GNF for certain grammars can be a time consuming process. This becomes a problem when changes to a grammar are applied frequently. In that case the same steps to obtain the GNF are executed over and over again.

The GNF is not only defined for context-free grammars. There are definitions for Hyperedge Replacement Grammars (HRG) by Engelfriet, Heyker, Leih [2] and by Jansen, Heinen, Noll, Katoen [4]. The latter define a Local Greibach Normal Form (LGNF) which is closer related to the definition of the GNF used in this thesis than the first. Applications of those definitions will suffer from the same problems as the original definition does.

In this thesis an algorithm is developed that improves this runtime problem. The algorithm employs structures that let the user work with an original grammar and memorize the steps that were executed to build an equivalent grammar in GNF. If the user modifies the original grammar only those steps are reevaluated that are affected by the modification. Since such an approach utilizes previous results by modifiying these results with respect of changes to the original grammar we call it *incremental*.

An advantage beside the increased efficiency of this incremental approach is that when including additional productions, the algorithms establishes the GNF without modifying the resulting GNF productions of previous runs. This means no productions are deleted or modified, only new productions are introduced.

Based on the existing *basic algorithm* to construct the GNF, this work develops data structures that preserve the information of how the basic algorithm obtained the GNF for an input grammar. The incremental algorithm that is constructed operates on this data structures allowing to add productions to the original input grammar, establishing the GNF for the resulting new input grammar. At the same time the information of how this was accomplished is augmented such that further productions can be added.

The data structures and the incremental algorithm itself are formally defined. The algorithm is proved correct in the sense of grammar equivalence.

Additional concepts to improve the runtime of the algorithm further and to increase its flexibility are developed and shown by example. This includes adding of multiple productions at once when adding productions incrementally as well as reducing the amount of intermediate productions and operations performed on an input grammar.

An implementation of the incremental algorithm is done in context of this thesis and its performance is tested. The incremental implementation shows significantly improved performance in every respect when compared to an implementation of the

basic algorithm. The performance is discussed with respect to the total runtime and the amount of performed modifications on the input grammar.

Structure of the Thesis

In Section 2 the Greibach Normal Form is defined and a well-known algorithm to obtain it is presented. Restrictions to the input grammars are discussed.

In Section 3 an incremental version of the *basic algorithm* of Section 2 is constructed. Data structures to preserve “incremental” information are developed on which the basic algorithm is simulated. The incremental algorithm is proved correct and its complexity is analyzed.

In Section 4 several methods to improve the performance as well as the flexibility of the incremental algorithm are developed and shown by example.

In Section 5 an implementation of the incremental algorithm is tested on different grammars. The results are compared to those of an implementation of the basic algorithm with respect to the runtime as well as the amount of executed operations and intermediate steps.

In Section 6 a brief summary of what has been accomplished in thesis is presented as well as suggestions to what this incremenal algorithm can contribute to in the future.

2 The Greibach Normal Form

Before giving an exact definition of the GNF we will give a convention of naming alphabets and symbols. A nonterminal symbol is denoted by a capital roman letter with probably an index written as a subscript. Some finite alphabet of nonterminal symbols is also written as a capital roman letter, normally as N ("N" for nonterminals). Terminal symbols are denoted by small roman letters while a finite terminal alphabet is denoted by capital greek letters, mostly Σ. Words over some alphabet are written as small greek letters like $\alpha \in \Sigma^*$ or $\gamma \in (\Sigma \cup N)^+$. The empty word is denoted by ε.

We will assume that the grammars considered here are always productive, i.e. for each nonterminal symbol of a grammar a terminal word can be derived.

Definition 2.1 (Productive Grammar)**.** *Let $G = (N, \Sigma, P, S)$ be an arbitrary context-free grammar. G is called* productive *if all productions $p \in P$ contain terminating symbols only.*

A symbol $X \in (\Sigma \cup N)$ is called terminating *if a terminal word can be derived from it, i.e. $X \Rightarrow_G^* \gamma$ with $\gamma \in \Sigma^+$. Otherwise the symbol is called* nonterminating.

Considering the fact that omitting productions containing nonterminating symbols does not change the language of the grammar barely restricts the scope of possible input grammars.

Requiring the grammar being productive after the insertion of a single production however is not a restriction at all, because when adding a set of new productions that keeps a grammar productive, there is a way to add the new productions in a sequence so that the grammar never contains any nonterminating symbols after each insertion.

Lemma 2.1. *Let $G = (N, \Sigma, P, S)$ be a productive context-free grammar and $P' = \{p_1, \ldots, p_k\}$ be a set of productions so that $G' = (N', \Sigma, P \cup P', S)$ is productive again. The productions in P' might contain new nonterminals so that adding those new nonterminals to N yields N'.*

There exists a permutation $(i_1, \ldots, i_k)$ of the sequence $(1, \ldots, k)$ so that

$$G' := (N, \Sigma, P \cup \{p_{i_j} \mid j \in \{1, \ldots, h\}\}, S)$$

is productive for every $h \in \{1, \ldots, k\}$.

This means that there exists an order for P' so that if the productions of P' are added to P following this order, G will stay productive after each single insertion.

Proof of Lemma 2.1. If the set of new productions $P' = \{p_1, \ldots, p_k\}$ contains no new nonterminal symbols the claim holds trivially, since no right hand side contains a nonterminating symbol. Therefore any sequence of productions will keep the grammar productive.

Assume the non-trivial case that there is at least one production $p \in P'$ that contains new nonterminals on its right hand side.

The upper bound for the number of new nonterminals is k because there has to be at least one production in P' for each new nonterminal for which this nonterminal is the left hand side symbol. Otherwise the symbol would be nonterminating for $P \cup P'$.

Suppose all right hand sides of productions of new nonterminals contain new nonterminals. In this case all new nonterminals would not be terminating. So there has to be at least one production in P' for which a new nonterminal produces terminating symbols only and is therefore itself terminating.

This terminating production is made first in the sequence and the problem is reduced to $k-1$ productions with at most $k-1$ new nonterminals since the symbol we just identified as terminating is no longer considered new.

The same conditions as above apply for $k-1$ again so the sequence can be built until no production is left or there are only productions left that contain no new nonterminals and are terminating anyway. □

Also we will only consider ε-free grammars which means that the grammar is not allowed to contain a production $A \to \varepsilon$ that produces the empty word. This does also not reduce the number of possible grammars with repspect to the languages of them since grammars containing ε-productions can be transfered to an equivalent grammar without ε-productions. For a proof of Lemma 2.2 cf. Wood [6].

Lemma 2.2. *Every context free grammar G with $\varepsilon \notin L(G)$ that contains ε-productions can be transfered to an ε-free grammar G' such{ that $L(G) = L(G')$.*

If it is required that $\varepsilon \in L(G)$ we can work with an ε-free grammar $G = (N, \Sigma, P, S)$ and in the end introduce a new startsymbol $S' \notin N$ and add the productions $S' \to \varepsilon \mid S$.

Another special case that is of interest are productions of the form $A \to A$. Those productions are either not productive if they are the only production for nonterminal A or they obviously do not not change the language of the grammar. So a production of that form is either not valid with respect to the productivity or does not change the language of the grammar and therefore can be omitted safely.

Definition 2.2 (Greibach Normal Form)**.** *Let G be a context-free grammar, Σ a terminal alphabet, N a nonterminal alphabet and $\Sigma_N := \Sigma \cup N$. The grammar G is in Greibach Normal Form iff all productions in G are of the form*

$$A \to a\gamma$$

where $a \in \Sigma$ is a terminal symbol and $\gamma \in \Sigma_N^*$ is a possibly empty word over Σ_N. Single productions have the Greibach property *if they follow this restriction.*

If $\gamma \in N^*$ for every production, i.e. all right hand sides contain no terminal symbols beside the leftmost symbol, G is in "strong" GNF.

2.1 An Algorithm for Constructing the Greibach Normal Form

One method to obtain the GNF for a context-free string grammar consists of two steps (cf. Asteroth, Baier [1]). In order to obtain the strong GNF an additional step has to be applied before or after the original two.

The basic idea of this algorithm is to establish the GNF by modifying the productions of an input grammar based on a total order of its nonterminals while preserving the language of the grammar.

Given some context-free grammar G with a finite set N' of nonterminals with $|N'| = n$, i.e. N' containing n symbols. Those symbols can be renamed without changing the language of G. We therefore w.l.o.g. let the nonterminal alphabet be $N = \{A_1, A_2, \ldots A_n\}$ and assume a total order of N so that $A_i < A_j$ if $i < j$, i.e. A_1 is the least and A_n the greatest nonterminal symbol.

In the first step the productions are modified such that when left deriving a nonterminal a certain nonterminal can only occur once at the leftmost position, more precisely

allowing only terminal symbols or greater nonterminal symbols being produced at the leftmost position, i.e.

$$A_{i_1}\gamma_1 \Rightarrow_L A_{i_2}\gamma_2 \Rightarrow_L \ldots \Rightarrow_L A_{i_n}\gamma_n$$

so that $A_j \neq A_k$ for all $j, k \in \{i_1, \ldots, i_n\}$ and $j \neq k$. Any grammar fulfilling this restriction is free of left recursions.

Definition 2.3 (Left Recursion)**.** *Let $G = (N, \Sigma, P, S)$ be a context-free grammar and $A \in N$ a nonterminal of G. A production $A \to \alpha$ with $\alpha \in (\Sigma \cup N)^*$ is called* left recursive *if $\alpha \Rightarrow_G^* A\beta$ with $\beta \in (\Sigma \cup N)^*$. A production of the form $A \to A\alpha$ is called* direct left recursive.

This means that if there is a production $A \to \alpha$ the nonterminal A can be derived at the leftmost position again by starting with α and using left derivations only.

Once all productions of a grammar follow that condition all productions of the greatest nonterminal are automatically in GNF since only greater nonterminals are allowed at the leftmost position. By proceeding down in the total order of nonterminals the GNF can be established for all productions in the second step of the algorithm.

This total order can be chosen arbitrarily but has to stay fixed once it has been chosen. It is relevant for the performance of the algorithm how this order is chosen, but this problem will not be discussed here.

The interdependency regarding left derivation can be established by requiring $i < j$ for each production of the form $A_i \to A_j\gamma$, i.e. when building left derivations of some nonterminal symbol A_i only symbols A_j with $j > i$ or terminal symbols are created at the leftmost position.

Definition 2.4 (Ordered Production)**.** *Given a grammar $G \in CFG$ with terminal alphabet Σ and an ordered nonterminal alphabet $N = \{A_1, A_2, \ldots A_n\}$ such that $A_i < A_j$ for $i < j$.*

A production of G is called ordered *if it is either of the form $A_i \to a\gamma$ or $A_i \to A_j\gamma$ with $i \leq j$ where $\gamma \in (\Sigma \cup N)^*$ and $a \in \Sigma$.*

Ordering Productions *in this context means that the productions of a grammar are modified so that each production is of that form.*

If all productions are ordered and all direct left recursions are removed, the productions for the greatest nonterminal A_n are in GNF. Now all productions that are not in GNF get their leftmost nonterminal replaced by doing a left derivation of their right hand side resulting in a terminal symbol at the leftmost position. This process starts with the A_{n-1} productions and proceeds down to A_1.

Ordering all productions, removing direct left recursions and establishing the Greibach property for the productions that are ordered and free of direct left recursion is accomplished by two language preserving transformations for context-free grammars.

Definition 2.5 (Language Preserving Operations)**.** *Let $G \in CFG$ be a context-free grammar. We define two operations that transform G by changing the productions of G and also introducing new nonterminals for the latter.*

1. *Derivation of the right hand side*

   Let $A \to \alpha B\gamma$ be a production in G and let $B \to \beta_1 \mid \ldots \mid \beta_k$ be all productions in G with left hand side B. The production $A \to \alpha B\gamma$ can be replaced by the

productions $A \to \alpha\beta_1\gamma \mid \ldots \mid \alpha\beta_k\gamma$ *yielding the modified grammar* G'.
Since the algorithm uses this transformation only to replace a nonterminal at the leftmost position (i.e. $\alpha = \varepsilon$*) this transformation is often refered to as "replacing a production by its left derivations".*

2. *Removing direct left recursion*

   Let $A \to A\alpha$ *be a direct left recursive production and let* $A \to \beta_1 \mid \ldots \mid \beta_k$ *be all productions with left hand side A that are not left recursive.*

   The production $A \to A\alpha$ *can be removed by introducing a new nonterminal symbol B and the new productions* $B \to \alpha$ *and* $B \to \alpha B$ *as well as the productions* $A \to \beta_1 B \mid \ldots \mid \beta_k B$.

Modifying a context-free grammar in this way is language preserving, which has been shown in Wegener [5].

Lemma 2.3. *Transforming a context-free grammar G to G′ by using the operations as in Definition 2.5 does not change the language generated by that grammar such that* $L(G) = L(G')$.

The algorithm transforms the original context-free grammar by modifying productions using those operations until all productions are ordered and free of left recursions and then established the Greibach property for each production. Algorithm 2.1 shows the general operation method of this basic approach (cf. Asteroth, Baier [1]).

Algorithm 2.1 A basic algorithm for transfering a CFG grammar to GNF

Require: productive, ε-free, context-free grammar $G = (N, \Sigma, P, S)$ with an ordered set of nonterminals $N = \{A_1, \ldots, A_n\}$

```
 1: for i = 1 to n do
 2:    for all productions A_i → α do
 3:       order the production A_i → α
 4:    end for
 5:    for all left recursive productions A_i → A_iα do
 6:       remove left recursion for A_i → A_iα
 7:    end for
 8: end for
 9: for j = n to 1 do
10:    for all productions A_j → α do
11:       establish Greibach property for A_j → α
12:    end for
13: end for
14: for k = 1 to n do
15:    for all productions B_k → α do
16:       establish Greibach property for B_k → α
17:    end for
18: end for
```

Ensure: a modified grammar G' in GNF with $L(G) = L(G')$

In the first step of the algorithm the productions are being ordered by replacing productions by their left derivations if they are not ordered yet. If resulting new productions are still not ordered they need to be replaced by their left derivations again and so on.

The goal of the ordering is to enable the algorithm to establish the Greibach property for each production by replacing it with its left derivations only once which happens in the second part of the algorithm. This does obviously not work for direct left recursive productions, but those are considered ordered so they need to be removed seperately. After all productions for some nonterminal A_i have been ordered, the left recursive productions are replaced by right recursive productions using the second operation of Definition 2.5.

If there are direct left recursive productions for A_i a new nonterminal symbol B_i is introduced and two new productions of B_i for each left recursive production of A_i.

$$A_i \rightarrow A_i\gamma \quad \text{results to} \quad B_i \rightarrow \gamma \text{ and } B_i \rightarrow \gamma B_i$$

The original left recursive productions are deleted. In order to give the A_i productions access to the recursive parts again a new right recursive production is added for each non-recursive A_i production, i.e. for each $A_i \rightarrow \gamma$ a new production $A_i \rightarrow \gamma B_i$ is added if $A_i \rightarrow \gamma$ is not left recursive as in Definition 2.3.

Note that this removal of direct left recursions is done for the A_i productions before any A_j production with $i < j$ is processed.

Once all production are ordered and no longer left recursive the algorithm can establish the Greibach property starting with the productions of the greatest nonterminal symbol A_n. Those productions are already in GNF because they have a terminal symbol at their leftmost position of their right hand sides otherwise they would be either direct left recursive or not ordered. Productions of A_{n-1} can have the nonterminal A_n at their leftmost position so replacing those productions with their left derivations results in productions that are in GNF. In contrast to the ordering process the resulting productions of such a replacement need not to be checked again. In this way the algorithm proceeds down to A_1 establishing the Greibach property for all productions that have some A_i on their left hand side.

This leaves the B_i productions that might have been introduced when the direct left recursions were removed. But since those productions are either in GNF already or have some A_i at their leftmost position the Greibach property can be established by applying the replacement by left derivations once at most. Also different B_i productions are not interdependent so the order in which they are processed is irrelevant.

The two steps (or three steps for strong GNF) can now be defined more precisely. Let $G' = (N', \Sigma, P, S)$ denote an arbitrary context-free grammar and let N be the ordered set of nonterminals as defined earlier renaming only N' so that $L(G) = L(G')$ where $G = (N, \Sigma, P, S)$. Furthermore $\gamma \in \Sigma_N^*$ denotes a (possibly empty) word over terminals and nonterminals.

1. Order all productions and remove direct left recursions.

   (a) (ordering of productions) All productions of G are transfered to either the form

   $$A_i \rightarrow a\gamma \quad \text{or} \quad A_i \rightarrow A_j\gamma$$

   where $i \leq j$.

(b) (remove direct left recursions) All productions of the form $A_i \to A_i\gamma$ are removed and replaced by

$$B_i \to \gamma \mid \gamma B_i \quad \text{and} \quad A_i \to \beta_1 B_i \mid \ldots \mid \beta_k B_i$$

where $A_i \to \beta_1 \mid \ldots \mid \beta_k$ are all productions of A_i that are not direct left recusive and B_i is a distinct new nonterminal symbol.

2. Establish the Greibach property for all productions.

   (a) (bring original nonterminals into GNF) All productions of the form $A_i \to A_j\gamma$ where $j > i$ are transfered to GNF by replacing the production by its left derivations.

   (b) (bring new nonterminals into GNF) All productions of the form $B_i \to A_j\gamma$ are transfered to GNF in the same way.

3. (replace terminal symbols for strong GNF) All productions that contain a terminal symbol $a \in \Sigma$ on their right hand side other than the leftmost position get that symbol replaced by a new nonterminal A_a and a new production $A_a \to a$ is added. This optional step can be included to obtain a grammar in strong GNF.

Example 2.1. *Let $G = (\Sigma, N, P, S)$ be a grammar with terminals $\Sigma = \{a, b, c\}$, nonterminals $N = \{A_1, A_2, A_3\}$ and productions*

$$\begin{aligned} A_1 &\to A_2b \mid A_3c \\ A_2 &\to a \\ A_3 &\to A_1A_3 \mid A_3A_4 \mid b \\ A_4 &\to c \end{aligned}$$

The start symbol S will not be used in this context.

The algorithm starts with the least nonterminal symbol A_1 and repeats the ordering procedure followed by the removal of direct left recursions for all nonterminal symbols up to A_4.

The first changes to the original grammar are done to the A_3 productions because all A_1 and A_2 productions are ordered and not direct left recursive. The production $A_3 \to A_1A_3$ is replaced by two new productions $A_3 \to A_2bA_3 \mid A_3cA_3$, where $A_3 \to A_2bA_3$ has to be ordered again and is replaced by $A_3 \to abA_3$. The productions for A_3 are ordered but there are two left recursive productions.

$$A_3 \to abA_3 \,\Big|\, A_3cA_3 \,\Big|\, A_3A_4 \,\Big|\, b$$

Now in order to remove the left recursions a new nonterminal symbol B_3 is introduced and each recursive production is replaced by two new productions. These are $B_3 \to cA_3 \mid cA_3B_3$ and $B_3 \to A_4 \mid A_4B_3$. This now right recursive part has to be added to the remaining non-recursive A_3 productions.

$$\begin{aligned} A_1 &\to A_2b \mid A_3c \\ A_2 &\to a \\ A_3 &\to abA_3 \mid abA_3B_3 \mid b \mid bB_3 \\ A_4 &\to c \\ B_3 &\to cA_3 \mid cA_3B_3 \mid A_4 \mid A_4B_3 \end{aligned}$$

The only A_4 production as well as the productions for A_3 and A_2 are already in GNF. When checking the A_1 productions, the algorithm finds $A_1 \to A_2b$ first. There is only one A_2 production to apply, so the only new production is $A_1 \to ab$ replacing the old one. Then $A_1 \to A_3c$ is found and is replaced by $A_1 \to abA_3c \mid abA_3B_3c \mid bc \mid bB_3c$. So all productions with one of the original nonterminals on the left hand side are in GNF now and the grammar reads

$$\begin{aligned}
A_1 \to &\quad ab \mid abA_3c \mid abA_3B_3c \mid bc \mid bB_3c \\
A_2 \to &\quad a \\
A_3 \to &\quad abA_3 \mid abA_3B_3 \mid b \mid bB_3 \\
A_4 \to &\quad c \\
B_3 \to &\quad cA_3 \mid cA_3B_3 \mid A_4 \mid A_4B_3
\end{aligned}$$

In the final step the productions for the new nonterminal B_3 have to be transfered to GNF. The productions $B_3 \to A_4 \mid A_4B_3$ are replaced by their left derivations yielding $B_3 \to c \mid cB_3$. So the final grammar in GNF reads

$$\begin{aligned}
A_1 \to &\quad ab \mid abA_3c \mid abA_3B_3c \mid bc \mid bB_3c \\
A_2 \to &\quad a \\
A_3 \to &\quad abA_3 \mid abA_3B_3 \mid b \mid bB_3 \\
A_4 \to &\quad c \\
B_3 \to &\quad cA_3 \mid cA_3B_3 \mid c \mid cB_3
\end{aligned}$$

To obtain the strong GNF the new nonterimnals A_b and A_c need to be introduced in the additional step along with the productions $A_b \to b$ and $A_c \to c$. All terminals symbols b and c on the right hand sides except those at the leftmost position are replaced by A_b and A_c respectively.

$$\begin{aligned}
A_1 \to &\quad aA_b \mid aA_bA_3A_c \mid aA_bA_3B_3A_c \mid bA_c \mid bB_3A_c \\
A_2 \to &\quad a \\
A_3 \to &\quad abA_3 \mid abA_3B_3 \mid b \mid bB_3 \\
A_4 \to &\quad c \\
B_3 \to &\quad cA_3 \mid cA_3B_3 \mid c \mid cB_3 \\
A_b \to &\quad b \\
A_c \to &\quad c
\end{aligned}$$

Note that the order in which the productions are processed for a certain left hand side A_i does not matter. There are only three rules that have to be followed.

1. When ordering the productions no A_i production is allowed to be processed before all A_j productions for any $j < i$ have been processed.

2. When removing a direct left recursion for left hand side A_i all productions for A_i must have been ordered.

3. When establishing the Greibach property no A_i production is allowed to be processed before all A_j productions for some $j > i$ have been processed.

3 An Incremental Approach

An incremental algorithm that generates a grammar in GNF from a context-free string grammar has to utilize previous results, i.e. changes applied to the original grammar only change and extend the previous GNF grammar at those points where it is necessary instead of constructing it from scratch. This is accomplished by modeling a data structure that holds information about how the basic algorithm established the GNF for a grammar. Basically the incremental algorithm simulates the original algorithm on the new data structure and avoids performing unnecessary operations.

As stated earlier the approach followed here generates a grammar in GNF for a given original grammar and allows changes to this original grammar only. Basic operations are adding of new productions, removing existing ones and change existing ones where the latter can be implemented by combining a remove and add operation.

The essential operation for modifiying the original grammar is adding new productions with possibly new nonterminal symbols on the left hand side (not on the right hand side because of the productivity constraint). The question is how the new production affects previous "replacement chains" in the different steps.

The other basic modification, the removal of productions, enables changing of productions as well by removing the old production and adding the changed version as a new one. We will try to keep the algorithm as flexible as possible in order to support this additional modifications, but they will not be examined any further for now.

3.1 Basic Concept and Data Structures

Let G be a grammar with a finite nonterminal alphabet $N = \{A_1, \ldots, A_n\}$ consisting of n symbols. A new production is of the form $A_x \to \gamma$ with $\gamma \in {\Sigma_N}^+$ and $x \in [0, n]$ where $x = 0$ denotes a new nonterminal symbol. Lets first consider the left hand side A_0. If the new nonterminal is inserted as the least symbol it is already ordered and does not need being considered for establishing the Greibach property since it does not occur in the original productions. The new production could however have a direct left recursion, but this would not affect any other productions. So adding a new production with a new nonterminal on the left hand side in this way is easy since the new production will always be ordered already.

If a new production with the left hand side symbol A_i with $i \in [1, n]$ is added this could have affected other productions while ordering, but only those with left hand side A_j with $j > i$. A naive approach would be reordering all $A_j \to \gamma$ with $j > i$. This is not a particulary efficient way since adding A_1 productions for instance would cause the algorithm to reconstruct the entire GNF grammar.

A better approach is to remember at which points productions for a certain symbol A_i were used. To accomplish this we create a so called production tree for each nonterminal symbol A_i of the original grammar. Such a tree has its root node labeled with A_i and will be refered to as the A_i tree. The children of a root node A_i are nodes that are labeled with the right hand sides of the productions of A_i. So the nodes (except the root nodes) represent production, e.g. a node within the A_i tree that is labeled with γ represents the production $A_i \to \gamma$. Due to that the terms *node* and *production* are used synonymously in this context. Another property of the production trees is that child nodes always represent a left derivation of their parent node.

So the current set of productions during the execution of the algorithm is always given by the leaf nodes of the production trees. Inner nodes represent intermediate steps that were further processed by the algorithm, i.e. inner nodes represent productions

that are replaced by the productions represented by their child nodes using the first operation described earlier (replacing a production by its left derivations).

Definition 3.1 (Production Tree)**.** *Let $G = (N, \Sigma, P, S)$ be a productive context-free string grammar and $A \in N$.*

The production tree $T_A^G = (V, E, lab)$ *is an acyclic tree with a finite set of nodes V and a set of edges $E \subseteq (V \times V)$ that is defined for the nonterminal A of the grammar G. $lab : V \to (\Sigma \cup N)^+$ is a labeling function that assigns a non-empty word over $(\Sigma \cup N)$ to each node in V.*

The production tree T_A^G *is characterized by the following properties:*

- *The root node $r \in V$ is labeled with $lab(r) = A$.*
- *Let $v \in V$ be a node in T_A^G labeled with $lab(v) = B\gamma$ where $B \in N$ and $\gamma \in (\Sigma \cup N)^*$. If v has children and $B \to \beta_1 \mid \ldots \mid \beta_k$ are all productions in P with left hand side B, there has to be a exactly one child that is labeled with $\beta_i\gamma$ for each $i \in [1, \ldots, k]$.*
- *A node $v \in V$ that is labeled with $a\gamma$ where $a \in \Sigma$ and $\gamma \in (\Sigma \cup N)^*$ is a leaf node.*

A node $v \in V$ that is labeled with $A\gamma$ where $\gamma \in (\Sigma \cup N)^+$ is refered to as a recursive node *otherwise as a* non-recursive node.

As abbreviation we sometimes write $v \in T_A^G$ when meaning $v \in V$ for $T_A^G = (V, E, lab)$ or omit G and write T_A if the grammar is known from the context.

A leaf node in such a production tree can be marked as blind leaf to exclude it from the set of productions that are represented by a production tree which is defined below.

Definition 3.2 (Represented Productions)**.** *Let T_A^G be a production tree that has its leaf nodes that are not marked as blind leaves labeled with $\gamma_1, \ldots, \gamma_n$.*

The set of represented productions *of T_A^G is defined as*

$$P(T_A^G) := \left\{A \to \gamma_1, \ldots, A \to \gamma_n\right\}$$

Direct left recursive productions are never used for replacement by left derivations and are not part of GNF grammars. A recursive node, however, cannot be deleted because this could make its parent node a leaf node which would therefore represent a production that is not supposed to be part of the grammar. Also cf. Example 3.4 which describes this matter.

Example 3.1. *Consider Figure 1 where both trees are valid production trees for the grammar G from Example 2.1 with productions*

$$\begin{aligned} A_1 &\to A_2 b \mid A_3 c \\ A_2 &\to a \\ A_3 &\to A_1 A_3 \mid A_3 A_4 \\ A_4 &\to c \end{aligned}$$

The root nodes are drawn as circles and are shaded yellow while the other nodes are rectangles and are not shaded. All nodes contain their label and are tagged at the top left with their identification in the set of nodes V. We will only use those identifications here to ilustrate the formal definition of production trees and will omit them later.

The production trees are defined as

$$\begin{array}{lll} V_{left} & = \{ & v_1, v_2, v_3 \quad \} \\ E_{left} & = \{ & (v_1, v_2),\ (v_1, v_3) \quad \} \\ lab_{left} & = \{ & v_1 \mapsto A_1,\ v_2 \mapsto A_2 b, \\ & & v_3 \mapsto A_3 c \quad \} \end{array} \qquad \begin{array}{lll} V_{right} & = \{ & v_1, v_2, v_3, v_4 \quad \} \\ E_{right} & = \{ & (v_1, v_2),\ (v_1, v_3),\ (v_2, v_4) \quad \} \\ lab_{right} & = \{ & v_1 \mapsto A_1,\ v_2 \mapsto A_2 b, \\ & & v_3 \mapsto A_3 c,\ v_4 \mapsto a \quad \} \end{array}$$

Note that the first tree is also a valid production tree for a grammar with arbitrary A_2 productions while the second tree requires a grammer that contains the production $A_2 \to a$ and no other A_2 productions.

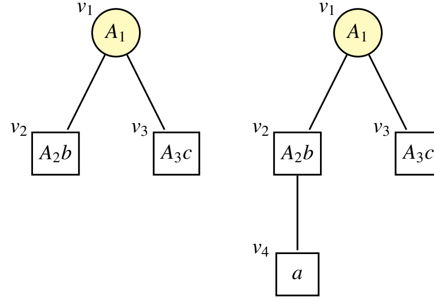

Figure 1: Two different production trees defined for nonterminal A_1

The incremental algorithm will create a production tree for each nonterminal in the grammar. Multiple production trees can only be combined to a *forest of production trees* under certain conditions. They need to be defined for the same alphabet and start symbol. Also there has to be exactly one tree for each nonterminal, otherwise the forest would not define a grammar properly.

Definition 3.3 (Forest of Production Trees)**.** *Let $F = \left\{T_{A_1}^{G_1}, \ldots, T_{A_n}^{G_n}\right\}$ be a set of production trees with $G_i = (N_i, \Sigma_i, P_i, S_i)$ and $A_i \in N_i$ for $i \in [1, \ldots, n]$.*

F is a forest of production trees *if*

- *Each grammar G_i that defines a production tree of F has the same alphabet so that $N := N_1 = N_2 = \ldots = N_n$ and $\Sigma := \Sigma_1 = \Sigma_2 = \ldots = \Sigma_n$.*
- *All grammars G_i have the same start symbol $S \in N$.*
- *For each $A \in N$ there is exactly one production tree T_A^G in F.*

Defining a forest of production trees with these constraints allows to uniquely define a context-free grammar for a given forest.

Definition 3.4 (Grammar of a Forest)**.** *Let $F = \{T_{A_1}^{G_1}, T_{A_2}^{G_2}, \ldots, T_{A_n}^{G_n}\}$ be a forest of production trees and $P(T_{A_i}^{G_i}), i \in [1, \ldots, n]$ the sets of productions represented by the trees in F.*

The grammar of a forest *F with $|N| = n$ is defined as*

$$G(F) := (N, \Sigma, \cup_{i=1}^{n} P(T_{A_i}^{G_i}), S)$$

Before defining the language preserving operations from Definition 2.5 we need to define how new productions are added to an existing forest of production trees.

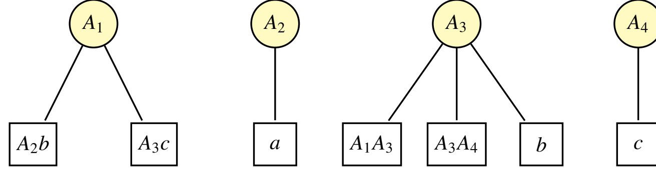

Figure 2: Grammar in tree representation

Definition 3.5 (Adding a Production to a Forest)**.** *Let F be a forest of production trees representing the grammar $G(F) = (N, \Sigma, P, S)$. A new production $(A \to \gamma) \notin P$ is added to F by the following steps.*

1. *If $A \notin N$ add A to N as the least element.*
2. *If the tree $T_A^{G_A} = (V, E, lab)$ does not exist yet it is created containing its root node only such that $V = \{r_A\}$, $E = \emptyset$ and $lab(r_A) = A$.*
3. *Modify $T_A^{G_A}$ by introducing a new node v to V and extending the labeling function such that $lab(v) = \gamma$.*
4. *Make v a child of the root node, i.e. add the edge (r_A, v).*

The forest of production trees in Figure 2 represents the grammar G known from Example 2.1. A forest containing only trees with depth one is a special case and will be refered to as *direct representation*. The grammars $G_1, \ldots, G_n$ for which the different production trees in the forest are defined (as in Definition 3.3) can be chosen as $G_i = G$ for all $i \in \{1, \ldots, n\}$. Note that in this work two context-free grammars G and G' are considered equal if in addition to requiring $L(G) = L(G')$ their sets of productions P and P' have to be equal as well.

When starting with an empty forest and adding the productions of an input grammar G as in Definition 3.5 the algorithm can start to simulate the original algorithm on the forest which is a direct representation of G. To do that the language preserving operations from Definition 2.5 need to be defined for a forest of production trees.

Definition 3.6 (Replacing a Node by its Left Derivations)**.** *Let F be a forest of production trees, $\{T_{A_i}^{G}, T_{A_j}^{G'}\} \subseteq F$ being two different trees in F and let $v \in V_{A_i}$ be a leaf node in $T_{A_i}^{G}$ labeled with $lab^{A_i}(v) = A_j\gamma$ where $\gamma \in (\Sigma \cup N)^*$. Also let $P(T_{A_j}^{G}) = \{A_j \to \beta_1 \mid \ldots \mid \beta_k\}$ be the productions represented by $T_{A_j}^{G}$.*

The replacement by left derivations *for the leaf node $v \in V_{A_1}$ is defined by creating k child nodes $w_1, \ldots, w_k$ of v and extend the labeling function lab^{A_i} so that*

$$lab^{A_i}(w_m) = \beta_m\gamma \quad for \quad m \in \{1, \ldots, k\}$$

In Definition 3.6 we see that replacement by left derivations can be easily transfered to the concept of production trees.

Example 3.2 (Replacement by Left Derivations)**.** *Figure 3 shows the T_{A_1} and T_{A_2} trees as an Figure 2 where the node labeld A_2b (marked green) is replaced by its left derivations. In this case this is just one since the only represented production of T_{A_2} is $P(T_{A_2}) = \{A_2 \to a\}$. The resulting trees are shown on the right side with the resulting child node labeld ab (marked green again).*

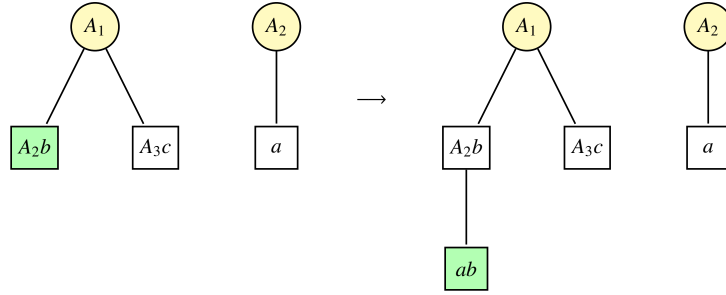

Figure 3: Applying replacement by left derivations on node A_2b

The fact that Definition 3.6 is an exact simulation of Definition 2.5(1) on a forest of production trees is stated in Lemma 3.1.

Lemma 3.1. *Let $G \in CFG$ be a productive, context-free grammar and let F be a forest of production trees such that $G = G(F)$. Now let the grammar G' and the forest F' be the results of replacing a production of G by its left derivations as in Definition 2.5(1) and replacing the node which represents the same production in F as in Definition 3.6.*

The grammar $G(F')$ represented by F' equals G'.

Proof. Let $G \in CFG$ and F a forest of production trees such that $G(F) = G$. Further let $A_i, A_j \in N$ be nonterminals of the grammars and let $p = (A_i \to A_j\gamma)$ be a production in G.

Since G is productive the nonterminal A_j has to be terminating, i.e. there have to be productions of A_j in G. Let those productions be $A_j \to \beta_1 \mid \ldots \mid \beta_k$. According to Definition 2.5(1) the production p is deleted and replaced by $A_i \to \beta_1\gamma \mid \ldots \mid \beta_k\gamma$ resulting in the grammar G'.

Because of $G = G(F)$ there have to be production trees T_{A_i} and T_{A_j} in F such that $p \in P(T_{A_i})$ and $P(T_{A_j}) = \{A_j \to \beta_1 \mid \ldots \mid \beta_k\}$. The node v which represents p has to be a leaf node in T_{A_i}. Replacing v as in Definition 3.6 creates k child nodes $w_1, \ldots, w_k$ labeled $lab^{A_i}(w_m) = \beta_m\gamma$ for $m \in \{1, \ldots, k\}$. This makes v being no longer a leaf node and therefore the production p is no longer part of the represented grammar of the resulting forest F'. The new children of v are leaves now and are part of the represented grammar such that $\{A_i \to \beta_1\gamma \mid \ldots \mid \beta_k\gamma\} \subseteq P(T_{A_i})$. Therefore the set P' of productions for G' as well as $G(F')$ is

$$P' \;=\; \{A_i \to \beta_1\gamma \mid \ldots \mid \beta_k\gamma\} \;\cup\; P \setminus \{p\}$$

where P is the set of productions before the replacement. So it is $G' = G(F')$. □

The removal of left recursive productions proves to be more complicated than the replacement by left derivations. Transfering the operation of Definition 2.5(2) to the concept of production trees requires the definition of production trees (Definition 3.1) being extended.

The reason is that the nodes for the right recursive copies of the form $A_i \to \gamma B_i$ cannot be inserted as siblings of the original $A_i \to \gamma$ node. This is because γB_i will not be a left derivation of its parent since the additional symbol B_i appears at the rightmost position.

Definition 3.7 (Extended Production Tree). *Let $T_A = (V, E, lab)$ be a production tree. An* extended production tree *$T'_A := (V, E, lab, lab_R)$ contains an additional labeling function $lab_R : V \to \mathcal{P}((\Sigma \cup N)^*)$ where $\mathcal{P}$ denotes the powerset. So lab_R maps each node in V to a set of labels.*

Let $v \in V$ and $lab_R(v) = \{\alpha_1, \dots, \alpha_k\}$. Write $A \to lab_R(v)$ for $A \to \alpha_1 | \dots | \alpha_k$.

The set of represented productions of an extened tree T'_A with leaf nodes $v_1, \dots, v_n$ is defined as

$$P(T'_A) := \bigcup_{i=1}^{n} (A \to lab_R(v_i))$$

This allows the insertion of productions without adding nodes and without violating the left derivation constraint of the production trees. A forest containing extended production trees behaves exactly like a forest of common production trees regarding the represented grammar.

Definition 3.8 (Removing a Left Recursive Node). *Let F be a forest of production trees representing the grammar $G(F) = (N, \Sigma, P, S)$ and let F contain the production tree T_{A_i} with $v_{rec} \in T_{A_i}$ being a leaf node with $lab(v_{rec}) = A_i\gamma$ where $\gamma \in (\Sigma \cup N)^+$. The node v_{rec}, which is denoted as* left recursive node*, is removed as follows*

1. *If $B_i \notin N$ extend N by B_i making B_i the greatest symbol in the order of nonterminals and create a new extended derivation tree $T'_{B_i} = (V, E, lab, lab_R)$ containing a root node only such that $V = \{r_{B_i}\}$, $E = \emptyset$, $lab = \{r_{B_i} \mapsto B_i\}$ and $lab_R = \emptyset$.*

2. *Add a new node b to T_{B_i} such that $(r_{B_i}, b) \in E$ and $lab(b) = \gamma$. The additional labeling function lab_R is defined for each node $v \in V$ such that*

$$lab_R(v) := \begin{cases} \{lab(v), lab(v) \cdot B_i\} & \text{if } v \text{ is a non-recursive node different to root} \\ \{lab(v)\} & \text{otherwise} \end{cases}$$

3. *In T_{A_i} mark v_{rec} as blind leaf and if T_{A_i} is not an extended tree yet, make it extended by supplying it with an additional labeling function lab'_R which is defined for each node $v \in T_{A_i}$ analogously to lab_R.*

The concept of extended derivation trees avoids the constraints of the original production trees of Definition 3.1 by increasing the amount of labels without adding any new nodes. The additional productions appear only when the represented productions $P(T_{A_i})$ of a tree T_{A_i} are requested. To emphasize that these right recursive copies are not actual nodes we say that they are *generated on query*.

The right recursive counterparts are required in two situations only. Either the productions are used to replace some production by its left derivations or the production is already part of the grammar in GNF i.e. part of the final representation.

If the algorithm encounters a recursive node in T_{A_i} the first time it extends the tree as in Definition 3.8 creating a new extened production T_{B_i} whose root node has only one child node representing solely the terminal production since the right recursive copy is created *on query*. In all figures of this work extended trees get their root node shaded red instead of yellow in order to indicate that the tree is extended.

Example 3.3 (Using Extended Production Trees). *Let T_{A_i} be a production tree as shown in Figure 4 representing the productions $A_i \to \alpha$ and $A_i \to A_i\beta$ with $\alpha, \beta \neq \varepsilon$.*

The removal of the direct left recursive production $A_i \to A_i\beta$ results in the additional extended production tree T'_{B_i} and in T_{A_i} becoming the extended tree T'_{A_i}.

The supplementary productions that are a result of the augmented labeling are shown as dashed nodes, they are not actual nodes and only used in this example to illustrate the productions being actually represented.

Left recursive nodes like $A_i \rightarrow A_i\beta$ in the extended tree T'_{A_i} that have been processed are marked as a blind leaf and will not be considered for the represented productions according to Definition 3.2. In all figures those nodes are drawn with a red frame.

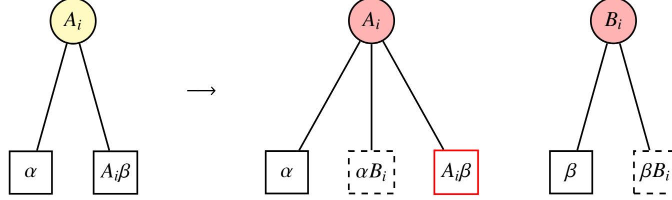

Figure 4: Removal of a direct left recursive production

Like for the replacement by left derivations it has to be shown that the removal of a direct left recursive production is an exact simulation of the operation as described in Definition 2.5(2).

Lemma 3.2. *Let $G \in CFG$ be a productive, context-free grammar and let F be a forest of production trees such that $G = G(F)$. Now let the grammar G' and the forest F' be the results of removing a direct left recursive production of G as in Definition 2.5(2) and replacing the node which represents the same production in F as in Definition 3.8.*

The grammar $G(F')$ represented by F' equals G'.

Proof. Let $G \in CFG$ and F a forest of production trees such that $G(F) = G$. Further let $A_i \in N$ be nonterminal of the grammars and let $p = (A_i \rightarrow A_i\gamma)$ be a production in G with $\gamma \in (\Sigma \cup N)^+$. The productions of G and $G(F)$ are denoted by P.

When p is removed in G according to Definition 2.5(2) a new nonterminal B_i and the productions $B_i \rightarrow \gamma \mid \gamma B_i$ are added to G. Since A_i has to be a terminating symbol there have to be A_i productions beside $p = (A_i \rightarrow A_i\gamma)$ that are not direct left recursive. Let those productions be $A_i \rightarrow \beta_1 \mid \ldots \mid \beta_m$. For those productions Definition 2.5(2) states that there has to be an additional right recursive copy. So the productions $P_B(A_i) := \{A_i \rightarrow \beta_1 B_i \mid \ldots \mid \beta_m B_i\}$ are added to G. The resulting grammar is denoted by G' and its productions by P'. If G had already another direct left recursive production removed such that $P_B(A_i) \subseteq P$ including them once more has no effect since P and P' are sets.

The forest F is modified according to Definition 3.8 resulting in the forest F'. Because of $G = G(F)$ there has to be a leaf node v in T_{A_i} representing p. This node v is made a blind leaf so p is not a production of $G(F')$. The extended tree T_{B_i} is created if not already part of F and a child node b labeled $lab(b) = \gamma$ is added to T_{B_i}. Since b is a leaf node and T_{B_i} is an extended tree, the extended labeling function will map b to $\{\gamma, \gamma B_i\}$, so it is $\{B_i \rightarrow \gamma \mid \gamma B_i\} \subseteq P(T_{B_i}) \subseteq P'$.

Because of $G = G(F)$ the tree $T_{A_i} \in F$ has to contain the same non-recursive productions as G such that $\{A_i \rightarrow \beta_1 \mid \ldots \mid \beta_m\} \subseteq P(T_{A_i})$. When making T_{A_i} an extended tree Definition 3.8 states that the additional labeling function is defined such that only the nodes representing those productions receive an additional right recursive copy. Therefore it is $\{A_i \rightarrow \beta_1 B_i \mid \ldots \mid \beta_m B_i\} \subseteq P(T_{A_i})$.

So the set of productions for G' and $G(F')$ reads

$$P' = \{A_i \to \beta_1 B_i \mid \dots \mid \beta_m B_i\} \cup \{B_i \to \gamma \mid \gamma B_i\} \cup P \setminus \{p\}$$

and it follows that $G' = G(F')$. □

Now that the two language preserving operations have been defined for forests of production trees the original algorithm (whose steps were described on page 7) can be transfered to this data structure. As in the original algorithm the production trees are modified in two steps each consisting of two parts In the first step all productions are ordered and direct left recursions are removed. In the second step the Greibach property is established first for all productions of nonterminals of the original grammar then for all productions of newly introduced nonterminals.

Those steps will be administered in three *passes*. The first pass carries out the ordering of the productions as well as the removal of direct left recursions and is refered to as the *ordering pass*. The second pass establishes the GNF for productions with an original nonterminal symbol on the left hand side and is refered to as the *Greibach pass*. The third and last pass transfers the new nonterminals, that might have been introduced while removing left recursions, to GNF. This pass is refered to as the *final pass*.

Within these passes the production trees are only modified by the language preserving operations from Definition 3.6 and Definition 3.8.

For the sake of simplicity we assume that only the "simple" GNF and not the strong GNF is required. This simplification can be allowed because establishing the strong GNF is an easy task since terminal symbols $a \in \Sigma$ that would violate the strong GNF property are replaced by a new nonterminal A_a for which only a single production $A_a \to a$ exists. Those A_a pose no problem in subsequent adding operations as the productions $A_a \to a$ are in GNF already and no other A_a productions are generated at any point.

Each of the passes described above usually has to process only a certain subset of all productions. This suggests the use of a worklist for each pass on which the productions have to appear in the right order to simulate the basic algorithm. These worklists will be refered to as *ordering list*, *Greibach list* and *final list*.

Definition 3.9 (Worklist)**.** *Let F be a forest of production trees. A* worklist *operating on F is defined as a sequence of nodes that belong to arbitrary trees in F*

$$W := v_1, v_2, \dots, v_k$$

where $v_i \neq v_j$ for $i \neq j$ with $i, j \in [1, \dots, k]$ and no v_i is a root node.

Worklists are always processed from left to right where nodes that have been processed are removed from the list. Processing a worklist *means that nodes are processed until the worklist is empty. An empty worklist is denoted by ε.*

Processing a node in the context of production trees means that one of the language preserving operations is applied to this node. Note that the second operation, the removal of left recursions, is only applied within the ordering pass when a direct left recursive node is encountered. Otherwise the algorithm always applies replacement by left derivations. Where the child nodes of a node v (with $lab(v) = A_j\gamma$ in $T^G_{A_i}$) that are generated in the process are determined by $P(T^{G'}_{A_j})$.

The primary objective of the incremental approach is to know at which nodes in the forest the productions represented by a certain tree $T^G_{A_i}$ were applied. This information is held by *usage pointers*.

Those usage pointers always point from a root node labeled with A_i to an inner node of some tree with root node A_j or B_j where $j \neq i$. Pointers from A_i to A_j with $i < j$ are refered to as *forward pointers* and are used in the ordering pass only, while pointers with $i > j$ are called *backward pointers* and are used in the Greibach pass only. A third kind of usage pointers are those that point from an A_i to a an inner node of a tree with root node labeled B_j. Those pointers are refered to as *final pointers* and are used in the final pass only. Usage pointers are inserted if the production corresponding to the destination node is not yet in GNF.

Definition 3.10 (Usage Pointers)**.** *Let $F = \{T^{G_1}_{A_1}, \dots, T^{G_n}_{A_n}\}$ be a forest of production trees with $R := \{r_1, \dots, r_n\}$ being the set of all root nodes of the trees in F. For a production tree $T^{G_i}_{A_i}$ let $L_i := \{v_{i_1}, \dots, v_{i_{k_i}}\}$ denote the set of all leave nodes in the tree.*

A usage pointer *in the forest F is an element of the set*

$$U^F := \Big(R \times \bigcup_{i=1}^{n} L_i\Big) \setminus \bigcup_{i=1}^{n} \Big(\{r_i\} \times L_i\Big)$$

When adding new productions these usage pointers are used to quickly find the (probably intermediate) productions within the algorithm (represented by nodes in the production tree) at which the new productions have to be taken into account. This means that whenever a node is created within a production tree a usage pointer might have to be added.

If the new production is of the form $A_i \to A_j\gamma$ (with $i \neq j$) a usage pointer has to be inserted from the root node of the A_j tree to the node representing the new production. This is because this production will at some point be replaced by its left derivations since it is not in GNF. When this happens all A_j productions will be used for that. So if there are any changes to the A_j productions the algorithm will follow the usage pointer from A_j to the node representing $A_i \to A_j\gamma$ and will have to modify its children according to the changes made to the A_j productions. When the new production is of the form $A_i \to a\gamma$ with $a \in \Sigma$ no usage pointer has to be added because this production is already in GNF and will undergo no further modification by the algorithm. Also direct left recursive nodes of the form $A_i \to A_i\gamma$ need no usage pointers because they are not replaced by their left derivations. Instead those nodes are deleted by using the language preserving operation for removing direct left recursion. When doing so for a node $A_i \to A_i\gamma$ a new extended tree $\tilde{T}^{G_i}_{B_i}$ is created with the node b with $lab(b) = \gamma$. If γ starts with a nonterminal A_j so that $\gamma = A_j\gamma'$ the usage pointer (r_j, b) from the root node of $T^{G_j}_{A_j}$ to the node b would have to be inserted.

As mentioned above we distinguish between three different kinds of usage pointers. We can tell by the source and destination tree what kind of pointer we are dealing with.

Definition 3.11 (Sets of Usage Pointers)**.** *Let G be a context-free grammar with non-terminals $\{A_1, \dots, A_n\}$ so that the set of possible right recursive nonterminal symbols is $\{B_1, \dots, B_n\}$. Let $B \subseteq \{B_1, \dots, B_n\}$ and $F = \{T^{G_1}_{A_1}, \dots, T^{G_n}_{A_n}\} \cup \{T^{G_{B'}}_{B'} \mid B' \in B\}$ be a forest of production trees.*

Based on the set of possible usage pointers U^F as in Definition 3.10 we define three different sets of usage pointers.

- *The set of* forward pointers $U^F_{fwd} \subseteq U^F$ *contains usage pointers (r_i, v) with v in $T^{G_j}_{A_j}$ so that $i < j$.*

- *The set of* backward pointers $U^F_{bck} \subseteq U^F$ *contains usage pointers* (r_i, v) *with* v *in* $T^{G_j}_{A_j}$ *so that* $i > j$.
- *The set of* final pointers $U^F_{fin} \subseteq U^F$ *contains usage pointers* (r_i, v) *with* v *in* $T^{G_j}_{B_j}$ *so that* $i \neq j$.

All those sets are subsets of U^F so that $(U^F_{fwd} \cup U^F_{bck} \cup U^F_{fin}) \subseteq U^F$. Since the label of a node starts with at most one nonterminal there can be at most one usage pointer pointing at this node. So the different sets of pointers are also disjoint so that $(U^F_{fwd} \cap U^F_{bck}) = (U^F_{fwd} \cap U^F_{fin}) = (U^F_{bck} \cap U^F_{fin}) = \emptyset$.

3.2 The Incremental Algorithm

We will now construct an incremental algorithm that holds the information of how the GNF is obtained by the basic algorithm for a certain grammar in a way that the productions of that grammar are added ony by one in a sequence according to Lemma 2.1. This incremental algorithm also allows extending this grammar by a production and reestablishing the GNF while also extending the information of how this was accomplished.

Using the structures and definitions of the previous section by name the concept of forests of production trees, the two language preserving operations on those forests, the worklists and the usage pointers such an algorithm will be defined by splitting the algorithm in several procedures.

The incremental algorithm has two basic parts. In the first part a new production is added as in Definition 3.5. If necessary, the resulting new node is put on the appropriate worklist and a usage pointers is added. In the second part the worklists are processed resulting in a forest that represents a grammar in GNF.

At all times the incremental algorithm maintains a forest of production trees, three worklists (ordering-, Greibach- and final list) and three sets of usage pointers (forward-, backward- and final pointers).

Definition 3.12 (Configuration)**.** *A* configuration of the incremental algorithm *is defined by a tuple of the form*

$$(F, W, U) \in \mathcal{S}$$

where F *is a forest of production trees,* $W = (W_O, W_G, W_F)$ *a tuple of worklists representing the ordering-, Greibach- and final list,* $U = (U_{fwd}, U_{bck}, U_{fin})$ *a tuple of sets of usage pointers defined over* F *such that* $U_{fwd} \subseteq U^F_{fwd}$, $U_{bck} \subseteq U^F_{bck}$ *and* $U_{fin} \subseteq U^F_{fin}$ *and* $\mathcal{S}$ *denotes the space of all possible configurations.*

The start configuration *or* empty configuration *is defined by the tuple*

$$(\emptyset, W_\emptyset, U_\emptyset) \in \mathcal{S}$$

where $W_\emptyset := (\varepsilon, \varepsilon, \varepsilon)$ *and* $U_\emptyset := (\emptyset, \emptyset, \emptyset)$ *while a* terminating configuration *is any configuration in* $\mathcal{S}$ *with empty worklists*

$$(F, W_\emptyset, U) \in \mathcal{F}$$

where $\mathcal{F} \subseteq \mathcal{S}$ *denotes the set of all* terminating configurations.

Starting with the empty configuration containing an empty forest $F_\emptyset$ such that $G(F_\emptyset) = (N, \Sigma, \emptyset, S)$, the algorithm allows to add productions p that keep the grammar productive. At first the concept will be shown for adding a production to an empty forest. Later further concepts are explained when the algorithm adds new productions to an existing, non-empty forest, i.e. operates incrementally.

Algorithm 3.1 Procedure "extendGrammar" adds a production to an existing forest of production trees F and modifies F so that the result is in GNF.

Require: a new production $p = (A \to \gamma)$, a configuration $C = (F, W_\emptyset, U)$ such that $C \in \mathcal{F}$ a terminating configuration

1: C_p = insertProduction(C, p)
2: C' = processWorklist(C_p, orderingList)
3: C'' = processWorklist(C', GreibachList)
4: C_T =processWorklist(C'', finalList)

Ensure: a terminating configuration $C_T = (F'_p, W_\emptyset, U'_p)$ such that F'_p represents G'_p in GNF which is obtained by the basic algorithm from G plus p

---

The main procedure that extends a given forest by adding a new production is shown in Algorithm 3.1. The input configuration holds the information how the basic algorithm constructed the GNF grammar G' from the grammar G. What this procedure does is adding the procedure p to G (yielding G_p) by calling the procedure "insertProduction" in line 1. Then the basic algorithm on G is retraced using the forest and the usage pointers. During this retracing the forest is extended such that the production p and productions resulting from it are inserted as if the basic algorithm was executed on G_p. Thereby the GNF is reestablished and the forest and the usage pointers are extended such that they hold the information of how the basic algorithm established the GNF for the grammar G_p.

After inserting the new production to the forest the GNF will be reestablished by processing all worklists. Remember that the ordering pass (as invoked in line 2 of Algorithm 3.1) simulates the ordering as well as the removal of left recursions of the basic algorithm while establishing the Greibach property for productions of original A-symbols is simulated in the Greibach pass (line 3) and for new introduced B-symbols in the final pass (line 4).

The part of the algorithm that adds a new production to the forest is made the separate procedure "insertProduction" shown in Algorithm 3.2. This insertion is an exact implementation of Definition 3.5 that introduces a new production by creating a child of the corresponding root node.

Note that inserting new nonterminal symbols as least symbol in the total order of the nonterminals is a heuristic that attempts to minimize the cost of establishing the GNF. This will be examined more closely in Section 4 (Optimization). As statet earlier we will not examine the effects of different total orders of the nonterminals any closer, but will choose this approach in an attempt to make the introduction of new nonterminals less costly.

Every time a new node is inserted the algorithm checks if new pointers have to be added. If a new node v in a tree T_{A_i} is labeled with $A_j\gamma$ with $\gamma \in (\Sigma \cup N)^*$ and $j < i$ a forward pointer from the root node of T_{A_j} to v is inserted and a backward pointer is inserted if $j > i$. For left recursive productions with $j = i$ no pointer is required because the algorithm does not replace those productions by their left derivations.

Algorithm 3.2 Procedure “insertProduction” adds a production to an existing forest.

Require: a configuration $C = (F, W_\emptyset, U)$, a new production $p = (A \to \gamma)$ where $C \in \mathcal{F}$ is a terminating configuration and $G := G(F) = (N, \Sigma, P, S)$

1: **if** $A \notin N$ **then**
2: add A to N as least element of N
3: **end if**
4: **if** no tree $T_A^{G_A}$ exists in F **then**
5: create derivation tree T_A in F containing root node only
6: **end if**
7: create new node v with $lab(v) = \gamma$
8: add v as child of the root node of T_A
9: C_p = checkPointerAndWorklist(C, v)

Ensure: a configuration $C_p = (F_p, W_p, U_p)$ with updated worklists and usage pointers and F_p representing the grammar G plus the new production p

As implemented in Algorithm 3.3 nodes are put on a worklist if a usage pointers was added for them because they need further processing then. Before actually processing the worklists the method of how nodes are placed on them has to be examined more closely.

Algorithm 3.3 Procedure “checkPointerAndWorklist”.

Require: a configuration $C = (F, W, U)$, a node $v \in T_R$ such that $T_R \in F$, $lab(v) = X\gamma$ where $X \in (\Sigma \cup N)$ and $\gamma \in (\Sigma \cup N)^*$

1: **if** $R \to X\gamma$ is direct left recursive **then**
2: add v to the ordering list $W_O \in W$ as last R-element
3: **else if** $R \to X\gamma$ is not ordered **then**
4: add a forward pointer to $U_{fwd} \in U$ pointing from root of T_X to v
5: add v to the ordering list $W_O \in W$ as first R-element
6: **else if** $R \to X\gamma$ is not in GNF **then**
7: add a backward pointer to $U_{bck} \in U$ pointing from root of T_X to v
8: add v to the Greibach list $W_G \in W$ as last R-element
9: **end if**

Ensure: a configuration $C' = (F, W', U')$ with updated worklists and usage pointers depending on v but with the same forest F

In order to explain how the nodes are added to the respective worklist in line 2, line 5 and line 8 of Algorithm 3.3 the order in which the nodes need to be processed to simulate the basic algorithm is of concern. In the basic algorithm all productions of A_i with $i < j$ needed to be ordered first and then got their direct left recursions removed before any A_j production with $j > i$ was allowed being ordered. To ensure the productions are processed in an ascending order for the ordering pass and in a descending order for the Greibach pass regarding the A_i it is necessary to insert the nodes at the proper position within a worklist.

Since the ordering worklist is used to order the productions as well as to remove

direct left recursions the algorithm has to distinguish between those different types of A_i productions when adding them to the ordering list. For the Greibach list no further distinction beside the ordering of the list by left hand side symbols is required. Algorithm 3.3 implements this behaviour.

The way how nodes are put on a worklist will be of particular interest when allowing to add multiple productions at once. In Section 4 (Optimization) we will get back to this in detail.

Now that the new production has been inserted into the forest, the GNF is reestablished by processing the worklists starting with the ordering list. A node is processed by applying either the replacement by left derivations as in Definition 3.6 or the replacement of direct left recursive nodes as in Definition 3.8. The procedures for those operations are shown in Algorithm 3.4 and Algorithm 3.5.

Algorithm 3.4 Procedure "leftDerivation" to replace a node by its left derivations.

Require: a configuration $C = (F, W, U)$, a node $v \in T_R$
where $T_R, T_X \in F,\ lab(v) = X\gamma,\ X \in N$ and $\gamma \in (\Sigma \cup N)^*$

1: **for all** productions $p \in P(T_X)$ {let $p = X \to \beta$} **do**
2: create node v' with $lab(v') = \beta\gamma$
3: add v' to T_R as child of v yielding configuration C'
4: C = checkPointerAndWorklist(C', v')
5: **end for**

Ensure: a configuration (F', W', U') where F' represents the grammar that results in replacing $R \to X\gamma$ in $G(F)$ by its left derivations

Note that in line 4 of Algorithm 3.4 a new node created by left derivation is checked for usage pointer and further processing right after it has been created, so the information in which pass that node was processed is preserved.

Algorithm 3.5 on the other hand creates child nodes for extended B_i-trees only. So these nodes, if not in GNF already, receive final pointers only and are processed solely in the final pass. For that reason the check for GNF, worklist and pointer when removing a direct left recursive node is done in lines 12-15 of Algorithm 3.5 directly and is not part of the "checkPointerAndWorklist" procedure.

There is a special case Algorithm 3.5 takes care of. Whenever there is a direct left recursive production with $\gamma = \varepsilon$ (i.e. the production is of the form $A_i \to A_i$) the algorithm just ignores it. As mentioned earlier such productions are either not valid or do not change the language of the grammar. Like every direct left recursive node the $A_i \to A_i$ nodes cannot be deleted nor can the nodes causing this production as Example 3.4 shows.

Example 3.4 ($A_i \to A_i$ productions)**.** *Consider the forest in Figure 5 where the node $A_2 \to A_1$ is added resulting in the useless production $A_2 \to A_2$ which is ignored. The original production $A_2 \to A_1$ has to be kept since new A_1 productions will create new productions as a result of the left derivation of the node $A_2 \to A_1$ that will change the language of the grammar if left out.*

Note that the tree T_{A_2} is not made an extended production tree since creating T_{B_2} with $P(T_{B_2}) = \varepsilon \mid B_2$ and extending T_{A_2} such that $P(T_{A_2}) = a \mid aB_2$ will not change the language but will instead create a forbidden ε-production and also the production $B_2 \to B_2$ that cannot be transfered to GNF by the means of this algorithm.

Algorithm 3.5 Procedure “removeRecursion” to remove a direct left recursive node.

Require: a configuration $C = (F, W, U)$, a leaf node $v_r \in T_{A_i}$
where $T_{A_i} \in F$, $lab(v_r) = A_i\gamma$, $\gamma \in (\Sigma \cup N)^*$

1: make v_r a blind leaf

2: **if** $\gamma = \varepsilon$ **then**
3: do nothing and end procedure
4: **end if**

5: **if** tree T_{B_i} does not exist **then**
6: add symbol B_i to N as greatest symbol
7: create extended production tree T_{B_i} containing root node only
8: make T_{A_i} an extended production tree
9: **end if**

10: create node $v \in T^G_{B_i}$ with $lab(v) = \gamma$
11: add v as child of the root node of T_{B_i}

12: **if** $B_i \to \gamma$ is not in GNF **then**
13: add v to final list $W_F \in W$
{let $X \in N$ be the leftmost symbols of γ}
14: add a final pointer to $U_{fin} \in U$ pointing from root of T_X to v
15: **end if**

Ensure: a configuration (F', W', U') where F' represents the grammar that results in removing the direct left recursion $A_i \to A_i\gamma$ in $G(F)$

This example also shows why direct left recursive nodes cannot just be deleted. Doing so with the node $A_2 \to A_2$ would result in $A_2 \to A_1$ becoming a leaf node again making that production part of the represented grammar.

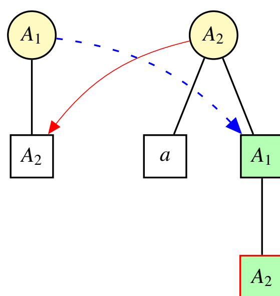

Figure 5: Nodes causing $A_i \to A_i$ productions have to be kept.

The actual processing of a worklist can be made a procedure that is shared by all of the different passes since the algorithm only needs to distinguish between nodes that are direct left recursive and those that are not. So for the processing calls in lines 2-4 of Algorithm 3.1 the procedure Algorithm 3.6 can be used passing the current configuration and the information which worklist is supposed to be processed.

Algorithm 3.6 Procedure “processWorklist”.

Require: a configuration $C = (F, W, U)$, the identification of the worklist where $\Gamma \in W$ denotes the worklist and v the first node in the list

```
1: while list Γ not empty do
2:    remove v from Γ yielding C′
3:    if node v is direct left recursive then
4:       C″ = removeRecursion( C′, v )
5:    else
6:       C″ = leftDerivation( C′, v )
7:    end if
8: end while
```

Ensure: a configuration $C'' = (F'', W'', U'')$ where W'' contains the now empty Γ

Operating Incrementally

When adding a production to a non-empty forest the corresponding node is added in the same way as for an empty forest. The goal beside establishing the Greibach property for the new production is to find the points at which the new production would have been applied if it already had been part of the original grammar. At this time the algorithm can make use of the production tree structure and the usage pointers.

When adding a production p to a tree T_{A_i} the algorithm can tell by the usage pointers at which nodes the productions $P(T_{A_i})$ were used for replacement by left derivation by following all usage pointers originating in T_{A_i}. The nodes at the destinations of these pointers need to apply this new production p as well. For this the algorithm has to distinguish between new and old productions to tell which productions are new. This could be accomplished by marking new nodes for the duration of the insertion process and remove those marks once the GNF is reestablished. But unfortunately this does not solve the problem. If an A_i production is added that turns out to be left recursive the right recursive copies of A_i productions that are not left recursive could not be found by marking because those productions are generated on query only.

A solution to this problem is to memorize the sets of represented productions $P(T_{A_i})$ for each nonterminal before the insertion, denoted by $P_{old}(T_{A_i})$. Now the new productions can be found easily.

$$P_{new}(T_{A_i}) \;:=\; P(T_{A_i}) \setminus P_{old}(T_{A_i})$$

The marking however helps to easily identify new nodes and decide which productions need to be applied. In all following figures nodes that are marked as new are shaded green.

When adding a new production, the incremental algorithm has to act like the original algorithm would have at any point. Since a forest in a start configuration has to represent a grammar in GNF its trees represent Greibach productions only. This turns out being a problem in the ordering pass when simulating the basic algorithm. The reason is that if the new production would already had been part of the original grammar the trees were usually not entirely representing GNF productions at that stage, because the Greibach pass would not have been executed yet.

Therefore when ordering a new node the algorithm is neither allowed to use the GNF leaves nor the productions of the original grammar. Instead the ordered inter-

mediate nodes have to be used, because those nodes were the leaf nodes back in the ordering pass. These could however be leaf nodes or original rules if they were ordered or in GNF already. These nodes are refered to as *first ordered nodes*.

Definition 3.13 (First Ordered Productions)**.** *Let $T_{A_i}^{G_i}$ be a (possibly extended) production tree. The set of* first ordered productions *for that tree $P_{fon}(T_{A_i}^{G_i})$ is defined as*

$$P_{fon}(T_{A_i}^{G_i}) \;:=\; P(\overline{T_{A_i}^{G_i}})$$

where $\overline{T_{A_i}^{G_i}}$ is the production tree that is derived from $T_{A_i}^{G_i}$ by traversing $T_{A_i}^{G_i}$ and deleting all children for all ordered nodes that are found.

Note that within a forest of production trees that was generated by this algorithm an ordered node possibly has children but no grandchilden since the Greibach property is established within a single step once a node is ordered. Therefore in Definition 3.13 it is sufficient to delete the children of the ordered nodes instead of entire subtrees.

Example 3.5 (First Ordered Nodes)**.** *Consider the production trees T_{A_1} and T_{A_2} as shown in Figure 6. When adding the new, not ordered production $A_2 \rightarrow A_1$ the incremental algorithm has to use the nodes A_2b and A_3c in the T_{A_1} tree (shaded blue) for replacement by left derivations, because those productions would have been available in the original algorithm at that stage and not the final GNF productions.*

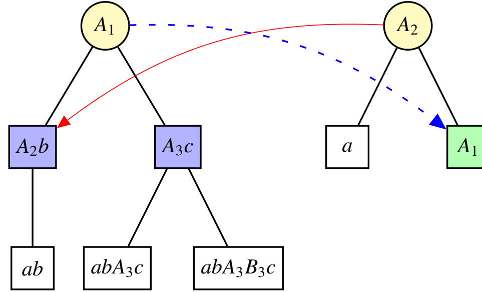

Figure 6: Ordering new productions using first ordered nodes only

Now the usage pointers need to be followed to tell where the new productions would have been applied if they had already been part of the original grammar. Since following a usage pointer from a tree T_{A_i} to a tree T_{A_j} and applying at least the new productions also yields new productions in T_{A_j} the usage pointers originating in T_{A_j} have to be followed as well. Such a chain is refered to as a *path of usage pointers*. The incremental algorithm needs to follow all those paths and add the destinations to the proper worklist. This has to be done once for each type of usage pointer before the corresponding worklist is processed. Note that all these destinations need processing, otherwise no usage pointer would have been inserted in the first place. However, new nodes that have just been added were already put on the proper worklist by “checkPointerAndWorklist”. So when following usage pointers only old nodes have to be put on a worklist if necessary.

The incremental algorithm as shown in Algorithm 3.1 has to implement this concept of following usage pointers. A procedure to gather all nodes that are located on a pointer path originating in a given production tree T_{A_i} is shown in Algorithm 3.7.

Algorithm 3.7 Procedure "followPointers" for a certain production tree T_{A_i}.

Require: a configuration $C = (F, W, U)$, a production tree $T_{A_i} \in F$, the *pointertype* where *pointertype* $\in \{forward, backward, final\}$.

```
1:  if T_{A_i} is marked as "followed from" then
2:     do nothing
3:  else
4:     mark T_{A_i} as "followed from"
5:     for all usage pointers (r_{A_i}, v) of pointertype do
6:        if v is not a new node then
7:           put v on the list W_{pointertype} ∈ W list yielding C'
8:        else
9:           C' := C
10:       end if
          {let v ∈ T_{A_j}}
11:       C_F = followPointers( C', T_{A_j}, pointertype )
12:    end for
13: end if
```

Ensure: a configuration $C_F = (F, W', U)$ with unchanged forest and pointer lists but with a modified worklist for the *pointertype* holding all destination nodes on all pathes originating in T_{A_i}

We let the algorithm insert new nodes for the ordering worklist ordered by the left hand side of their represented production and distinguish between nodes that are direct left recursive and those that are not as in Algorithm 3.3 ("checkPointerAndWorklist"). Inserting nodes ordered by their left hand side to the ordering list allows Algorithm 3.7 to follow all forward pointer paths in an arbitrary order. As we will see later in Section 4.3 (Managing Worklists) this will prove being no disadvantage.

Following the forward pointers can result in multiple items on the ordering list allowing to illustrate Algorithm 3.3 more closely as Example 3.6 does.

Example 3.6 (Inserting Nodes into Worklists)**.** *Consider the ordering list shown in Figure 7 for an arbitrary a grammar $G \in CFG$ that will not be defined in detail. Suppose a new node representing the production $A_4 \rightarrow A_3$ has to be added to the ordering list.*

Since $A_4 \rightarrow A_3$ is not direct left recursive but also not ordered Algorithm 3.3 states (in line 5) that the node has to be added to the ordering list as first A_4 element. That would be at the position right after the first element $A_2 \rightarrow A_1$.

Now suppose the new production $A_4 \rightarrow A_4 b$ has to be inserted which is direct left recursive so according to Algorithm 3.3 (line 2) the node has to be inserted as last A_4 element which would be the position right before the last element $A_5 \rightarrow A_2$.

$A_2 \rightarrow A_1$	$A_4 \rightarrow A_2$	$A_4 \rightarrow A_1$	$A_4 \rightarrow A_4 a$	$A_5 \rightarrow A_2$

Figure 7: An ordering worklist for a grammar G

The worklists can now be processed as they were for an empty forest. When pro-

cessing a node in this incremental context we need to keep in mind that each *new* node, including those that are created when processing worklists during the insertion process, needs to use all available productions. Solely old nodes that were already part of the forest before the insertion process need to apply the new productions P_{new} only. When processing the ordering list the algorithm has to use the first ordered nodes as seen earlier. Now with the knowledge how usage pointers have to be followed this can be illustrated by an example.

Example 3.7 (Using only new productions at old nodes)**.** *Consider the forest of production trees as shown in Figure 8. When adding the new production* $A_3 \rightarrow A_2$ *the corresponding node is ordered by using the first ordered node* $A_2 \rightarrow A_4$*. In the Greibach pass the new node* $A_3 \rightarrow A_4$ *and the backward destination* $A_1 \rightarrow A_3$ *need to be processed. The new node* $A_3 \rightarrow A_4$ *has to apply all available* A_4 *productions while the old node at the backward destination* $A_1 \rightarrow A_3$ *has to apply only the new* A_3 *productions.*

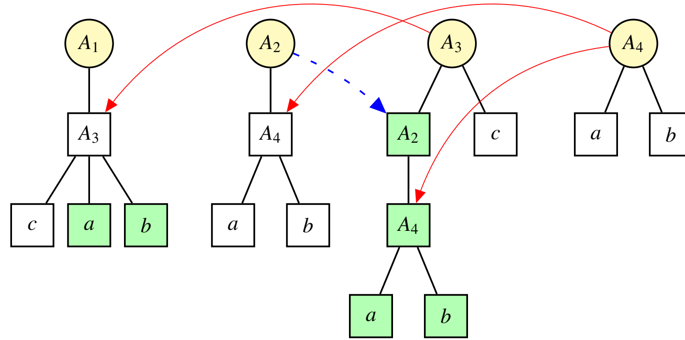

Figure 8: At old nodes only new productions are applied

The main procedure for adding new productions as in Algorithm 3.1 has to be modified to implement the following of usage pointers. The new procedure is shown in Algorithm 3.8.

The different usage pointers have to be followed at different stages of the algorithm because the ordering may result in new nodes in additional trees. So there are more trees to follow backward pointers from which again results in more trees containing new nodes.

As seen in Example 3.7 the policy regarding new nodes for the ordering pass applies for the Greibach pass as well. Old nodes at pointer destinations need to apply new productions only while new nodes always need to apply all available productions.

To implement this behaviour the procedure “leftDerivation” has to be extended as shown in Algorithm 3.9 so it chooses the correct productions based on the current pass and whether the current node is new or old.

The incremental algorithm ensures that all productions that are modified by the basic algorithm are either destinations of usage pointers if they are replaced by their left derivations or are blind leaves if they are direct left recursive productions that are removed.

Algorithm 3.8 Procedure "extendGrammar" that follows usage pointers.

Require: a configuration $C = (F, W_\emptyset, U)$, a new production $p = (A \to \gamma)$ such that $T_A \in F$ and $C \in \mathcal{F}$ a terminating configuration

1: C_p = insertProduction(C, p)

2: C_F = followPointers(C_p, T, forward)
3: C_O = processWorklist(C_F, orderingList)

4: **for all** production trees T containing new nodes **do**
5: C_F = followPointers(C_O, T, backward)
6: **end for**
7: C_G = processWorklist(C_F, GreibachList)

8: **for all** production trees T containing new nodes **do**
9: C_F = followPointers(C_G, T, final)
10: **end for**
11: C_T = processWorklist(C_F, finalList)

Ensure: a terminating configuration $C_T = (F'_p, W_\emptyset, U'_p)$ such that F'_p represents G'_p in GNF which is obtained by the basic algorithm from G plus p

Algorithm 3.9 Modified procedure "leftDerivation" distinguishing between new and old nodes and between different passes.

Require: a configuration $C = (F, W, U)$, a node $v \in T_R$ where $T_R, T_X \in F$, $lab(v) = X\gamma$, $X \in N$ and $\gamma \in (\Sigma \cup N)^*$

1: **if** v is a new node **then**
2: **if** in ordering pass **then**
3: $P := P_{fon}(T_X)$
4: **else**
5: $P := P(T_X)$
6: **end if**
7: **else**
8: $P := P_{new}(T_X)$
9: **end if**

10: **for all** productions $p \in P$ { let $p = X \to \beta$ } **do**
11: create new node v' with $lab(v') = \beta\gamma$
12: add v' to T_R as child of v yielding configuration C'
13: C = checkPointerAndWorklist(C', v')
14: **end for**

Ensure: a configuration (F', W', U') where F' represents the grammar that results in replacing $R \to X\gamma$ in $G(F)$ by its left derivations by selecting the used productions so that the basic algorithm is simulated

Lemma 3.3. *Let F' be a forest of production trees representing $G' \in GNF$ that was obtained from $G \in CFG$ by the basic algorithm and let $P(v)$ denote the productions represented by the node v.*

a) The set $P_{fwd} = \{ P(v) \mid (r, v) \in U_{fwd} \}$ holds exactly those productions the are replaced by their left derivations in the ordering pass of the basic algorithm.

b) The set $P_{rec} = \{ P(v) \mid v \text{ is a blind leaf} \}$ represents all direct left recursive productions removed by the basic algorithm.

c) $P_{bck} = \{ P(v) \mid (r, v) \in U_{bck} \}$ holds the productions modified in the Greibach pass.

d) $P_{fin} = \{ P(v) \mid (r, v) \in U_{fin} \}$ holds the productions modified in the final pass.

Proof. Let the forest of production trees F' represent the grammar $G' \in GNF$ that is obtained from $G \in CFG$ by the basic algorithm.

To prove that P_{fwd} represents all productions modified in the ordering pass of the basic algorithm we have to show that each production that is ordered within the basic algorithm is represented by a node v in F'. This node v has to receive a forward pointer and has to be ordered by the incremental algorithm as well yielding the same productions as the corresponding production in the basic algorithm.

Only A-productions are ordered by the algorithm. There are exactly two points in the incremental algorithm which create new A-productions. The first one is at line 7 of the procedure "insertProduction" (Algorithm 3.2) and the second one is at line 11 of "leftDerivation" (Algorithm 3.9).

Original productions $A \to X\gamma$ of G with $A \in N$, $X \in (N \cup \Sigma)$, $\gamma \in (N \cup \Sigma)^*$ are added using "insertProduction" which, after creating a corresponding node v in line 7, calls "checkPointerAndWorklist" (Algorithm 3.3) in line 9. If the production is not ordered, i.e. $X = A'$ where $A' < A$, the node v is put on the ordering list and the forward pointer $(r_{A'}, v)$ is added.

When v is replaced by the incremental algorithm it yields the same productions as the basic algorithm as stated in Lemma 3.1. Since the replacement is performed by the procedure "leftDerivation" the nodes created there are analyzed by "checkPointerAnd-Worklist" which again adds a forward pointer if necessary and put the nodes on the ordering list.

□

To conclude this section we will examine an example using the productions from Example 2.1.

Example 3.8. *Suppose that the productions of the grammar of Example 2.1 are added in the following sequence*

$$(A_4 \to c, A_3 \to b, A_3 \to A_3A_4, A_2 \to a, A_1 \to A_3c, A_1 \to A_2b, A_3 \to A_1A_3)$$

Since the algorithm uses the heuristic to make unknown nonterminals the least element in the total order of nonterminals, this total order will depend on the sequence in which the productions are added. The sequence has been chosen as above for readability because it maintains the order $A_1 < A_2 < A_3 < A_4$ of the nonterminals.

Suppose the incremental algorithm already added the sequence of productions except the last entry $A_3 \to A_1A_3$. The resulting forest of production trees is shown in

Figure 9. There are two backward and one final pointer. The trees T_{A_3} and T_{B_3} are extended due to the direct left recursive production $A_3 \to A_3A_4$.

According to Lemma 2.1 the grammar resulting in the adding of the sequence above is also productive after each single insertion. So omitting the last element yields a productive grammar for which the GNF can be constructed as in Figure 9.

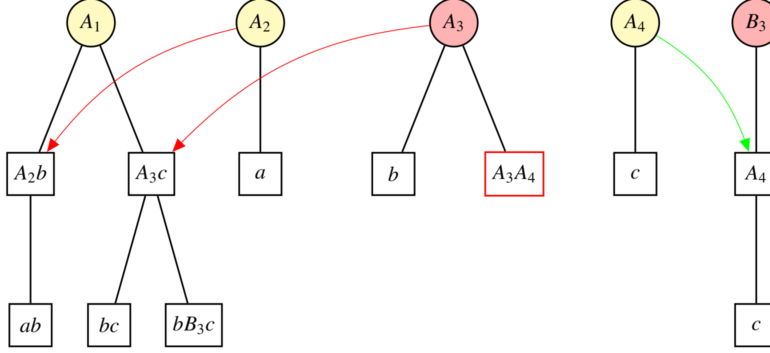

Figure 9: Forest after adding all productions from Example 2.1 except $A_3 \to A_1A_3$

We will now add the remaining production step by step using the incremental algorithm by running the main procedure "extendGrammar" (Algorithm 3.8) with the production $A_3 \to A_1A_3$ as parameter.

First "insertProduction" (Algorithm 3.2) is called with $A_3 \to A_1A_3$. The nonterminal A_3 is already known and the tree T_{A_3} does already exist, so the procedure creates a node v labeled A_1A_3 as child of the root node of T_{A_3} and calls "checkPointerAndWorklist" (Algorithm 3.3) with this node v as parameter. There, since $A_3 \to A_1A_3$ is not ordered, a forward pointer from T_{A_1} to v is added and v is put on the ordering list.

Now the main procedure follows all forward pointers before processing the ordering list, but as there are no forward pointers originating in T_{A_3} no new items for the ordering list are found.

When processing the ordering list using the procedure "processWorklist" (Algorithm 3.6) the node v is removed from the ordering list and is replaced by its left derivations using the procedure "leftDerivation" (Algorithm 3.9).

Since v is a new node that is processed in the ordering pass, Algorithm 3.9 uses the productions represented by the first ordered nodes of the corresponding tree T_{A_1} which are $P_{fon}(T_{A_1}) = \{A_1 \to A_2b,\ A_1 \to A_3c\}$. The replacement results in two new nodes v_1 and v_2 labeled $lab(v_1) = A_2bA_3$ and $lab(v_2) = A_3vA_3$ that are made child nodes of v. Both nodes are put on the ordering list by the procedure "checkPointerAndWorklist" which also adds a new forward pointer from T_{A_2} to v_1.

The ordering list is processed further, calling "leftDerivation" for node v_1 using $P_{fon}(T_{A_2}) = \{A_2 \to a\}$ resulting in a single child of v_1 labeled $lab(v_1) = abA_3$ which represents a production in GNF.

When processing node v_2 the procedure "removeRecursion" (Algorithm 3.5) is used as v_2 is a direct left recursive node. Since T_{B_3} already exists the procedure just creates a new node v_3 labeled $lab(v_3) = cA_3$ as child of the root node of T_{B_3} and ends the procedure because $B_3 \to cA_3$ is already in GNF.

The ordering list is empty now so "processWorklist" ends as well. The current forest of productions trees is shown in Figure 10.

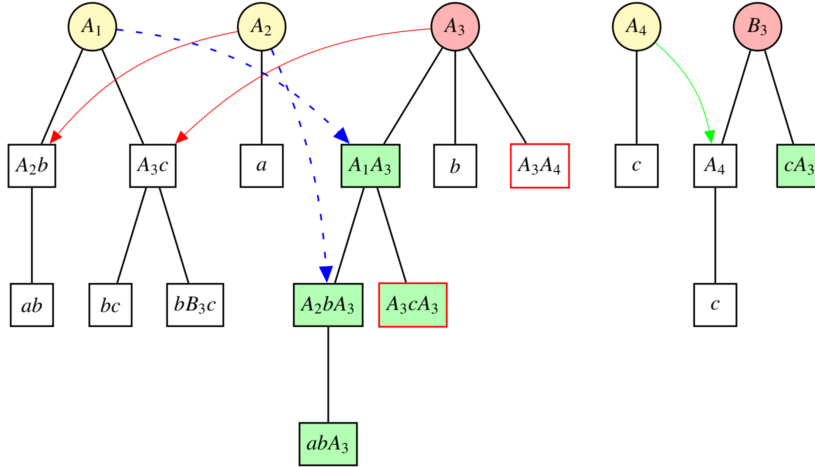

Figure 10: Forest after completing the ordering pass while adding $A_3 \to A_1A_3$

Now the main procedure follows all paths of backward pointers originating in trees containing new nodes by calling "followPointers" (Algorithm 3.7) for each of these trees.

For the tree T_{A_3} one backward pointer is found pointing to the node v' in T_{A_1} representing $A_1 \to A_3c$. So v' is put on the Greibach list and T_{A_1} is checked for backward pointers as well, but as there are none Algorithm 3.7 ends for T_{A_3}. The tree T_{B_3} has no backward pointer either so the main procedure ends the following of pointers and starts processing the Greibach list containing the node v' only.

Since v' is not direct left recursive it is processed by Algorithm 3.9. As v' is not a new node the procedure "leftDerivation" uses the new productions of T_{A_3} only. Those are $P_{new}(T_{A_3}) = \{A_3 \to abA_3,\ A_3 \to abA_3B_3\}$. So two new nodes v_3 and v_4 labeled $lab(v_3) = abA_3c$ and $lab(v_4) = abA_3B_3c$ are created as child nodes of v'. So v' has four child nodes now.

This ends the procedure "processWorklist" as the Greibach list is empty. Since there is no final pointer originating in any of the trees containing new nodes the main procedure and therefore the algorithm terminates. The resulting forest of production trees is shown in Figure 11.

The represented productions correspond to those of Example 2.1.

$$
\begin{aligned}
P(T_{A_1}) = &\quad \{A_1 \to ab \mid bc \mid bB_3c \mid abA_3c \mid abA_3B_3c\} \\
P(T_{A_2}) = &\quad \{A_2 \to a\} \\
P(T_{A_3}) = &\quad \{A_3 \to abA_3 \mid abA_3B_3 \mid b \mid bB_3\} \\
P(T_{A_4}) = &\quad \{A_4 \to c\} \\
P(T_{B_3}) = &\quad \{B_3 \to cA_3 \mid cA_3B_3 \mid c \mid cB_3\}
\end{aligned}
$$

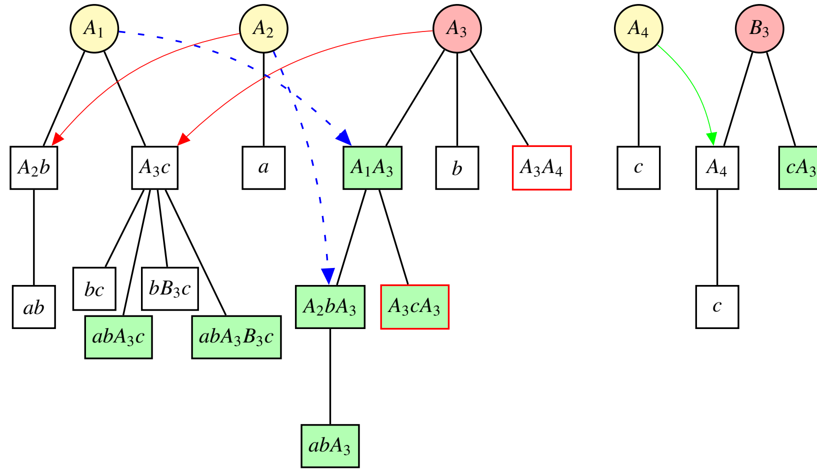

Figure 11: Forest after incrementally inserting the production $A_3 \rightarrow A_1A_3$

3.3 Correctness

The correctness of the incremental approach can be stated in the following theorem.

Theorem 1 (Correctness of Incremental Algorithm)**.** *Given a context-free, productive grammar G and a grammar G' in GNF that is obtained by the basic algorithm as shown in Algorithm 2.1.*

The incremental Algorithm 3.8 is correct in a sense that the productions of G added in a sequence following Lemma 2.1 yields a forest F' with $G(F') = G'$.

To prove Theorem 1 we have to show that the incremental algorithm obtains the same results as the basic algorithm. To do that we show that for a grammar G for which the basic algorithm generates the grammar G' in GNF the incremental algorithm constructs a forest F' representing G' such that $G(F') = G'$.

This will be shown by mathematical induction where the inductive step is that when adding a new production p to G yielding the grammar G_p for which the basic algorithm generates the GNF grammar G'_p and adding this production p to F' using the incremental algorithm will result in the forest F'_p that represents a grammar that equals G'_p. That means is has to be shown that G'_p equals $G(F'_p)$. A sketch of this interrelation is shown in Figure 12.

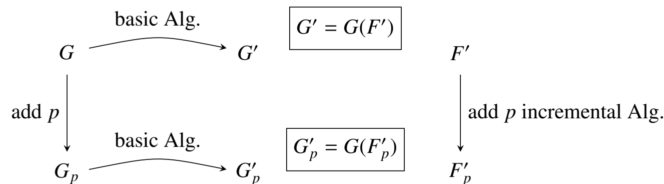

Figure 12: Proving the correctness of the incremental algorithm.

It is sufficient to prove that the claim holds for adding one production at a time because Lemma 2.1 states that when adding a set of productions that keep a gram-

mar productive there exists a sequence to add those productions so that the resulting grammar is productive after each single insertion.

We assume that the terminal and nonterminal alphabets Σ and N have been supplied. The start symbol of the grammar is not important when constructing the GNF, it will be denoted by S and can be any symbol of the set N. Also without loss of generality we let the basic algorithm use the same ordering of nonterminals as the incremental algorithm.

Before starting with the actual proof lets revisit the definition of the production trees and of forests of them. A production tree $T_A^{G_A}$ is defined for a nonterminal A and a grammar G_A. Multiple trees can be combined to a forest if all grammars used to define a single tree share the same alphabet. So when talking about N or Σ in the context of a forest of production trees (abbreviatory just denoted by *forest*) this refers to the alphabet that is shared by all involved trees. Also, unless stated otherwise, let the grammars G_A be minimal with respect to their productions, which means G_A contains only those productions that are required to fulfill Definition 3.1.

The method of the proof is mathematical induction over the number n of productions in the grammar. Starting with $n = 0$ which corresponds to an empty grammar and an empty forest. There are four types of new productions or more precisely the basic as well as the incremental algorithm distinguishes between four different types of productions.

1. The production already has the Greibach property.
2. The production is direct left recursive.
3. The production is ordered and free of left recursion but not in GNF.
4. The production is not ordered.

With the constraints regarding the productivity of the grammar we will see that these different types can only be covered when adding a minimum of three productions.

Proof. Proof of Theorem 1 by mathematical induction over the number $n \in \mathbb{N}$ of productions of the grammar.

Induction Basis We will show that the claim holds for the induction basis by showing that it holds for all $n \in \{0, 1, 2, 3\}$ where $n = 0$ corresponds to the empty grammar.

Case n = 0 Let $G = (N, \Sigma, \emptyset, S)$ be an empty grammar which is obviously in GNF since there is no production that has not the Greibach property. So the basic algorithm does not modify this grammar and the resulting grammar G' is G again. The empty forest $F' = \emptyset$ is representing G' such that $G(F') = G'$ so the claim holds for $n = 0$.

Case n = 1 Let $p = (A \to \gamma)$ be a production and let $G_p := (N \cup \{A\}, \Sigma, \{p\}, S)$ be a grammar that extends G by the new production p. The GNF grammar that results in the application of the basic algorithm on G_p is denoted by G'_p.

What has to be shown is that the forest F'_p that results from adding $p = (A \to \gamma)$ to $F' = \emptyset$ using the incremental algorithm represents the grammar G'_p, that is $G(F'_p) = G'_p$. Since A is a new nonterminal the production $A \to \gamma$ has to contain only terminating symbols and since no nonterminal is terminating at the moment γ has to consist solely

of terminal symbols. Otherwise $A \to \gamma$ would result in G_p being not productive. This means that $A \to \gamma$ is a production in GNF and therefore G_p is already in GNF so the basic algorithm obtains the same grammar again which is $G'_p = G_p$. Adding p to F' results in a new tree $T_A = (V, E, lab)$ with $V = \{r_A, v\}$, $E = \{(r_A, v)\}$ and $lab = \{r_A \mapsto A, v \mapsto \gamma\}$. Since $A \to \gamma$ has the Greibach property the resulting forest is $F'_p = \{T_A\}$ with T_A as above. With v being the only leaf in T_A and T_A being the only tree in the forest, the set of productions of the represented grammar equals $P(T_A) = \{A \to \gamma\}$. Therefore the represented grammar $G(F'_p)$ equals G'_p so the claim holds for $n = 1$.

Case n = 2 Adding a second production requires further case distinction. While for the first production in $n = 1$ the new production had to be already in GNF as seen above there are three possibilities when adding a second production. The second production could again be in GNF already, it could be direct left recursive or it could be ordered and free of left recursion but not having the Greibach property. Adding a second production that is in GNF already works exactly like in case $n = 1$ and will not be shown again. We will show the correctness for adding a direct left recursive production first and then for a production that does not have the Greibach property. The left recursive production will require further case distinction so the plan for the case $n = 2$ is as follows.

- Left Recursive Production $A \to AX\gamma$
  - i) Terminal case ($X \in \Sigma$)
  - ii) Nonterminal case ($X \in N$)
- Non-Greibach Production (ordered and free of left recursion) $A_1 \to A_2\gamma$

Case "Direct Left Recursion" Assume a grammar G with a single GNF production as in case $n = 1$ such that $G = (N, \Sigma, \{A \to a\beta\}, S)$ which is again in GNF already so it equals G'. In the case that the second production is supposed to be a direct left recursive production it has to be of the form $p = (A \to AX\gamma)$ where $X \in (\Sigma \cup N)$ and $\gamma \in (\Sigma \cup N)^*$ [1]. Adding p to G yields the grammar $G_p = (N, \Sigma, \{A \to a\beta, A \to AX\gamma\}, S)$.

How the GNF grammar G'_p will look like depends on the symbol X. In any case the basic algorithm introduces a new nonterminal B with the productions $B \to X\gamma \mid X\gamma B$ and deletes the production $A \to AX\gamma$ while adding the production $A \to a\beta B$.

Case X ∈ Σ If $X \in \Sigma$ the basic algorithm terminates with $G'_p = (N \cup \{B\}, \Sigma, P', S)$ where $P' = \{\ A \to a\beta,\ A \to a\beta B,\ B \to X\gamma,\ B \to X\gamma B\ \}$.

Case X ∈ N X could also be in N but because of the productivity X would have to be a terminating symbol then. The only terminating symbol is A at the moment. So both B productions would be replaced by their left derivation yielding the productions

$$B \to\ a\beta\gamma \mid a\beta\gamma B \mid a\beta B\gamma \mid a\beta B\gamma B$$

It has to be shown that for both cases the forest F'_p which is obtained by incrementally adding p to F' represents G'_p again such that $G(F'_p) = G'_p$.

[1] Note that there has to be a symbol X behind A. The production $A \to A$ is either non-productive if it is the only A production or it is useless in a sense that omitting it would not change the language of the grammar. So this case will not be examined any further.

The starting point is F' which is the forest that represents G' so it contains the single tree $T_A = (\{r_A, v_1\}, \{(r, v_1)\}, \{r \mapsto A, v_1 \mapsto a\beta\})$. The incremental algorithm adds a new node v_2 to T_A such that $T_A = (\{r_A, v_1, v_2\}, \{(r_A, v_1), (r_A, v_2)\}, \{r_A \mapsto A, v_1 \mapsto a\beta, v_2 \mapsto AX\gamma\})$. The new node v_2 is put on the ordering list and processed by Algorithm 3.5 where the direct left recursion is removed.

There, the new nonterminal B is introduced and the extended production tree $T_B = (V, E, lab, lab_R)$ is created with vertices $V = \{r_B, v\}$, edges $E = \{(r_B, v)\}$, labeling function $lab = \{r_B \mapsto B, v \mapsto \gamma\}$ and the additional labeling function as in Definition 3.8 which is $lab_R = \left\{r_B \mapsto \{B\}, v \mapsto \{X\gamma, X\gamma B\}\right\}$. So the represented productions for T_B are $P(T_B) = \{B \to X\gamma, B \to X\gamma B\}$. Node v_2 in T_A is deleted and T_A is made an extended tree such that $T_A = (\{r_A, v_1\}, \{(r_A, v_1)\}, \{r_A \mapsto A, v_1 \mapsto a\beta\}, \{r_A \mapsto \{A\}, v_1 \mapsto \{a\beta, a\beta B\}\})$. and therefore $P(T_A) = \{A \to a\beta, A \to a\beta B\}$.

In the case that $X \in \Sigma$ the algorithm will terminate with $F'_p = \{T_A, T_B\}$ representing the productions $P(T_A) \cup P(T_B) = \{A \to a\beta, A \to a\beta B, B \to X\gamma, B \to X\gamma B\}$ that correspond to those of G'_p so the claim holds for the case that X is a terminal symbol.

For the case that $X \in N$, i.e. $X = A$, creating the node v in T_B will result in a final pointer (r_A, v) and v being added to the final list where it is replaced by its left derivations by creating two nodes named v' and v'' in T_B as children of v so that $(v, v'), (v, v'') \in E$ and $v' \mapsto a\beta\gamma, v'' \mapsto a\beta B\gamma \in lab$. As v' and v'' are leaf nodes now and v is not any longer, the extended labeling function is now

$$lab_R = \left\{ r_B \mapsto \{B\},\ v \mapsto \{X\gamma, X\gamma B\},\ v' \mapsto \{a\beta\gamma, a\beta\gamma B\},\ v'' \mapsto \{a\beta B\gamma, a\beta B\gamma B\} \right\}$$

This results in

$$P(T_B) = \{\ B \to a\beta\gamma,\ B \to a\beta\gamma B,\ B \to a\beta B\gamma,\ B \to a\beta B\gamma B\ \}$$

which corresponds to the B productions as in G'_p for the basic algorithm so the claim also holds for $X \in N$.

Case "Non-Greibach production" As in the left recursive case assume a grammar G with a single GNF production as in case $n = 1$ such that $G = (N, \Sigma, \{A_2 \to a\beta\}, S)$ which is again in GNF already so it equals G'.

A new production that is ordered and free of direct left recursion but does not have the Greibach property is of the form $p = (A_1 \to A_2\gamma)$. where the new nonterminal A_1 is introduced as least symbol so the leading A_2 on the right hand side is greater making the production ordered but not in GNF.

Adding p to G results in the grammar $G_p = (N \cup \{A_1\}, \Sigma, \{A_2 \to a\beta, A_1 \to A_2\gamma\}, S)$ which is transformed by the basic algorithm to the GNF grammar G'_p by replacing $A_1 \to A_2\gamma$ by its left derivations. Resulting in $P'_p = \{A_2 \to a\beta, A_1 \to a\beta\gamma\}$ being the productions of G'_p.

Again it has to be shown that these productions correspond to those represented by the forest F'_p that is obtained by incrementally adding p to F' where F' contains the single tree $T_{A_2} = (\{r_{A_2}, v\}, \{(r_{A_2}, v)\}, \{r_{A_2} \mapsto A_2, v \mapsto a\beta\})$.

Adding $p = (A_1 \to A_2\gamma)$ with the new nonterminal symbol A_1 results in a new tree $T_{A_1} = (\{r_{A_1}, v'\}, \{(r_{A_1}, v')\}, \{r_{A_1} \mapsto A_1, v' \mapsto A_2\gamma\})$ and a backward pointer (r_{A_2}, v'). When v' is created it is put in the Greibach list where the replacement by left derivations results in the T_{A_1} tree being modified to $T_{A_1} = (V', E', lab')$ where $V' = \{r_{A_1}, v', v''\}$, $E' = \{(r_{A_1}, v'), (v', v'')\}$ and $lab' = \{r_{A_1} \mapsto A_1, v' \mapsto A_2\gamma, v'' \mapsto a\beta\gamma\})$.

With v'' being the only leaf node of T_{A_1} the represented productions of F'_p is the set $P(T_{A_1}) \cup P(T_{A_2}) = \{A_1 \to a\beta\gamma\} \cup \{A_2 \to a\beta\}$ which equals P'_p of the grammar G'_p. So

the claim holds for the non-Greibach case as well and since this covered all passible second productions the claim holds for the case $n = 2$.

Case n = 3 While there were already three different possibilities for the second production in case $n = 2$ there are now even more possible "configurations" of production types. We need to examine this case in detail because a production that is not ordered can only occur at the earliest when adding the third production.

In order to obtain a non-ordered production there have to be already two different symbols in N that are terminating. As Table 1 shows there are only two possible configurations if the third production is supposed to be a non-ordered production. Note that $|N_A|$ refers only to the original nonterminals and not to the B symbols that are introduced when removing direct left recursion. The first production has to be always in GNF so it is not mentioned in Table 1 because there is no case distinction for it.

Case	2nd production	$\lvert N_A \rvert$	3rd production not ordered
i)	in GNF (same symbol)	1	no
ii)	in GNF (different symbol)	2	yes
iii)	direct left recursive	1	no
iv)	ordered but not in GNF	2	yes

Table 1: Possible configurations when adding the third production.

We will show that the claim holds for both cases when adding a non-ordered production. The remaining cases for other types of productions work exactly as in case $n = 2$ and will be omitted here.

Assume a grammar G as in case *ii*) that contains productions for two nonterminals that are both in GNF such that $G = (N, \Sigma, \{A_1 \to a\alpha, A_2 \to b\beta\}, S)$. Since G is in GNF already the basic algorithm will not modify it so the resulting grammar G' equals G. Let the order of the nonterminals be $A_1 < A_2$.

A non-ordered production has to be of the form $p = (A_2 \to A_1\gamma)$ because the left hand side symbol needs to be greater than the leftmost right hand side symbol. If the left hand side would be a new nonterminal it would have to be introduced as least symbol and therefore would be ordered in any case. Choosing A_1 as left hand side would result in an ordered production as well since A_1 is the current least symbol, so the left hand side has to be A_2. The right hand sides of new productions are only allowed to contain symbols that are terminating but as the leftmost symbol has to be a lesser nonterminal to make the production non-ordered it has to be A_1. In Table 1 the cases *i*) and *iii*) cannot have a third production that is not ordered because there is only one nonterminal with terminating productions and the construction as above does obviously not work in those cases.

So let the new production be $p = (A_2 \to A_1\gamma)$ resulting in the extended grammar $G_p = (N, \Sigma, \{A_1 \to a\alpha, A_2 \to b\beta, A_2 \to A_1\gamma\}, S)$ which is modified by the basic algorithm yielding the GNF grammar $G'_p = (N, \Sigma, \{A_1 \to a\alpha, A_2 \to b\beta, A_2 \to a\alpha\gamma\}, S)$. The forest F' corresponding to G' consists of the trees T_{A_1} and T_{A_2} representing the productions $P(T_{A_1}) = \{A_1 \to a\alpha\}$ and $P(T_{A_2}) = \{A_2 \to b\beta\}$ where $T_{A_2} = (\{r_{A_2}, v\}, \{(r_{A_2}, v)\}, \{r_{A_2} \mapsto A_2, v \mapsto b\beta\})$.

New productions are always added as children of a root node so adding p extends T_{A_2} by a new node v' with $lab(v') = A_1\gamma$ and a new edge (r_{A_2}, v'). Since v' is not

ordered it is put on the ordering list and the forward pointer (r_{A_1}, v') is inserted. When v' is processed on the ordering list Algorithm 3.9 uses $P_{fon}(T_{A_1})$ for generating the left derivations because v' is a new node and the algorithm is in the ordering pass. The productions $P_{fon}(T_{A_1})$ however, coincide with $P(T_{A_1}) = \{A_1 \to a\alpha\}$ in this case anyway, so just one new node v'' with $lab(v'') = a\alpha\gamma$ is created as child of v'. The represented productions are now $P(T_{A_1}) \cup P(T_{A_2}) = \{A_1 \to a\alpha\} \cup \{A_2 \to b\beta, A_2 \to a\alpha\gamma\}$ which match those of G'_p so the claim holds for case *ii*).

Now assume a grammar G as in case *iv*) such that $G = (N, \Sigma, \{A_1 \to A_2\gamma, A_2 \to a\beta\}, S)$ as for the non-Greibach case for $n = 2$. The non-Greibach production $A_1 \to A_2\gamma$ is replaced by its left derivations once, resulting in the GNF grammar $G' = (N, \Sigma, \{A_1 \to a\beta\gamma, A_2 \to a\beta\}, S)$ and a forest $F' = \{T_{A_1}, T_{A_2}\}$ with the trees

$$\begin{aligned} T_{A_1} &= (V^{A_1}, E^{A_1}, lab^{A_1}) \\ V^{A_1} &= \{r_{A_1}, v_1, v_2\} \\ E^{A_1} &= \{(r_{A_1}, v_1), (v_1, v_2)\} \\ lab^{A_1} &= \{r_{A_1} \mapsto A_1, v_1 \mapsto A_2\gamma, v_2 \mapsto a\beta\gamma\}) \end{aligned} \qquad \begin{aligned} T_{A_2} &= (V^{A_2}, E^{A_2}, lab^{A_2}) \\ V^{A_2} &= \{r_{A_2}, v\} \\ E^{A_2} &= \{(r_{A_2}, v)\} \\ lab^{A_2} &= \{r_{A_2} \mapsto A_2, v \mapsto a\beta\}) \end{aligned}$$

containing also a backward pointer (r_{A_2}, v_1) since v_1 is not in GNF (cf. Figure 13 for an illustration).

When adding a non-ordered production it has to be of the form $p = (A_2 \to A_1\delta)$ following the argumentation as in case *ii*) above. So adding p to G results in G_p with the productions $P_p = \{A_1 \to A_2\gamma, A_2 \to a\beta, A_2 \to A_1\delta\}$.

The basic algorithm transforms G_p to the GNF grammar $G'_p = (N \cup \{B_2\}, \Sigma, P'_p, S)$ where the productions are transformed from P_p to P'_p such that

$$P_p = \{ \begin{array}{ll} A_1 \to & A_2\gamma \\ A_2 \to & a\beta \mid A_1\delta \quad \} \end{array} \qquad P'_p = \{ \begin{array}{ll} A_1 \to & a\beta\gamma \mid a\beta B_2\gamma \\ A_2 \to & a\beta \mid a\beta B_2 \\ B_2 \to & \gamma\delta \mid \gamma\delta B_2 \quad \} \end{array}$$

The new production $A_2 \to A_1\delta$ turns out to be left recursive which results in a new symbol B_2 and several new productions. Note that the question whether this grammar is in GNF already or not depends on γ and δ. Before distinguishing between these cases we will do the incremental insertion of p up to this point.

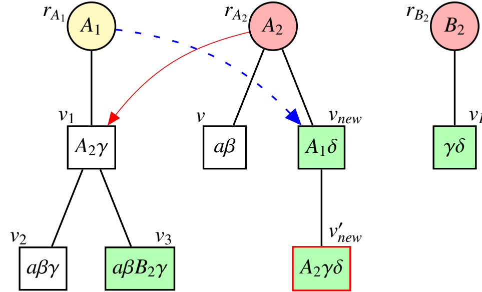

Figure 13: Forest F' is transformed to F'_p by inserting $A_2 \to A_1\delta$

The incremental insertion process of the production $p = (A_2 \to A_1\delta)$ to the forest F' is shown in Figure 13 where new nodes are shaded green again and red root nodes indicate that the respective tree is an extended production tree.

Inserting p results in a new node v_{new} in T_{A_2} with edge (r_{A_2}, v_{new}) and labeling $lab^{A_2}(v_{new}) = A_1\delta$. A forward pointer (r_{A_1}, v_{new}) is added and v_{new} is put on the ordering list. In the ordering pass v_{new} is replaced by its left derivations using the first ordered productions $P_{fon}(T_{A_1}) = \{A_1 \to A_2\gamma\}$ resulting in a new node v'_{new} with edge (v_{new}, v'_{new}) and labeling $lab(v'_{new}) = A_2\gamma\delta$. Since v'_{new} is direct left recursive it is put on the ordering list where it is processed immediately.

Algorithm 3.5 removes this direct left recursion by introducing the new nonterminal B_2 and a new extended production tree $T_{B_2} = (\{r_{B_2}, v_B\}, \{(r_{B_2}, v_B)\}, lab^{B_2}, lab^{B_2}_R)$ with $lab^{B_2} = \{r_{B_2} \mapsto B_2, v_B \mapsto \gamma\delta\}$ and $lab^{B_2}_R = \{r_{B_2} \mapsto \{B_2\}, v_B \mapsto \{\gamma\delta,\ \gamma\delta B_2\}\}$ representing the productions $P(T_{B_2}) = \{B_2 \to \gamma\delta, B_2 \to \gamma\delta B_2\}$.

The production tree T_{A_2} is made an extended tree as in Definition 3.8 with the extended labeling function

$$
\begin{aligned}
lab^{A_2}_R = \{\quad & r_{A_2} \mapsto \{A_2\}, && \text{since } r_{A_2} \text{ is the root node} \\
& v \mapsto \{\alpha\beta, \alpha\beta B_2\}, && \\
& v_{new} \mapsto \{A_1\delta, A_1\delta B_2\}, && \\
& v'_{new} \mapsto \{A_2\gamma\delta\} && \text{since } v'_{new} \text{ is recursive} \quad \}
\end{aligned}
$$

So according to Definition 3.2 and Definition 3.7 the represented productions of the tree T_{A_2} are $P(T_{A_2}) = \{A_2 \to \alpha\beta,\ \alpha\beta B_2\}$.

Now the ordering pass is complete and all worklists are empty. However, Algorithm 3.8 follows the different usage pointers before processing the respective list. In this case the backward pointer (r_{A_2}, v_1) is found when following the pointers of T_{A_2}, so the node v_1 is put on the Greibach list. The node v_1 was already part of F' and is therefore not a new node. Also the algorithm is not in the ordering pass, so the productions used for replacement by left derivations are

$$P_{new}(T_{A_2}) = P(T_{A_2}) \setminus P_{old}(T_{A_2}) = \{A_2 \to \alpha\beta, A_2 \to \alpha\beta B_2\} \setminus \{A_2 \to \alpha\beta\} = \{A_2 \to \alpha\beta B_2\}$$

which results in a single new node v_3 in T_{A_1} as child of v_1 such that $(v_1, v_3) \in E^{A_1}$ and $lab^{A_1}(v_3) = \alpha\beta B_2\gamma$. So $P(T_{A_1}) = \{A_1 \to \alpha\beta\gamma,\ A_1 \to \alpha\beta B_2\gamma\}$ are the new represented productions of T_{A_1}. If there are no more pointers to follow the algorithm terminates. and the set of all productions represented by F'_p is

$$
\begin{aligned}
P(T_{A_1}) \cup P(T_{A_2}) \cup P(T_{B_2}) = \quad & \{A_1 \to \alpha\beta\gamma,\ A_1 \to \alpha\beta B_2\gamma\} \cup \\
& \{A_2 \to \alpha\beta,\ \alpha\beta B_2\} \cup \\
& \{B_2 \to \gamma\delta, B_2 \to \gamma\delta B_2\}
\end{aligned}
$$

which corresponds to the set P'_p of the basic algorithm. Therefore the claim holds for this case as well.

As mentioned ealier the productions P'_p might not have been in GNF. This depends on the words $\gamma, \delta \in (\Sigma \cup N)^*$ for which Table 2 shows a case distinction where X denotes the first symbol in γ or δ.

The cases for which the grammar is already in GNF have already been covered while the cases that result in a final pointer, i.e. $lab^{B_2}(v_B)$ starts with a nonterminal, work analogously to the case $X \in N$ when inserting a direct left recursive production in case $n = 2$ and will not be shown again.

This leaves the case $\gamma \cdot \delta = \varepsilon$ for which a $A_2 \to A_2$ production is generated. The node representing this production is just ignored by Algorithm 3.5 and is not processed. As shown earlier leaving out productions of the form $A_i \to A_i$ does not change the language of a grammar.

This concludes case $n = 3$ and therefore the induction basis has been proven.

γ	δ	Consequence
ε	ε	results in a $A_2 \to A_2$ production
ε	$X \in \Sigma$	grammar is in GNF
ε	$X \in N$	final pointer (r_X, v_B)
$X \in \Sigma$	any word	grammar is in GNF
$X \in N$	any word	final pointer (r_X, v_B)

Table 2: Influence of γ and δ on the grammar G'_p.

Induction Hypothesis For a grammar G with an arbitrary but fixed number of productions n there exists a forest of production trees F' along with usage pointers U' that represent the grammar G'. This grammar G' is identical to the result of the basic algorithm executed on G such that F' and U' hold the information which operations were used by the basic algorithm to obtain G'.

Inductive Step Let $G, G' \in CFG$ be grammars and F' be a forest as in the induction hypothesis. In the inductive step the $(n+1)^{st}$ production p is added to G resulting in the grammar G_p which is transfered to the GNF grammar G'_p by the basic algorithm. It has to be shown that the claim holds, i.e. that adding p to F' incrementally results in a forst F'_p that represents G'_p.

The claim of the inductive step can be put into a lemma.

Lemma 3.4. *Let the forest F' along with the usage pointers U' which is required to represent the grammar $G' \in GNF$ that was obtained from a grammar $G \in CFG$ by the basic algorithm and that F' and U' hold the information of how G' was constructed.*

Executing the incremental algorithm for a new production p using

$$C \quad = \quad extendGrammar(\ (F', W_\emptyset, U'),\ p\ )$$

yields a configuration $C = (F'_p, W_\emptyset, U'_p)$ such that the requirement holds for F'_p, U'_p and the grammars $G'_p \in GNF$, $G_p \in CFG$ again, where G_p denotes the grammar that is obtained by adding the production p to G.

According to the induction hypothesis there is a forest and usage pointers as required by Lemma 3.4. Therefore with Lemma 3.4 the claim of the inductive step holds. □

It remains to show that Lemma 3.4 actually holds. The algorithms distinguish between four types of productions. In this proof we will execute the incremental algorithm for the new production p while distinguishing between the different types of productions if the algorithms do.

The input configuration C changes very often. We will trace every change to this configuration be describing how its elements are changed and will always refer with C_{cur} to the current configuration.

Proof. Let $G, G_p \in CFG$, $G', G'_p \in GNF$, F', F'_p and U', U'_p denote the same objects as in Lemma 3.4. Further let p be an A_i-production.

Lemma 3.4 states that if the incremental algorithm is executed on an input configuration $C = (F', W_\emptyset, U')$ such that F' and U' fullfil the requirements of Lemma 3.4

the forest and usage pointers contained in the resulting configuration fullfil the requirements as well. So to prove Lemma 3.4 we execute the algorithm and show that this claim holds.

The incremental algorithm is started by calling “extendGrammar” (Algorithm 3.8) with the input configuration $C = (F', W_\emptyset, U')$ and the new production p. In line 1 “insertProduction” (Algorithm 3.2) is called with the current configuration C and the production p. A new node v_p is created (line 7, “insertProduction”) representing p. Since v_p is a child of the corresponding root node (line 8, “insertProduction”) this operation adds p to the productions of the grammar represented by F'. In line 9 of “insertProduction” the procedure “checkPointerAndWorklist” (Algorithm 3.3) is called with C and the new node v_p.

There, the algorithm distinguishes between different types of productions.

- If v_p is a direct left recursive node, i.e. p is of the form $A_i \to A_i\gamma$, no usage pointer is inserted which is correct since the basic algorithm does not replace such productions by their left derivations. The node v_p is put on the ordering list held in C (line 2) such that the recursion is removed later. Note that for the follwing cases $i \neq j$ holds, otherwise this case would have applied (line 1, “checkPointerAndWorklist”).
- If v_p is not ordered, i.e. if the production is of the form $p = (A_i \to A_j\gamma)$ with $i > j$, a forward pointer (r_{A_j}, v_p) is added to the forward pointers held by C (line 4, “checkPointerAndWorklist”) and v_p is put on the ordering list held by C (line 5, “checkPointerAndWorklist”).
- If v_p is found being not in GNF (line 6, “checkPointerAndWorklist”) p has to be of the form $A_i \to A_j\gamma$ with $i < j$. A backward pointer (r_{A_j}, v_p) is added to the backward pointers held by C (line 7, “checkPointerAndWorklist”) and v_p is put on the Greibach list (line 8, “checkPointerAndWorklist”).

The procedure “checkPointerAndWorklist” ends and yields a configuration that might have a non-empty ordering or Greibach list and probably new usage pointers. The forest also contains the new production p now.

Now the main procedure “extendGrammar” follows all forward pointers by calling “followPointers” (Algorithm 3.8) with C_{cur}, the tree T_{A_i} and the pointer type *forward* as parameters. The procedure is called to find all those productions that were replaced by their left derivations in the basic algorithm using A_i productions. To prevent following the pointers originating in a tree twice, a tree is marked as *followed from* (line 4) and the procedure exits (line 1-3, “followPointers”) preventing the usage pointers of such a tree being followed again.

The forward pointers originating in T_{A_i} are of the form (r_{A_i}, v_f) with $v_f \in T_{A_j}$. For each such pointer the node v_f is put on the ordering list of C_{cur}. When such a node is replaced it changes the productions of its corresponding tree T_{A_j}. The A_j-production might have been used in the ordering pass of the basic algorithm themselves. Therefore “followPointers” is called with C_{cur}, T_{A_j} and *forward* as well.

The assumption states that the forest and the usage pointers describe the original run of the basic algorithm on G, so by following the forward pointers this way the ordering list contains all nodes now that are influenced by A_i-productions at the ordering stage.

After all forward destinations have been gathered the procedure “followPointers” ends and the main procedure invokes the procedure “processWorklist” (Algorithm 3.6) with C_{cur} and *oderingList* (line 3).

By definition of how nodes are put on it, the nodes on the ordering list are sorted ascending by their left hand side nonterminal where non-ordered nodes for a certain nonterminal are in front of the direct left recursive nodes. Note that the ordering list cannot have any productions of nonterminals A_j with $j < i$, because we started with an A_i productions and followed forward pointers only.

While the ordering list is not empty (line 1), "processWorklist" removes the current node v_f from the ordering list and calls "removeRecursion" (Algorithm 3.5) (line 4) or "leftDerivation" (Algorithm 3.9) (line 6) depending on the type of production (direct left recursive or not, line 3).

Consider the case that the production is not direct left recursive. The parameters for "leftDerivation" are C_{cur} and v_f. Now the productions that are used for this derivation are selected. At first "leftDerivation" distinguishes between new and old nodes. The only new node that has been created so far is the node v_p representing the new production p while all forward destinations are nodes that were already part of F'.

If v_f is a new node, it represents a production that was not part of the original run of the basic algorithm on G. Therefore all available productions have to be used to simulate a run of the basic algorithm on G_p.

If v_f is an old node, it was already part of the original run on G and was already replaced by its left derivations. Since the operation of replacing a node by its left derivations has to use all available productions, the incremental algorithm has just to complete this derivation by using the new productions that were not part of the original run on G.

This selection regarding new and old nodes are implemented by "leftDerivation" with the condition in lines 1 such that "leftDerivation" selects the new productions only (line 8) or old productions (lines 2-6). By either using all available productions or new nodes only, the incremental algorithm has to perform a left derivation that meets the requirements of Lemma 3.1 and will therefore create the same productions as the basic algorithm.

Before this holds there is another case distinction that has to be carried out. There is a difference if the incremental algorithm is in the ordering pass or not. Since the forest F' holds the information of how G' was obtained from G by the basic algorithm the trees in F' were modified by the Greibach pass in the original run so they do not represent the productions the basic algorithm had available during the ordering pass. If clipping those nodes created in the Greibach pass as in Definition 3.13 resulting in the productions P_{fon} as used by "leftDerivation" (line 3).

Therefore according to Lemma 3.1 the incremental algorithm creates the same productions as the basic one. In line 13 "checkPointerAndWorklist" is called for each new node assigning the proper usage pointer and worklist to them so the resulting forest and usage pointers describe this step of the basic algorithm in G_p properly.

The second type of node on the ordering list are recursive node that are handled by "removeRecursion" (Algorithm 3.5). Let the current node v_f represent $A_i \to A_i\gamma$ with $\gamma \in (\Sigma \cup N)^+$ (case $\gamma = \varepsilon$ is ignored by lines 2-4). At first the recursive node is made a blind leaf so it is no longer part of the represented grammar. The node v_b created in the extended T_{B_i} (ensured to exist in lines 5-9) represents $B_i \to \gamma \mid \gamma B_i$ and receives a final pointer if it is not in GNF (and also put on the final list) so that the information of how the run on G_p was performed is preserved.

Lemma 3.2 ensures that the same productions are created as in the basic algorithm where the argumentation for the productions P_{fon} still applies. Note that nodes that are representing a right recursive copy after the call of "removeRecursion" but were put

on the Greibach list before are still properly processed, since their left derivations also represent a right recursive copy each.

After all nodes on the ordering list have processed in this way the procedure ends. The main procedure gathers the backward destinations of all trees holding new nodes (lines 4-6) because left derivations of those destinations need to be completed as described above.

When processing the Greibach list by calling "processWorklist" with C_{cur} and *GreibachList* only the procedure "leftDerivation" is used when processing the elements v_g in the list since recursive nodes only occur on the ordering list.

Lemma 3.1 now applies without any restriction since the trees used derivation now represent the productions actually used. Therefore using "leftDerivation" creates the same productions as the basic algorithm and as shown above creates the required usage pointers as well.

The same holds for the final pass for which final pointer destinations have to be gathered (line 8-10) analogously to the Greibach pass. Using "leftDerivation" along with Lemma 3.2 yields the same result when processing the final list in line 11.

So the incremental algorithm terminates with the worklists of C_{cur} being empty and the forest F'_p with $G(F'_p) = G'_p$ along with the usage pointers U'_p are holding the information of how the basic algorithm obtained G'_p from G_p.

Therefore Lemma 3.4 holds. $\square$

3.4 Complexity

The complexity of the incremental algorithm will be examined for a context-free grammar $G = (N, \Sigma, P, S)$ for which $G' = (N', \Sigma, P', S)$ is the GNF grammar that is obtained from G by the basic algorithm while the forest F' hold this information.

Let $|N| \in \mathbb{N}$ denote the number of nonterminals and $|P| \in \mathbb{N}$ the total number of productions in G. As mentioned earlier only the ordering pass generates productions that might need further processing while in the Greibach and final pass the Greibach property is established in a single step by replacing a production by its left derivations once.

We will analyze the overall complexity in terms of the productions represented by the trees of a forest of the incremental algorithm. The result for the basic algorithm is the same since the incremental algorithm simulates the basic one.

When establishing the Greibach property for a production $p \in P$ the number of nonterminals $|N|$ is an upper bound for the number of replacements by left derivations that are required until the production is in GNF since there are at most $|N| - 1$ replacements required for ordering p as in Definition 2.4 plus one step for establishing the Greibach property. So the maximum tree depth is $|N|$ for each production tree within the forest.

The values we are interested in are upper bounds for the new number of productions $|P'|$ and the number of inner nodes, i.e. the number of intermediate productions.

At first only consider the ordering pass. The number of productions $|P|$ is an upper bound for the number of productions for a certain nonterminal $A_i \in N$ in the original grammar. The least nonterminal A_1 is ordered, but if it contains a direct left recursion the number of productions doubles such that $|P(A_1)| < 2|P|$. This factor two has to be considered for the upper bound of each symbol such that, in the worst case, after ordering all productions of a certain A_i a single production is direct left recursive.

The worst case for ordering productions of a nonterminal A_i is that all those productions got an A_1 as leftmost symbol which in turn produce A_2 at the leftmost position and so on up to A_i. For the number of productions after the ordering pass this means that the upper bound is a product such that

$$|P_{fon}(A_i)| \quad < \quad \underbrace{|P|}_{A_i\ productions} \cdot \underbrace{|P_{fon}(A_1)|}_{A_1\ productions} \cdot \ldots \cdot \underbrace{|P_{fon}(A_{i-1})|}_{A_{i-1}\ productions} \cdot \underbrace{2}_{left\ recursive\ factor}$$

which can be written as

$$|P_{fon}(A_i)| \quad < \quad 2|P| \cdot \prod_{k=1}^{i-1} |P_{fon}(A_k)|$$

The worst case for the entire algorithm is that this is the case for each nonterminal in the grammar. Since $|P_{fon}(A_1)| = 2|P|$ the maximum number of productions for the greatest nonterminal A_n after the ordering pass in the worst case would be

$$\begin{aligned} |P_{fon}(A_n)| \quad < \quad 2|P| \cdot \prod_{k=1}^{n-1} |P_{fon}(A_k)| \quad &= \quad 2|P| \cdot \prod_{k=1}^{|N|-1} |P_{fon}(A_k)| \\ = \; 2|P| \cdot \underbrace{(2|P|)^1}_{|P_{fon}(A_1)|} \cdot \underbrace{(2|P|)^2}_{|P_{fon}(A_2)|} \cdot \underbrace{(2|P|)^4}_{|P_{fon}(A_3)|} \cdot \ldots \cdot \underbrace{(2|P|)^{2^{|N|-2}}}_{|P_{fon}(A_{n-1})|} \; &= \; (2|P|)^{2^{|N|-1}} \end{aligned}$$

In the Greibach pass there is only one additional replacement step such that the Greibach property for A_i is established by inserting the GNF productions of A_{i+1}. An upper bound of the number of the final GNF productions for a nonterminal A_i is

$$|P(A_i)| \; < \; |P_{fon}(A_i)| \cdot \prod_{k=i+1}^{n} |P_{fon}(A_k)|$$

and since the productions for A_n are in GNF after the ordering pass the final productions of A_n are $P(A_n) = P_{fon}(A_n)$ and an upper bound for the productions of the least nonterminal A_1 is

$$\begin{aligned} |P(A_1)| \quad &< \quad |P_{fon}(A_1)| \cdot \prod_{k=2}^{n} |P(A_k)| \\ &= \quad \underbrace{2|P|}_{|P_{fon}(A_1)|} \cdot \underbrace{(2|P|)^2}_{|P_{fon}(A_2)|} \cdot \ldots \cdot \underbrace{(2|P|)^{2^{|N|-2}}}_{|P_{fon}(A_{n-1})|} \cdot \underbrace{(2|P|)^{2^{|N|-1}}}_{|P_{fon}(A_n)|} \\ &= \quad (2|P|)^{2^0+2^1+\ldots+2^{|N|-1}} \quad = \quad (2|P|)^{2^{|N|}-1} \\ &< \quad (2|P|)^{2^{|N|}} \end{aligned}$$

The productions of newly introduced B-symbols can be ignored when determining an upper bound for all productions in the grammar since additional direct left recursive productions for a certain A_i insert just one additional B_i-production which is transfered to GNF with at most one replacement but reduces the factor of the A_i productions by one, because the direct left recursive production is removed.

Therefore the number of productions $|P'|$ of the GNF grammar G' for the worst case is of

$$O(\; |N| \; \cdot \; |P|^{2^{|N|}} \;)$$

The maximum number of intermediate nodes, i.e. intermediate productions, depends from the tree depth which is $|N|$. With the total number of the final productions as above this value has to be of the order

$$O(\ |N|^2\ \cdot\ |P|^{2^{|N|}}\ )$$

In the best case all productions of G are in GNF already and the productions are $O(|P|)$ while the intermediate productions are $O(0)$ since no replacements take place.

The worst case when adding a production p to F', i.e. when analyzing the incremental insertion of a single production, we find that it is not different from the overall result. This is because we only loose the factor $2|P|$ once which is just a minus one in the exponent and therefore is constant since the exponent grows with $2^{|N|}$.

Note that $O(\ |N| \cdot |P|^{2^{|N|}}\ )$ is an upper bound of the complexity in terms of the number of resulting productions, but it is certainly not the best. The double exponential blowup is probably not constructable in practice so the worst case cemplextity might be closer to

$$O(\ |N|\ \cdot\ |P|^{|N|}\ )$$

which is still grows exponential with $|N|$.

4 Optimization

The incremental algorithm can be optimized and improved with respect to speed and flexibility. In this section several such enhancements or general ideas will be introduced but will not be proven as correct.

4.1 Adding Multiple Productions

One improvement that affects flexibility as well as speed is to allow the main procedure shown in Algorithm 3.8 to add multiple productions at once by passing a set of productions $P' = \{p_1, \ldots, p_k\}$. The condition that adding the set P' has to keep the grammar productive still applies, but there is no need to bother finding a correct sequence since all productions are added at once.

When adding multiple productions Algorithm 3.8 has to be changed with respect to the call of "insertProduction" (shown in Algorithm 3.2) which has just to be called for every production in P' before the usage pointers are followed and the worklists are processed. The only thing that has to be taken care of is that the respective production tree exists if the algorithm tries to add usage pointers.

4.2 Treating Recursive Nodes Like Non-Ordered Nodes

Another optimization regards the handling of direct left recursive productions when putting them on the ordering list. In the basic algorithm the operation that removed direct left recursive nodes had to be executed after all A_i productions had been ordered because for creating the right recursive copies of the form $A_i \rightarrow \gamma B_i$ the algorithm had to know which A_i production is left recursive, i.e. which production eventually becomes direct left recursive and which does not.

Creating a separate node for each right recursive copy doubles the amount of non-recursive A_i nodes and all of those nodes might need further processing as well. So handling the copy via the additional labeling function as in Definition 3.8 is already an inherent optimization of the incremental approach.

There is however more potential for optimization in this step. For this incremental approach direct left recursions do not necessarily be processed after all productions for the same left hand side have been ordered.

Definition 3.8 states that additional right recursive labels are only included for for all nodes except recursive nodes and the root node. This might create copies of nodes that are not ordered yet and thereby creating productions that are not created by the basic algorithm. Even so this does not pose a problem, because if nodes that are not ordered yet are replaced by their left derivations, their descendants which will eventually become ordered and will represent the proper productions then.

The optimization is to treat nodes representing direct left recursive productions just as any node that is not ordered, so there is no need for separate insertion in the ordering list and the productions are just ordered by their left hand side symbol.

The general concept of adding multiple productions while putting direct left recursive productions to the ordering list just as any non-ordered production will be shown by adding a set of productions to an empty grammar in the following example.

Example 4.1 (Multiple new productions with direct recursion handling)**.** *Consider an empty grammar* $G = (N, \Sigma, \emptyset, S)$ *with* $N = \{A_1, A_2, A_3, A_4\}$ *in the usual order.*

Adding and a set of productions P corresponding to those of Example 2.1 to an empty forest results in a forest as shown in Figure 14 solely containing new nodes and three usage pointers.

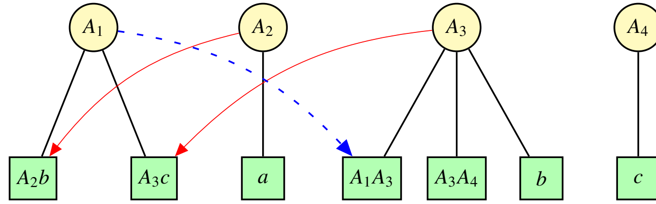

Figure 14: Production trees after adding the original productions of Example 2.1

Since new productions are always added as nodes that are children of root nodes, the first call of the procedure results in a direct representation of the grammar with the productions P that have been passed.

Figure 15 shows the ordering and Greibach list after all productions have been inserted. Note that the order of the productions on the ordering list does not matter since there are only two A_3 productions. However, the direct left recursive production $A_3 \to A_3A_4$ is processed first in order to show that it does not make any difference.

Before the ordering list is processed Algorithm 3.8 follows all paths of forward pointers. The only pointer to follow is (r_{A_1}, v_1^3) with $lab^{A_3}(v_1^3) = A_1A_3$ but this node is a new node and therefore has already been put on the ordering list.

Figure 15: Ordering and Greibach list after adding the productions of Example 2.1

Processing the node $A_3 \to A_3A_4$ on the ordering list results in a forest as shown in Figure 16 with the new extended tree T_{B_3} containing the node v_{B_3} representing the production $B_3 \to A_4$. There is also a new final pointer (r_{B_3}, v_{B_3}) and the node v_{B_3} is put on the final list.

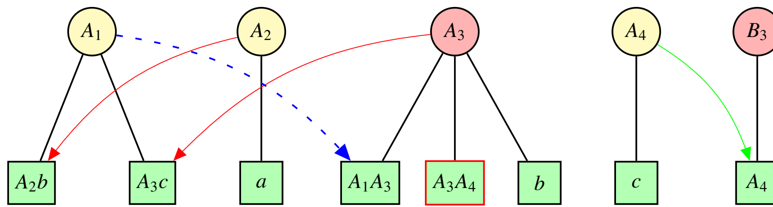

Figure 16: Production trees after processing $A_3 \to A_3A_4$ on the ordering list

The productions $P(T_{A_3}) = \{A_3 \to A_1A_3,\ A_3 \to A_1A_3B_3,\ A_3 \to b,\ A_3 \to bB_3\}$ which are the current A_3 productions would lead to an invalid grammar in the basic algorithm. In the incremental algorithm this does not pose a problem since the right recursive copies were only created on query.

Processing the next item on the ordering list $A_3 \to A_1A_3$ results in the two children $A_3 \to A_2bA_3$ and $A_3 \to A_3cA_3$. These are both put on the ordering list and are

processed at once which completes the ordering pass. The resulting forest is shown in Figure 17.

The tree T_{A_3} is now representing the same productions as the basic algorithm would after ordering. These are $P(T_{A_3}) = \{A_3 \rightarrow abA_3,\ A_3 \rightarrow abA_3B_3,\ A_3 \rightarrow b,\ A_3 \rightarrow bB_3\}$.

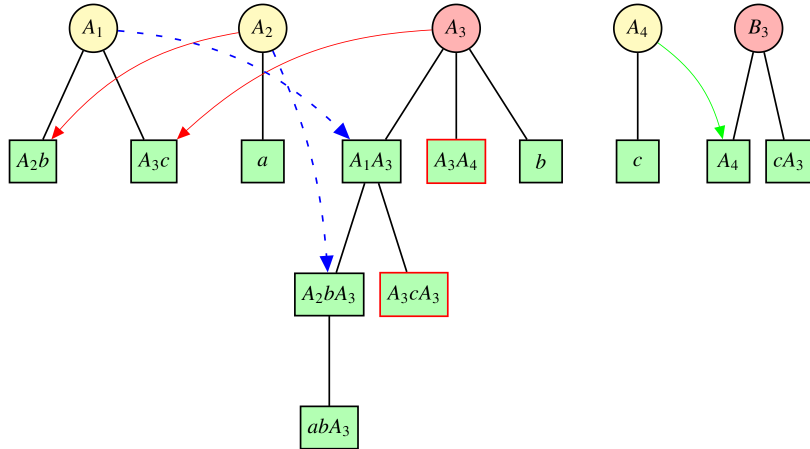

Figure 17: Production trees after completing the ordering pass

Before processing the Greibach list the algorithm follows all backward pointers. But as for the forward pointers the algorithm only finds new nodes that are already on the Greibach list.

As nodes on the Greibach list are replaced by their left derivations yielding soley nodes that have the Greibach property the Greibach pass will finish after processing the two nodes on the Greibach list. The production $A_1 \rightarrow A_2b$ is replaced using the single A_2 production $P(T_{A_2}) = \{A_2 \rightarrow a\}$ while $A_1 \rightarrow A_3c$ is replaced using the four productions from $P(T_{A_3})$ as above.

Since all nodes are new nodes in this example there is no difference when following the final pointers before processing the final list. As for the Greibach pass items on the final list need only be replaced by their left derivations once. So after replacing $B_3 \rightarrow A_4$ with $B_3 \rightarrow c$ the incremental algorithm terminates. The final forest F is shown in Figure 18 and the represented GNF grammar $G(F)$ matches the grammar that was obtained in Example 2.1.

4.3 Managing Worklists

As we just saw the algorithm could already be improved by not distinguishing between nodes that are not ordered and nodes that are direct left recursive. In Section 3.2 (The Incremental Algorithm) new nodes were added to the worklists so that the nodes were ordered by their left hand side symbol.

This order has to be maintained of course since the first step of the algorithm (ordering and removal of direct left recursion) has to be performed in ascending order with respect to the nonterminals while the second step (establishing the Greibach property for all productions) has to be performed in descending order.

There might, however, be the possiblity to process new nodes in a sequence that maintains this order while operating the worklists FIFO or LIFO based which would

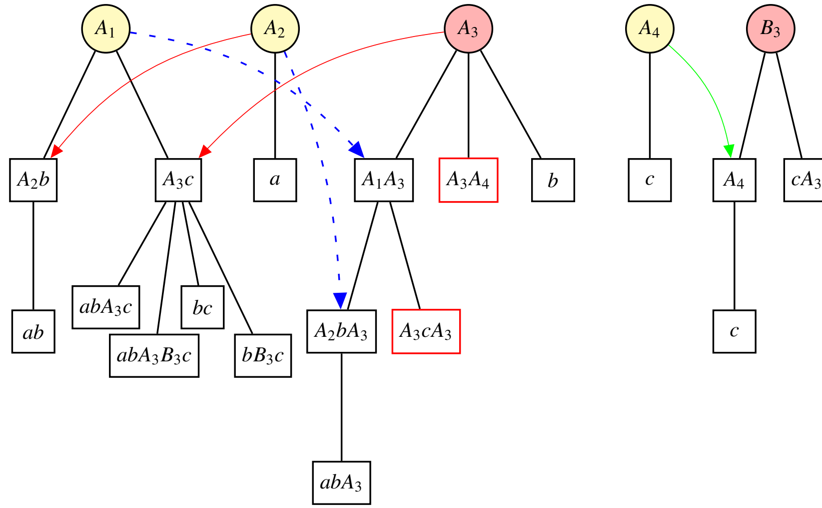

Figure 18: Forest of production trees after completing the final pass

be an improvement since the algorithm would not need to seek the correct position within a worklist for a new node. We will show that this could be done for the final list, that it is not possible for the Greibach list and that it is too costly for the ordering list.

The final list only holds items of the form $B_i \to A_j\gamma$ for some newly introduced nonterminal B_i and an original nonterminal A_j. Those A_j are never new "B-symbols" that were introduced when removing direct left recursion but always nonterminals of the original grammar. For all productions of those original "A-symbols" the Greibach property is established in the Greibach pass. Since the final pass is executed after the Greibach pass and "B-symbols" never occur at the leftmost position, the order in which the final list is processed is of no concern for the output grammar. So the algorithm could indeed just use a stack or a queue for the final list saving on the ordered insertion.

When adding nodes to the Greibach list it is not possible to maintain a proper order by left hand side symbol that would allow to operate the worklist FIFO or LIFO based. Example 4.2 illustrates this fact.

Example 4.2. *Consider a grammar $G \in CFG$ with the productions*

$$\begin{aligned} A_1 &\to A_4a \\ A_2 &\to A_4b \mid A_1c \\ A_3 &\to A_4d \\ A_4 &\to a \end{aligned}$$

So after adding each of these productions using Algorithm 3.2 ("insertProduction") the Greibach list contains three nodes, one for left hand sides A_1, A_2 and A_3 each while the ordering list contains just $A_2 \to A_1c$. Figure 19 shows the ordering and Greibach list at this stage. In the ordering pass the production $A_2 \to A_1c$ is replaced by $A_2 \to A_4ac$ which has to be put on the Greibach list but can neither be appended at the head nor at the tail.

The problem that prevents the use of a stack or a queue is that during the processing of the ordering pass new nodes, i.e. productions, are generated that might need to be

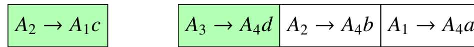

Figure 19: Ordering and Greibach list after adding the productions of Example 4.2

put on some worklist again. That is what happened to the Greibach list in Example 4.2 when the ordering list was being processed.

While it is impossible to operate the Greibach list FIFO or LIFO based, this could be achieved for the ordering list. As seen earlier in this section the sequence when adding multiple nodes does not matter anymore, so productions could now be added starting with productions of the greatest nonterminal proceeding down to those of the least nonterminal. When operating the ordering list like a stack the least items would be atop and would be processed first. If a production would not be ordered after one replacement, the procedure "leftDerivation" (Algorithm 3.9) results in new nodes representing productions of the same nonterminal, so putting them on the top of the stack would result in a proper sequence for processing.

This becomes difficult however when following paths of forward pointers. The algorithm has to deal with a set of production trees that contain new nodes and secondly with different paths of forward pointers originating in those trees. The forward pointers for each tree would need to be ordered so that the algorithm could determine the production tree of the next lower nonterminal without searching all destinations. But even then the task would not be simple as Example 4.3 shows.

Example 4.3 (Following Forward Pointers)**.** *Consider a configuration of production trees as in Figure 20 containing certain forward pointers that will not be defined in detail. Suppose a new A_1 production is added. At first the forward pointers to nodes in T_{A_2} need to be followed, but before the pointers from T_{A_1} to T_{A_4} are followed the algorithm needs to switch to T_{A_2} because the destinations in T_{A_3} need to be processed before those in T_{A_4}. The algorithm would have to maintain a lot of information in order to process all new nodes in sequence.*

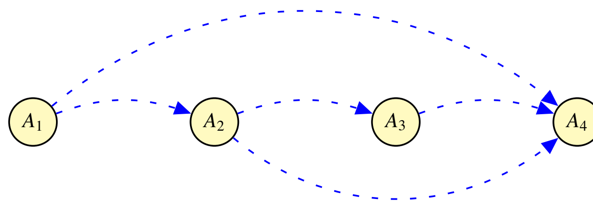

Figure 20: Inserting nodes in ordering worklist in a proper sequence proves difficult

In addition considering that the nodes with greater nonterminal need to be put on the stack first makes this approach impractical. Using a queue does not work either because new nodes created by processing a node on the ordering list results in productions of the current nonterminal which have to be processed first.

In practice seeking the proper position for a new node within the ordering list can more easily be achieved by splitting the ordering list in $|N| = n$ ordering lists where

there is a list for each nonterminal in N. Since the nodes need only be ordered with respect to their left hand side symbol, a new node can just be appended to the list of this symbol. To process such a *combined ordering list* the algorithm has to start with the list of the least nonterminal proceeding to the list of the next nonterminal once the current list is empty.

4.4 Checking for Usage Pointer and Worklist

As seen above only the ordering pass can create any nodes that might need further processing. The nodes generated in the Greibach and final pass are always in GNF and therefore need no further processing. This means they do not have to be put on a worklist and no usage pointer is created for them. It is therefore reasonable to use two different procedures to process the ordering list on the one hand and the Greibach and final list on the other hand using different procedures to do a replacement by left derivations. The left derivation of nodes from the Greibach and final list do not need to call "checkPointerAndWorklist" (Algorithm 3.3). Also there is no need for the case distinction as in the original "leftDerivation" (Algorithm 3.9) since the ordering pass would have its own procedure.

4.5 Inserting New Nonterminals

When adding a production of a new nonterminal $A \notin N$ we defined in Section 3.2 that A is made the least element in N.

This heuristic exploits the fact that a new nonterminal is always introduced as left hand side of a new production and therefore has only known, terminating symbols on its right hand side that will all be greater in the total order if the new nonterminal is made the least symbol. As a result the new production will be ordered in any case. The ordering pass was the only pass that could result in additional nodes that need processing again, so the justification for choosing this heuristic is that nodes for new nonterminals would require one replacement by left derivations at most.

4.6 Finding Represented Productions

Every time the algorithm needs to query nodes, when using them for replacement by left derivations for example, the corresponding tree needs to be traversed using depth-first or breadth-first search to find the leaf nodes. For a production tree $T_A = (V, E, lab)$ the time complexity is $O(|V| + |E|)$. Maintaining a doubly-linked list for each tree containing all leaf nodes and another list containing all first ordered nodes could improve the runtime of the algorithm since there would be no need to search for these nodes.

In return these lists need to be changed every time a node is replaced by its left derivations, i.e. when it is no longer a leaf node. The complexity of maintaining such a list for a tree $T_A = (V, E, lab)$ is $O(|V|)$ since the cost for changing the doubly-linked list at a single node is constant.

Suppose a node v in a list is replaced by left derivations which generates k child nodes $v_1, \ldots, v_k$. This requires changing the four pointers in the doubly-linked list involving v and setting four pointers for each new node v_i where $i \in \{1, \ldots, k\}$. A single node within a tree $T_A = (V, E, lab)$ is linked at most once into the list and once out of the list. So for a single node there are at most eight list pointer operations such that the complexity is $O(8 \cdot |V|)$ which equals $O(|V|)$.

An alternativ solution for using an extra list for first ordered nodes is to use the list for the leaf nodes and check the parent nodes. Since there is only one derivation necessary to obtain the Greibach property for an ordered node. For a given leaf node a first ordered node will either be the parent of the leaf node or it will correspond to the leaf node if the Greibach property was already established during the ordering.

5 Testing an Implementation of the Incremental Algorithm

The incremental algorithm was implemented and its performance was tested and compared to an implementation of the basic algorithm. There were no optimizations for both algorithms except the inherent optimization of the incremental algorithm which creates right recursive copies on query only.

At first the number of used operations and the number of intermediate productions of the algorithms were counted and compared to each other. More precisely the number of replacements by left derivations and the number of removals of direct left recursions was counted. The number of intermediate productions is the total number of productions created by the simple algorithm and respectively the total number of new nodes created by the incremental algorithm.

Secondly, the runtime was measured. The runtime for a single step was measured for both algorithms and the total runtime when adding productions one by one was calculated, i.e. the sum of all steps so far.

The system the algorithms were tested on was running Ubuntu 10.04.2 and had an Intel Pentium 4 CPU at 2.53 GHz with 1 GB RAM.

The algorithms were tested on the grammar $G = (N, \Sigma, P, S)$ with $N = \{A_2, \ldots, A_5\}$, $\Sigma = \{a, b, c, \ldots\}$ and productions

$$\begin{aligned} A_2 &\rightarrow a \mid A_3 b \mid A_4 c \\ A_3 &\rightarrow b \mid A_4 c \\ A_4 &\rightarrow c \\ A_5 &\rightarrow d \end{aligned}$$

that were added in a sequence following Lemma 2.1. After that the grammar was extended by the productions (one at a time)

$A_4 \rightarrow A_3 a$	causes A_4 becoming left recursive
$A_5 \rightarrow aa$	GNF production of A_5 that is not used in any pass
$A_1 \rightarrow A_4 ba$	ordered production of the new symbol A_1
$A_1 \rightarrow bb$	GNF production of A_1 that is not used in any pass
$A_3 \rightarrow A_2 cc$	not ordered production of A_3 that makes all A-symbols left recursive
$A_3 \rightarrow abc$	GNF production of A_3 which is used in the Greibach pass
$A_2 \rightarrow A_5 ab$	ordered production of A_2 used for ordering

The productions have been chosen so that the first seven productions are ordered and free of recursion such that productions that are more costly with respect to the operations being executed are added at a later point.

Table 3 shows the statistics that were determined for the grammar G where $|P|$ is the number of productions in the original grammar and $|P'|$ for the resulting GNF grammar. The columns for the numbers of operations and intermediate productions show the numbers for both the basic and incremental algorithm where those of the basic algorithm are on the left side of the slash while those of the incremental algorithm are right of it.

Note that the number of replacements by left derivations is the number of how many times this operation was used and not how many productions were created during the process which is covered by the intermediate productions.

When analyzing these values it is noticable that those of the incremental algorithm are at most equal to those of the basic algorithm, in most cases they are considerably

$\|P\|$	new production	$\|P'\|$	replacements by left derivations	direct left recursions	intermediate productions
1	$A_5 \to d$	1	0 / 0	0 / 0	0 / 0
2	$A_4 \to c$	2	0 / 0	0 / 0	0 / 0
3	$A_3 \to b$	3	0 / 0	0 / 0	0 / 0
4	$A_2 \to a$	4	0 / 0	0 / 0	0 / 0
5	$A_3 \to A_4c$	5	1 / 1	0 / 0	1 / 1
6	$A_2 \to A_3b$	7	3 / 1	0 / 0	4 / 2
7	$A_2 \to A_4c$	8	4 / 1	0 / 0	5 / 1
8	$A_4 \to A_3a$	22	5 / 4	1 / 1	20 / 11
9	$A_5 \to aa$	23	5 / 0	1 / 0	20 / 0
10	$A_1 \to A_4ba$	27	6 / 1	1 / 0	24 / 4
11	$A_1 \to bb$	28	6 / 0	1 / 0	24 / 0
12	$A_3 \to A_2cc$	132	13 / 7	5 / 4	138 / 76
13	$A_3 \to abc$	180	13 / 6	5 / 0	185 / 36
14	$A_2 \to A_5ab$	278	62 / 11	5 / 0	331 / 79

Table 3: Statistics when adding one production at a time.

less. For the 8th production for instance it is clearly visible that even if the incremental algorithm uses almost the same amount of language preserving operations it creates considerably less productions because it only completes its previous result by new productions. For the removal of left recursions it becomes apparent that the incremental algorithm never removes the same recursion twice.

Table 4 shows the runtimes measured for the algorithms. The columns *basic algorithm* and *incremental algorithm* hold the runtimes for establishing the GNF in the current step while the columns *basic sum* and *incremental sum* hold the sums of that times up to the current step. The runtime for a certain step was determined by running the respective algorithm between five and seven times, removing the value deviating the most of the average and calculating the average of the remaining values. The unit of the runtime is nanoseconds.

To know the total runtime for establishing the GNF for a grammar holding the productions up to a certain row in Table 4 the value in *basic algorithm* has to be compared to *incremental sum*. The basic algorithm is expected being faster when constructing the GNF for a certain grammar. This, however, is not true for the first, the seventh and the last three productions. Especially for the last production the difference becomes significant and reaches a factor of about twelve in favour of the incremental algorithm while for all other cases the factor was below two in each direction.

This unexpected result might be caused by two advantages of the incremental algorithm that were not shared by the implementation of the basic algorithm used for the tests. The first advantage is the already mentioned inherent optimization that right recursive copies are only created on query and are not processed separately. Especially in the last three steps when all A-symbols become left recursive the amount of productions processed by the basic algorithm can become almost twice the amount processed by the incremental version. The second advantage is probably the use of worklists by the incremental algorithm. The basic algorithm checks each rule if its ordered then checks if the rule is free of direct left recursion and finaly checks if the rule has the Greibach property. The implementation of the incremental algorithm used for the tests

$\lvert P \rvert$	new production	basic algorithm	incremental algorithm	basic sum	incremental sum
1	$A_5 \to d$	0.062	0.038	0.062	0.038
2	$A_4 \to c$	0.048	0.033	0.110	0.071
3	$A_3 \to b$	0.061	0.033	0.171	0.104
4	$A_2 \to a$	0.074	0.032	0.245	0.136
5	$A_3 \to A_4c$	0.107	0.062	0.352	0.198
6	$A_2 \to A_3b$	0.171	0.066	0.523	0.264
7	$A_2 \to A_4c$	0.386	0.053	0.909	0.317
8	$A_4 \to A_3a$	0.389	0.180	1.298	0.497
9	$A_5 \to aa$	0.384	0.037	1.682	0.534
10	$A_1 \to A_4ba$	0.472	0.082	2.154	0.616
11	$A_1 \to bb$	0.485	0.038	2.639	0.654
12	$A_3 \to A_2cc$	2.157	0.640	4.796	1.294
13	$A_3 \to abc$	2.480	0.332	7.276	1.626
14	$A_2 \to A_5ab$	26.971	0.645	34.247	2.271

Table 4: Runtimes when adding one production at a time.

checks if a production has the Greibach property and if it does the algorithm performs no further checks.

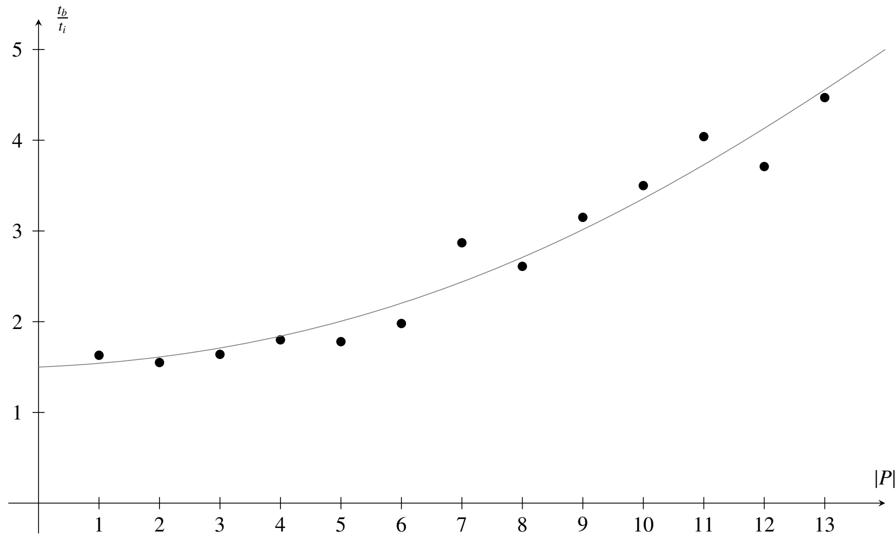

Figure 21: Factor of the total runtime when establishing GNF after each production

In a context in which productions are added consecutively and requiring the grammar being transformed to GNF after each insertion the value in *basic sum* has to be compared to *incremental sum* in order to know the total runtime. This is more meaningful than comparing the costs for a single step since it is averaged over all insertion steps up to the current production.

Figure 21 shows the numbers of productions in the original grammar that are

mapped to the quotient of the *basic sum* and the *incremental sum*. This quotient is the factor by which the incremental algorithm is faster than the basic algorithm and is obviously growing with the amount of added productions. Although the factor seems to grow stronger than linear it might vary with the order in which the productions are added and also might change with the type of the productions.

A suggested improvement of the incremental algorithm was to allow adding multiple productions at once. There was made a test for the grammar of Example 2.1 as for the grammar above, but with adding the entire grammar at once in the first step and then inserting three additional productions one by one again. When looking at the statistics for this grammar which is shown in Table 5 we find the same characteristics as for the grammar tested above. Direct left recursions are never removed twice by the incremental algorithm. Also adding GNF productions as in step $|P| = 9$ that are rarely used in replacement operations can be inserted with almost zero cost.

$\|P\|$	new productions	$\|P'\|$	replacements by left derivations	direct left recursions	intermediate productions
7	Example 2.1	15	6 / 5	2 / 2	16 / 9
8	$A_2 \to A_1$	51	11 / 5	5 / 3	59 / 26
9	$A_4 \to aa$	54	11 / 1	5 / 0	61 / 1
10	$A_1 \to bb$	95	11 / 6	5 / 0	101 / 30

Table 5: Statistics for establishing the GNF for Example 2.1.

Table 6 shows the runtimes for this second grammar that were determined in the same way as those of the first grammar. When adding the seven productions of Example 2.1 in one step it is noticable that the incremental algorithm has again slightly the edge over the basic algorithm. This holds for each of the four steps that were tested.

$\|P\|$	new productions	basic algorithm	incremental algorithm	basic sum	incremental sum
7	Example 2.1	0.411	0.389	0.411	0.389
8	$A_2 \to A_1$	1.092	0.316	1.503	0.705
9	$A_4 \to aa$	0.981	0.063	2.484	0.768
10	$A_1 \to bb$	1.423	0.295	3.907	1.063

Table 6: Runtimes for establishing the GNF for Example 2.1.

The factor obtained by dividing the *basic algorithm* by the *incremental sum* is comparatively small and is at most 1.5 for all steps This time however in favour of the incremental algorithm. This still corresponds well to the factors acquired for the first grammar.

Also the factors of the total runtime are similar to those for the first grammar regarding the size of the total amount of productions of the original grammar. A comparison of the factors for the steps tested for both grammars is shown in Table 7 where in columns two and three the factor for a single step in the basic algorithm is compared to the sum of the steps for the incremental version while in columns four and five the sums for both algorithms are compared.

Both tests suggest an at least linear increase of the runtime factor compared to the

$\|P\|$	step / sum		sum / sum	
	1st	2nd	1st	2nd
7	1.2	1.1	1.2	1.1
8	0.8	1.5	2.6	2.1
9	0.7	1.3	3.1	3.2
10	0.8	1.3	3.5	3.7

Table 7: Comparing the factors for both tested grammars.

basic approach. The measured values of the first test shown in Figure 21 suggest a growth of this factor that is better than linear.

6 Conclusion

6.1 Summary

In this thesis an incremental algorithm for constructing the Greibach Normal Form (GNF) has been developed. It allows to construct a grammar G' in GNF from a productive, context-free grammar G such that the information of how the GNF was established is preserved. This allows the algorithm to add new productions to the original grammar G and using this preserved information to perform only those operations that are necessary to extend the resulting GNF grammar. While rebuilding the GNF the algorithm updates this information at the same time such that further productions can be added in this manner.

There are two main benefits of this incremental approach. The first is an increase in speed since the algorithm does not need to construct the GNF from scratch again, it rather extends the previous result by new productions, i.e. it never creates the same production twice.

The second benefit of this approach is that all productions of a previous GNF result G' are also productions of the GNF result G'_P that was constructed when adding a set of productions P incrementally. This follows from the fixed order of the nonterminals.

Furthermore the complexity of the incremental algorithm was analyzed. While best case runtime improves from linear with the number of productions $O(|P|)$ to constant $O(1)$ the worst case runtime does not change. In practice, however, the overall runtime improves considerably as seen in Section 5 when testing an implementation for different input grammars. The factor of the runtime when comparing the basic to the incremental approach grows at least linear with the number of productions in the grammar in favor of the incremental approach.

The performance tests, however, were not makig use of the optimizations developed in this thesis. Those include adding multiple productions at once and omitting case distinctions between direct left recursive nodes and non-ordered nodes. Improvements regarding the ordering of the worklists have been developed such as the final lists provably needs no ordering at all. The lookup of productions in the incremental algorithm can be improved from $O(|V| + |E|)$ per query to $O(|V|)$ per lifetime of the forest.

All in all the incremental algorithm for constructing the GNF appears to be a useful instrument which bears advantages that might prove valueable for certain fields of todays computer science.

6.2 Future Work

A field that has been omitted in this work is the impact of the order on the nonterminals. The incremental algorithm has to maintain the order of nonterminals, once this has been chosen, to operate correctly. The heuristic to insert new nonterminals as least symbol is a good choice if the algorithm does not know what productions are added in the future. There is, however, room for improvements when adding multiple productions especially when such a set of new productions contains more than one new nonterminal or more than one production containing new nonterminals including the right hand side. It might be a good idea to avoid forward pointers since ordering can result in multiple ordering steps and also in left recursions which usually generate more productions. There is potential to noticeably improve the runtime by choosing different orders of the nonterminals especially when the productions rules to be added are known before.

In the future it is interesting to adapt the incremental algorithm to be applicable to hyperedge replacement grammars (HRG). For those a GNF has been defined by Engelfriet, Heyker, Leih [2] which derives graphs from the outside to the inside similar to the Double Greibach Normal Form for string grammars which creates terminal symbols both from the left and right. A definition which is closer related to the GNF used in this thesis is the Local Greibach Normal Form that was defined by Jansen, Heinen, Noll, Katoen [4]. There, terminal symbols are only required to be created at the left side, so transfering the incremental algorithm to it is supposed to be easier.

The algorithm to obtain the LGNF is carried out with significant correspondence to the basic algorithm investigated in this thesis. Therefore the incremental algorithm presented here is supposed to be transferable one-by-one.

References

[1] A. Asteroth and C. Baier. *Theoretische Informatik*. Pearson, 2003.

[2] J. Engelfriet, L. Heyker, and G. Leih. Context-free graph languages of bounded degree are generated by apex graph grammars. *Acta Informatica*, 31:341–378, 1994.

[3] S. A. Greibach. A new normal-form theorem for context-free phrase structure grammars. *J. ACM*, 12:42–52, January 1965.

[4] C. Jansen, J. Heinen, J.-P. Katoen, and T. Noll. A local greibach normal form for hyperedge replacement grammars. Technical Report AIB-2011-04, RWTH Aachen, Jan. 2011.

[5] I. Wegener. *Theoretische Informatik*. Teubner, 2005.

[6] D. Wood. *Theory of Computation*. Harper and Row, 1987.