YET ANOTHER $\mathcal{O}(n^6)$ RECOGNITION ALGORITHM FOR MILDLY CONTEXT-SENSITIVE LANGUAGES

Pierre BOULLIER
INRIA-Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France
E-mail: Pierre.Boullier@inria.fr

Abstract

Vijay-Shanker and Weir have shown in [17] that Tree Adjoining Grammars and Combinatory Categorial Grammars can be transformed into equivalent Linear Indexed Grammars (LIGs) which can be recognized in $\mathcal{O}(n^6)$ time using a Cocke-Kasami-Younger style algorithm. This paper exhibits another recognition algorithm for LIGs, with the same upper-bound complexity, but whose average case behaves much better. This algorithm works in two steps: first a general context-free parsing algorithm (using the underlying context-free grammar) builds a shared parse forest, and second, the LIG properties are checked on this forest. This check is based upon the composition of simple relations and does not require any computation of symbol stacks.

Keywords: context-sensitive parsing, ambiguity, parse tree, shared parse forest.

1 Introduction

It is well known that natural language processing cannot be described by purely context-free grammars (CFGs). On the other hand, general context-sensitive formalisms are powerful enough but cannot be parsed in reasonable time. Therefore, various intermediate frameworks have been investigated, the trade-off being between expressiveness power and computational tractability. One of these formalism classes is the so-called mildly context-sensitive languages which can be described by several equivalent grammar types. Among these types, Tree Adjoining Grammars (TAGs) are attractive because they can express some natural language phenomena (see Abeillé and Schabes [1]) and many systems are based upon this framework (see for example [10] and [6]). Formal properties of TAGs have been studied (see Vijay-Shanker and Joshi [16], and Vijay-Shanker [15]) and a recognizer for TAGs (see [17]), based upon a Cocke-Kasami-Younger method ([4] and [18]), works in $\mathcal{O}(n^6)$ worst time. Unfortunately, with this algorithm, this complexity is always reached. More practical methods, which are usually based upon the Earley parsing algorithm [3], have also been investigated (see for example Schabes [13] and Poller [11]). Though the $\mathcal{O}(n^6)$ worst-case time is not improved, for some inputs, the actual complexity may be much better. However, the design of a better worst-case recognizer remains an open problem.

In [17], Vijay-Shanker and Weir have shown that mildly context-sensitive grammars have the same formal power and that TAGs and Combinatory Categorial Grammars (CCGs) can be transformed into equivalent Linear Indexed Grammars (LIGs).

An Indexed Grammar [2] is a CFG in which each object is a non-terminal associated with a stack of symbols. The productions of this grammar class define on the one hand a derived relation in the usual sense and, on the other hand, the way symbols are pushed or popped on top of the stacks which are associated with each non-terminal. A restricted form of Indexed Grammar called LIG allows only for the stack associated with the left-hand side (LHS) non-terminal of a given production to be associated with at most one non-terminal in the right-hand side (RHS).

This paper presents a new recognition algorithm for LIGs. It works in two main steps:

1. a CF-parsing algorithm, working on the underlying CF-grammar, builds a shared parse forest;
2. the LIG conditions (see Section 3) are checked on that forest.

Since that second step does not depend on the way the shared parse forest is built, any general CF-parsing algorithm can be used in the first step. As previously mentioned, CKY or Earley parsing

algorithms are candidates but others too. In particular, generalized LR parsing methods (see Lang [7], Tomita [14], and Rekers [12]) are good challengers. In fact, the CF-parsing algorithm of our prototype system upon which this paper ideas have been tested, implements a non-deterministic LR (or LC at will) parsing algorithm using a graph-structured stack. Under certain conditions, for a given input string of length n, the CF-parsing takes a time $\mathcal{O}(n^3)$ in the worst case and moreover the output shared parse forest of size $\mathcal{O}(n^3)$ (in the worst case) is such that each elementary tree can be seen as an occurrence (after some non-terminal renaming) of a production in the underlying CFG. Following Lang [8], in Section 2, we stress this analogy by defining a shared parse forest as a CFG.

Obviously, the checking of the LIG properties can be performed on a forest by computing the stacks of symbols along paths, and in associating with each (shared) node a set of such stacks. As in [17], in devising an elaborate sub-stack sharing mechanism, this check could be performed in $\mathcal{O}(n^6)$ time. In Section 4, we take a different approach: this check is performed without the computation of any stack of symbols (and hence without having to design any sub-stack sharing structure). Given a single tree of the (unfolded) shared parse forest, we identify *spines* as being paths along which individual stack of symbols are evaluated. The origin of such a spine corresponds to the birth of a stack which evolves according the LIG stack schemas and which finally vanishes at the end of the spine. The checking of LIG conditions relies on the simple observation that, for a given spine, the stack actions must be bracketed. Each time a push or pop occurs at a node, there is a twin node where the opposite action, acting on the same symbol at the same stack level, should take place. In a shared parse forest, different spines may share nodes. In particular, a given couple of twin nodes may be shared among several spines, with the corresponding check being done only once. In Section 5 we show that this check sharing, expressed as relations between twin nodes, results in a worst case $\mathcal{O}(n^6)$-time LIG recognition. Our algorithm is illustrated by an example in Section 6. Since TAGs or CCGs can be transformed into equivalent LIGs [17], this complexity extends over mildly context-sensitive languages.

2 Parse Tree and Shared Parse Forest

The goal of this section is to set up the vocabulary and to define our vision of shared parse forests.

Let $G = (V_N, V_T, P, S)$ be a CFG where:

- V_N is a non-empty finite set of *non-terminal* symbols.
- V_T is a finite set of *terminal* symbols; V_N and V_T are disjoint; $V = V_N \cup V_T$ is the *vocabulary*.
- S is an element of V_N called the *start symbol*.
- $P \subseteq V_N \times V^*$ is a finite set of productions. Each production is denoted by $A \rightarrow \sigma$ or by $r_p, 1 \leq p \leq |P|$; such a production is called an A-production.

We adopt the convention that A, B, C denote non-terminals, a, b, c denote terminals, w, x denote elements of V_T^*, X denotes elements of V, and β, σ denote elements of V^*.

On V^* we define $|P|$ disjoint binary relations named *right derive by* $B \rightarrow \beta$[1] and denoted by $\underset{G}{\overset{B\rightarrow\beta}{\Rightarrow}}$ (or simply $\overset{B\rightarrow\beta}{\Rightarrow}$ when G is understood) as the set $\{(\sigma B x, \sigma\beta x) \mid B \rightarrow \beta \in P\}$.

The relation *derive* denoted by $\Rightarrow$ is defined by:

$$\Rightarrow \;=\; \bigcup_{B\rightarrow\beta\in P} \overset{B\rightarrow\beta}{\Rightarrow}$$

Let $\sigma_1, \ldots, \sigma_i, \sigma_{i+1}, \ldots, \sigma_l$ be strings in V^* such that $\forall i, 1 \leq i < l, \exists r_p \in P, \sigma_i \overset{r_p}{\Rightarrow} \sigma_{i+1}$ then the sequence of strings $(\sigma_1, \ldots, \sigma_i, \sigma_{i+1}, \ldots, \sigma_l)$ is called a *derivation*. Conversely, since $\overset{r_p}{\Rightarrow}$ and $\overset{r_q}{\Rightarrow}$ are disjoint when p and q are different, between any two consecutive strings σ_i and σ_{i+1} in a derivation, the relation $\overset{r_p}{\Rightarrow}$ whose (σ_i, σ_{i+1}) is an element is uniquely known.

[1] In the sequel the qualifier *right* will disappear since only right derivations, right sentential forms, etc ... are introduced.

A *σ-derivation* is a derivation starting with σ. A σ-derivation whose last element in the sequence is β is called a *σ/β-derivation*. The elements of a σ-derivation are called a *σ-phrase*. A σ-phrase in V_T^* is a *σ-sentence*. On the other hand an S-phrase is a *sentential form* and an S-sentence is a *sentence*.

The language defined by G is the set of its sentences:

$$\mathcal{L}(G) = \{x \mid S \underset{G}{\overset{+}{\Rightarrow}} x \wedge x \in V_T^*\}$$

In an S/x-derivation, to accurately define the contribution of any symbol X (its X-sentence) to the sentence x, we will define the notion of split of x by X.

A triple (x_1, x_2, x_3) is called *3-split* (or more simply *split*) of x when $x = x_1x_2x_3$. If n is the length of x, such a triple can also be denoted by $x_{i..j}$ with $0 < i \leq j \leq n+1$, $|x_1| = i-1$, $|x_2| = j-i$, and $|x_3| = n-j+1$. Two splits of x, say (x'_1, x'_2, x'_3) and (x''_1, x''_2, x''_3) can be composed into one split iff we have $x'_1x'_2 = x''_1$. In such a case the resulting split of x is $(x'_1, x'_2x''_2, x''_3)$. Assume that $x_{i..j}$ denotes (x'_1, x'_2, x'_3) and that $x_{k..l}$ denotes (x''_1, x''_2, x''_3), the previous condition simply means that $j = k$ and that the composed split is denoted by $x_{i..l}$. We allow x itself to designate the split $(\varepsilon, x, \varepsilon) = x_{1..|x|+1}$.

We call *split of x by X* the couple $(X, (x_1, x_2, x_3))$ if there is an S/x-derivation $S \overset{*}{\Rightarrow} \sigma X x_3 \overset{*}{\Rightarrow} \sigma x_2 x_3 \overset{*}{\Rightarrow} (x_1x_2x_3 = x)$. This couple, when x is understood, could be denoted by $[X]_i^j$ if $(x_1, x_2, x_3) = x_{i..j}$ or even by $[X]$ when the split of x is not necessary. This definition and notations extends from symbols to strings.

Now we can define our vision of parse trees.

Definition 1 *Let $G = (V_N, V_T, P, S)$ be a CFG, x a sentence in $\mathcal{L}(G)$, and d^x an S/x-derivation. We call parse tree (w.r.t. G and d^x) the CFG $G^{d^x} = (V_N^x, V_T^x, P^{d^x}, S^x)$ where:*

- $V_N^x = \{(B, (x_1, x_2, x_3)) \mid B \in V_N \wedge x = x_1x_2x_3\}$.
- $V_T^x = \{(a, (x_1, a, x_3)) \mid a \in V_T \wedge x = x_1ax_3\}$.
- $S^x = [S] = (S, (\varepsilon, x, \varepsilon))$.
- $P^{d^x} = \{[B] \rightarrow [X_1]\ldots[X_k]\ldots[X_p] \mid B \rightarrow X_1\ldots X_k\ldots X_p \in P\}$ *and* $d^x = S \overset{*}{\Rightarrow} \sigma B x_3 \Rightarrow \sigma X_1\ldots X_k\ldots X_p x_3 \overset{*}{\Rightarrow} \sigma X_1\ldots X_{k-1}x_2^k\ldots x_2^p x_3 \overset{*}{\Rightarrow} \sigma x_2^1\ldots x_2^k\ldots x_2^p x_3 \overset{*}{\Rightarrow} x$ *where:* $x = x_1x_2x_3$, $x_2 = x_2^1\ldots x_2^k\ldots x_2^p$, $[X_k] = (X_k, (x_1x_2^1\ldots x_2^{k-1}, x_2^k, x_2^{k+1}\ldots x_2^p x_3))$, *and* $[B] = (B, (x_1, x_2, x_3))$.

Parse trees are trees in which the start symbol S^x is the root, non-terminal symbols are the internal nodes while terminal symbols are the leaves. Obviously we have $\mathcal{L}(G^{d^x}) = \{[x]\}$. In fact, for any two consecutive strings $\sigma B x_3$ and $\sigma\beta x_3$ in d^x, we have $[\sigma][B][x_3] \underset{G^{d^x}}{\overset{[B]\rightarrow[\beta]}{\Rightarrow}} [\sigma][\beta][x_3]$. It is easy to see that this definition of a parse tree is a tree in the usual sense only when the derivation d^x does not involve any cycle (i.e. $\nexists A, A \overset{+}{\Rightarrow} A$). If there is a cycle, our definition denotes, by a single parse tree, the ambiguities denoted by the unbounded number of (usual) trees when this cycle is taken 0, 1, 2, ... times. The whole notion of ambiguity will be captured by the following definition of shared parse forest.

Definition 2 *Let $G = (V_N, V_T, P, S)$ be a CFG, and x a sentence in $\mathcal{L}(G)$. The shared parse forest for x (w.r.t. G) is the CFG, $G^x = (V_N^x, V_T^x, P^x, S^x)$ where:*

- $V_N^x = \{(B, (x_1, x_2, x_3)) \mid B \in V_N \wedge x = x_1x_2x_3\}$.
- $V_T^x = \{(a, (x_1, a, x_3)) \mid B \in V_T \wedge x = x_1ax_3\}$.
- $S^x = (S, (\varepsilon, x, \varepsilon))$.
- $P^x = \bigcup_{d^x \in D^x} P^{d^x}$ *where D^x is the set of all S/x-derivations, and P^{d^x} is the production set of the parse tree $G^{d^x} = (V_N^x, V_T^x, P^{d^x}, S^x)$ associated with any derivation d^x in D^x.*

Any production $r_p = [B] \rightarrow [X_1] \ldots [X_q]$ in P^x is mapped by the unary operator $^-$ to its associated production $\overline{r_p} = B \rightarrow X_1 \ldots X_q$ in P.

This vision of a set of parse trees as a CFG has several formal and practical advantages (thanks to [9]). It exhibits a particular case of a general result: the intersection of CF-languages (defined by G) and regular languages (the input string x) are CF-languages (the resulting shared parse forest G^x). G^x can also be seen as a specialization of G (productions in G^x are productions in G, up to some renaming), which only defines (in all the same possible ways as G) the string x. This CFG allows to define an unbounded number of derivations (when G is cyclic) in a finite way. A *context sharing* occurs when there are several occurrences of the same non-terminal in RHSs, while a *sub-tree sharing* occurs when there are several occurrences of the same non-terminal in LHSs. This sharing may even be considered as optimal if we impose (as done here) that productions (elementary trees) in a parse tree, have the same structure as their corresponding production in G.

Without any restriction on G, the size of P^x is $\mathcal{O}(n^{l+1})$ where l is the length of the longest RHS in P. If G is unambiguous, (or if the parsing of x does not exhibit any ambiguity,) this size is linear in n.

3 Linear Indexed Grammars (LIGs)

An indexed grammar is a CFG in which stack of symbols are associated with non-terminals. The derive relation, in addition to its usual meaning, handles these stacks of symbols. LIGs are a restricted form of indexed grammars in which the stack associated with the non-terminal in the LHS of any production is associated with at most one non-terminal in the RHS. Other non-terminals are associated with stacks of bounded size.

In fact, in a production, it is not a stack which is associated with a non-terminal, but rather a stack schema expressing a way to compute a stack. Let V_l denotes a finite set of (stack) symbols, a stack is an element of V_l^*. A stack schema is an element of $V_b \times V_l^*$ where $V_b = \{\varepsilon, ..\}$. The stack schema $(..\alpha)$ where $\alpha \in V_l^*$ matches all the stacks whose prefix (bottom) part is left unspecified and whose suffix (top) part is α. A stack may be considered as a stack schema whose first component (the element of V_b) is ε.

Following [17], we formally defined a LIG as follows:

Definition 3 *A LIG, L is denoted by (V_N, V_T, V_l, P_L, S) where:*

- *V_N is a non-empty finite set of non-terminal symbols.*
- *V_T is a finite set of terminal symbols, V_N and V_T are disjoint, and $V = V_N \cup V_T$ is the vocabulary.*
- *V_l is a finite set of stack symbols.*
- *P_L, the production set, is a finite subset of $(V_N \times V_b \times V_l^*) \times ((V_N \times V_b \times V_l^*) \cup V_T)^*$.*
- *$S \in V_N$ is the start symbol.*

We adopt the convention that α will denote members of V_l^*, π elements of V_b, and γ elements of V_l.

A triple (A, ε, α) in $V_N \times V_b \times V_l^*$ is called a *secondary object* and is denoted by $A(\alpha)$ while a triple $(A, .., \alpha)$ is called a *primary object* and is denoted by $A(..\alpha)$. The disjoint sets of primary and secondary objects are respectively denoted by V_O^P and V_O^S. The set of objects denoted V_O is $V_O^P \cup V_O^S$. The object $A(\pi\alpha)$, whose non-terminal component part is A, is called an A-object. We use Γ to denote strings in $(V_O \cup V_T)^*$. $A(..\alpha)$ denotes an object whose stack suffix (stack top) is α and with an arbitrary prefix (stack bottom). $A()$ denotes that an empty stack schema is associated with the non-terminal A. $A(\alpha)$ denotes that the stack α is associated with the non-terminal A. Each production in P_L is denoted by $A(\pi\alpha) \rightarrow \Gamma$ or $r_p()$[2] where $1 \leq p \leq |P_L|$.

The general form of a production in a LIG is:

$$r_p() = A(\pi\alpha) \rightarrow w_1 A_1(\alpha_1) \ldots w_{i-1} A_{i-1}(\alpha_{i-1}) w_i A_i(\pi\alpha_i) w_{i+1} A_{i+1}(\alpha_{i+1}) \ldots w_p A_p(\alpha_p) w_{p+1}$$

[2]The parentheses reminds us that we are in a LIG!

If the LHS object $A(\pi\alpha)$ is secondary (i.e. $\pi = \varepsilon$), we observe that all the objects (if any) in the RHS should also be secondary, while if $A(\pi\alpha)$ is primary (i.e. $\pi = ..$), there must be exactly one primary object in the RHS.

The above production is called an A-production. If this production is used, for any $\Gamma_1, \Gamma_2 \in (V_O \cup V_T)^*$ and $\alpha' \in V_l^*$, we define the binary relation *derive by* $r_p()$ on LIGs by:

$$\Gamma_1 A(\alpha'\alpha)\Gamma_2 \overset{r_p()}{\Rightarrow} \Gamma_1 w_1 A_1(\alpha_1) \ldots w_{i-1}A_{i-1}(\alpha_{i-1})w_i A_i(\alpha'\alpha_i)w_{i+1}A_{i+1}(\alpha_{i+1}) \ldots w_p A_p(\alpha_p)w_{p+1}\Gamma_2$$

when $\pi = .. \vee \alpha' = \varepsilon$.

We observe that the stack $\alpha'\alpha$ associated with the non-terminal A in the LHS and the stack $\alpha'\alpha_i$ associated with the non-terminal A_i in the RHS have the same prefix α'.

Definition 4 *We define the CF-backbone of a LIG as being its underlying CFG. Formally, if* $L = (V_N, V_T, V_l, P_L, S)$ *is a LIG, its CF-backbone is the CFG,* $G_L = (V_N, V_T, P_G, S)$, *or simply* G *when* L *is understood, where:*

$$P_G = \{A \rightarrow w_1 A_1 \ldots w_{i-1}A_{i-1}w_i A_i w_{i+1}A_{i+1} \ldots w_p A_p w_{p+1} \mid A(\pi\alpha) \rightarrow w_1 A_1(\alpha_1) \ldots w_{i-1}A_{i-1}(\alpha_{i-1})w_i A_i(\pi\alpha_i)w_{i+1}A_{i+1}(\alpha_{i+1}) \ldots w_p A_p(\alpha_p)w_{p+1} \in P_L\}.$$

If there is a one to one mapping between P_L and P_G the LIG is said to be *fair*. It is not very difficult to find an algorithm which transforms any LIG into an equivalent fair LIG. In the sequel we will only consider fair LIGs.

Due to the one to one mapping between (fair) LIGs and their CF-backbones we assume that iff $r_p()$ is a production in P_L, then r_p, with the same index p, denotes the corresponding production in its CF-backbone P_G.

Let $L = (V_N, V_T, V_l, P_L, S)$ be a LIG, $G = (V_N, V_T, P_G, S)$ its CF-backbone, x a string in $\mathcal{L}(G)$, and $G^x = (V_N^x, V_T^x, P_G^x, S^x)$ its shared parse forest for x. Consider the LIG $L^x = (V_N^x, V_T^x, V_l, P_L^x, S^x)$ s.t. G^x is its CF-backbone and each stack schema $(\pi_k\alpha_k)$ associated with the non-terminal $[A_k]$, occurring at position k in production $r_p() \in P_L^x$ is the stack schema of the object at position k in $\overline{r_p}() \in P_L$. More formally we have:

$$P_L^x = \{r_p() = [A_0](\pi_0\alpha_0) \rightarrow [w_1][A_1](\pi_1\alpha_1) \ldots [w_k][A_k](\pi_k\alpha_k) \ldots [w_{m+1}] \mid r_p = [A_0] \rightarrow [w_1][A_1] \ldots [w_k][A_k] \ldots [w_{m+1}] \in P_G^x \wedge \overline{r_p}() = A_0(\pi_0\alpha_0) \rightarrow w_1 A_1(\pi_1\alpha_1) \ldots w_k A_k(\pi_k\alpha_k) \ldots w_{m+1} \in P_L\}$$

L^x is called the *LIGed forest* for x.

By construction, any LIGed forest is fair and between a LIG L and its LIGed forest L^x for x, we have $x \in \mathcal{L}(L) \iff x \in \mathcal{L}(L^x)$. (Recall that x can designate the split $(\varepsilon, x, \varepsilon)$).

An object is said *initial* (resp. *final*) if it is secondary and occurs in the RHS (resp. LHS) of a production. V_O^I (resp. V_O^F) denotes the set of initial (resp. final) objects.

Definition 5 *For a given LIGed forest for* x, *we call spine, any sequence of* $2p$ $(1 \leq p)$ *objects* $(o_1, o_2, \ldots, o_{2i-1}, o_{2i}, o_{2i+1}, \ldots, o_{2p})$ *such that:*

- o_1 *(resp.* o_{2p}*) is an initial (resp. final) object.*
- *Inside objects* o_j *(if any)* $(\forall j, 1 < j < 2p)$ *are primary.*
- $\forall i, 1 \leq i \leq p$, *two consecutive objects* $o_{2i-1} = X_1(\pi_1\alpha_1)$, *and* $o_{2i} = X(\pi\alpha)$ *are such that* $X_1 = X$, *and* o_{2i-1} *(resp.* o_{2i}*) occurs in the RHS (resp. LHS) of a* P_L^x *production.*

This notion of spine is fundamental in LIG theory since it represents a path upon which stacks of symbols are evaluated. For example, followed in the direct way (top-down), the spine $(o_1 = X_1(\alpha_1), o_2 = X_1(..\alpha_1'), \ldots, o_{2i-1} = X_i(..\alpha_i), o_{2i} = X_i(..\alpha_i'), \ldots, o_{2p} = X_p(\alpha_p'))$ indicates that:

- a stack s is created and initialized with α_1 on the initial object o_1;

- if α'_1 is a suffix of s, then α'_1 is popped from s on object o_2;

⋮

- the string of symbols α_i is pushed on s on object o_{2i-1};
- if α'_i is a suffix of s, then α'_i is popped from s on object o_{2i};

⋮

- on the final object o_{2p}, if α'_p is a suffix of s, then α'_p is popped from s and the stack s is checked for emptiness.

A spine is said to be *valid* if each check sketched above succeeds[3].

4 Our LIG Recognition Algorithm

In this paper we restrict our attention to LIGs with the following characteristics:

1. the RHS of a production contains at most two symbols;
2. the stack schema ($\pi\alpha$) of any object (primary or secondary) is such that $0 \leq |\alpha| \leq 1$.

Recall that our recognition algorithm works on shared parse forests. Therefore, it is assumed that such a forest has been built by any general CF-parsing algorithm, working on the associated CF-backbone grammar, with a string x as input.

The reason why we allow at most two symbols in the RHS of the CF-backbone is to build the forest in time $\mathcal{O}(n^3)$. Moreover, in such a case, the parameters of the shared parse forest are kept within some suitable upper bounds: in particular the number of productions is $\mathcal{O}(n^3)$, the number of non-terminal symbols is $\mathcal{O}(n^2)$, the number of X-productions for any given $X = (A, x_{i..j})$ is $\mathcal{O}(n)$ and the number of occurrences of such a non-terminal symbol X in the RHSs is also $\mathcal{O}(n)$.

The restriction on stack schemas, have been chosen only for pedagogic facilities. This restriction does not change neither our algorithm principle nor its upper bound complexity. Moreover, it is easy to see that this form of LIG constitutes a normal form.

We will restrict our attention to non-cyclic CF-backbones. This restriction will guarantee that in any parse (sub-)tree, internal nodes are different from the root node. Nevertheless, this restriction is not mandatory and slight modifications of our algorithms allow to also handle cyclic grammars without changing their complexities.

Contrary to the previous section where we saw that a stack of symbols can be evaluated along spines, we choose not to compute stacks explicitly. The idea of our algorithm is based upon the remark that each time a symbol γ is pushed on a stack at a given place, this very symbol should be popped at some other place. The converse should also be true. The following will exhibit a mean by which this property could be checked without explicitly computing neither stacks nor spines.

We could remark that we are not interested in finding all the valid spines between any pair of objects (o_1, o_2), but only if there is at least one such valid spine. As a first consequence we will only consider *abridged spines* (a-*spine* for short) $(o_1, o_2, o_4, \ldots, o_{2i}, o_{2i+2}, \ldots, o_{2p})$ which summarize all the spines $(o_1, o_2, o_3, o_4, \ldots, o_{2i}, o_{2i+1}, o_{2i+2}, \ldots, o_{2p})$ where the RHSs (odd) objects (except the initial one) have been erased. If the length of a spine is $2p$, we see that the length of its a-spine is $p + 1$.

The first purpose of our algorithm is to compute the relation *valid spine* denoted by $\bowtie$ and which is the set of all couples (o_1, o_2) s.t. o_1 is an initial object, o_2 is a final object, and there is at least one valid spine between o_1 and o_2.

In order to reach this goal, for a given LIGed forest for x, we define on its objects V_O, $2|V_l| + 1$ binary relations noted (for some γ in V_l) $\overset{\gamma}{\prec}$, $\overset{\gamma}{\succ}$, and $\multimap\!\!\diamond\!\!\multimap$. These relations between objects indicate the evolution of an imaginary stack between the first and the second object.

[3]Of course it is possible to adopt the dual vision and to evaluate stacks along spines in the opposite (bottom-up) way. A stack is created and initialized with α'_p on the final object o_{2p}. Elements are pushed on LHS objects while they are checked and popped on RHS objects, and finally α_1 is popped on o_1 and the stack is checked for emptiness.

The element (o_1, o_2) of $\overset{\gamma}{\prec}$ (resp. $\overset{\gamma}{\succ}$) means that the stack associated with o_2 is built by pushing γ (resp. popping γ if possible) on top of the stack associated with o_1. The element (o_1, o_2) of $\multimap$ means that the stacks associated with o_1 and o_2 are identical.

Let $[X_1](\pi_1\alpha_1) \rightarrow \ldots [X_2'](\pi_2'\alpha_2') \ldots$ and $[X_2](\pi_2\alpha_2) \rightarrow \Gamma$ be two productions in P_L^x with $[X_2'] = [X_2]$. Moreover, assume that o_1, o_2', and o_2 respectively denotes the objects $[X_1](\pi_1\alpha_1)$, $[X_2'](\pi_2'\alpha_2')$, and $[X_2](\pi_2\alpha_2)$. The Table 1 indicates precisely the way these relations are defined. All other couples of objects are non comparable.

π_1	π_2'	π_2	Conditions	Relations
any	ε	ε	$\alpha_2' = \alpha_2$	$o_2' \bowtie o_2$
any	ε	..	$\alpha_2' = \alpha_2$	$o_2' \multimap o_2$
any	ε	..	$\alpha_2' = \gamma \wedge \alpha_2 = \varepsilon$	$o_2' \overset{\gamma}{\prec} o_2$
..	..	any	$\alpha_2' = \alpha_2$	$o_1 \multimap o_2$
..	..	any	$\alpha_2' = \gamma \wedge \alpha_2 = \varepsilon$	$o_1 \overset{\gamma}{\prec} o_2$
..	..	any	$\alpha_2' = \varepsilon \wedge \alpha_2 = \gamma$	$o_1 \overset{\gamma}{\succ} o_2$

Table 1: $\overset{\gamma}{\prec}$, $\overset{\gamma}{\succ}$, and $\multimap$ definitions.

Our algorithm will simply compose the previous relations in order to relate an object where a symbol is pushed to the object(s) where this very symbol is popped in order to finally answer the question: is there at least one valid spine between o_1 and o_2 where o_1 is initial and o_2 is final?

Formally the valid spine relation is defined by $\bowtie = \{(o_1, o_2) \mid o_1 \in V_O^I \wedge o_2 \in V_O^F \wedge o_1 \overset{+}{\approx} o_2\}$ where the $\approx$ relation is the smallest solution of the set of recursive equations $\approx = \multimap$ and $\approx = \overset{\gamma}{\prec} \overset{*}{\approx} \overset{\gamma}{\succ}$.

We will implement this computation as a limit of the composition of the $\overset{\gamma}{\prec}$, $\multimap$, and $\overset{\gamma}{\succ}$ relations and we will show that our algorithm has an $\mathcal{O}(n^6)$-time upper bound complexity.

The laws governing this composition are shown in Table 2 where o_1 and o_3 are any sorts of objects and o_2 always designates a primary object[4].

These composition rules are applied until no more new element can be added to any of these relations.

$o_1 \multimap o_2$	and	$o_2 \multimap o_3$	and	$o_1 \in V_O^I \wedge o_3 \in V_O^F$	$\Longrightarrow$	$o_1 \bowtie o_3$
$o_1 \overset{\gamma}{\prec} o_2$	and	$o_2 \overset{\gamma}{\succ} o_3$	and	$o_1 \in V_O^I \wedge o_3 \in V_O^F$	$\Longrightarrow$	$o_1 \bowtie o_3$
$o_1 \multimap o_2$	and	$o_2 \multimap o_3$	and	$o_1 \notin V_O^I \vee o_3 \notin V_O^F$	$\Longrightarrow$	$o_1 \multimap o_3$
$o_1 \overset{\gamma}{\prec} o_2$	and	$o_2 \overset{\gamma}{\succ} o_3$	and	$o_1 \notin V_O^I \vee o_3 \notin V_O^F$	$\Longrightarrow$	$o_1 \multimap o_3$
$o_1 \overset{\gamma}{\prec} o_2$	and	$o_2 \multimap o_3$			$\Longrightarrow$	$o_1 \overset{\gamma}{\prec} o_3$
$o_1 \multimap o_2$	and	$o_2 \overset{\gamma}{\succ} o_3$			$\Longrightarrow$	$o_1 \overset{\gamma}{\succ} o_3$

Table 2: Valid Composition of relations.

[4] If unrestricted stack schemas have been used, for example, the composition of $\overset{\alpha_1}{\prec}$ and $\overset{\alpha_2}{\succ}$ would have led to three possibilities, depending upon the stack suffixes α_1 and α_2, namely $\multimap$ if $\alpha_1 = \alpha_2$, $\overset{\alpha_2'}{\prec}$ if $\alpha_1 = \alpha_2'\alpha_2$, and $\overset{\alpha_1'}{\succ}$ if $\alpha_1'\alpha_1 = \alpha_2$.

If an initial object o_1 and a final object o_2 are such that $o_1 \bowtie o_2$, this means that there is (at least) one valid spine between these objects. Conversely if there are initial objects with no corresponding final object (in $\bowtie$), or final objects with no initial object, this means that there is no valid spine starting (or ending) at that object and that the productions where these objects occur are invalid w.r.t. the LIG conditions and therefore should be erased. This erasing of productions in the LIGed forest L^x for x, creates a new LIG say $\underline{L^x}$.

The string x is an element of the initial LIG L iff the language of the CF-backbone for $\underline{L^x}$ is non empty.

In order to facilitate the evaluation of our algorithm complexity we will add a parameter k to the previous relations $\diamond$, $\overset{\gamma}{\prec}$, and $\overset{\gamma}{\succ}$. The value $k = 1$ is assigned to the initial relations defined in Table 1, and the k-relations are achieved by composing 1-relations and $(k-1)$-relations. In fact this value k expresses the existence of sub-strings of length $k+1$ in a-spines.

The procedure in Table 3 implements the definition of the level 1 relations given in Table 1.

```
(1)   procedure 1-relations ()
(2)     V_O^LHS = {o | o → Γ ∈ P_L^x}
(3)     for each o' = [X](π'α') in V_O^LHS do
(4)       for each o → ... [X](π2α2) ... in P_L^x ∪ {S'() → S^x()} do
(5)         if π2 = ε then o = [X](π2α2) end if
(6)         if π2 = ε and π' = ε then I = I ∪ {o}, F = F ∪ {o'}
(7)         else if α2 = α' then ◇=◇ ∪{(o, o')}
                                 1 1
(8)         else if α2 = γ and α' = ε then ≺=≺ ∪{(o, o')}
                                           1 1
(9)         else if α2 = ε and α' = γ then ≻=≻ ∪{(o, o')}
                                           1 1
(10)        end if
(11)      end do
(12)    end do
(13)  end procedure
```

Table 3: The 1-relations $\overset{\gamma}{\underset{1}{\prec}}$, $\overset{\gamma}{\underset{1}{\succ}}$, and $\underset{1}{\diamond}$.

Line (2) collects in V_O^{LHS} the LHS objects. The loop at lines (3–12) examines each such LHS object o' which is supposed to be an X-object. The embedded loop at lines (4–11) selects the productions with an X-object in RHS. Note that we have added a new production $S'() \rightarrow S^x()$ which introduces a new initial object $S^x()$ called the *start* object. This augmented LIG and its start object allow us to handle spines whose initial object non-terminal symbol is the LIG start symbol. The first component of a relation is an LHS object o, except when the RHS object $[X](\pi_2\alpha_2)$ is secondary (and therefore initial), this case is processed at line (5). The choice of the relations is governed by the relative values of the stack schemas $(\pi_2\alpha_2)$ and $(\pi'\alpha')$. Instead of building up the $\bowtie$ relation, we choose to build I (resp. F) which is the set of valid initial (resp. final) objects. At line (6), when o and o' are secondary (o is initial and o' is final), they are respectively put into I and F. Lines (7–9) select the appropriate valid level 1 relation. The case where $\alpha = \gamma$, $\alpha' = \gamma'$, and $\gamma \neq \gamma'$ (push of γ immediately followed by a pop of γ') is erroneous.

The function in Table 4 describes the way the level k relations are computed from the level 1 and level $k-1$ relations. If a couple (o_1, o_3) is a member of a level k relation, this means that there is at least one string of length $k+1$, starting at o_1 and ending at o_3, which is a valid sub-string of an a-spine.

The loop body at lines (2–31) is executed twice[5]. Complete valid a-spines may only be reached by composing $\underset{h}{\diamond}$ and $\underset{k-h}{\diamond}$ relations at line (7) or by composing $\overset{\gamma}{\underset{h}{\prec}}$ and $\overset{\gamma}{\underset{k-h}{\succ}}$ relations at line (23). All other

[5]only once when $k = 2$.

valid compositions, as stated in Table 2, participate in the level k relations. A new couple of objects is entered into its level k relation at lines (9, 13, 20, or 25) only if this element is not already a member of the same relation at level h with $h \leq k$. Though this condition can only be seen here as an optimization, it is mandatory when cyclic grammars are considered. This function returns true iff one of its level k relation is not empty (i.e. there is sub-strings of length $k+1$ which are not yet complete valid a-spines).

```
(1)  function k-relations(k) return boolean
(2)    for each h in {1, k − 1} do
(3)      for each (o1, o2) in ◇_h do
(4)        if o2 in V_O^P then
(5)          for each (o2, o3) in ◇_{k−h} do
(6)            if o1 in V_O^S and o3 in V_O^S then
(7)              I = I ∪ {o1}, F = F ∪ {o3}
(8)            else
(9)              ◇_k = ◇_k ∪{(o1, o3)}
(10)           end if
(11)         end do
(12)         for each γ in V_l do
(13)           for each (o2, o3) in ≻^γ_{k−h} do ≻^γ_k = ≻^γ_k ∪{(o1, o3)} end do
(14)         end do
(15)       end if
(16)     end do
(17)     for each γ in V_l do
(18)       for each (o1, o2) in ≺^γ_h do
(19)         if o2 in V_O^P then
(20)           for each (o2, o3) in ◇_{k−h} do ≺^γ_k = ≺^γ_k ∪{(o1, o3)} end do
(21)           for each (o2, o3) in ≻^γ_{k−h} do
(22)             if o1 in V_O^S and o3 in V_O^S then
(23)               I = I ∪ {o1}, F = F ∪ {o3}
(24)             else
(25)               ◇_k = ◇_k ∪{(o1, o3)}
(26)             end if
(27)           end do
(28)         end if
(29)       end do
(30)     end do
(31)   end do
(32)   return ⋃_γ ≺^γ_k ∪ ⋃_γ ≻^γ_k ∪ ◇_k ≠ ∅
(33)  end function
```

Table 4: The k-relations $\overset{\gamma}{\prec}_k$, $\overset{\gamma}{\succ}_k$, and $\diamond_k$.

The main function which describes our recognizing algorithm is in Table 5.

Its parameters are a LIG L and an input string x. At line (3), G denotes its CF-backbone. The shared parse forest G^x at line (4) is supposed to have been computed by any general CF-parsing algorithm. If $x \notin \mathcal{L}(G)$, it will not be in $\mathcal{L}(L)$ either (line (5)). At line (6), L^x denotes the corresponding LIGed forest. The sets I and F, which are going to hold the initial and final valid objects, are initialized to

(1) **function** *recognize* (L, x) **return boolean**
(2) **let** $L = (V_N, V_T, V_I, P_L, S)$
(3) **create** $G = (V_N, V_T, P_G, S)$ /* its CF-backbone */
(4) **create** $G^x = (V_N^x, V_T^x, P_G^x, S^x)$ /* its shared parse forest for x */
(5) **if** $\mathcal{L}(G^x) = \emptyset$ **then return false end if**
(6) **create** $L^x = (V_N^x, V_T^x, V_I, P_L^x, S^x)$ /* its LIGed forest */
(7) $I = F = \emptyset$
(8) $\overset{\gamma}{\underset{1}{\prec}} = \overset{\gamma}{\underset{1}{\succ}} = \underset{1}{\diamond} = \emptyset$
(9) $k = 1$
(10) **call** 1-relations()
(11) **do**
(12) $k = k + 1$
(13) $\overset{\gamma}{\underset{k}{\prec}} = \overset{\gamma}{\underset{k}{\succ}} = \underset{k}{\diamond} = \emptyset$
(14) **while** k-relations(k)

(15) **if** $S^x()$ **not in** I **then return false end if**
(16) **for each** $r_p() = o_0 \rightarrow \ldots o_h \ldots$ **in** P_L^x **do**
(17) **if** o_0 **in** V_O^S **and** o_0 **not in** F **or**
(18) o_h **in** V_O^S **and** o_h **not in** I **then**
(19) erase r_p **in** P_G^x
(20) **end if**
(21) **end do**
(22) **return** useless-symbol-elimination$(P_G^x) \neq \emptyset$
(23) **end function**

Table 5: The Recognition Algorithm.

the empty set at line (7), so are the collection of level 1 relations at line (8). The loop at lines (11–14) computes all the level k relations. The ultimate goal is the computation of sets I and F. When the start object $S^x()$ is not an element of I, this means that there is no valid spine starting at the root and therefore the recognizer failed (line (15)).

Since G^x is the CF-backbone of L^x, each time a production $r_p()$ in P_L^x contains a non valid initial or final object, its corresponding production r_p in P_G^x is erased (see lines (16–21)). At line (22) we assume that a classical algorithm eliminates from P_G^x all the useless symbols[6]. If the resulting production set is not empty, it contains a production of the form $(S, x_{1..n+1}) \rightarrow \ldots$ which shows that x is a sentence of that reduced production set and therefore that x is an element of L^x and hence an element of L.

5 Its Complexity

Objects in LIGed forest are of the form $(A, x_{i..j})(\pi\alpha)$. The maximum number of split $x_{i..j}$ is $\mathcal{O}(n^2)$ where $|x| = n$. All other parameters (non-terminals and stack schemas) are constant for a given LIG L. Therefore, the size of any set which contains objects has an $\mathcal{O}(n^2)$ upper bound, especially I, F, and V_O^{LHS}.

5.1 Complexity of the 1-*relations* procedure

In Table 3, we have:

line (2) A single pass over P_L^x computes V_O^{LHS}, whose size is $\mathcal{O}(n^2)$, in time $\mathcal{O}(n^3)$.

[6] A symbol X is useless if it does not appear in any S/x-derivation.

lines (5–10) Each activation of this body is performed in constant time.

lines (4–11) For a given non-terminal $[X]$ there are at most $\mathcal{O}(n)$ occurrences of $[X]$ in the RHSs of $P_L^{\sharp}$. Therefore, each activation of that block takes $\mathcal{O}(n)$ time.

lines (3–12) The body of that loop is executed $\mathcal{O}(n^2)$ time so that block takes $\mathcal{O}(n^3)$ time.

lines (1–13) At the end the time complexity of the *1-relations* is $\mathcal{O}(n^3)$.

Since the body part (lines (7–9)) where the 1-relations are computed is executed at most $\mathcal{O}(n^3)$ time, the size of these relations is $\mathcal{O}(n^3)$. We notice that in each such relation, for a given object, say o, there are at most $\mathcal{O}(n)$ pairs whose first member or second member is o.

5.2 Complexity of the *k-relations* function

When $k > 1$, the size of the level k relations, since they contain pair of objects is at most $\mathcal{O}(n^4)$.

In Table 4, we have:

lines (5–11), (13), (20), (21–27) For each intermediate primary object o_2, these loops are executed a number of time which depends on the value of h since we refer to either level 1 relations or level $k-1$ relations. In the case where $k - h > 1$, these loops are executed $\mathcal{O}(n^2)$ time, else, when $k - h = 1$, these loops are executed $\mathcal{O}(n)$ time. Since their body sets individual relations in constant time, the overall complexity is not changed.

lines (12–14) Since line (13) is executed a bounded (i.e $|V_l|$) number of times, the complexity of the body extends to this loop.

lines (3–16), (18–29) For each activation of these loops, their body is executed $\mathcal{O}(n^3)$ time when $h = 1$ or $\mathcal{O}(n^4)$ when $h > 1$. We see that, in all cases, we get an execution time of $\mathcal{O}(n^5)$ for each activation.

lines (17–30) The complexity of its body extends to this loop (i.e. $\mathcal{O}(n^5)$).

lines (2–33) This loop is executed at most twice (when $k > 2$), therefore this block takes $\mathcal{O}(n^5)$.

line (34) This return condition may easily be get as a side effect of the setting of the relations (is there at least one element?), and therefore does not change the overall complexity.

In the worst case, the time complexity of the *k-relations* function is $\mathcal{O}(n^5)$.

5.3 Complexity of the Recognition Algorithm

In Table 5, we have:

line (4) Can take $\mathcal{O}(n^3)$ with the appropriate CF-parsing algorithm since the length of the longest RHS is two.

line (6) The LIGed forest is almost simply a copy of the shared parse forest and therefore takes $\mathcal{O}(n^3)$.

line (10) Takes $\mathcal{O}(n^3)$ (see 5.1).

lines (11–14) In order to evaluate the complexity of that loop we should know the maximum value of k. Recall that $k + 1$ is the length of valid sub-strings and therefore its maximum value corresponds to the length of the longest a-spine. Since spines are specialized path in parse trees and the height of parse trees (for non cyclic grammar) is $\mathcal{O}(n)$, this loop is executed $\mathcal{O}(n)$ times and since each execution of the *k-relations* function takes $\mathcal{O}(n^5)$ (see 5.2), this loop takes at most $\mathcal{O}(n^6)$.

lines (16–21) Takes $\mathcal{O}(n^3)$.

line (23) The classical algorithm for the elimination of useless symbol is performed in time linear with the size of the grammar, so in our case it will take $\mathcal{O}(n^3)$.

So, in the worst case, for a non cyclic grammar, the time complexity of our recognition algorithm is $\mathcal{O}(n^6)$.

If the CF-backbone of a LIG is unambiguous, the shared parse forest can be built in time $\mathcal{O}(n^2)$ by an Earley or generalized LR parsing algorithm (see [5]). In such a case, the shared parse forest is a simple (parse) tree whose size is $\mathcal{O}(n)$. Therefore, the objects cannot be shared among spines, and the cumulated length of all the spines is $\mathcal{O}(n)$. With this hypothesis, the size of our k-relations for $k \geq 1$ is $\mathcal{O}(n)$ and it can easily be seen that, for a given value k, their construction takes $\mathcal{O}(n)$ time, and that the complete check could therefore be performed in $\mathcal{O}(n^2)$ time[7]. Therefore, for unambiguous grammars, a total recognition time of $\mathcal{O}(n^2)$ is reached by our algorithm.

We can wonder whether intermediate values between $\mathcal{O}(n^2)$ and $\mathcal{O}(n^6)$ are reached for some sub-classes of LIGs. When the number of non-terminal symbols is $\mathcal{O}(n)$ in a shared parse forest it is not difficult to see that our recognizer has an $\mathcal{O}(n^4)$ worst time bound, but unfortunately we are not aware of any grammatical characterization of such a sub-class!

It should be pointed out that our algorithm is valid, even without restricting the maximum length l of the RHSs. The only consequence is that the recognizing time can be increased since the CF-parsing time (and the size of the shared parse forest) can be of the order $\mathcal{O}(n^{l+1})$. Moreover, though the cardinalities of the k-relations with $k \geq 2$ stay in $\mathcal{O}(n^4)$, the cardinalities of the 1-relations increase to $\mathcal{O}(n^4)$ and therefore induce a checking of the LIG conditions in time $\mathcal{O}(n^7)$. Finally, without restriction, a fair LIG can be recognized by our algorithm in time $\max(\mathcal{O}(n^{l+1}), \mathcal{O}(n^7))$.

6 An Example

In this section, we illustrate our algorithm with a LIG $L = (\{S, T\}, \{a, b, c\}, \{\gamma_a, \gamma_b, \gamma_c\}, P_L, S)$ where P_L contains the following productions:

$$\begin{array}{llll} S(..) \rightarrow S(..\gamma_a)a & S(..) \rightarrow S(..\gamma_b)b & S(..) \rightarrow S(..\gamma_c)c & S(..) \rightarrow T(..) \\ T(..\gamma_a) \rightarrow aT(..) & T(..\gamma_b) \rightarrow bT(..) & T(..\gamma_c) \rightarrow cT(..) & T() \rightarrow c \end{array}$$

It is easy to see that its CF-backbone G, whose production set P_G is:

$$\begin{array}{llll} S \rightarrow Sa & S \rightarrow Sb & S \rightarrow Sc & S \rightarrow T \\ T \rightarrow aT & T \rightarrow bT & T \rightarrow cT & T \rightarrow c \end{array}$$

defines the language $\mathcal{L}(G) = \{wcw' \mid w, w' \in \{a, b, c\}^*\}$. We remark that the stacks of symbols in L constrain the string w' to be equal to w and therefore the language $\mathcal{L}(L)$ is $\{wcw \mid w \in \{a, b, c\}^*\}$.

We can remark that in L the key part is played by the middle c, introduced by the last production $T() \rightarrow c$, and that this grammar is non ambiguous, while in G the symbol c, introduced by the last production $T \rightarrow c$, is only a separator between w and w' and that this grammar is ambiguous (any occurrence of c may be this separator).

Let $x = ccc$ be an input string, we wish to know whether x is an element of $\mathcal{L}(L)$.

Since x is an element of $\mathcal{L}(G)$, its shared parse forest G^x is not empty. Its production set P_G^x is:

$$\begin{array}{ll} (S, x_{1..4}) \rightarrow (S, x_{1..3})x_{3..4} & (S, x_{1..4}) \rightarrow (T, x_{1..4}) \\ (S, x_{1..3}) \rightarrow (S, x_{1..2})x_{2..3} & (S, x_{1..3}) \rightarrow (T, x_{1..3}) \\ (S, x_{1..2}) \rightarrow (T, x_{1..2}) & (T, x_{1..4}) \rightarrow x_{1..2}(T, x_{2..4}) \\ (T, x_{2..4}) \rightarrow x_{2..3}(T, x_{3..4}) & (T, x_{3..4}) \rightarrow x_{3..4} \\ (T, x_{1..3}) \rightarrow x_{1..2}(T, x_{2..3}) & (T, x_{2..3}) \rightarrow x_{2..3} \\ (T, x_{1..2}) \rightarrow x_{1..2} & \end{array}$$

We can observe that this shared parse forest denotes in fact three different parse trees. Each one corresponding to a different cutting out of $x = wcw'$ (i.e. $w = \varepsilon$ and $w' = cc$, or $w = c$ and $w' = c$, or $w = cc$ and $w' = \varepsilon$).

The corresponding LIGed forest whose start symbol is $S^x = (S, x_{1..4})$ and production set P_L^x is:

[7] Remark that an obvious algorithm which evaluates stacks on this single parse tree will take $\mathcal{O}(n)$.

$(S, x_{1..4})(..) \rightarrow (S, x_{1..3})(..\gamma_c)x_{3..4}$ $(S, x_{1..4})(..) \rightarrow (T, x_{1..4})(..)$
$(S, x_{1..3})(..) \rightarrow (S, x_{1..2})(..\gamma_c)x_{2..3}$ $(S, x_{1..3})(..) \rightarrow (T, x_{1..3})(..)$
$(S, x_{1..2})(..) \rightarrow (T, x_{1..2})(..)$ $(T, x_{1..4})(..\gamma_c) \rightarrow x_{1..2}(T, x_{2..4})(..)$
$(T, x_{2..4})(..\gamma_c) \rightarrow x_{2..3}(T, x_{3..4})(..)$ $(T, x_{3..4})() \rightarrow x_{3..4}$
$(T, x_{1..3})(..\gamma_c) \rightarrow x_{1..2}(T, x_{2..3})(..)$ $(T, x_{2..3})() \rightarrow x_{2..3}$
$(T, x_{1..2})() \rightarrow x_{1..2}$

In this LIGed forest there are three a-spines which are shown below with their objects separated by the appropriate level 1 relation:

$$(S, x_{1..4})() \underset{1}{\diamond\!\!-} (S, x_{1..4})(..) \overset{\gamma_c}{\underset{1}{\prec}} (S, x_{1..3})(..) \overset{\gamma_c}{\underset{1}{\prec}} (S, x_{1..2})(..) \underset{1}{\diamond\!\!-} (T, x_{1..2})()$$
$$(S, x_{1..4})() \underset{1}{\diamond\!\!-} (S, x_{1..4})(..) \overset{\gamma_c}{\underset{1}{\prec}} (S, x_{1..3})(..) \overset{\gamma_c}{\underset{1}{\succ}} (T, x_{1..3})(..\gamma_c) \underset{1}{\diamond\!\!-} (T, x_{2..3})()$$
$$(S, x_{1..4})() \underset{1}{\diamond\!\!-} (S, x_{1..4})(..) \overset{\gamma_c}{\underset{1}{\succ}} (T, x_{1..4})(..\gamma_c) \overset{\gamma_c}{\underset{1}{\succ}} (T, x_{2..4})(..\gamma_c) \underset{1}{\diamond\!\!-} (T, x_{3..4})()$$

Though these a-spines are not computed by our algorithm, it is easier to see what happens directly on them. In particular we can see that the first and last a-spine are not valid since there is $\overset{\gamma_c}{\prec}$ (or $\overset{\gamma_c}{\succ}$) without corresponding $\overset{\gamma_c}{\succ}$ (or $\overset{\gamma_c}{\prec}$) and that only the middle a-spine is valid. In fact the algorithm computes the level 1 relations (shown within the a-spines), the level 2 relations:

$$(S, x_{1..3})(..) \overset{\gamma_c}{\underset{2}{\prec}} (T, x_{1..2})() \quad (S, x_{1..4})(..) \underset{2}{\diamond\!\!-} (T, x_{1..3})(..\gamma_c) \quad (S, x_{1..4})() \overset{\gamma_c}{\underset{2}{\succ}} (T, x_{1..4})(..\gamma_c)$$

the level 3 relations:

$$(S, x_{1..4})() \underset{3}{\diamond\!\!-} (T, x_{1..3})(..\gamma_c) \quad (S, x_{1..4})(..) \underset{3}{\diamond\!\!-} (T, x_{2..3})()$$

while the computing of the level 4 leads to empty relations with sets $I = \{(S, x_{1..4})()\}$ and $F = \{(T, x_{2..3})()\}$.

Since the start object $(S, x_{1..4})()$ is in I, the execution of lines (16–21) in Table 5 leads to erase the productions $(T, x_{1..2}) \rightarrow x_{1..2}$, and $(T, x_{3..4}) \rightarrow x_{3..4}$ in P_G^x. The *useless-symbol-elimination* function called at line (22) returns the following (non empty) production set:

$(S, x_{1..4}) \rightarrow (S, x_{1..3})x_{3..4}$ $(S, x_{1..3}) \rightarrow (T, x_{1..3})$
$(T, x_{1..3}) \rightarrow x_{1..2}(T, x_{2..3})$ $(T, x_{2..3}) \rightarrow x_{2..3}$

which shows that $ccc \in \mathcal{L}(L)$.

We can remark that, with that example, our recognition algorithm is in fact a parsing algorithm (i.e. all resulting a-spines are valid). This is not always the case. Assume a LIGed forest with the following four a-spines: $s_1 = (o_1, \ldots, o_3)$, $s_2 = (o_1, \ldots, o_4)$, $s_3 = (o_2, \ldots, o_3)$, and $s_4 = (o_2, \ldots, o_4)$. Moreover assume that the only valid a-spines are s_1 and s_4, therefore, the algorithm will consider that o_1 and o_2 are valid initial objects and that o_3 and o_4 are valid final objects and that no production elimination should take place. Therefore, the LIGed forest is left unchanged but could not be considered as a representation of the shared parse forest for the initial LIG since there are a-spines s_2 and s_3 which are not valid.

7 Conclusion

In this paper we have presented a new recognition algorithm which works for the class of mildly context-sensitive languages. Though its worst case complexity does not improve over previous ones (i.e. a $\mathcal{O}(n^6)$ time is achieved), the recognizer behaves in practice much faster than its worst case.

The advantages of this algorithm can mainly be summarized as follows:

- parsing of the input string with the underlying CFG and checking of the LIG conditions are split into separate phases;

- LIG conditions checking relies upon a very simple principle which can be expressed by binary relations;
- the recognition test is simply performed by composition of the previous relations;
- therefore, no symbol stack computation is needed;
- it can be applied to unrestricted fair LIGs (though the $\mathcal{O}(n^6)$ limit can then be exceeded).

We can wonder whether the first point is really an advantage since it can be retorted that illegal paths should be aborted as soon as possible. Our argument is that it wastes time to compute symbol stacks in $\mathcal{O}(n^6)$ along paths which can be discovered as syntactically illegal in $\mathcal{O}(n^3)$.

This algorithm is implemented in a prototype system which is part of an ongoing effort to get a set of parsers for various NL formalisms.

References

[1] ABEILLÉ, A., and SCHABES, Y. 1989. Parsing idioms in lexicalized TAGs. *Proceedings of the fourth conference of the ACL.*

[2] AHO, A. V. 1968. Indexed grammars—An extension to context free grammars. *J. ACM*, Vol. 15, pp. 647-671.

[3] EARLEY, Jay C. 1968. An efficient context-free parsing algorithm. *Ph.D. thesis*, Carnegie-Mellon University, Pittsburgh, PA.

[4] KASAMI, T. 1965. An efficient recognition and syntax algorithm for context-free languages. *Technical Report AF-CRL-65-758*, Air Force Cambridge Research Laboratory, Bedford, MA.

[5] KIPPS, J. R. 1989. Analysis of Tomita's algorithm for general context-free parsing. *International Parsing Workshop'89*, pp. 193-202.

[6] KILGER, A., and FINKLER, W. 1993. TAG-based incremental generation. *German Research Center for Artificial Intelligence (DFKI), Technical Report*, Saarbrücken (Germany).

[7] LANG, B. 1974. Deterministic techniques for efficient non-deterministic parsers. *Automata, Languages and Programming, 2nd Colloquium*, Lectures Notes in Computer Science, Springer-Verlag, Vol. 14, pp. 255-269.

[8] LANG, B. 1991. Towards a uniform formal framework for parsing. *Current Issues in Parsing Technology*, edited by M. Tomita, Kluwer Academic Publishers, pp. 153-171.

[9] LANG, B. 1994. Recognition can be harder than parsing. *Computational Intelligence*, Vol. 10, No. 4, pp. 486-494.

[10] PAROUBEK, P., SCHABES, Y., and JOSHI, A. K. 1992. XTAG-a graphical workbench for developing tree-adjoining grammars. *Third Conference on Applied Natural Language Processing*, Trento (Italy).

[11] POLLER, P. 1994. Incremental parsing with LD/TLP-TAGs. *Computational Intelligence*, Vol. 10, No. 4, pp. 549-562.

[12] REKERS, J. 1992. Parser generation for interactive environments. *Ph.D. thesis*, University of Amsterdam.

[13] SCHABES, Y. 1994. Left to right parsing of lexicalized tree-adjoining grammars. *Computational Intelligence*, Vol. 10, No. 4, pp. 506-524.

[14] TOMITA, M. 1987. An efficient augmented context-free parsing algorithm. *Computational Linguistics*, Vol. 13, pp. 31-46.

[15] VIJAY-SHANKER, K. 1987. A study of tree adjoining grammars. *PhD. thesis*, University of Pennsylvania.

[16] VIJAY-SHANKER, K., and JOSHI, A. K. 1985. Some computational properties of tree adjoining grammars. *23rd Meeting of the Association for Computational Linguistics*, Chicago, pp. 82-93.

[17] VIJAY-SHANKER, K., and WEIR D. J. 1994. Parsing some constrained grammar formalisms. *ACL Computational Linguistics*, Vol. 19, No. 4, pp. 591-636.

[18] YOUNGER, D. H. 1965. Recognition and parsing of context-free languages in time n^3. *Information and Control*, Vol. 10, No. 2, pp. 189-208.