HAL
archives-ouvertes.fr

# Graph Interpolation Grammars as Context-Free Automata

John Larchevêque

▶ **To cite this version:**

John Larchevêque. Graph Interpolation Grammars as Context-Free Automata. [Research Report] RR-3456, INRIA. 1998. inria-00073234

**HAL Id: inria-00073234**
**https://hal.inria.fr/inria-00073234**

Submitted on 24 May 2006

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

# *Graph Interpolation Grammars as Context-Free Automata*

John Larchevêque

**N° 3456**

July 1998

THÈME 3

ISSN 0249-6399

INRIA
ROCQUENCOURT

# Graph Interpolation Grammars as Context-Free Automata

John Larchevêque

Thème 3 — Interaction homme-machine,
images, données, connaissances
Projet Atoll

Rapport de recherche n° 3456 — July 1998 — 20 pages

**Abstract:** A derivation step in a Graph Interpolation Grammar has the effect of scanning an input token. This feature, which aims at emulating the incrementality of the natural parser, restricts the formal power of GIGs. This contrasts with the fact that the derivation mechanism involves a context-sensitive device similar to tree adjunction in TAGs. The combined effect of input-driven derivation and restricted context-sensitiveness would be conceivably unfortunate if it turned out that Graph Interpolation Languages did not subsume Context Free Languages while being partially context-sensitive. This report sets about examining relations between CFGs and GIGs, and shows that GILs are a proper superclass of CFLs. It also brings out a strong equivalence between CFGs and GIGs for the class of CFLs. Thus, it lays the basis for meaningfully investigating the amount of context-sensitiveness supported by GIGs, but stops short of embarking on this investigation.

**Key-words:** parsing, natural language processing, linguistics, grammar, syntactic representation

*(Résumé : tsvp)*

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : 01 39 63 55 11 - International : +33 1 39 63 55 11
Télécopie : (33) 01 39 63 53 30 - International : +33 1 39 63 53 30

# Etude des Grammaires à Interpolations de Graphes en tant qu'Automates Non Contextuels

**Résumé :** Une étape de dérivation dans une Grammaire à Interpolations de Graphes a pour effet d'intégrer un lexème de la chaîne d'entrée. Cette propriété, qui vise à émuler l'incrémentalité de l'analyseur syntaxique naturel, restreint la puissance formelle des GIG. Ceci est à contraster avec le fait que la dérivation d'une chaîne par une GIG met en jeu un mécanisme à effet contextuel analogue à l'adjonction d'arbre dans une TAG. L'effet conjoint du comportement d'automate des dérivations et d'une certaine mesure de contextualité pourrait aboutir à une situation potentiellement fâcheuse dans laquelle les languages à interpolations de graphes ne subsumeraient pas les languages non contextuels, tout en étant partiellement contextuels. Ce rapport examine les relations entre les CFG et les GIG, et montre que les langages à interpolations de graphes constituent une surclasse stricte des langages non contextuels. Il y est également montré que les GIG sont fortement équivalentes aux CFG pour la classe des languages non contextuels. Ces résultats donnent un sens à la question du degré de contextualité des GIG, qui pourra ainsi faire l'objet de travaux futurs.

**Mots-clé :** syntaxe, langage naturel, linguistique, grammaire, représentation syntaxique

# 1 Introduction

## 1.1 Graph Interpolation Grammars as automata

Graph Interpolation Grammars are indifferently viewed as grammars or automata. That is to say that a derivation step in a GIG is an automaton move during which one token of the input string is scanned. In this perspective, GIG rules can be viewed as the finite control of an automaton.

## 1.2 Rationale for GIG graphs

The output of a GIG is a graph rather than a tree. If trimmed down to its bare essentials, a GIG graph can be considered as a particular representation of a tree, in which a node is connected to its leftmost child through a *parent-of* edge, and each child is connected to its right sibling through a *predecessor-of* edge. Now, the advantages of this representation are the following

1. A parse subgraph is allowed to assign a smaller arity to a given node than the complete graph that includes it, so that arities can be determined incrementally. For example, given a sentence starting with *John thinks Mary ...*, what can be hypothesized is that *Mary* is the subject of a verb, but the arity of this verb is unknown at this stage of the parsing.

2. In addition to tree adjunction, which shifts a subtree further down the parse tree, parse graphs support a similar operation which shifts a subgraph further right, thus allowing sets of constructions such as the following to be viewed as paradigms. (Alternating paradigm constituents appear in square brackets and gaps are represented by squares).

    Who did you [invite] □?
    Who did you [happen to invite] □?
    Who did you [think I invited] □?

Such a paradigm is captured in GIG by considering that the basic construction (with the constituent *invite*) can be expanded by interpolating a subgraph to the left of the gap. One can expect it to be hard, if possible at all, to represent such a constituent as *think I invited* (without the object gap) by a tree, especially if intuitive phrase structure is to be preserved. Hence the need for a more versatile structure than standard parse trees.

## 1.3 Expressive power of graph interpolation

The operation on which GIGs are based, graph interpolation, is basically a tree adjunction [JLT75, VSW94] applied to a pair of GIG graphs. On the one hand, graph interpolation is more general than tree adjunction, but, on the other hand, its application obeys a very strong constraint, namely that it be performed in response to the scanning of a token. Owing

to this constraint, it is even legitimate to ask whether Graph Interpolation Languages subsume Context-Free Languages. So, before solving the important problem of GIGs relation to TAGs in term of formal power, it is worth comparing GIGs with CFGs. This comparison is the subject of the present report.

### 1.4 Content of the report

This report proves that

- Graph Interpolation Languages include Context-Free Languages (Section 3),
- Graph Interpolation Languages properly include Context-Free Languages (Section 4),
- GIGs are strongly equivalent to CFGs on the class of CFLs (Section 5),
- the intersection of CFLs and deterministic GILs includes the class LL(1) (Section 6).

A preliminary to these results is the definition of a GIG, which is given in the next section.

## 2 Definition of a Graph Interpolation Grammar

A linguistic presentation of Graph Interpolation Grammars was given in [Lar98]. The definition given here deliberately ignores all features not directly relevant to the purpose of defining GIGs as automata.

- Multiple-interpolation rules will not be considered, for the formal power they contribute is excessive in the perspective of a comparison with CFGs.
- A fixed word order will be assumed.
- Phrase heads and grammatical functions will be ignored, which shows that they can be decoupled from parsing issues.
- Node label subtyping, which has no impact on formal power, will not be made use of.

Apart from these introductory remarks, the present report can be read without any prior knowledge of Graph Interpolation Grammars.

### 2.1 Parse graphs

A parse graph is a representation of a tree in which the relation *parent-of* is broken down into a *first-child* edge and zero or more *right-sibling* edges. More precisely, a parse graph $\Gamma$ is a tuple $\langle N, L, r, V, \Sigma, \gamma, E, F \rangle$, where

- $N$ and $L$ are disjoint sets of nodes (respectively nonlexical nodes and lexemes),
- $r$ is a distinguished member of $N$, called the graph *root*,

- $V$ and $\Sigma$ are disjoint sets of node labels (respectively variables and symbols),
- $\gamma$ is the label assignment, i.e. a function in $N \times V \cup L \times \Sigma$,
- $E$ is the set of *first-child* edges, i.e. a function in $N \times (N \cup L)$,
- $F$ is the set of *right-sibling* edges, i.e. a function in $N \times N$. (This definition precludes lexical nodes, i.e. members of $L$, from having siblings.)

#### 2.1.1 Constraints on $E$ and $F$

- $E$ and $F$ are functions, i.e. $|E(n)| \leq 1$ and $|F(n)| \leq 1$ for all $n$ in $N$. (The notation $\mathcal{R}(x)$, where $\mathcal{R}$ is a relation, denotes the set $\{y \mid \langle x, y\rangle \in \mathcal{R}\}$.) In a *complete parse graph*, $N$ is the domain of $E$, i.e. $|E(n)| = 1$ for all $n$ in $N$.
- Every node except the root has exactly one immediate dominator[1] by $E \cup F$, i.e. $|(E \cup F)^{-1}(n)| = 1$ for all $n$ in $N - \{r\} \cup L$.
- Every node in $L$ is a single child with no descendant, i.e. $E(l) = F(l) = F^{-1}(l) = \emptyset$ for all $l$ in $L$.

## 2.2 Path interpolation

A path interpolation consists in plugging into a parse graph $\Gamma_1 = \langle N_1, L_1, r_1, V, \Sigma, \gamma_1, E_1, F_1\rangle$ a disjoint parse graph $\Gamma_2 = \langle N_2, L_2, s, V, \Sigma, \gamma_2, E_2, F_2\rangle$ by substituting a path of $\Gamma_2$ for a node $q$ of $\Gamma_1$ that is called the *anchor* of the interpolation. The path that is substituted is defined by 3 nodes of $\Gamma_2$, $s$, the source of the path (which is the root of $\Gamma_2$), $t$, the target of the path, and $s'$, the successor of $s$ in the interpolation path[2].
Owing to properties of $E_2$ and $F_2$ (i.e. existence of at most one immediate dominator for any node), there is at most one path from $s$ to $t$. A constraint on path interpolation is that there be exactly one path (of length 0 or more) from $s$ to $t$.
There are 3 types of interpolation.

- In a *down-shift*, $\langle s, s'\rangle$ is in $E_2$. A down-shift has the effect of a tree adjunction.
- In a *right-shift*, $\langle s, s'\rangle$ is in $F_2$. A right-shift has the effect of shifting a sibling edge of $\Gamma_1$ further right in the resulting graph.
- In an $\epsilon$-*shift*, $s = s' = t$. The effect of an $\epsilon$-shift is to insert a subgraph at the anchor $q$.

[1] A node $n$ is said to *dominate* a node $m$ in a rooted graph whenever every path from the root to $m$ goes through $n$. The term *dominator* is here preferred to *predecessor* or *successor* to avoid confusions with the *successor* relation introduced in Section 2.2.2.

[2] Examples of interpolations are shown on Figure 1.

Formally, an interpolation $\mathcal{I}$ is a function that maps a tuple of the form $\langle \Gamma_1, \Gamma_2, q, t \rangle$ to a parse graph $\Gamma_3$. ($q$ is the anchor of the interpolation, and $t$ the target, i.e. the endpoint, of the interpolation path.) Let $\Gamma_1$ and $\Gamma_2$ be defined as above, then $\Gamma_3 = \langle N_3, L_3, r_3, V, \Sigma, \gamma_3, E_3, F_3 \rangle$ is defined as follows, where $s$ is the root of $\Gamma_2$ (i.e. the interpolation source) and $s'$ is the node that immediately follows $s$ in the interpolation path.

- $N_3 = N_1 - \{q\} \cup N_2$
- $L_3 = L_1 \cup L_2$
- $r_3 = r_1$
- $\gamma_3 = \gamma_1 \cup \gamma_2$
-

$$
\begin{array}{lll}
E_3 = & E_1 \cup E_2 & \\
\multicolumn{3}{l}{\textit{// parent of q, if any, becomes parent of s}} \\
- & \{\langle n, q \rangle \mid & n \in N_1\} \\
\cup & \{\langle n, s \rangle \mid & \langle n, q \rangle \in E_1\} \\
\multicolumn{3}{l}{\textit{// child of q, if any, becomes child of t (down-shift) or s (right-shift)}} \\
- & \{\langle q, n \rangle \mid & n \in N_1\} \\
\cup & \{\langle m, n \rangle \mid & \langle q, n \rangle \in E_1 \text{ and} \\
 & & ((\langle s, s' \rangle \in E_2 \text{ and } m = t) \\
 & & \text{ or } (\langle s, s' \rangle \notin E_2 \text{ and } m = s))\}
\end{array}
$$

-

$$
\begin{array}{lll}
F_3 = & F_1 \cup F_2 & \\
\multicolumn{3}{l}{\textit{// left sibling of q, if any, becomes left sibling of s}} \\
- & \{\langle n, q \rangle \mid & n \in N_1\} \\
\cup & \{\langle n, s \rangle \mid & \langle n, q \rangle \in F_1\} \\
\multicolumn{3}{l}{\textit{// right sibling of q, if any, becomes right sibling of t (right-shift) or s (down-shift)}} \\
- & \{\langle q, n \rangle \mid & n \in N_1\} \\
\cup & \{\langle m, n \rangle \mid & \langle q, n \rangle \in F_1 \text{ and} \\
 & & ((\langle s, s' \rangle \in F_2 \text{ and } m = t) \\
 & & \text{ or } (\langle s, s' \rangle \notin F_2 \text{ and } m = s))\}
\end{array}
$$

#### 2.2.1 Definition constraints for $\mathcal{I}$

1. **Disjunction constraint** $\Gamma_1$ and $\Gamma_2$ are disjoint, i.e. $N_1 \cap N_2 = \emptyset$ (from which it follows that the other pairs of sets are disjoint too, given constraints on parse graph edges).
2. **Lexicalization constraint** $\Gamma_2$ is mono-lexicalized, i.e. $|L_2| = 1$.

3. **Context constraint** $\Gamma_1$ is leftmost-lexicalized, meaning that its frontier, i.e. its yield over $(N_1 \cup L_1)^*$, is in the regular language $L_1^* N_1^*$. (The concept of frontier is defined more precisely in the next section.)

4. **Frontier constraint** Given $x\alpha$ the frontier of $\Gamma_1$, with $x$ in $L_1^*$ and $\alpha$ in $N_1^*$, and $\delta a \beta$ the frontier of $\Gamma_2$, with $a$ in $L_2$ and $\delta$ and $\beta$ in $N_2^*$, the frontier of $\Gamma_3$ has prefix $xa$[3].

Figure 1 illustrates the three forms of interpolation.

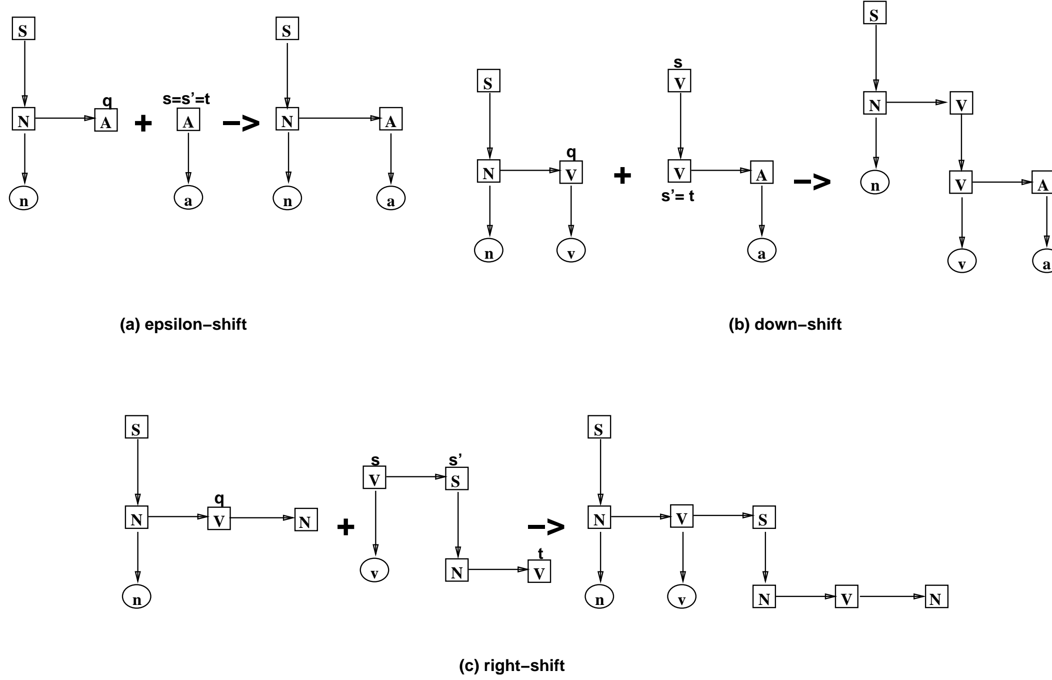


Figure 1: The three types of interpolation

### 2.2.2 Anchor position

Given a triple $\langle \Gamma_1, \Gamma_2, t \rangle$, the constraints just enumerated restrict the set of possible positions for the anchor $q$ with respect to the last lexeme of $\Gamma_1$ if $\langle \Gamma_1, \Gamma_2, q, t \rangle$ is to belong to the definition domain of interpolation. As a matter of fact, $q$ is either an ancestor or an immediate successor of the last lexeme of $\Gamma_1$, as this section will attempt to prove[4].

[3]For details on the possible forms of the frontier of $\Gamma_3$, see Section 6.
[4]This result does not imply that there is a single choice of $q$ for given $\Gamma_1$, $\Gamma_2$, and $t$.

**Immediate successor** The notion of "immediate successor" is defined as follows. Let $<$ be the "successor" relation in a parse graph $\Gamma$, defined as follows. (The operator $\circ$ denotes relation composition or, equivalently, edge concatenation.)

$$< \quad = \quad ((F^{-1})^* \circ E^{-1})^* \circ F \circ E^*$$

Then, the "immediate successor" relation, noted as $\ll$, holds between $m$ and $n$ if and only if $m < p < n$ implies that $m = p$ or $n = p$.

**Graph frontier** This relation makes it possible to define the frontier of a parse graph $\Gamma = \langle N, L, r, V, \Sigma, \gamma, E, F \rangle$ more precisely than was done in the previous section. Let $T$ equal $\{n \in N \cup L \mid E(n) = \emptyset\}$ in:

$$\begin{aligned} \text{frontier}(\Gamma) = \{w \mid \quad & w \in T^* \text{ and} \\ & \forall x, y \in T^*, \forall m, n \in T : w = xmny \Rightarrow m \ll n\} \end{aligned}$$

The frontier of a parse graph is a string containing all of and only its terminal nodes in $\ll$ order. (The definition of $\ll$ implies that $|w| = |T|$.)

**Anchor position with respect to the last lexeme** Let $\mu$ be the last lexeme of $\Gamma_1$ and $\nu$ the single lexeme of $\Gamma_2$. It will now be shown that the constraint $\mu \ll \nu$ (which results from the frontier constraint) can only be satisfied if $q$, the anchor, is an ancestor or immediate successor of $\mu$. This theorem does not cover the initial interpolation step, which starts from a context $\Gamma_1$ that consists of a single nonterminal node (Section 2.4).

**Theorem 1** *If $L_1$ is nonempty, $\mathcal{I}$ is defined for $\langle \Gamma_1, \Gamma_2, q, t \rangle$ only if $q$ is either an ancestor or an immediate successor of the last lexeme of $\Gamma_1$. I.e., let $\mu$ be the rightmost lexeme in the frontier of $\Gamma_1$, either $q \in (F_1^{-1*} \circ E_1^{-1})^*(\mu)$ or $\mu \ll q$ must hold.*

**Proof outline** The constraint $\mu \ll \nu$ can be decomposed as follows, where $m$ and $n$ are nodes of $\Gamma_3$.

$$\mu \quad \overset{*}{\underset{F_3^{-1*} \circ E_3^{-1}}{\Longrightarrow}} \quad m \quad \underset{F_3}{\Longrightarrow} \quad n \quad \overset{*}{\underset{E_3}{\Longrightarrow}} \quad \nu$$

In order to locate $q$ with respect to $\mu$, we will locate $s$, the root of $\Gamma_2$, and then use the fact that $s$ occupies in $\Gamma_3$ the position that $q$ occupies in $\Gamma_1$. More precisely, the unique $E_1 \cup F_1$-path from $r_1$ to $q$ contains the same nodes (down to $q$ itself) as the unique $E_3 \cup F_3$-path from $r_3$ to $s$.
Now, as the root of $\Gamma_2$, $s$ dominates $\nu$, which implies that it lies somewhere on a path containing $r_3$, $m$, $n$, and $\nu$ (in this order), since every node has a single immediate dominator by $E_3 \cup F_3$.
Supposing $s$ properly dominates $m$, then $m = t$, for the path from $s$ to $m$ occupies the place of an $N_1$ node, namely $q$, on the path from $r_1$ to $\mu$. (By definition of an interpolation, $r_1 = r_3$.) Conversely, $q$ occupies the place of $m$ in $\Gamma_1$; therefore $q$ **is an ancestor of** $\mu$.

The same conclusion can be made if $s$ equals $m$.
If $s$ is dominated by $n$, we have $\mu \ll s$, from which it follows that $\mu \ll q$ in $\Gamma_1$. Therefore, $q$ **is an immediate successor of** $\mu$**.** $\square$

## 2.3 Interpolation rule

An interpolation rule $\rho$ is a tuple $\langle A, \Gamma_2, t\rangle$, where $A$ is a variable in $V$ and $t$, the *interpolation target*, is a node of $\Gamma_2$. Rule $\rho$ is said to match a parse graph $\Gamma_1$ if the interpolation function is defined for $\langle \Gamma_1, \Gamma_2, q, t\rangle$ for some $q$ in $\gamma_1^{-1}(A)$.
The variable $A$ is called the *context pattern* of the rule, $\Gamma_2$ is called the *addendum*, and a matching $\Gamma_1$ the *context* to which the rule is applied.

### 2.3.1 Rule notation

The clearest way of representing an interpolation involves a 2-dimensional representation of the addendum, as on Figure 1. On the other hand, since a parse graph is a representation of a parse tree, which is itself a representation of a context-free derivation, an addendum can be represented as a CF derivation. Thus, a rule $\langle A, \Gamma_2, t\rangle$ can be represented as a left-hand side consisting of the context pattern $A$, a separating right arrow, and a right-hand side in which $\Gamma_2$ is represented as a leftmost derivation, and $t$ is dotted. For example, the $\epsilon$-shift on Figure 1 can be noted as $A \rightarrow (\dot{A} \underset{\mathrm{lm}}{\Longrightarrow} a)$. Likewise, the down-shift can be noted as $V \rightarrow (V \underset{\mathrm{lm}}{\Longrightarrow} \dot{V}A \underset{\mathrm{lm}}{\Longrightarrow} Va)$ and the right-shift as $V \rightarrow (VS \underset{\mathrm{lm}}{\Longrightarrow} vS \underset{\mathrm{lm}}{\Longrightarrow} vN\dot{V})$.

## 2.4 Graph Interpolation Grammar

A Graph Interpolation Grammar is a tuple $\langle V, \Sigma, P, S\rangle$, where $P$ is a set of interpolations rules, $V$ and $\Sigma$ are the label sets used by these rules, and $S$ is a member of $V$ called the *axiom*.
At the start of a GIG derivation, the *context* is a subgraph consisting of one node labeled with the axiom.
Each step matches a rule of $P$ with the context and applies the corresponding interpolation, yielding a new context.

# 3 Graph Interpolation Languages form a superclass of CFLs

## 3.1 A definition of Graph Interpolation Languages

Let an Instantaneous Description be defined as a tuple $\langle \Gamma, w\rangle$, where the string to parse is of the form $vw$, with $v$ the yield of the context $\Gamma$. Initially, $\Gamma$ consists of a single node labelled

with the axiom, and $v$ is empty. An interpolation $\mathcal{I}(\langle \Gamma, \Gamma_2, q, t \rangle) = \Gamma_3$ maps $\langle \Gamma, ax \rangle$ to $\langle \Gamma_3, x \rangle$, with $yield(\Gamma_3) = yield(\Gamma).a$. A string $w$ is accepted by a GIG if applicable interpolations map the initial ID to $\langle \Gamma_f, \epsilon \rangle$, with $yield(\Gamma_f) = w$. The definition of a Graph Interpolation Language follows in the classical way.

## 3.2 Graph Interpolation Languages as a superclass of Context-Free Languages

**Theorem 2** *Any CFL not containing the string $\epsilon$ can be recognized by a Graph Interpolation Grammar.*

**Proof outline** Any CFL not containing the string $\epsilon$ can be described by a grammar in Greibach Normal Form, i.e. a CFG $\langle V, \Sigma, P, S \rangle$ in which every production is of the form $A \rightarrow a\gamma$, with $A$ in $V$, $a$ in $\Sigma$, and $\gamma$ in $V^*$.
To build a GIG $G'$ from a GNF grammar $G$, consider each production $p$ as a GIG rule $\rho$ whose context pattern consists of the left-hand side of $p$, and whose addendum represents the subtree obtained in a CFG-derivation by substituting the right-hand side for the left-hand side, except that the leftmost symbol of the right-hand side is replaced by the preterminal symbol $X$, $X \notin V \cup \Sigma$, as shown in Figure 2.

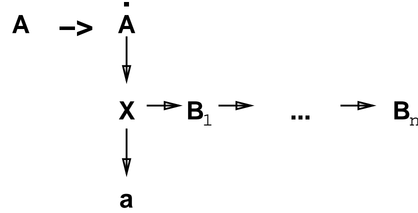


Figure 2: GIG rule built from the production $A \rightarrow aB_1...B_n$

The GIG thus built is a tuple $\langle V', \Sigma, P', S \rangle$, where $V'$ equals $V \cup \{X\}$ and $P'$ is the set of GIG-rules just described.
It is easy to prove by induction that the derivation produced by this GIG is a leftmost derivation of $G$.
Indeed, initially, the context consists of a node labeled $S$, whose yield is therefore a leftmost sentential form of $G$.
Supposing the yield of the current context is a leftmost sentential form $xA_1...A_n$, $x$ in $\Sigma^*$, then there will be an applicable GIG rule if and only if there is a production $A_1 \rightarrow aB_1...B_m$ in $P$, and this rule will map the context to a context whose yield is $xaB_1...B_mA_2...A_n$, i.e. a leftmost sentential form of $G$. $\square$

# 4 GILs form a proper superclass of CFLs

**Theorem 3** *The class of Context-Free Languages is properly included in the class of Graph Interpolation Languages.*

**Proof outline** Now it has been shown that every CFL not containing $\epsilon$ is recognized by some GIG, it remains to show that there are languages which can be recognized by Graph Interpolation Grammars but not by Context-Free Grammars.
The language $a^n b^n c^n d^n$, which is a classic example of a context-sensitive language, is recognized by the GIG $\langle \{S, A, B, C, D\}, \{a, b, c, d\}, P, S \rangle$, where $P$ consists of the following rules:

$$
\begin{array}{llll}
(1) & S & \rightarrow & (\underline{\dot{S}} \Rightarrow \underline{A}SD \Rightarrow a\underline{S}D \Rightarrow aB\underline{S}CD \Rightarrow aB\epsilon CD) \\
(2) & S & \rightarrow & (\underline{S} \Rightarrow \underline{A}SD \Rightarrow a\underline{S}D \Rightarrow aB\dot{S}CD) \\
(3) & B & \rightarrow & (\underline{\dot{B}} \Rightarrow b) \\
(4) & C & \rightarrow & (\underline{\dot{B}} \Rightarrow c) \\
(5) & D & \rightarrow & (\underline{\dot{B}} \Rightarrow d)
\end{array}
$$

A 2-dimensional representation of these rules appears on Figure 3.

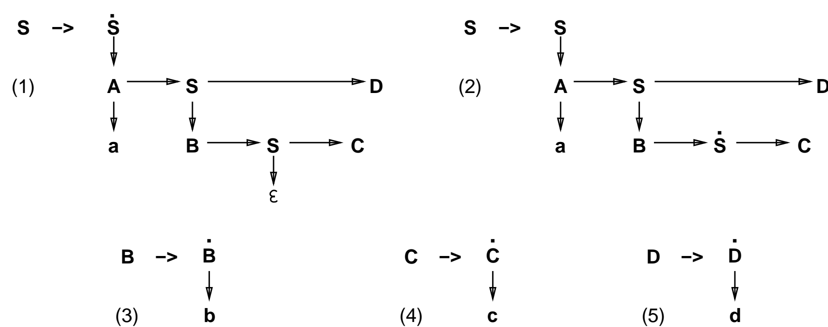


Figure 3: GIG rules for $a^n b^n c^n d^n$

As production (1) requires the presence of an $S$-node in the context frontier, it applies exclusively to the initial context. On the other hand, the frontier constraint requires that the down-shift (2) apply exclusively to the middle $S$ of the previous interpolation, i.e. the $S$ that is an immediate successor (by $\ll$) of the last lexeme. Figure 4 shows the first two steps of the unique derivation of *aabbccdd*. $\square$

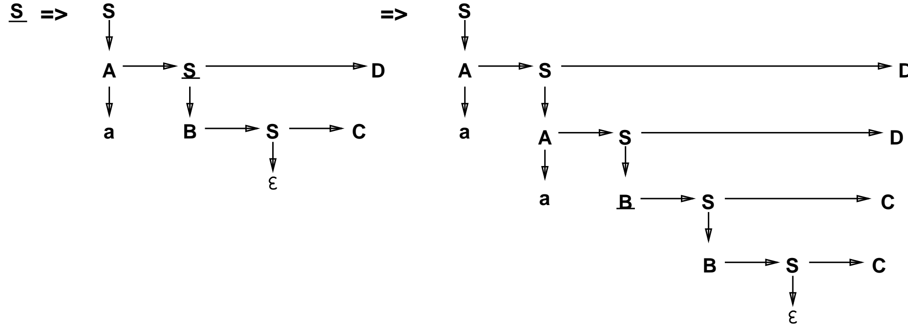


Figure 4: Initial GIG derivation steps for *aabbccdd*

# 5 GILs strongly subsume CFLs

Define a *2-tiered* Context-Free Grammar to be a CFG containing no production of the form $A \rightarrow \alpha a \beta$ in which $\alpha$ or $\beta$ is a nonempty string in $(V \cup \Sigma)^*$ and $a$ is a terminal symbol. The graph representations of trees generated by a 2-tiered CFG obey the constraint that lexical nodes have neither descendants nor siblings (Section 2.1.1). A simple way of making an arbitrary CFG $G = \langle V, \Sigma, P, S \rangle$ 2-tiered is to *(i)* add into $V$ a preterminal symbol $X_a$ for each $a$ in $\Sigma$, *(ii)* substitute $X_a$ for $a$ in all productions of $P$, and *(iii)* add into $P$ the production $X_a \rightarrow a$ for each $a$ in $\Sigma$. The resulting CFG $G'$ remains close enough to the original grammar $G$ for any semantic computation on a $G$ tree to map straightforwardly to its counterpart $G'$ tree, and vice versa. Thus, conversion to 2-tiered form is semantically neutral, and it seems therefore legitimate to say that GIGs are strongly equivalent to CFGs on the class of CFLs, even though the derivations obtained with a GIG are necessarily 2-tiered. In this sense, the following theorem asserts the strong equivalence of GIGs and CFGs on the class of CFLs.

**Theorem 4** *Any unambiguous 2-tiered CFG not recognizing the empty string can be mapped to a GIG that constructs exactly the same CF derivations on the same inputs.*

**Proof outline** Consider the following procedure for building a GIG $G' = \langle V, \Sigma, P', S \rangle$ from a 2-tiered CFG $G = \langle V, \Sigma, P, S \rangle$:

1. For each variable $A$ in $V$, unwind all possible leftmost derivations by grammar $G$ starting at $A$ until either *(i)* the empty string is derived ($A \underset{\mathrm{lm}}{\overset{+}{\Longrightarrow}} \epsilon$), *(ii)* the derived string has a leftmost terminal symbol ($A \underset{\mathrm{lm}}{\overset{+}{\Longrightarrow}} a\alpha$), or *(iii)* the leftmost symbol of the derived string is a variable of $V$ that has already been replaced in the derivation ($A \underset{\mathrm{lm}}{\overset{*}{\Longrightarrow}} B\alpha \underset{\mathrm{lm}}{\overset{+}{\Longrightarrow}} B\beta\alpha$). Let $\Delta$ be the set of derivations thus obtained.

Note that, when a leftmost $\epsilon$ is obtained, the derivation is pursued, if possible, until the leftmost symbol is a member of $V \cup \Sigma$.

These bounded derivations will be used later on to form addenda in GIG rules.

2. For each derivation $D_1$ in $\Delta$, of the form $A \overset{*}{\underset{\text{lm}}{\Longrightarrow}} AB\alpha$, for each derivation $D_2$ in $\Delta$, of the form $B \overset{*}{\underset{\text{lm}}{\Longrightarrow}} b\beta$, $b$ in $\Sigma$, replace $D_1$ in $\Delta$ by the derivation $A \overset{*}{\underset{\text{lm}}{\Longrightarrow}} AB\alpha \overset{*}{\underset{\text{lm}}{\Longrightarrow}} Ab\beta\alpha$.

   Here, use is made of the fact that $G$ is unambiguous and 2-tiered to write the yield of any self-embedding $A$-derivation as $AB\alpha$. This step puts self-embedding derivations into a form suitable to create down-shifts.

3. For each pair of derivations $\langle A \overset{*}{\underset{\text{lm}}{\Longrightarrow}} \epsilon, B \overset{*}{\underset{\text{lm}}{\Longrightarrow}} \alpha A\beta\rangle$ in $\Delta^2$, with $\alpha \neq \epsilon$, add the derivation $B \overset{*}{\underset{\text{lm}}{\Longrightarrow}} \alpha A\beta \overset{*}{\underset{\text{lm}}{\Longrightarrow}} \alpha\beta$ to $\Delta$.

   After this step, each derivation into $\epsilon$ is inserted into a derivation whose yield starts with a symbol in $\Sigma \cup V$.

4. From each derivation whose yield has a leftmost terminal symbol, build an $\epsilon$-shift and add it to $P'$.

   The $\epsilon$-shift is built in the obvious way. A derivation $D$ of the form $A \overset{*}{\underset{\text{lm}}{\Longrightarrow}} a\alpha$ is mapped to an interpolation rule $\langle A, \Gamma, r\rangle$, where $\Gamma$ is the parse graph representing $D$ and $r$ is the root of $\Gamma$.

5. From each derivation $D$ of the form $A \overset{*}{\underset{\text{lm}}{\Longrightarrow}} Ab\alpha$, build a down-shift $\langle A, \Gamma, t\rangle$, where $\Gamma$ is the parse graph representing $D$ and $t$ is the leftmost node (labeled $A$) in the frontier of $\Gamma$.

**Example** Consider the 2-tiered CFG $\langle\{A_1, A_2, B_1, B_2, C_1, C_2, D_1, D_2, E, S\}, \{a, b, c, d\}, P, S\rangle$, with $P$ consisting of the following productions:

$$
\begin{array}{lcl}
S & \rightarrow & A_1B_1 \mid C_1E \\
A_1 & \rightarrow & A_2 \mid \epsilon \\
A_2 & \rightarrow & a \\
B_1 & \rightarrow & B_1B_2 \mid B_2 \\
B_2 & \rightarrow & b \\
C_1 & \rightarrow & C_2D_1 \\
C_2 & \rightarrow & c \\
D_1 & \rightarrow & D_1D_2 \mid \epsilon \\
D_2 & \rightarrow & d \\
E & \rightarrow & \epsilon
\end{array}
$$

**Step 1** The following derivations are inserted into $\Delta$:

$$
\begin{array}{ll}
D_1: & S \underset{\rm lm}{\Longrightarrow} A_1B_1 \underset{\rm lm}{\Longrightarrow} A_2B_1 \underset{\rm lm}{\Longrightarrow} aB_1 \\
D_2: & S \underset{\rm lm}{\Longrightarrow} A_1B_1 \underset{\rm lm}{\Longrightarrow} B_1 \underset{\rm lm}{\Longrightarrow} B_1B_2 \\
D_3: & S \underset{\rm lm}{\Longrightarrow} A_1B_1 \underset{\rm lm}{\Longrightarrow} B_1 \underset{\rm lm}{\Longrightarrow} B_2 \underset{\rm lm}{\Longrightarrow} b \\
D_4: & S \underset{\rm lm}{\Longrightarrow} C_1E \underset{\rm lm}{\Longrightarrow} C_2D_1E \underset{\rm lm}{\Longrightarrow} cD_1E \\
D_5: & A_1 \underset{\rm lm}{\Longrightarrow} A_2 \underset{\rm lm}{\Longrightarrow} a \\
D_6: & A_1 \underset{\rm lm}{\Longrightarrow} \epsilon \\
D_7: & A_2 \underset{\rm lm}{\Longrightarrow} a \\
D_8: & B_1 \underset{\rm lm}{\Longrightarrow} B_1B_2 \\
D_9: & B_1 \underset{\rm lm}{\Longrightarrow} B_2 \underset{\rm lm}{\Longrightarrow} b \\
D_{10}: & B_2 \underset{\rm lm}{\Longrightarrow} b \\
D_{11}: & C_1 \underset{\rm lm}{\Longrightarrow} C_2D_1 \underset{\rm lm}{\Longrightarrow} cD_1 \\
D_{12}: & C_2 \underset{\rm lm}{\Longrightarrow} c \\
D_{13}: & D_1 \underset{\rm lm}{\Longrightarrow} D_1D_2 \\
D_{14}: & D_1 \underset{\rm lm}{\Longrightarrow} \epsilon \\
D_{15}: & D_2 \underset{\rm lm}{\Longrightarrow} d \\
D_{16}: & E \underset{\rm lm}{\Longrightarrow} \epsilon
\end{array}
$$

Note that derivations $D_2$ and $D_3$ involve the $\epsilon$-production for $A_1$.

**Step 2** The left-recursive derivations $D_8$ and $D_{13}$ are replaced by the following derivations.

$$
\begin{array}{ll}
{D_8}': & B_1 \underset{\rm lm}{\Longrightarrow} B_1B_2 \underset{\rm lm}{\Longrightarrow} B_1b \\
{D_{13}}': & D_1 \underset{\rm lm}{\Longrightarrow} D_1D_2 \underset{\rm lm}{\Longrightarrow} D_1d
\end{array}
$$

**Step 3** The empty derivations $D_6$, $D_{14}$ and $D_{16}$ are exhaustively plugged into the yield of other derivations in $\Delta$, yielding eventually the following set of additional derivations.

$$
\begin{array}{ll}
D_{17}: & S \underset{\rm lm}{\Longrightarrow} C_1E \underset{\rm lm}{\Longrightarrow} C_2D_1E \underset{\rm lm}{\Longrightarrow} cD_1E \Longrightarrow cE \\
D_{18}: & S \underset{\rm lm}{\Longrightarrow} C_1E \underset{\rm lm}{\Longrightarrow} C_2D_1E \underset{\rm lm}{\Longrightarrow} cD_1E \Longrightarrow cD_1 \\
D_{19}: & S \underset{\rm lm}{\Longrightarrow} C_1E \underset{\rm lm}{\Longrightarrow} C_2D_1E \underset{\rm lm}{\Longrightarrow} cD_1E \Longrightarrow cE \Longrightarrow c \\
D_{20}: & S \underset{\rm lm}{\Longrightarrow} C_1E \underset{\rm lm}{\Longrightarrow} C_2D_1E \underset{\rm lm}{\Longrightarrow} cD_1E \Longrightarrow cD_1 \Longrightarrow c \\
D_{21}: & C_1 \underset{\rm lm}{\Longrightarrow} C_2D_1 \underset{\rm lm}{\Longrightarrow} cD_1 \underset{\rm lm}{\Longrightarrow} c \\
D_{22}: & D_1 \underset{\rm lm}{\Longrightarrow} D_1D_2 \underset{\rm lm}{\Longrightarrow} D_1d \underset{\rm lm}{\Longrightarrow} d
\end{array}
$$

Note that $D_{19}$ and $D_{20}$ are created by plugging an $\epsilon$-production into a derivation created during the current step. On the other hand, the effect of $A_1 \rightarrow \epsilon$ is null during this step, but was taken into account during Step 1.

**Step 4** All derivations in $\Delta$ except *(i)* the self-embedding derivations $D_2$, $D_8{}'$ and $D_{13}{}'$, and *(ii)* the empty derivations $D_6$, $D_{14}$, and $D_{16}$ are mapped to $\epsilon$-shifts. Note that $D_{19}$ and $D_{20}$ map to the same addendum and therefore give rise to a single rule.

**Step 5** The self-embedding derivations $D_8{}'$ and $D_{13}{}'$ are mapped to down-shifts.

Note that $D_2$ does not appear as such in the set of interpolation rules $P'$, for it is redundant with the subset $\{D_8{}', D_1, D_3, D_9\}$. On the other hand, empty derivations are not retained either, for they are redundant with the subset $\{D_2, D_3, D_{17}, D_{18}, D_{19}, D_{20}, D_{21}, D_{22}\}$.

◇

**Lemma 1** *When the above procedure is applied to produce a GIG $G'$ from a CFG $G$, all and only derivations of $G$ can be constructed by applying rules of $G'$.*

**Proof outline**

To prove that only derivations of $G$ are built, consider that all addenda in the rules of $G'$ represent derivations of $G$. Now, supposing inductively that the context to which a rule of $G'$ applies represents a derivation of $G$, it can easily be shown that the resulting context will then also represent a derivation of $G$, whether the rule applied was a down-shift or an $\epsilon$-shift.

To prove that all derivations of $G$ can be built by applying rules of $G'$, we will first of all ignore the effect of left recursion and $\epsilon$-productions, and then factor them back in. We accordingly define $\overset{*}{\underset{\text{lm!}}{\Longrightarrow}}$ to denote a sequence of zero or more leftmost derivation steps that do not involve a left-recursive cycle or $\epsilon$-production, and will explicitly denote derivation steps that may involve an $\epsilon$-production as $\underset{\text{lm}_\epsilon}{\Longrightarrow}$ .

The proof will hinge on *leftmost* derivations. However, a parse graph represents every possible derivation order; therefore, showing that all leftmost derivations of $G$ can be represented by parse graphs of $G'$ is sufficient to prove that all derivations of $G$ can.

1. For any leftmost derivation of $G$ of the form $S \overset{*}{\underset{\text{lm!}}{\Longrightarrow}} xA\alpha \overset{*}{\underset{\text{lm!}}{\Longrightarrow}} xa\beta\alpha$, if we suppose inductively that there is a GIG context that represents $S \overset{*}{\underset{\text{lm!}}{\Longrightarrow}} xA\alpha$, then, by construction (Steps 1 and 4), there is an $\epsilon$-shift of $G'$ that maps this context to one that represents $S \overset{*}{\underset{\text{lm!}}{\Longrightarrow}} xA\alpha \overset{*}{\underset{\text{lm!}}{\Longrightarrow}} xa\beta\alpha$.

2. In a 2-tiered CFG, every leftmost derivation involving a left-recursive cycle contains a subderivation of the form

$S \overset{*}{\underset{\text{lm}}{\Longrightarrow}} x_1A\alpha \overset{+}{\underset{\text{lm}}{\Longrightarrow}} x_1AB\beta\alpha \overset{+}{\underset{\text{lm}}{\Longrightarrow}} x_1x_2B\beta\alpha \overset{+}{\underset{\text{lm}}{\Longrightarrow}} x_1x_2a\gamma\beta\alpha$, with $x_1$ and $x_2$ in $\Sigma^*$, and $a$ in $\Sigma$.

Now, supposing inductively there is a GIG context corresponding to $S \overset{*}{\underset{\text{lm}}{\Longrightarrow}} x_1A\alpha \overset{+}{\underset{\text{lm}}{\Longrightarrow}} x_1x_2\alpha$, then there is a down-shift of $G'$ mapping it to every possible context representing a derivation of the form

$$S \overset{*}{\underset{\text{lm}}{\Longrightarrow}} x_1A\alpha \overset{+}{\underset{\text{lm!}}{\Longrightarrow}} x_1AB\beta\alpha \overset{+}{\underset{\text{lm}}{\Longrightarrow}} x_1x_2B\beta\alpha \overset{+}{\underset{\text{lm!}}{\Longrightarrow}} x_1x_2a\gamma\beta\alpha$$

. Now, since every elementary left-recursive cycle is captured by a down-shift (due to exhaustive derivation down to a leftmost cyclic occurrence or a terminal node), further application of down-shifts can generate all possible derivations not involving $\epsilon$-productions.

3. To prove that all cases induced by the presence of $\epsilon$-productions are captured, it is sufficient to prove that, if $S \overset{*}{\underset{\text{lm}}{\Longrightarrow}} xA\alpha$ is an $\epsilon$-free derivation of $G$ and $A \to \epsilon$ is a member of $P$, then $G'$ can build the derivation $S \overset{*}{\underset{\text{lm}}{\Longrightarrow}} xA\alpha \underset{\text{lm}_\epsilon}{\Longrightarrow} x\alpha$.

   If $A$ appears in the frontier of an addendum built in Steps 1 and 2, then Step 3, together with the fact that any $\epsilon$-free derivation of $G$ can be built by $G'$, guarantees that the effect of $A \to \epsilon$ is captured. Otherwise, there necessarily exists a derivation $B \overset{*}{\underset{\text{lm}}{\Longrightarrow}} A\beta$, and therefore $B \overset{*}{\underset{\text{lm}}{\Longrightarrow}} A\beta \underset{\text{lm}_\epsilon}{\Longrightarrow} \beta$ will necessarily be part of a derivation produced during Step 1. Eventually, this derivation will either be the topmost part of an addendum or, if it is a derivation into $\epsilon$, this derivation, $B \overset{+}{\underset{\text{lm}_\epsilon}{\Longrightarrow}} \epsilon$, behaves like an empty production and its effect is taken into account in the same way as $A \to \epsilon$. Now, since we have made the hypothesis that $G$ does not recognize the empty string, there is no infinite regression; that is to say that $A \to \epsilon$ necessarily applies within a derivation $X \overset{*}{\underset{\text{lm}_\epsilon}{\Longrightarrow}} \alpha B\beta \overset{*}{\underset{\text{lm}_\epsilon}{\Longrightarrow}} \epsilon$ such that $X$ occurs in the frontier of an addendum built during Steps 1 and 2.

   Therefore, every possible incidence of an arbitrary empty production is captured by the procedure.

□

If $G'$ builds the same leftmost derivations as $G$, the language it recognizes is strongly equivalent to the language recognized by $G$. □

# 6 Deterministic GIGs and CFGs

If, for any instantaneous description $\langle \Gamma_1, w \rangle$, where $\Gamma_1$ is a GIG context whose yield is $x$ and the string to parse is $xw$, no more than one rule is applicable, the GIG has a deterministic behavior. This behavior is interesting for its efficiency; for this reason, this section examines the generative power of deterministic GIGs.

**Theorem 5** *The intersection of CFLs not containing $\epsilon$ and deterministic GILs includes the class LL(1).*

**Proof outline** To start with, we will prove that an LL(1) grammar can be put into Greibach Normal Form without losing its LL(1) property. In order to prove this, since an LL(1) grammar contains no left recursion, it is enough to prove the following lemma.

**Lemma 2** *Let $G = \langle V, \Sigma, P, S \rangle$ be an LL(1) grammar. Let $p = A \rightarrow \alpha_1 B \alpha_2$ be a production in $P$ and $B \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$ be the set of all $B$-productions, $B \neq A$. Let $G' = \langle V, \Sigma, P', S \rangle$ be obtained from $G$ by deleting production $p$ from $P$ and adding instead the productions $A \rightarrow \alpha_1 \beta_1 \alpha_2 \mid \alpha_1 \beta_2 \alpha_2 \mid \ldots \alpha_1 \beta_n \alpha_2$. Then $G'$ recognizes the same language as $G$ and is LL(1).*

**Proof outline** To adapt a recursive descent parser that parses according to $G$ to parse according to $G'$, inline all calls to the procedure $B$ inside the procedure $A$. Since these calls are nonrecursive, the behaviour of the parser will be unaffected. □

Now, a GNF grammar is LL(1) if and only if there is no more than one $A$-production whose right-hand side begins with $a$ for given $A$ and $a$, or else more than one symbol of lookahead would be required to select an $A$-production. It is easy to see that, if this condition obtains, the construction procedure outlined in Section 3 produces a deterministic GIG equivalent to the source LL(1) grammar. Indeed, in any instantaneous description, the anchor for any rule of the type described in Section 3 is the immediate successor of the last lexeme that is present in the frontier of the context, so at most one node is eligible as the anchor; now, as at most one rule corresponds to the anchor symbol, the GIG that is constructed is deterministic. □

Beyond this admittedly rather drab result, it is somewhat arduous to characterize the class of deterministic GILs. It should be noted that this class includes some context-sensitive languages, such as the language analyzed in section 4, which is recognized by a deterministic GIG.
On the other hand, one can conjecture that the intersection of CFLs and deterministic GILs properly includes LL(1) languages from observing the sequence of context yields obtained during a GIG derivation.

**Vertical $\epsilon$-shift** A vertical $\epsilon$-shift substitutes a subgraph for a dangling node immediately following the last lexeme of $\Gamma_1$, so it maps a context whose yield is of the form $xA\alpha$ to

a context whose yield is of the form $xb\beta\alpha$. The same mapping can be obtained by a leftmost derivation; and, as the decision to take it depends on one token of lookahead, a deterministic GIG using only vertical $\epsilon$-shifts can be emulated by an LL(1) parser.

**Horizontal $\epsilon$-shift** A horizontal $\epsilon$-shift substitutes a subgraph for a node dominating the last lexeme, so the new subgraph starts with an $F$-edge, as shown on Figure 5. The yield mapping corresponding to a horizontal $\epsilon$-shift is of the form $x\alpha \Rightarrow xa\beta\alpha$, and cannot therefore be emulated by a leftmost derivation.

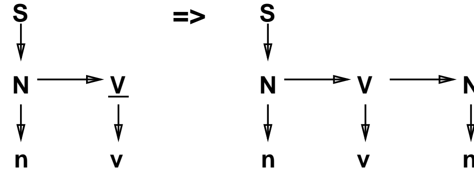


Figure 5: A derivation involving a horizontal $\epsilon$-shift

**Down-shift** Either the yield of the anchor is made up of terminal symbols, as in a down-shift that encapsulates a left-recursion, or it contains exclusively nonterminal symbols, as in the down-shift that was used in Section 4. In the first case, the mapping is of the form $xy\alpha \Rightarrow xya\beta\alpha$, where $y$ is the yield of the anchor. In the second case, the mapping is of the form $x\alpha_1\alpha_2 \Rightarrow xa\beta_1\alpha_1\beta_2\alpha_2$, where $\alpha_1$ is the yield of the anchor. Neither of these mappings can be emulated by a leftmost derivation.

**Right-shift** Either the anchor is a dangling node, as in the example that was shown on Figure 1, or it has descendants, as will be the case for the rule shown on figure 6. These cases correspond respectively to $xA\alpha \Rightarrow xa\beta\alpha$ and $x\alpha \Rightarrow xa\beta\alpha$. Only the former can be emulated by a leftmost derivation.

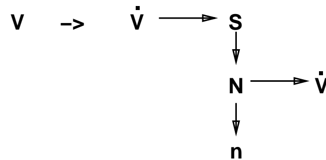


Figure 6: A right-shift that applies to an ancestor of the last lexeme

It seems possible to surmise that all yield mappings that cannot be accounted for by left derivations, when present in a deterministic GIG, are likely to allow it to recognize non

LL(1) languages; but what precedes can in no way be considered as a proof of this, just as a motivation for investigating this hypothesis.

## 7 Further research

The discussion that precedes has concentrated on context-free grammars, languages, and derivations. This can be seen as a preliminary step to determining the amount of context-sensitiveness supported by GIGs. In particular, the contribution of right-shifts and horizontal $\epsilon$-shifts to the formal power of GIGs remains to be determined.
A conclusion of this paper is that the class of deterministic graph interpolation languages is possibly too restricted to be satisfactory; so another important issue for further research is the complexity of nondeterministic GIG parsing.

## 8 Related works

It is interesting to determine how GIG-based parsing relates to other approaches to word-by-word incremental parsing, such as Link Grammars [ST95], Dependency Grammars [Mil92], or Applicative Combinatory Grammars [Mil95].
Its most distinctive feature with respect with these approaches is probably that GIG parsing has as its goal the building of a complete parse tree and each partial parse output is a representation of a parse tree. On the contrary, parse trees do underlie the dependency graphs produced by a link grammar or a dependency grammar, but the parsing process does not rely on this fact. Whether a Link Grammar or Dependency Grammar can be easily expressed as a GIG, and with what processing consequences, is a subject for further research.
On the other hand, incremental parsing based on Combinatory Grammars does not generate useful phrase structures, but incrementally builds semantic values, which constitute its real output. Consequently, meaningful comparison with GIG-parsing cannot be made until a semantic component is defined for GIGs.

## References

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation.* Addison-Wesley series in computer science. Addison-Wesley, 1979.

[JLT75] A. K. Joshi, L. S. Levy, and M. Takahashi. Tree adjunct grammars. *Journal of Computer and System Sciences*, 10:136–63, 1975.

[Lar98] John Larchevêque. Graph interpolation grammars: a rule-based approach to the incremental parsing of natural languages. Research Report RR-3390, INRIA,

Rocquencourt, France, March 1998. Available in the Computation and Language E-Print Archive under cmp-lg/9804001.

[Mil92] David Milward. Dynamics, dependency grammar, and incremental interpretation. In *COLING92: Proc. 14th Int. Conf. Comput. Ling., Nantes*, pages 1095–1099, 1992. Available from http://www.cam.sri.com/people/milward/.

[Mil95] David Milward. Incremental interpretation of categorial grammar. In *Proceedings of the 7th Conference of the European Chapter of the ACL, EACL95, Dublin*, 1995. Available from the Computation and Language archive under cmp-lg/9503015.

[ST95] Daniel D. K. Sleator and Davy Temperley. Parsing english with a link grammar. Technical Report CMU-CS-TR-91-126, Carnegie Mellon University, 1995. Available from the Computation and Language archive under cmp-lg/9508004.

[VSW94] K. Vijay-Shanker and David J. Weir. The equivalence of four extensions of context-free grammars. *Math. Systems Theory*, 27:511–546, 1994. Available from http://www.cogs.susx.ac.uk/lab/nlp/weir/weir.html.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399