# Greibach Normal Form

Alexander Haberl and Tobias Nipkow and Akihisa Yamada

September 12, 2025

**Abstract**

This theory formalizes Hopcroft and Ullman's algorithm [3] to transform a set of productions into Greibach Normal Form (GNF) [2]. We concentrate on the essential property of the GNF: every production starts with a terminal; the tail of a rhs may contain further terminals. The complexity of the algorithm can be exponential.

**theory** *Greibach_Normal_Form*
**imports**
*Context_Free_Grammar.Context_Free_Grammar*
*Fresh_Identifiers.Fresh_Nat*
**begin**

**declare** *relpowp.simps*(*2*)[*simp del*]

## 1 Aux Lemmas

**lemma** *Nts_mono*: $G \subseteq H \Longrightarrow Nts\ G \subseteq Nts\ H$
⟨*proof*⟩

**lemma** *derivern_prepend*: $R \vdash u \Rightarrow r(n)\ v \Longrightarrow R \vdash p\ @\ u \Rightarrow r(n)\ p\ @\ v$
⟨*proof*⟩

**lemma** *Lang_subset_if_Ders_subset*: $Ders\ R\ A \subseteq Ders\ R'\ A \Longrightarrow Lang\ R\ A \subseteq Lang\ R'\ A$
⟨*proof*⟩

**lemma** *Eps_free_deriven_Nil*:
⟦ *Eps_free R*; $R \vdash l \Rightarrow (n)\ []$ ⟧ $\Longrightarrow l = []$
⟨*proof*⟩

**lemma** *nts_syms_empty_iff*: $nts\_syms\ w = \{\} \longleftrightarrow (\exists u.\ w = map\ Tm\ u)$
⟨*proof*⟩

**lemma** *non_word_has_last_Nt*: *nts_syms w* ≠ {} ⟹ ∃ *u A v*. *w* = *u* @ [*Nt A*] @ *map Tm v*
⟨*proof*⟩

**lemma** *nts_syms_rev*: *nts_syms* (*rev w*) = *nts_syms w*
⟨*proof*⟩

Sentential form that is not a word has a first *Nt*.

**lemma** *non_word_has_first_Nt*: *nts_syms w* ≠ {} ⟹ ∃ *u A v*. *w* = *map Tm u* @ *Nt A* # *v*
⟨*proof*⟩

If there exists a derivation from $u$ to $v$ then there exists one which does not use productions of the form $A \to A$.

**lemma** *no_self_loops_derivels*: *P* ⊢ *u* ⇒*l*(*n*) *v* ⟹ {*p*∈*P*. ¬(∃ *A*. *p* = (*A*,[*Nt A*]))} ⊢ *u* ⇒*l*∗ *v*
⟨*proof*⟩

A decomposition of a derivation from a sentential form to a word into multiple derivations that derive words.

**lemma** *derivern_snoc_Nt_Tms_decomp1*:
*R* ⊢ *p* @ [*Nt A*] ⇒*r*(*n*) *map Tm q*
⟹ ∃ *pt At w k m*. *R* ⊢ *p* ⇒(*k*) *map Tm pt* ∧ *R* ⊢ *w* ⇒(*m*) *map Tm At* ∧ (*A*, *w*) ∈ *R*
∧ *q* = *pt* @ *At* ∧ *n* = *Suc*(*k* + *m*)
⟨*proof*⟩

A decomposition of a derivation from a sentential form to a word into multiple derivations that derive words.

**lemma** *word_decomp1*:
*R* ⊢ *p* @ [*Nt A*] @ *map Tm ts* ⇒(*n*) *map Tm q*
⟹ ∃ *pt At w k m*. *R* ⊢ *p* ⇒(*k*) *map Tm pt* ∧ *R* ⊢ *w* ⇒(*m*) *map Tm At* ∧ (*A*, *w*) ∈ *R*
∧ *q* = *pt* @ *At* @ *ts* ∧ *n* = *Suc*(*k* + *m*)
⟨*proof*⟩

Sentential form that derives to terminals and has a Nt in it has a derivation that starts with some rule acting on that Nt.

**lemma** *deriven_start_sent*:
*R* ⊢ *u* @ *Nt V* # *w* ⇒(*Suc n*) *map Tm x* ⟹ ∃ *v*. (*V*, *v*) ∈ *R* ∧ *R* ⊢ *u* @ *v* @ *w* ⇒(*n*) *map Tm x*
⟨*proof*⟩

**definition** *nts_syms_list* :: ('*n*,'*t*)*syms* ⇒ '*n list* ⇒ '*n list* **where**
*nts_syms_list sys* = *foldr* (λ*sy ns*. *case sy of Nt A* ⇒ *List.insert A ns* | *Tm* _ ⇒ *ns*) *sys*

**definition** *nts_prods_list* :: (*'n*,*'t*)*prods* ⇒ *'n list* **where**
*nts_prods_list ps* = *foldr* (λ(*A*,*sys*) *ns*. *List.insert A* (*nts_syms_list sys ns*)) *ps* []

**lemma** *set_nts_syms_list*: *set*(*nts_syms_list sys ns*) = *nts_syms sys* ∪ *set ns*
⟨*proof*⟩

**lemma** *set_nts_prods_list*: *set*(*nts_prods_list ps*) = *nts ps*
⟨*proof*⟩

**lemma** *distinct_nts_syms_list*: *distinct*(*nts_syms_list sys ns*) = *distinct ns*
⟨*proof*⟩

**lemma** *distinct_nts_prods_list*: *distinct*(*nts_prods_list ps*)
⟨*proof*⟩

**fun** *freshs* :: (*'a*::*fresh*) *set* ⇒ *'a list* ⇒ *'a list* **where**
*freshs X* [] = [] |
*freshs X* (*a*#*as*) = (*let a'* = *fresh X a in a'* # *freshs* (*insert a' X*) *as*)

**lemma** *length_freshs*: *finite X* ⟹ *length*(*freshs X as*) = *length as*
⟨*proof*⟩

**lemma** *freshs_disj*: *finite X* ⟹ *X* ∩ *set*(*freshs X as*) = {}
⟨*proof*⟩

**lemma** *freshs_distinct*: *finite X* ⟹ *distinct* (*freshs X as*)
⟨*proof*⟩

This theory formalizes a method to transform a set of productions into Greibach Normal Form (GNF) [2]. We concentrate on the essential property of the GNF: every production starts with a *Tm*; the tail of a rhs can contain further terminals. This is formalized as *GNF_hd* below. This more liberal definition of GNF is also found elsewhere [1].

The algorithm consists of two phases:

- *solve_tri* converts the productions into a *triangular* form, where Nt *Ai* does not depend on Nts *Ai*, ..., *An*. This involves the elimination of left-recursion and is the heart of the algorithm.
- *expand_tri* expands the triangular form by substituting in: Due to triangular form, *A0* productions satisfy *GNF_hd* and we can substitute them into the heads of the remaining productions. Now all *A1* productions satisfy *GNF_hd*, and we continue until all productions satisfy *GNF_hd*.

This is essentially the algorithm given by Hopcroft and Ullman [3], except that we can drop the conversion to Chomsky Normal Form because of our more liberal *GNF_hd*.

## 2 Function Definitions

Depend on: $A$ depends on $B$ if there is a rule $A \to B\ w$:

**definition** *dep_on* :: ('n,'t) *Prods* ⇒ 'n ⇒ 'n *set* **where**
*dep_on R A* = {*B*. ∃ *w*. (*A*,*Nt B* # *w*) ∈ *R*}

GNF property: All productions start with a terminal.

**definition** *GNF_hd* :: ('n,'t)*Prods* ⇒ *bool* **where**
*GNF_hd R* = (∀ (*A*, *w*) ∈ *R*. ∃ *t*. *hd w* = *Tm t*)

GNF property expressed via *dep_on*:

**definition** *GNF_hd_dep_on* :: ('n,'t)*Prods* ⇒ *bool* **where**
*GNF_hd_dep_on R* = (∀ *A* ∈ *Nts R*. *dep_on R A* = {})

**abbreviation** *lrec_Prods* :: ('n,'t)*Prods* ⇒ 'n ⇒ 'n *set* ⇒ ('n,'t)*Prods* **where**
*lrec_Prods R A S* ≡ {(*A'*,*Bw*) ∈ *R*. *A'*=*A* ∧ (∃ *w B*. *Bw* = *Nt B* # *w* ∧ *B* ∈ *S*)}

**abbreviation** *subst_hd* :: ('n,'t)*Prods* ⇒ ('n,'t)*Prods* ⇒ 'n ⇒ ('n,'t)*Prods* **where**
*subst_hd R X A* ≡ {(*A*,*v*@*w*) |*v w*. ∃ *B*. (*A*,*Nt B* # *w*) ∈ *X* ∧ (*B*,*v*) ∈ *R*}

Expand head: Replace all rules $A \to B\ w$ where $B \in Ss$ ($Ss$ = solved Nts in *triangular* form) by $A \to v\ w$ where $B \to v$. Starting from the end of $Ss$.

**fun** *expand_hd* :: 'n ⇒ 'n *list* ⇒ ('n,'t)*Prods* ⇒ ('n,'t)*Prods* **where**
*expand_hd A* [] *R* = *R* |
*expand_hd A* (*S*#*Ss*) *R* =
(*let R'* = *expand_hd A Ss R*;
*X* = *lrec_Prods R' A* {*S*};
*Y* = *subst_hd R' X A*
*in R'* − *X* ∪ *Y*)

**lemma** *Rhss_code*[*code*]: *Rhss P A* = *snd* ‘ {*Aw* ∈ *P*. *fst Aw* = *A*}
⟨*proof*⟩

**declare** *expand_hd.simps*(*1*)[*code*]
**lemma** *expand_hd_Cons_code*[*code*]: *expand_hd A* (*S*#*Ss*) *R* =
(*let R'* = *expand_hd A Ss R*;
*X* = {*w* ∈ *Rhss R' A*. *w* ≠ [] ∧ *hd w* = *Nt S*};
*Y* = (⋃(*B*,*v*) ∈ *R'*. ⋃ *w* ∈ *X*. *if hd w* ≠ *Nt B then* {} *else* {(*A*,*v* @ *tl w*)})
*in R'* − ({*A*} × *X*) ∪ *Y*)
⟨*proof*⟩

Remove left-recursions: Remove left-recursive rules $A \to A\ w$:

**definition** *rm_lrec* :: $'n \Rightarrow ('n,'t)Prods \Rightarrow ('n,'t)Prods$ **where**
*rm_lrec A R* = *R* − {(*A*,*Nt A* # *v*)|*v*. *True*}

**lemma** *rm_lrec_code*[*code*]:
*rm_lrec A R* = {*Aw* ∈ *R*. *let* (*A*′,*w*) = *Aw in A*′ ≠ *A* ∨ *w* = [] ∨ *hd w* ≠ *Nt A*}
⟨*proof*⟩

Make right-recursion of left-recursion: Conversion from left-recursion to right-recursion: Split $A$-rules into $A \to u$ and $A \to A\ v$. Keep $A \to u$ but replace $A \to A\ v$ by $A \to u\ A'$, $A' \to v$, $A' \to v\ A'$.

The then part of the if statement is only an optimisation, so that we do not introduce the $A \to u\ A'$ rules if we do not introduce any $A'$ rules, but the function also works, if we always enter the else part.

**definition** *rrec_of_lrec* :: $'n \Rightarrow 'n \Rightarrow ('n,'t)Prods \Rightarrow ('n,'t)Prods$ **where**
*rrec_of_lrec A A*′ *R* =
(*let V* = {*v*. (*A*,*Nt A* # *v*) ∈ *R* ∧ *v* ≠ []};
*U* = {*u*. (*A*,*u*) ∈ *R* ∧ ¬(∃ *v*. *u* = *Nt A* # *v*) }
*in if V* = {} *then R* − {(*A*, [*Nt A*])} *else* ({*A*} × *U*) ∪ (⋃ *u*∈*U*. {(*A*,*u*@[*Nt A*′])}) ∪ ({*A*′} × *V*) ∪ (⋃ *v*∈*V*. {(*A*′,*v* @ [*Nt A*′])}))

**lemma** *rrec_of_lrec_code*[*code*]: *rrec_of_lrec A A*′ *R* =
(*let RA* = *Rhss R A*;
*V* = *tl* ‘ {*w* ∈ *RA*. *w* ≠ [] ∧ *hd w* = *Nt A* ∧ *tl w* ≠ []};
*U* = {*u* ∈ *RA*. *u* = [] ∨ *hd u* ≠ *Nt A* }
*in if V* = {} *then R* − {(*A*, [*Nt A*])} *else* ({*A*} × *U*) ∪ (⋃ *u*∈*U*. {(*A*,*u*@[*Nt A*′])}) ∪ ({*A*′} × *V*) ∪ (⋃ *v*∈*V*. {(*A*′,*v* @ [*Nt A*′])}))
⟨*proof*⟩

Solve left-recursions: Solves the left-recursion of Nt $A$ by replacing it with a right-recursion on a fresh Nt $A'$. The fresh Nt $A'$ is also given as a parameter.

**definition** *solve_lrec* :: $'n \Rightarrow 'n \Rightarrow ('n,'t)Prods \Rightarrow ('n,'t)Prods$ **where**
*solve_lrec A A*′ *R* = *rm_lrec A R* ∪ *rrec_of_lrec A A*′ *R*

**lemmas** *solve_lrec_defs* = *solve_lrec_def rm_lrec_def rrec_of_lrec_def Let_def Nts_def*

Solve triangular: Put $R$ into triangular form wrt $As$ (using the new Nts $As'$). In each step $A\#As$, first the remaining Nts in $As$ are solved, then $A$ is solved. This should mean that in the result of the outermost *expand_hd A As*, $A$ only depends on $A$. Then the $A$ rules in the result of *solve_lrec A A*′ are already in GNF. More precisely: the result should be in *triangular* form.

**fun** *solve_tri* :: *'a list* ⇒ *'a list* ⇒ (*'a*, *'b*) *Prods* ⇒ (*'a*, *'b*) *Prods* **where**
*solve_tri* [] _ *R* = *R* |
*solve_tri* (*A*#*As*) (*A*′#*As*′) *R* = *solve_lrec A A*′ (*expand_hd A As* (*solve_tri As As*′ *R*))

Triangular form wrt [*A1*,...,*An*] means that *Ai* must not depend on *Ai*, ..., *An*. In particular: *A0* does not depend on any *Ai*, its rules are already

in GNF. Therefore one can convert a *triangular* form into GNF by backwards substitution: The rules for $Ai$ are used to expand the heads of all $A(i+1)$,...,$An$ rules, starting with $A0$.

**fun** *triangular* :: *'n list* ⇒ (*'n* × (*'n*, *'t*) *sym list*) *set* ⇒ *bool* **where**
*triangular* [] *R* = *True* |
*triangular* (*A*#*As*) *R* = (*dep_on R A* ∩ ({*A*} ∪ *set As*) = {} ∧ *triangular As R*)

Remove self loops: Removes all productions of the form $A \to A$.

**definition** *rm_self_loops* :: ('*n*,'*t*) *Prods* ⇒ ('*n*,'*t*) *Prods* **where**
*rm_self_loops P* = *P* − {*x*∈*P*. ∃ *A*. *x* = (*A*, [*Nt A*])}

Expand triangular: Expands all head-Nts of productions with a Lhs in *As* (*triangular* (*rev As*)). In each step *A*#*As* first all Nts in *As* are expanded, then every rule $A \to B\ w$ is expanded if $B \in$ *set As*. If the productions were in *triangular* form wrt *rev As* then $Ai$ only depends on $A(i+1)$, ..., $An$ which have already been expanded in the first part of the step and are in GNF. Then the all $A$-productions are also is in GNF after expansion.

**fun** *expand_tri* :: *'n list* ⇒ ('*n*,'*t*)*Prods* ⇒ ('*n*,'*t*)*Prods* **where**
*expand_tri* [] *R* = *R* |
*expand_tri* (*A*#*As*) *R* =
(*let R'* = *expand_tri As R*;
*X* = *lrec_Prods R' A* (*set As*);
*Y* = *subst_hd R' X A*
*in R'* − *X* ∪ *Y*)

**declare** *expand_tri.simps*(*1*)[*code*]
**lemma** *expand_tri_Cons_code*[*code*]: *expand_tri* (*S*#*Ss*) *R* =
(*let R'* = *expand_tri Ss R*;
*X* = {*w* ∈ *Rhss R' S*. *w* ≠ [] ∧ *hd w* ∈ *Nt* ' (*set Ss*)};
*Y* = (⋃(*B*,*v*) ∈ *R'*. ⋃*w* ∈ *X*. *if hd w* ≠ *Nt B then* {} *else* {(*S*,*v* @ *tl w*)})
*in R'* − ({*S*} × *X*) ∪ *Y*)
⟨*proof*⟩

The main function *gnf_hd* converts into *GNF_hd*:

**definition** *gnf_hd* :: ('*n*::*fresh*,'*t*)*prods* ⇒ ('*n*,'*t*)*Prods* **where**
*gnf_hd ps* =
(*let As* = *nts_prods_list ps*;
*As'* = *freshs* (*set As*) *As*
*in expand_tri* (*As'* @ *rev As*) (*solve_tri As As'* (*set ps*)))

# 3 Some Basic Lemmas

## 3.1 *Eps_free* preservation

**lemma** *Eps_free_expand_hd*: *Eps_free R* ⟹ *Eps_free* (*expand_hd A Ss R*)
⟨*proof*⟩

**lemma** *Eps_free_solve_lrec*: *Eps_free R* $\Longrightarrow$ *Eps_free* (*solve_lrec A A′ R*)
⟨*proof*⟩

**lemma** *Eps_free_solve_tri*: *Eps_free R* $\Longrightarrow$ *length As* $\leq$ *length As′* $\Longrightarrow$ *Eps_free* (*solve_tri As As′ R*)
⟨*proof*⟩

**lemma** *Eps_free_expand_tri*: *Eps_free R* $\Longrightarrow$ *Eps_free* (*expand_tri As R*)
⟨*proof*⟩

## 3.2 Lemmas about *Nts* and *dep_on*

**lemma** *dep_on_Un*[*simp*]: *dep_on* (*R* $\cup$ *S*) *A* = *dep_on R A* $\cup$ *dep_on S A*
⟨*proof*⟩

**lemma** *expand_hd_preserves_neq*: *B* $\neq$ *A* $\Longrightarrow$ (*B*,*w*) $\in$ *expand_hd A Ss R* $\longleftrightarrow$ (*B*,*w*) $\in$ *R*
⟨*proof*⟩

Let *R* be epsilon-free and in *triangular* form wrt *Bs*. After *expand_hd A Bs R*, *A* depends only on what *A* depended on before or what one of the *B* $\in$ *Bs* depends on, but *A* does not depend on the *Bs*:

**lemma** *dep_on_expand_hd*:
⟦ *Eps_free R*; *triangular Bs R*; *distinct Bs*; *A* $\notin$ *set Bs* ⟧
$\Longrightarrow$ *dep_on* (*expand_hd A Bs R*) *A* $\subseteq$ (*dep_on R A* $\cup$ ($\bigcup$ *B*$\in$*set Bs*. *dep_on R B*)) $-$ *set Bs*
⟨*proof*⟩

**lemma** *dep_on_subs_Nts*: *dep_on R A* $\subseteq$ *Nts R*
⟨*proof*⟩

**lemma** *Nts_expand_hd_sub*: *Nts* (*expand_hd A As R*) $\subseteq$ *Nts R*
⟨*proof*⟩

**lemma** *Nts_solve_lrec_sub*: *Nts* (*solve_lrec A A′ R*) $\subseteq$ *Nts R* $\cup$ {*A′*}
⟨*proof*⟩

**lemma** *Nts_solve_tri_sub*: *length As* $\leq$ *length As′* $\Longrightarrow$ *Nts* (*solve_tri As As′ R*) $\subseteq$ *Nts R* $\cup$ *set As′*
⟨*proof*⟩

## 3.3 Lemmas about *triangular*

**lemma** *tri_Snoc_impl_tri*: *triangular* (*As* @ [*A*]) *R* $\Longrightarrow$ *triangular As R*
⟨*proof*⟩

If two parts of the productions are *triangular* and no Nts from the first part depend on ones of the second they are also *triangular* when put together.

**lemma** *triangular_append*:

⟦*triangular As R*; *triangular Bs R*; ∀ *A*∈*set As*. *dep_on R A* ∩ *set Bs* = {}⟧
⟹ *triangular* (*As*@*Bs*) *R*
⟨*proof*⟩

# 4 Function *solve_tri*: Remove Left-Recursion and Convert into Triangular Form

## 4.1 Basic Lemmas

Mostly about rule inclusions in *solve_lrec*.

**lemma** *solve_lrec_rule_simp1*: *A* ≠ *B* ⟹ *A* ≠ *B'* ⟹ (*A*, *w*) ∈ *solve_lrec B B' R* ⟷ (*A*, *w*) ∈ *R*
⟨*proof*⟩

**lemma** *solve_lrec_rule_simp3*: *A* ≠ *A'* ⟹ *A'* ∉ *Nts R* ⟹ *Eps_free R*
⟹ (*A*, [*Nt A'*]) ∉ *solve_lrec A A' R*
⟨*proof*⟩

**lemma** *solve_lrec_rule_simp7*: *A'* ≠ *A* ⟹ *A'* ∉ *Nts R* ⟹ (*A'*, *Nt A'* # *w*) ∉ *solve_lrec A A' R*
⟨*proof*⟩

**lemma** *solve_lrec_rule_simp8*: *A'* ∉ *Nts R* ⟹ *B* ≠ *A'* ⟹ *B* ≠ *A*
⟹ (*B*, *Nt A'* # *w*) ∉ *solve_lrec A A' R*
⟨*proof*⟩

**lemma** *dep_on_expand_hd_simp2*: *B* ≠ *A* ⟹ *dep_on* (*expand_hd A As R*) *B* = *dep_on R B*
⟨*proof*⟩

**lemma** *dep_on_solve_lrec_simp2*: *A* ≠ *B* ⟹ *A'* ≠ *B* ⟹ *dep_on* (*solve_lrec A A' R*) *B* = *dep_on R B*
⟨*proof*⟩

## 4.2 Triangular Form

*expand_hd* preserves *triangular*, if it does not expand a Nt considered in *triangular*.

**lemma** *triangular_expand_hd*: ⟦*A* ∉ *set As*; *triangular As R*⟧ ⟹ *triangular As* (*expand_hd A Bs R*)
⟨*proof*⟩

Solving a Nt not considered by *triangular* preserves the *triangular* property.

**lemma** *triangular_solve_lrec*: ⟦*A* ∉ *set As*; *A'* ∉ *set As*; *triangular As R*⟧
⟹ *triangular As* (*solve_lrec A A' R*)
⟨*proof*⟩

Solving more Nts does not remove the *triangular* property of previously solved Nts.

**lemma** *part_triangular_induct_step*:
⟦*Eps_free R*; *distinct* ((*A*#*As*)@(*A'*#*As'*)); *triangular As* (*solve_tri As As' R*)⟧
⟹ *triangular As* (*solve_tri* (*A*#*As*) (*A'*#*As'*) *R*)
⟨*proof*⟩

Couple of small lemmas about *dep_on* and the solving of left-recursion.

**lemma** *rm_lrec_rem_own_dep*: *A* ∉ *dep_on* (*rm_lrec A R*) *A*
⟨*proof*⟩

**lemma** *rrec_of_lrec_has_no_own_dep*: *A* ≠ *A'* ⟹ *A* ∉ *dep_on* (*rrec_of_lrec A A' R*) *A*
⟨*proof*⟩

**lemma** *solve_lrec_no_own_dep*: *A* ≠ *A'* ⟹ *A* ∉ *dep_on* (*solve_lrec A A' R*) *A*
⟨*proof*⟩

**lemma** *solve_lrec_no_new_own_dep*: *A* ≠ *A'* ⟹ *A'* ∉ *Nts R* ⟹ *A'* ∉ *dep_on* (*solve_lrec A A' R*) *A'*
⟨*proof*⟩

**lemma** *dep_on_rem_lrec_simp*: *dep_on* (*rm_lrec A R*) *A* = *dep_on R A* − {*A*}
⟨*proof*⟩

**lemma** *dep_on_rrec_of_lrec_simp*:
*Eps_free R* ⟹ *A* ≠ *A'* ⟹ *dep_on* (*rrec_of_lrec A A' R*) *A* = *dep_on R A* − {*A*}
⟨*proof*⟩

**lemma** *dep_on_solve_lrec_simp*:
⟦*Eps_free R*; *A* ≠ *A'*⟧ ⟹ *dep_on* (*solve_lrec A A' R*) *A* = *dep_on R A* − {*A*}
⟨*proof*⟩

**lemma** *dep_on_solve_tri_simp*: *B* ∉ *set As* ⟹ *B* ∉ *set As'* ⟹ *length As* ≤ *length As'*
⟹ *dep_on* (*solve_tri As As' R*) *B* = *dep_on R B*
⟨*proof*⟩

Induction step for showing that *solve_tri* removes dependencies of previously solved Nts.

**lemma** *triangular_dep_on_induct_step*:
**assumes** *Eps_free R length As* ≤ *length As' distinct* ((*A*#*As*)@*A'*#*As'*) *triangular As* (*solve_tri As As' R*)
**shows** *dep_on* (*solve_tri* (*A* # *As*) (*A'* # *As'*) *R*) *A* ∩ ({*A*} ∪ *set As*) = {}
⟨*proof*⟩

**theorem** *triangular_solve_tri*: ⟦ *Eps_free R*; *length As* ≤ *length As'*; *distinct*(*As* @ *As'*)⟧

$\Longrightarrow$ *triangular As* (*solve_tri As As′ R*)
⟨*proof*⟩

**lemma** *dep_on_solve_tri_Nts_R*:
⟦*Eps_free R*; *B* ∈ *set As*; *distinct* (*As* @ *As′*); *set As′* ∩ *Nts R* = {}; *length As* ≤ *length As′*⟧
$\Longrightarrow$ *dep_on* (*solve_tri As As′ R*) *B* ⊆ *Nts R*
⟨*proof*⟩

**lemma** *triangular_unused_Nts*: *set As* ∩ *Nts R* = {} $\Longrightarrow$ *triangular As R*
⟨*proof*⟩

The newly added Nts in *solve_lrec* are in *triangular* form wrt *rev As′*.

**lemma** *triangular_rev_As′_solve_tri*:
⟦*set As′* ∩ *Nts R* = {}; *distinct* (*As* @ *As′*); *length As* ≤ *length As′*⟧
$\Longrightarrow$ *triangular* (*rev As′*) (*solve_tri As As′ R*)
⟨*proof*⟩

The entire set of productions is in *triangular* form after *solve_tri* wrt *As*@(*rev As′*).

**theorem** *triangular_As_As′_solve_tri*:
**assumes** *Eps_free R length As* ≤ *length As′ distinct*(*As* @ *As′*) *Nts R* ⊆ *set As*
**shows** *triangular* (*As*@(*rev As′*)) (*solve_tri As As′ R*)
⟨*proof*⟩

## 4.3 *solve_lrec* Preserves Language

### 4.3.1 *Lang R A* ⊆ *Lang* (*solve_lrec B B′ R*) *A*

If there exists a derivation from $u$ to $v$ then there exists one which does not use productions of the form $A \to A$.

**lemma** *rm_self_loops_derivels*: **assumes** $P \vdash u \Rightarrow l(n)\ v$ **shows** *rm_self_loops* $P \vdash u \Rightarrow l*\ v$
⟨*proof*⟩

Restricted to productions with one lhs ($A$), and no $A \to A$ productions if there is a derivation from $u$ to $A\ \#\ v$ then $u$ must start with Nt $A$.

**lemma** *lrec_lemma1*:
**assumes** $S = \{x.\ (\exists\, v.\ x = (A,\ v) \wedge x \in R)\}$ *rm_self_loops S* $\vdash u \Rightarrow l(n)$ *Nt A*#*v*
**shows** $\exists\, u'.\ u = Nt\ A\ \#\ u'$
⟨*proof*⟩

Restricted to productions with one lhs ($A$), and no $A \to A$ productions if there is a derivation from $u$ to $A\ \#\ v$ then $u$ must start with Nt $A$ and there exists a prefix of $A\ \#\ v$ s.t. a left-derivation from $[Nt\ A]$ to that prefix exists.

**lemma** *lrec_lemma2*:
**assumes** $S = \{x.\ (\exists\, v.\ x = (A,\ v) \wedge x \in R)\}$ *Eps_free R*
**shows** *rm_self_loops* $S \vdash u \Rightarrow l(n)$ *Nt A*$\#v \Longrightarrow$
$\exists\, u'\ v'.\ u = Nt\ A\ \#\ u' \wedge v = v' @ u' \wedge (rm\_self\_loops\ S) \vdash [Nt\ A] \Rightarrow l(n)\ Nt\ A\ \#\ v'$
⟨*proof*⟩

Restricted to productions with one lhs ($A$), and no $A \to A$ productions if there is a left-derivation from $[Nt\ A]$ to $A\ \#\ u$ then there exists a derivation from $[Nt\ A']$ to $u@[Nt\ A]$ and if $u \neq []$ also to $u$ in *solve_lrec A A' R*.

**lemma** *lrec_lemma3*:
**assumes** $S = \{x.\ (\exists\, v.\ x = (A,\ v) \wedge x \in R)\}$ *Eps_free R*
**shows** *rm_self_loops* $S \vdash [Nt\ A] \Rightarrow l(n)\ Nt\ A\ \#\ u$
$\Longrightarrow$ *solve_lrec A A' S* $\vdash [Nt\ A'] \Rightarrow (n)\ u @ [Nt\ A'] \wedge$
$(u \neq [] \longrightarrow$ *solve_lrec A A' S* $\vdash [Nt\ A'] \Rightarrow (n)\ u)$
⟨*proof*⟩

A left derivation from $p$ ($hd\ p = Nt\ A$) to $q$ ($hd\ q \neq Nt\ A$) can be split into a left-recursive part, only using left-recursive productions $A \to A\ \#\ w$, one left derivation step consuming Nt $A$ using some rule $A \to B\ \#\ v$ where $B \neq Nt\ A$ and a left-derivation comprising the rest of the derivation.

**lemma** *lrec_decomp*:
**assumes** $S = \{x.\ (\exists\, v.\ x = (A,\ v) \wedge x \in R)\}$ *Eps_free R*
**shows** $[\![\ hd\ p = Nt\ A;\ hd\ q \neq Nt\ A;\ R \vdash p \Rightarrow l(n)\ q\ ]\!]$
$\Longrightarrow \exists\, u\ w\ m\ k.\ S \vdash p \Rightarrow l(m)\ Nt\ A\ \#\ u \wedge S \vdash Nt\ A\ \#\ u \Rightarrow l\ w \wedge hd\ w \neq Nt\ A \wedge$
$R \vdash w \Rightarrow l(k)\ q \wedge n = m + k + 1$
⟨*proof*⟩

Every derivation resulting in a word has a derivation in *solve_lrec B B' R*.

**lemma** *tm_derive_impl_solve_lrec_derive*:
**assumes** *Eps_free R B* $\neq B'$ $B' \notin$ *Nts R*
**shows** $[\![\ p \neq []; R \vdash p \Rightarrow (n)\ map\ Tm\ q]\!] \Longrightarrow$ *solve_lrec B B' R* $\vdash p \Rightarrow *\ map\ Tm\ q$
⟨*proof*⟩

**corollary** *Lang_incl_Lang_solve_lrec*:
$[\![$ *Eps_free R*; $B \neq B'$; $B' \notin$ *Nts R*$]\!] \Longrightarrow$ *Lang R A* $\subseteq$ *Lang* (*solve_lrec B B' R*) *A*
⟨*proof*⟩

### 4.3.2 *Lang* (*solve_lrec B B' R*) *A* $\subseteq$ *Lang R A*

Restricted to right-recursive productions of one Nt ($A' \to w\ @\ [Nt\ A']$) if there is a right-derivation from $u$ to $v\ @\ [Nt\ A']$ then u ends in Nt $A'$.

**lemma** *rrec_lemma1*:
**assumes** $S = \{x.\ \exists\, v.\ x = (A',\ v\ @\ [Nt\ A'])\ \wedge\ x \in solve\_lrec\ A\ A'\ R\}$ $S \vdash u \Rightarrow r(n)\ v\ @\ [Nt\ A']$

**shows** $\exists\, u'.\ u = u' @ [Nt\ A']$
⟨*proof*⟩

*solve_lrec* does not add productions of the form $A' \to Nt\ A'$.

**lemma** *solve_lrec_no_self_loop*: *Eps_free R* $\Longrightarrow$ $A' \notin Nts\ R \Longrightarrow (A', [Nt\ A']) \notin$ *solve_lrec A A' R*
⟨*proof*⟩

Restricted to right-recursive productions of one Nt ($A' \to w @ [Nt\ A']$) if there is a right-derivation from $u$ to $v @ [Nt\ A']$ then u ends in Nt $A'$ and there exists a suffix of $v @ [Nt\ A']$ s.t. there is a right-derivation from $[Nt\ A']$ to that suffix.

**lemma** *rrec_lemma2*:
**assumes** $S = \{x.\ (\exists\, v.\ x = (A', v @ [Nt\ A']) \wedge x \in$ *solve_lrec A A' R*$)\}$ *Eps_free R* $A' \notin Nts\ R$
**shows** $S \vdash u \Rightarrow r(n)\ v @ [Nt\ A']$
$\Longrightarrow \exists\, u'\ v'.\ u = u' @ [Nt\ A'] \wedge v = u' @ v' \wedge S \vdash [Nt\ A'] \Rightarrow r(n)\ v' @ [Nt\ A']$
⟨*proof*⟩

Restricted to right-recursive productions of one Nt ($A' \to w @ [Nt\ A']$) if there is a restricted right-derivation in *solve_lrec* from $[Nt\ A']$ to $u @ [Nt\ A']$ then there exists a derivation in $R$ from $[Nt\ A]$ to $A \#\ u$.

**lemma** *rrec_lemma3*:
**assumes** $S = \{x.\ (\exists\, v.\ x = (A', v @ [Nt\ A']) \wedge x \in$ *solve_lrec A A' R*$)\}$ *Eps_free R*
$A' \notin Nts\ R\ A \neq A'$
**shows** $S \vdash [Nt\ A'] \Rightarrow r(n)\ u @ [Nt\ A'] \Longrightarrow R \vdash [Nt\ A] \Rightarrow (n)\ Nt\ A \#\ u$
⟨*proof*⟩

A right derivation from $p @ [Nt\ A']$ to $q$ ($last\ q \neq Nt\ A'$) can be split into a right-recursive part, only using right-recursive productions with Nt $A'$, one right derivation step consuming Nt $A'$ using some rule $A' \to as @ [Nt\ B]$ where $Nt\ B \neq Nt\ A'$ and a right-derivation comprising the rest of the derivation.

**lemma** *rrec_decomp*:
**assumes** $S = \{x.\ (\exists\, v.\ x = (A', v @ [Nt\ A']) \wedge x \in$ *solve_lrec A A' R*$)\}$ *Eps_free R*
$A \neq A'\ A' \notin Nts\ R$
**shows** $[\![A' \notin$ *nts_syms p*; *last q* $\neq Nt\ A'$; *solve_lrec A A' R* $\vdash p @ [Nt\ A'] \Rightarrow r(n)\ q]\!]$
$\Longrightarrow \exists\, u\ w\ m\ k.\ S \vdash p @ [Nt\ A'] \Rightarrow r(m)\ u @ [Nt\ A']$
$\wedge$ *solve_lrec A A' R* $\vdash u @ [Nt\ A'] \Rightarrow r\ w \wedge A' \notin$ *nts_syms w*
$\wedge$ *solve_lrec A A' R* $\vdash w \Rightarrow r(k)\ q \wedge n = m + k + 1$
⟨*proof*⟩

Every word derived by *solve_lrec B B' R* can be derived by $R$.

**lemma** *tm_solve_lrec_derive_impl_derive*:

**assumes** *Eps_free R* $B \neq B'$ $B' \notin$ *Nts R*
**shows** $[\![ p \neq []; B' \notin$ *nts_syms p*; (*solve_lrec B B' R*) $\vdash p \Rightarrow(n)$ *map Tm q*$]\!] \Longrightarrow$ *R* $\vdash p \Rightarrow*$ *map Tm q*
⟨*proof*⟩

**corollary** *Lang_solve_lrec_incl_Lang*:
**assumes** *Eps_free R B* $\neq B'$ $B' \notin$ *Nts R A* $\neq B'$
**shows** *Lang* (*solve_lrec B B' R*) *A* $\subseteq$ *Lang R A*
⟨*proof*⟩

**corollary** *solve_lrec_Lang*:
$[\![$ *Eps_free R*; *B* $\neq B'$; $B' \notin$ *Nts R*; *A* $\neq B'$$]\!] \Longrightarrow$ *Lang* (*solve_lrec B B' R*) *A* = *Lang R A*
⟨*proof*⟩

## 4.4 *expand_hd* Preserves Language

Every rhs of an *expand_hd R* production is derivable by *R*.

**lemma** *expand_hd_is_deriveable*: (*A*, *w*) $\in$ *expand_hd B As R* $\Longrightarrow$ *R* $\vdash$ [*Nt A*] $\Rightarrow*$ *w*
⟨*proof*⟩

**lemma** *expand_hd_incl1*: *Lang* (*expand_hd B As R*) *A* $\subseteq$ *Lang R A*
⟨*proof*⟩

This lemma expects a set of quadruples (*A*, *a1*, *B*, *a2*). Each quadruple encodes a specific Nt in a specific rule $A \rightarrow$ *a1* @ *Nt B* # *a2* (this encodes Nt *B*) which should be expanded, by replacing the Nt with every rule for that Nt and then removing the original rule. This expansion contains the original productions Language.

**lemma** *exp_includes_Lang*:
**assumes** *S_props*: $\forall x \in S.\ \exists A\ a1\ B\ a2.\ x = (A,\ a1,\ B,\ a2) \wedge (A,\ a1$ @ *Nt B* # *a2*) $\in R$
**shows** *Lang R A*
$\subseteq$ *Lang* ($R - \{x.\ \exists A\ a1\ B\ a2.\ x = (A,\ a1$ @ *Nt B* # *a2*) $\wedge$ (*A*, *a1*, *B*, *a2*) $\in S\ \}$
$\cup \{x.\ \exists A\ v\ a1\ a2\ B.\ x = (A,a1@v@a2) \wedge (A,\ a1,\ B,\ a2) \in S \wedge (B,v) \in R\})$ *A*
⟨*proof*⟩

**lemma** *expand_hd_incl2*: *Lang* (*expand_hd B As R*) *A* $\supseteq$ *Lang R A*
⟨*proof*⟩

**theorem** *expand_hd_Lang*: *Lang* (*expand_hd B As R*) *A* = *Lang R A*
⟨*proof*⟩

### 4.5 *solve_tri* Preserves Language

**lemma** *solve_tri_Lang*:
⟦ *Eps_free R*; *length As* ≤ *length As'*; *distinct*(*As* @ *As'*); *Nts R* ∩ *set As'* = {}; *A* ∉ *set As'*⟧
⟹ *Lang* (*solve_tri As As' R*) *A* = *Lang R A*
⟨*proof*⟩

## 5 Function *expand_hd*: Convert Triangular Form into GNF

### 5.1 *expand_hd*: Result is in *GNF_hd*

**lemma** *dep_on_helper*: *dep_on R A* = {} ⟹ (*A*, *w*) ∈ *R* ⟹ *w* = [] ∨ (∃ *T wt*. *w* = *Tm T* # *wt*)
⟨*proof*⟩

**lemma** *GNF_hd_iff_dep_on*:
**assumes** *Eps_free R*
**shows** *GNF_hd R* ⟷ (∀ *A* ∈ *Nts R*. *dep_on R A* = {}) (**is** *?L*=*?R*)
⟨*proof*⟩

**lemma** *helper_expand_tri1*: *A* ∉ *set As* ⟹ (*A*, *w*) ∈ *expand_tri As R* ⟹ (*A*, *w*) ∈ *R*
⟨*proof*⟩

If none of the expanded Nts depend on *A* then any rule depending on *A* in *expand_tri As R* must already have been in *R*.

**lemma** *helper_expand_tri2*:
⟦*Eps_free R*; *A* ∉ *set As*; ∀ *C* ∈ *set As*. *A* ∉ (*dep_on R C*); *B* ≠ *A*; (*B*, *Nt A* # *w*) ∈ *expand_tri As R*⟧
⟹ (*B*, *Nt A* # *w*) ∈ *R*
⟨*proof*⟩

In a triangular form no Nts depend on the last Nt in the list.

**lemma** *triangular_snoc_dep_on*: *triangular* (*As*@[*A*]) *R* ⟹ ∀ *C* ∈ *set As*. *A* ∉ (*dep_on R C*)
⟨*proof*⟩

**lemma** *triangular_helper1*: *triangular As R* ⟹ *A* ∈ *set As* ⟹ *A* ∉ *dep_on R A*
⟨*proof*⟩

**lemma** *dep_on_expand_tri*:
⟦*Eps_free R*; *triangular* (*rev As*) *R*; *distinct As*; *A* ∈ *set As*⟧
⟹ *dep_on* (*expand_tri As R*) *A* ∩ *set As* = {}
⟨*proof*⟩

Interlude: *Nts* of *expand_tri*:

**lemma** *Lhss_expand_tri*: *Lhss* (*expand_tri As R*) ⊆ *Lhss R*
⟨*proof*⟩

**lemma** *Rhs_Nts_expand_tri*: *Rhs_Nts* (*expand_tri As R*) ⊆ *Rhs_Nts R*
⟨*proof*⟩

**lemma** *Nts_expand_tri*: *Nts* (*expand_tri As R*) ⊆ *Nts R*
⟨*proof*⟩

If the entire *triangular* form is expanded, the result is in GNF:

**theorem** *GNF_hd_expand_tri*:
**assumes** *Eps_free R triangular* (*rev As*) *R distinct As Nts R* ⊆ *set As*
**shows** *GNF_hd* (*expand_tri As R*)
⟨*proof*⟩

Any set of productions can be transformed into GNF via *expand_tri* (*solve_tri*).

**theorem** *GNF_of_R*:
**assumes** *assms*: *Eps_free R distinct* (*As* @ *As'*) *Nts R* ⊆ *set As length As* ≤ *length As'*
**shows** *GNF_hd* (*expand_tri* (*As'* @ *rev As*) (*solve_tri As As' R*))
⟨*proof*⟩

### 5.2 *expand_tri* Preserves Language

Similar to the proof of Language equivalence of *expand_hd*.

All productions in *expand_tri As R* are derivable by *R*.

**lemma** *expand_tri_prods_deirvable*: (*B*, *bs*) ∈ *expand_tri As R* ⟹ *R* ⊢ [*Nt B*] ⇒∗ *bs*
⟨*proof*⟩

Language Preservation:

**lemma** *expand_tri_Lang*: *Lang* (*expand_tri As R*) *A* = *Lang R A*
⟨*proof*⟩

## 6 Function *gnf_hd*: Conversion to *GNF_hd*

All epsilon-free grammars can be put into GNF while preserving their language.

Putting the productions into GNF via *expand_tri* (*solve_tri*) preserves the language.

**lemma** *GNF_of_R_Lang*:
**assumes** *Eps_free R length As* ≤ *length As' distinct* (*As* @ *As'*) *Nts R* ∩ *set As'* = {} *A* ∉ *set As'*
**shows** *Lang* (*expand_tri* (*As'* @ *rev As*) (*solve_tri As As' R*)) *A* = *Lang R A*
⟨*proof*⟩

Any epsilon-free Grammar can be brought into GNF.

**theorem** *GNF_hd_gnf_hd*: *eps_free ps* $\Longrightarrow$ *GNF_hd* (*gnf_hd ps*)
⟨*proof*⟩

**lemma** *distinct_app_freshs*: ⟦ *As* = *nts_prods_list ps*; *As'* = *freshs* (*set As*) *As* ⟧
$\Longrightarrow$
*distinct* (*As* @ *As'*)
⟨*proof*⟩

*gnf_hd* preserves the language:

**theorem** *Lang_gnf_hd*: ⟦ *eps_free ps*; *A* ∈ *nts ps* ⟧ $\Longrightarrow$ *Lang* (*gnf_hd ps*) *A* = *lang ps A*
⟨*proof*⟩

Two simple examples:

**lemma** *gnf_hd* [(*0*, [*Nt*(*0*::*nat*), *Tm* (*1*::*int*)])] = {(*1*, [*Tm 1*]), (*1*, [*Tm 1*, *Nt 1*])}
⟨*proof*⟩

**lemma** *gnf_hd* [(*0*, [*Nt*(*0*::*nat*), *Tm* (*1*::*int*)]), (*0*, [*Tm 2*])] =
{ (*0*, [*Tm 2*, *Nt 1*]), (*0*, [*Tm 2*]), (*1*, [*Tm 1*, *Nt 1*]), (*1*, [*Tm 1*]) }
⟨*proof*⟩

Example 4.10 [3]: *P0* is the input; *P1* is the result after Step 1; *P3* is the result after Step 2 and 3.

**lemma**
*let*
*P0* =
[(*1*::*int*, [*Nt 2*, *Nt 3*]), (*2*,[*Nt 3*, *Nt 1*]), (*2*, [*Tm* (*1*::*int*)]), (*3*,[*Nt 1*, *Nt 2*]), (*3*,[*Tm 0*])];
*P1* =
[(*1*, [*Nt 2*, *Nt 3*]), (*2*, [*Nt 3*, *Nt 1*]), (*2*, [*Tm 1*]),
(*3*, [*Tm 1*, *Nt 3*, *Nt 2*, *Nt 4*]), (*3*, [*Tm 0*, *Nt 4*]), (*3*, [*Tm 1*, *Nt 3*, *Nt 2*]), (*3*, [*Tm 0*]),
(*4*, [*Nt 1*, *Nt 3*, *Nt 2*]), (*4*, [*Nt 1*, *Nt 3*, *Nt 2*, *Nt 4*])];
*P2* =
[(*1*, [*Tm 1*, *Nt 3*, *Nt 2*, *Nt 4*, *Nt 1*, *Nt 3*]), (*1*, [*Tm 1*, *Nt 3*, *Nt 2*, *Nt 1*, *Nt 3*]),
(*1*, [*Tm 0*, *Nt 4*, *Nt 1*, *Nt 3*]), (*1*, [*Tm 0*, *Nt 1*, *Nt 3*]), (*1*, [*Tm 1*, *Nt 3*]),
(*2*, [*Tm 1*, *Nt 3*, *Nt 2*, *Nt 4*, *Nt 1*]), (*2*, [*Tm 1*, *Nt 3*, *Nt 2*, *Nt 1*]),
(*2*, [*Tm 0*, *Nt 4*, *Nt 1*]), (*2*, [*Tm 0*, *Nt 1*]), (*2*, [*Tm 1*]),
(*3*, [*Tm 1*, *Nt 3*, *Nt 2*, *Nt 4*]), (*3*, [*Tm 1*, *Nt 3*, *Nt 2*]),
(*3*, [*Tm 0*, *Nt 4*]), (*3*, [*Tm 0*]),
(*4*, [*Tm 1*, *Nt 3*, *Nt 2*, *Nt 4*, *Nt 1*, *Nt 3*, *Nt 3*, *Nt 2*, *Nt 4*]), (*4*, [*Tm 1*, *Nt 3*, *Nt 2*, *Nt 4*, *Nt 1*, *Nt 3*, *Nt 3*, *Nt 2*]),
(*4*, [*Tm 0*, *Nt 4*, *Nt 1*, *Nt 3*, *Nt 3*, *Nt 2*, *Nt 4*]), (*4*, [*Tm 0*, *Nt 4*, *Nt 1*, *Nt 3*, *Nt 3*, *Nt 2*]),
(*4*, [*Tm 1*, *Nt 3*, *Nt 3*, *Nt 2*, *Nt 4*]), (*4*, [*Tm 1*, *Nt 3*, *Nt 3*, *Nt 2*]),
(*4*, [*Tm 1*, *Nt 3*, *Nt 2*, *Nt 1*, *Nt 3*, *Nt 3*, *Nt 2*, *Nt 4*]), (*4*, [*Tm 1*, *Nt 3*, *Nt 2*, *Nt 1*, *Nt 3*, *Nt 3*, *Nt 2*]),

```
      (4, [Tm 0, Nt 1, Nt 3, Nt 3, Nt 2, Nt 4]), (4, [Tm 0, Nt 1, Nt 3, Nt 3, Nt
2])]
  in
    solve_tri [3,2,1] [4,5,6] (set P0) = set P1 ∧ expand_tri [4,1,2,3] (set P1)
= set P2
⟨proof⟩
```

# 7 Complexity

Our method has exponential complexity, which we demonstrate below. Alternative polynomial methods are described in the literature [1].

We start with an informal proof that the blowup of the whole method can be as bad as $2^{n^2}$, where $n$ is the number of non terminals, and the starting grammar has $4n$ productions.

Consider this grammar, where $a$ and $b$ are terminals and we use nested alternatives in the obvious way:

$A0 \rightarrow A1\ (a \mid b) \mid A2\ (a \mid b) \mid ... \mid An\ (a \mid b) \mid a \mid b$

$A(i+1) \rightarrow Ai\ (a \mid b)$

Expanding all alternatives makes this a grammar of size $4n$.

When converting this grammar into triangular form, starting with $A0$, we find that $A0$ remains the same after *expand_hd*, and *solve_lrec* introduces a new additional production for every $A0$ production, which we will ignore to simplify things:

Then every *expand_hd* step yields for $Ai$ these number of productions:

(1) $2\hat{}(i+1)$ productions with rhs $Ak\ (a \mid b)\hat{}(i+1)$ for every $k \in [i+1,\ n]$,

(2) $2\hat{}(i+1)$ productions with rhs $(a \mid b)\hat{}(i+1)$,

(3) $2\hat{}(i+1)$ productions with rhs $Ai\ (a \mid b)\hat{}(i+1)$.

Note that $(a \mid b)\hat{}(i+1)$ represents all words of length $i+1$ over $\{a,b\}$. Solving the left recursion again introduces a new additional production for every production of form (1) and (2), which we will again ignore for simplicity. The productions of (3) get removed by *solve_lrec*. We will not consider the productions of the newly introduced nonterminals.

In the triangular form, every $Ai$ has at least $2\hat{}(i+1)$ productions starting with terminals (2) and $2\hat{}(i+1)$ productions with rhs starting with $Ak$ for every $k \in [i+1,\ n]$.

When expanding the triangular form starting from $An$, which has at least the $2\hat{}(i+1)$ productions from (2), we observe that the number of productions of $Ai$ (denoted by $\#Ai$) is $\#Ai \geq 2\hat{}(i+1) * \#A(i+1)$ (Only considering the productions of the form $A(i+1)\ (a \mid b)\hat{}(i+1)$). This yields that $\#Ai \geq 2\hat{}(i+1) * 2\hat{}((i+2) + ... + (n+1)) = 2\hat{}((i+1) + (i+2) + ... (n+1))$. Thus $\#A0 \geq 2\hat{}(1 + 2 + ... + n + (n+1)) = 2\hat{}((n+1)*(n+2)/2)$.

Below we prove formally that *expand_tri* can cause exponential blowup.

Bad grammar: Constructs a grammar which leads to a exponential blowup when expanded by *expand_tri*:

**fun** *bad_grammar* :: *'n list* ⇒ (*'n*, *nat*)*Prods* **where**
*bad_grammar* [] = {}
|*bad_grammar* [*A*] = {(*A*, [*Tm 0*]), (*A*, [*Tm 1*])}
|*bad_grammar* (*A*#*B*#*As*) = {(*A*, *Nt B* # [*Tm 0*]), (*A*, *Nt B* # [*Tm 1*])} ∪ (*bad_grammar* (*B*#*As*))

**lemma** *bad_gram_simp1*: *A* ∉ *set As* ⟹ (*A*, *Bs*) ∉ (*bad_grammar As*)
⟨*proof*⟩

**lemma** *expand_tri_simp1*: *A* ∉ *set As* ⟹ (*A*, *Bs*) ∈ *R* ⟹ (*A*, *Bs*) ∈ *expand_tri As R*
⟨*proof*⟩

**lemma** *expand_tri_iff1*: *A* ∉ *set As* ⟹ (*A*, *Bs*) ∈ *expand_tri As R* ⟷ (*A*, *Bs*) ∈ *R*
⟨*proof*⟩

**lemma** *expand_tri_insert_simp*:
*B* ∉ *set As* ⟹ *expand_tri As* (*insert* (*B*, *Bs*) *R*) = *insert* (*B*, *Bs*) (*expand_tri As R*)
⟨*proof*⟩

**lemma** *expand_tri_bad_grammar_simp1*:
*distinct* (*A*#*As*) ⟹ *length As* ≥ *1*
⟹ *expand_tri As* (*bad_grammar* (*A*#*As*))
= {(*A*, *Nt* (*hd As*) # [*Tm 0*]), (*A*, *Nt* (*hd As*) # [*Tm 1*])} ∪ (*expand_tri As* (*bad_grammar As*))
⟨*proof*⟩

**lemma** *finite_bad_grammar*: *finite* (*bad_grammar As*)
⟨*proof*⟩

**lemma** *finite_expand_tri*: *finite R* ⟹ *finite* (*expand_tri As R*)
⟨*proof*⟩

The last Nt expanded by *expand_tri* has an exponential number of productions.

**lemma** *bad_gram_last_expanded_card*:
⟦*distinct As*; *length As* = *n*; *n* ≥ *1*⟧
⟹ *card* ({*v*. (*hd As*, *v*) ∈ *expand_tri As* (*bad_grammar As*)}) = *2* ^ *n*
⟨*proof*⟩

The productions resulting from *expand_tri* (*bad_grammar*) have at least exponential size.

**theorem** *expand_tri_blowup*: **assumes** *n* ≥ *1*

**shows** *card* (*expand_tri* [*0*..<*n*] (*bad_grammar* [*0*..<*n*])) ≥ *2*^*n*
⟨*proof*⟩

**end**

# References


[1] N. Blum and R. Koch. Greibach normal form transformation revisited. *Inf. Comput.*, 150(1):112–118, 1999.

[2] S. A. Greibach. A new normal-form theorem for context-free phrase structure grammars. *J. ACM*, 12(1):42–52, 1965.

[3] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.