MITSUBISHI ELECTRIC RESEARCH LABORATORIES
http://www.merl.com

Lexicalized Context-Free Grammars

Yves Schabes

TR93-01 December 1993

Abstract

This paper introduces a tree generating system called Lexicalized Context-Free Grammar (LCFG) and describes a cubic-time parser for it. LCFG embeds the elegance of the analyses found in lexicalized tree-adjoining grammar without allowing for context-sensitive languages and therefore without requiring more computational resources than context-free grammars.

This work may not be copied or reproduced in whole or in part for any commercial purpose. Permission to copy in whole or in part without payment of fee is granted for nonprofit educational and research purposes provided that all such whole or partial copies include the following: a notice that such copying is by permission of Mitsubishi Electric Research Laboratories, Inc.; an acknowledgment of the authors and individual contributions to the work; and all applicable portions of the copyright notice. Copying, reproduction, or republishing for any other purpose shall require a license with payment of fee to Mitsubishi Electric Research Laboratories, Inc. All rights reserved.

Copyright © Mitsubishi Electric Research Laboratories, Inc., 1993
201 Broadway, Cambridge, Massachusetts 02139

Publication History:–

Submitted to the 31st Meeting of the Associatiation for Computational Lingusitics. January 5, 1993.

1 Motivations

Lexicalized tree-adjoining grammar (LTAG) [1, 8, 4] is an interesting framework for natural language processing mainly because of its extended domain of locality and also because of its lexicalized property. These characteristics allow for localizing most of the syntactic and semantic dependencies (such as filler-gap and predicate-argument relationships). An LTAG lexicon consists of a set of trees each one associated with one or more lexical items. These elementary trees can be viewed as elementary clauses (or phrases) in which the word(s) associated with them participate. Some samples trees that can be found in a LTAG for English are shown in Figure 1.

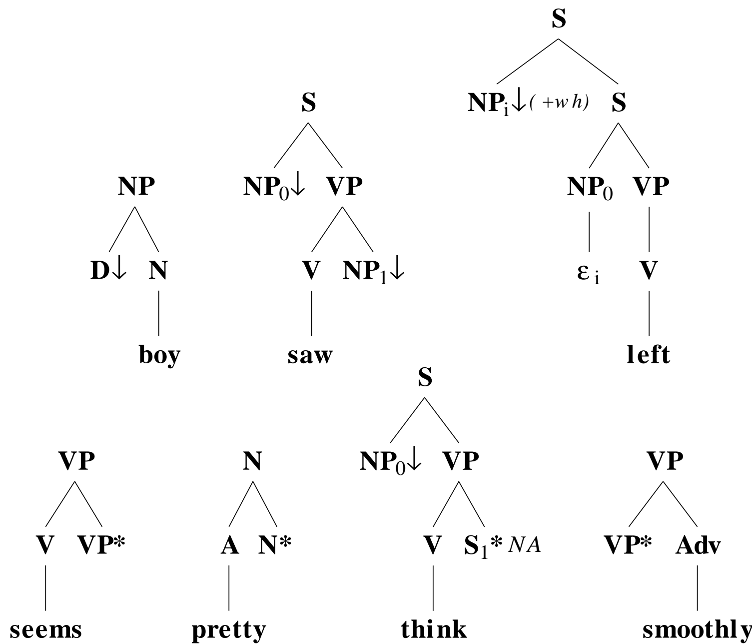


Figure 1: Sample trees.

The trees in an LTAG can be combined with two operations, substitution and adjunction. Substitution (Figure 2) in LTAG is the equivalent for trees to the string substitution operation used in context-free grammars.

Adjunction (Figure 3) inserts an auxiliary tree into the middle of another tree. This operation in effect encodes string wrapping and is therefore more powerful than concatenation. Adjunction generates languages and trees which cannot be obtained with the use of substitution (or equivalently with context-free grammars). Although the ability to generate context-sensitive features may be of interest for some natural languages, the extended domain of locality and the lexicalized property of LTAG are its primary features. Furthermore, in the current LTAG for English no context-sensitive phenomena are required.

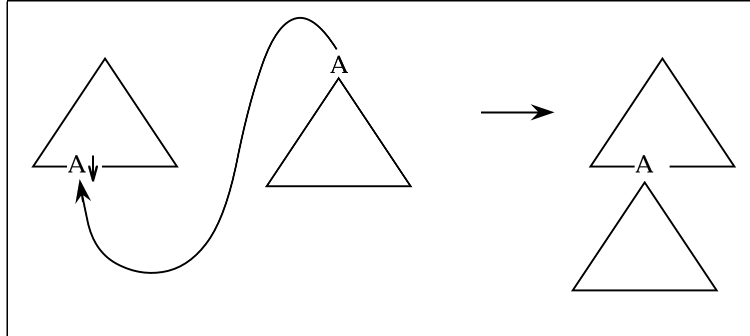


Figure 2: Substitution operation.

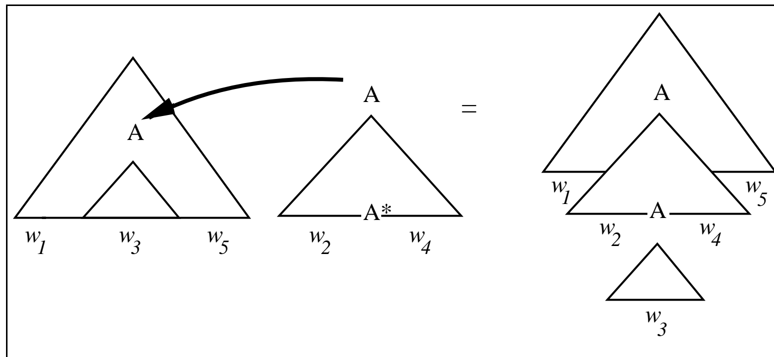


Figure 3: Adjunction.

The attractive aspects of LTAGs come at some cost. LTAGs require more processing power than context-free grammars: in the worst case $O(n^6)$-time for LTAGs [12, 7, 6] v.s. $O(n^3)$-time for CFGs [5, 3].

Another interesting aspect of LTAG is its ability to lexicalize CFGs. One can rewrite a CFG into a lexicalized TAG which preserves the original trees [4]. Until now, the existence of a less powerful formalism for lexicalizing CFG remained open.

This paper presents a context-free formalism called lexicalized context-free grammar (LCFG) which enables one to lexicalize context-free grammars. LCFG combines the elegance of LTAG and the efficiency of CFG. Similarly to LTAG, LCFG is a tree-rewriting system, it is lexicalized and it allows one for

an extended domain of locality compared to CFG. LCFG uses two combining operations, substitution and a restricted form of adjunction. The restricted form of adjunction still allows to insert a tree into another one, however it disallows any non-concatenative operation at the string level. Consequently, the set of languages generated by LCFG corresponds exactly to the set of context-free languages, but the set of trees it generates is a strict superset of the set of trees generated by CFG. It is this additional capability that enables LCFG to lexicalize CFG [10].

This paper describes LCFG and compares it with CFG and LTAG. Then, the computational efficiency of LCFG is illustrated by the description of a $O(n^3)$-time left to right parser for LCFG.

2 LCFG

Lexicalized context-free grammar is a restricted form of lexicalized tree-adjoining grammars. Informally speaking, the grammar consists of two sets of trees, the set of initial trees I and the set of auxiliary trees A. At least one terminal symbol (the anchor) must appear at the frontier of all initial or auxiliary trees.
As in LTAG, the set of initial trees, I, is characterized as follows:

- (i) interior nodes are labeled by non-terminal symbols;
- (ii) the nodes on the frontier of initial trees are labeled by terminals or non-terminals; non-terminal symbols on the frontier of the trees in I are marked for substitution; by convention, we annotate nodes to be substituted with a down arrow ($\downarrow$);

The set of auxiliary trees, A, is characterized as follows:

- (i) interior nodes are labeled by non-terminal symbols;
- (ii) the nodes on the frontier of auxiliary trees are labeled by terminal symbols or non-terminal symbols. Non-terminal symbol on the frontier of the trees in A are marked for substitution except for one node, called the *foot node*; by convention, we annotate the foot node with an asterisk ($*$); the label of the foot node must be identical to the label of the root node.
- (iii) the foot node is only allowed to appear to the leftmost or rightmost position on the frontier.

The auxiliary trees are characterized as in LTAG except for the third point which restricts their shape. Auxiliary trees whose foot node is to the rightmost (resp. leftmost) position are called left (resp. right) auxiliary trees since they introduce structure to the left (resp. right) of the foot node.

The trees can be combined with substitution (Figure 2), and by two restricted types of adjoining operations: *left adjoining* and *right adjoining*. The left adjoining operation adjoins a left auxiliary tree (Figure 4) and the right adjoining operation adjoins a right auxiliary tree (Figure 5).

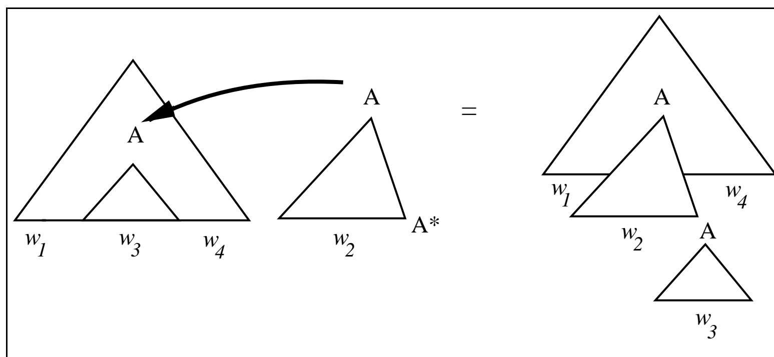


Figure 4: Left Adjunction.

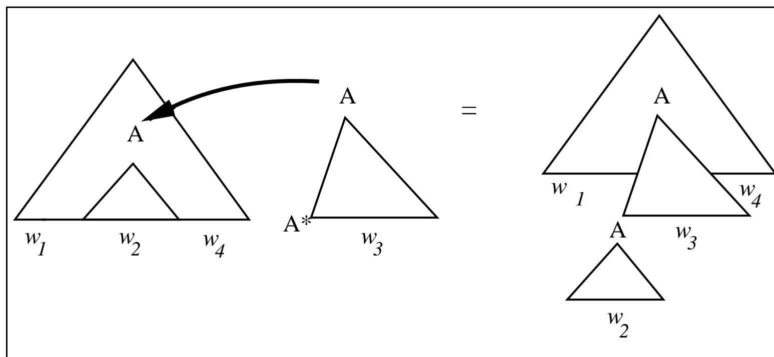


Figure 5: Right Adjunction.

In addition, LCFG impose that no left (resp. right) auxiliary can be adjoined on any node on the spine (the path from the root node to the foot node) of a right (resp. left) auxiliary tree. This restriction on the interaction of left and right auxiliary trees guarantees that no string wrapping can be

indirectly achieved and therefore that only concatenation of strings can be performed.

2.1 LCFG and CFG

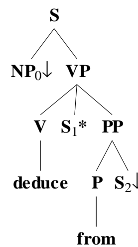
One can show that any LCFG generates only context free languages. However, the set of trees generated by LCFGs strictly includes the recognizable sets i.e. the sets of trees obtained by CFG as defined in [11].

Furthermore, Schabes and Waters [10] have shown that for any finitely ambiguous context-free language not generating the empty string there is a LCFG which generates the same trees (and therefore the same strings). This result is new, since until now only a construction requiring the full (context-sensitive) power of adjunction was known [4]. This result shows that any context-free grammar can be lexicalized by LCFG. The resulting normal form may be regarded as a stronger version of the Greibach Normal Form for CFGs, in the sense that the structures are preserved and not just the string sets.

2.2 LCFG and LTAG

The set of tree-adjoining languages is obviously a strict superset of the set of lexicalized context-free languages. However, it is of practical interest to see how much of the current LTAG grammar for English [1] falls into LCFG. Most of the English grammar falls into LCFG.

LCFG imposes two conditions: (1) only left or right types of auxiliary trees; (2) no interaction on the spine between the two types of auxiliary trees. In the current English LTAG, all auxiliary trees except for the ones associated with some verbs taking more than one sentential complement satisfy the first condition. For example, the verb *deduce* is associated with the following auxiliary tree:



The first sentential complement is required to be a foot node if one keeps the standard LTAG wh-analyses in which there exists an initial tree where filler and gap are local. This tree accounts for sentences like:

(1) John deduces that Mary invited Bob from smelling smoke.
(2) Who does John deduce that Mary invited from smelling smoke?

Since the tree contains a foot node in the center of its frontier, it is not part of an LCFG. Other analyses (other than the traditional LTAG's) of wh questions may not require this tree to be an auxiliary tree.

Most interactions between the two types of trees in the current LTAG follow the second condition. There are however cases where this condition is violated. For example, in the sentence:

(3) John said Bill left yesterday.

the attachment of *yesterday* is ambiguous. The LTAG derivations are shown in (Figure 6).

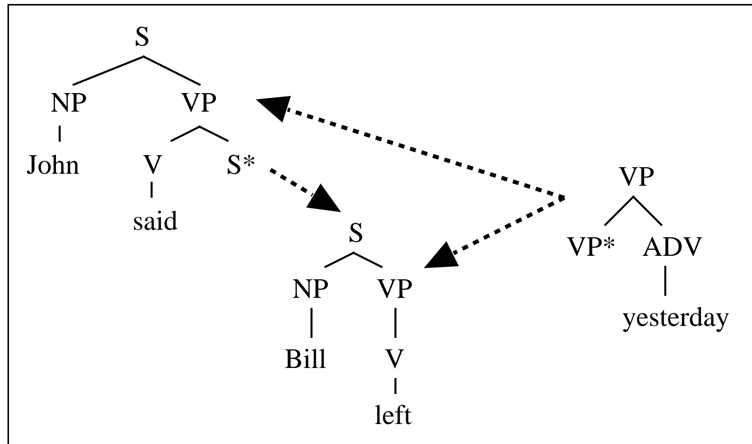


Figure 6: Ambiguous LTAG derivations for *John said Bill left yesterday.*

In LCFG, the high attachment of *yesterday* is forbidden since a right auxiliary tree (corresponding to *yesterday*) adjoins on the spines of a left auxiliary tree (corresponding to *John said*). However, one could design a mechanism to recover the high attachment reading from the low one.

Besides those two cases, the current LTAG for English falls into LCFG. This agrees with the intuition that most English analyses do not require a context-sensitive operation.

3 Parsing LCFG

Since LCFG is a restricted case of tree-adjoining grammar, standard TAG parsers (such as [6, 7, 12]) can be used for parsing LCFG. However, these parsers do not take full advantage of the context-freeness of LCFG. Although they require $O(n^6)$-time for TAG parsing, they can be made very easily to require at most $O(n^4)$-time for LCFG. This bound is still too high since we know that LCFGs generate only context-free languages. This section describes a left to right parsing algorithm for LCFG which requires $O(n^3)$ time in the worst case for LCFG. The algorithm is an extension of Earley's algorithm [3] to LCFG.

3.1 Preliminary Concepts

The following concepts are used in the description of the parsing algorithm.

In the following $a_1 \cdots a_n$ is the input string given to the parser and the LCFG being used is $G = (\Sigma, NT, I, A, \mathrm{S})$ where Σ is the finite set of terminal symbols, NT is the set of non-terminal symbols ($\Sigma \cap NT = \emptyset$), I is the set of initial trees, A is the set of auxiliary trees and $S \in NT$ is the start symbol.

A tree α will be considered to be a function from tree addresses to symbols of the grammar (terminal and non-terminal symbols): if x is a valid address in α, then $\alpha(x)$ is the label of the node at address x in the tree α.[1]

$Ladj(\alpha, ad)$ (resp. $Radj(\alpha, ad)$) is defined to be the set of left (resp. right) auxiliary trees that can be adjoined at the node at address ad in the tree α. For LCFGs with no constraints on adjunction, this set is the set of elementary auxiliary trees whose root node is labeled by $\alpha(ad)$.

For sake of simplicity, it is assumed that there are no constraint on adjoining. The derivation is relaxed from its standard TAG definition to allow one left auxiliary tree and one right auxiliary both adjoined at the same node. The algorithm can be easily extended to handle constraints on adjunction and to handle the alternative definition of TAG derivation found in [9].

Similarly, $Subst(\alpha, ad)$ is defined to be the set of initial trees that can be substituted at the node at address ad in the tree α.

[1] Addresses of nodes in a tree are encoded by Gorn-positions defined as follows: 0 is the address of the root node, k ($k \in \mathcal{N}$) is the address of the kth child of the root node, $x \cdot y$ (x is an address, $y \in \mathcal{N}$) is the address of the yth child of the node at address x.

$\mathcal{A}(\alpha)$ is defined as the set of addresses of all interior non-terminals (including the root node) of the elementary tree α including the foot node when the tree is an auxiliary tree. $\mathcal{A}(\alpha)$ is the set of addresses of the nodes on which adjunction can occur.

$\mathcal{S}(\alpha)$ is defined as the set of addresses of all non-terminal nodes of α on the frontier of the elementary tree α excluding the foot node when α is an auxiliary tree. $\mathcal{S}(\alpha)$ is the set of addresses of the nodes on which substitution can occur.

$foot(\beta)$ is defined to be the address of the foot node of the auxiliary tree β.

A *dotted tree* is defined as a tree associated with a dot above or below and either to the left or to the right of a given node. The four positions of the dot are annotated by la, lb, ra, rb (resp. left above, left below, right above, right below): ${}^{la}_{lb}A^{ra}_{rb}$. We write $< \alpha, dot, pos >$ for a dotted tree in which the dot is at address dot and at position pos in the tree α.

The left to right tree traversal used for parsing LCFGs consists of moving a dot in an elementary tree in a manner consistent with the left to right scanning of the fruntier while still being able to recognize adjunctions on interior nodes of the tree. The tree traversal starts with the dot above and to the left of the root node and ends with the dot above and to the right of the root node. At any time, there is only one dot in the dotted tree. An example of tree traversal is shown in Figure 7.

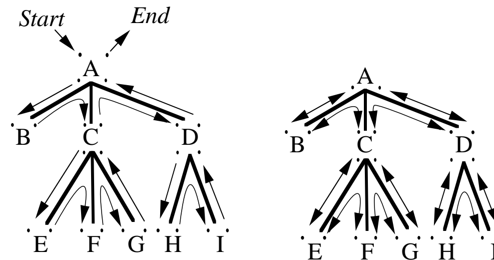


Figure 7: *Left*, left to right tree traversal; *right*, equivalent dot positions.

Two successive (according to the tree traversal) dot positions that do not cross a node in the tree are treated as equivalent (see Figure 7). For example the following equivalences hold for the tree α pictured in Figure 7: $< \alpha, 0, lb > \equiv < \alpha, 1, la >, < \alpha, 1, ra > \equiv < \alpha, 2, la >,$

$< \alpha, 2, lb > \equiv < \alpha, 2 \cdot 1, la >, \cdots$.

A dotted tree separates a tree into two parts (see Figure 8): a *left context* consisting of nodes that have been already traversed and a *right context* that still needs to be traversed.

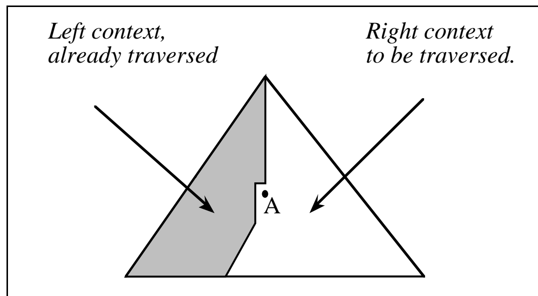


Figure 8: Left and right contexts of a dotted tree.

The algorithm uses the left context of a dotted tree to filter out the possibilities by only considering left contexts consistent with the prefix of the input string seen so far.

3.2 Data Structures

The algorithm collects states into a set called the chart $\mathcal{C}$. A *state* s is a 5-tuple, $[\alpha, dot, pos, i, j]$, where:

- α is an elementary tree, initial or auxiliary tree: $\alpha \in I \cup A$.
- *dot* is the address of the dot in the tree α.
- *pos* is the position of the dot: to the left and above, or to the left and below, or to the right and below, or to the right and above; $pos \in \{la, lb, rb, ra\}$.
- i, j are integers ranging over positions in the input string $[0, n]$.

The components α, dot, pos of a state define a dotted tree. Similarly to a dotted rule for context-free grammars defined by Earley [2], a dotted tree splits a tree into two contexts: a left context that has been traversed and a right context that needs to be recognized. The additional indices i, j record

the portion of the input string that the left context of the dotted tree covers. The fact that the auxiliary trees are either right or left auxiliary trees and that no other type of auxiliary trees can be derived guarantees that only two indices will be required.

In the following, each of two equivalent dot positions for the dotted tree are used interchangeably. For example, if the dot at address *dot* and at position *pos* in the tree α is equivalent to the dot at address dot' at position pos' in α, then $s = [\alpha, dot, pos, i, j]$ and $s' = [\alpha, dot', pos', i, j]$ refer to the same state. In this case, s or s' will be used interchangeably.

3.3 The Recognizer

The algorithm is a bottom-up parser that uses top-down prediction. It is a general recognizer for LCFGs and it requires no condition on the grammar.

The algorithm collects states in the chart $\mathcal{C}$. Initially, the chart $\mathcal{C}$ consists of all states of the form $[\alpha, 0, la, 0, 0]$, with α an initial tree rooted in S:

$$\mathcal{C} = \{[\alpha, 0, la, 0, 0] | \; \alpha \in I \text{ rooted in } S\}$$

These initial states (see Figure 9) correspond to dotted initial trees with the dot above and to the left of the root node (at address 0). They encode the fact that any valid derivation must start from an S-type initial tree.

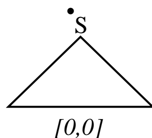


Figure 9: Initialization.

Depending on the existing states in the chart $\mathcal{C}$, new states are added to the chart by ten steps until no more states can be added to the chart. The steps of the algorithm are stated as inference rules: the state under the horizontal line of the inference rule is added to the chart if the states above the line are in the chart and if the side conditions are satisfied.

The recognizer for LCFGs follows:

```
program recognizer
begin
```

$\mathcal{C}$ = $\{[\alpha, 0, la, 0, 0] | \ \alpha \in I$ rooted in S $\}$

Apply one of the following steps (1-10) on each state in the chart $\mathcal{C}$ until no more states can be added to the chart $\mathcal{C}$:

- Scanner

$$(1) \quad \frac{[\alpha, dot, la, i, j]}{[\alpha, dot, ra, i, j+1]} \quad \alpha(dot) = a_{j+1}$$

$$(2) \quad \frac{[\alpha, dot, la, i, j]}{[\alpha, dot, ra, i, j]} \quad \alpha(dot) = \epsilon$$

- Left Predictor

$$(3) \quad \frac{[\alpha, dot, la, i, j]}{[\beta, 0, la, j, j]} \quad dot \in \mathcal{A}(\alpha), \beta \in Ladj(\alpha, dot)$$

$$(4) \quad \frac{[\alpha, dot, la, i, j]}{[\alpha, dot, lb, i, j]} \quad dot \in \mathcal{A}(\alpha)$$

- Right Predictor

$$(5) \quad \frac{[\alpha, dot, rb, i, j]}{[\beta, foot(\beta), rb, j, j]} \quad dot \in \mathcal{A}(\alpha), \beta \in Radj(\alpha, dot)$$

$$(6) \quad \frac{[\alpha, dot, rb, i, j]}{[\alpha, dot, ra, i, j]} \quad dot \in \mathcal{A}(\alpha)$$

- Left Completor

$$(7) \quad \frac{[\alpha, dot, la, i, j] \quad [\beta, foot(\beta), lb, j, k]}{[\alpha, dot, lb, i, k]} \quad dot \in \mathcal{A}(\alpha), \beta \in Ladj(\alpha, dot)$$

- Right Completor

$$(8) \quad \frac{[\alpha, dot, rb, i, j] \quad [\beta, 0, ra, j, k]}{[\alpha, dot, ra, i, k]} \quad dot \in \mathcal{A}(\alpha), \beta \in Radj(\alpha, dot)$$

- `Substitution Predictor`

$$(9) \quad \frac{[\alpha, dot, la, i, j]}{[\delta, 0, la, j, j]} \quad dot \in \mathcal{S}(\alpha), \delta \in Subst(\alpha, dot)$$

- `Substitution Completor`

$$(10) \quad \frac{[\alpha, dot, la, i, j] \quad [\delta, 0, ra, j, k]}{[\alpha, dot, ra, i, k]} \quad dot \in \mathcal{S}(\alpha), \delta \in Subst(\alpha, dot)$$

`If there is a state of the form` $[\alpha, 0, ra, 0, n]$ `in` $\mathcal{C}$ `with` $\alpha \in I$ `rooted in` S `then return acceptance otherwise return rejection.`
`end`

The input is recognized if in the final chart there is a state corresponding to an S-type initial tree completely recognized (with the dot to the right and above the root node) which spans the input form position 0 to n , i.e. if there is in the chart a state of the form $[\alpha, 0, ra, 0, n]$ where α is rooted in S.

The steps of the algorithm are pictorially explained in Figure 10.

The Scanner, Substitution Predictor and the Substitution Completor steps recognize substitutions of trees and are similar to the steps found in Earley's parser for CFGs [3]. The Left Predictor Right Predictor, Left Completor and Right Completor steps recognize left or right auxiliary trees that are combined. These steps are stated and explained below.

The predictor steps (3, 6, 5, 6) predict top down adjunctions of left and right auxiliary trees or trees to be substituted. The completor steps (7, 8) are bottom-up components which complete combining (by adjunction or substitution) of elementary trees. These steps are further explained below.

The scanner step scans the input string. It applies when the dot is to the left and above a terminal symbol. It consists of two cases: one when the terminal is not the empty string, and the other when it is.

The left predictor steps predict in a top-down fashion left recursive auxiliary trees. They predict new states accordingly to the left context that has been read. They consist of two steps that can be applicable simultaneously. The first step (3') predicts a left adjunction on the dotted node and the second step (4') considers the possibility that no left adjunction occurs on that node.

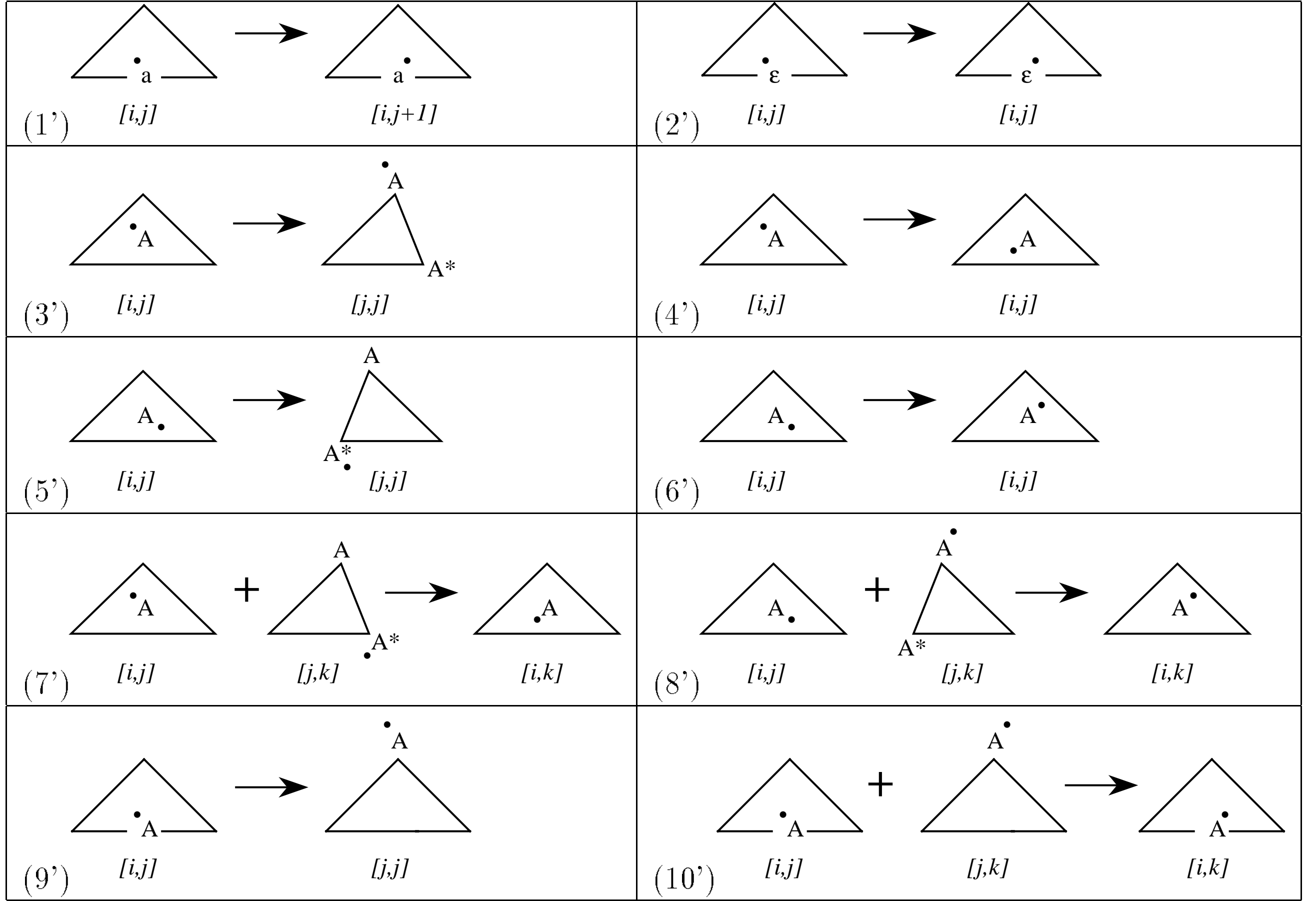


Figure 10: Pictorial explanation of each step

Similarly, the right predictor steps predict in a top-down fashion right recursive auxiliary trees. They consist of two steps that can be applicable simultaneously. The first step (5') predicts a right adjunction on the dotted node and the second step (6') considers the possibility that no right adjunction occurs on that node.

The left completor step completes the adjunction of a left recursive tree. It is a bottom-up step that concatenates the boundaries of a left auxiliary tree fully recognized and the ones of a partially recognized tree.

The right completor step completes the adjunction of a right recursive tree. It is a bottom-up step which concatenates the boundaries of a right

auxiliary tree fully recognized and the ones of a partially recognized tree.

The substitution predictor step predicts in a top-down fashion initial trees to be substituted. It predicts new states accordingly to the left context that has been read.

The substitution completor step completes the substitution of an initial tree. It is a bottom-up step that concatenates the boundaries of a left auxiliary tree fully recognized and the ones of a partially recognized tree.

3.4 Complexity

The algorithm takes in the worst case $O(n^3)$ time and $O(n^2)$ space. The worst case complexity comes from the left and right completor steps. An intuition of the validity of this result can be obtained by observing that these steps may be applied at most n^3 times since there are at most n^3 instances of states (corresponding the possible ranges of the indices i, j, k).

3.5 The Parser

The algorithm that we described in section 3.3 is a recognizer. However, if one includes pointers from a state to the other states (to a pair of states for the completor steps or to a state for the scanner and the predictor steps) that caused it to be placed in the chart, the recognizer can be modified to record all parse trees of the input string. The representation is similar to a shared forest.

4 Conclusion

Although the ability of LTAGs to generate context-sensitive languages may be of interest for some natural languages, the extended domain of locality and the lexicalized property of LTAG are its primary features.

This paper described a restricted form of LTAG, called Lexicalized Context-Free Grammar (LCFG), which embeds the elegance of most analyses found in LTAG without allowing for context-sensitive languages and without requiring more computational resources than context-free grammars. Most of the current LTAG for English falls into LCFG. This agrees with the intuition that most English analyses do not require a context-sensitive operation.

Since standard parsers for LTAG behave in $O(n^4)$-time when applied to LCFG, a new algorithm needed to be design in order to take fully advantage of the context-freeness of LCFGs. The paper described the design of a left to right parser for LCFG that requires in the worst case $O(n^3)$-time to process a sentence of length n.

Since the attractive aspects of LTAGs come at some computational cost, LCFG provides an efficient alternative which does not sacrifice the elegance of the LTAG analyses and which may be useful in different areas of computational linguistics.

References

[1] Anne Abeillé, Kathleen M. Bishop, Sharon Cote, and Yves Schabes. A lexicalized tree adjoining grammar for English. Technical Report MS-CIS-90-24, Department of Computer and Information Science, University of Pennsylvania, 1990.

[2] Jay C. Earley. *An Efficient Context-Free Parsing Algorithm*. PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, 1968.

[3] Jay C. Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102, 1970.

[4] Aravind K. Joshi and Yves Schabes. Tree-adjoining grammars and lexicalized grammars. In Maurice Nivat and Andreas Podelski, editors, *Tree Automata and Languages*. Elsevier Science, 1992.

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

[6] Bernard Lang. The systematic constructions of Earley parsers: Application to the production of $O(n^6)$ Earley parsers for Tree Adjoining Grammars. In *Proceedings of the 1st International Workshop on Tree Adjoining Grammars*, Dagstuhl Castle, FRG, August 1990.

[7] Yves Schabes. The valid prefix property and left to right parsing of tree-adjoining grammar. In *Proceedings of the second International Workshop on Parsing Technologies*, Cancun, Mexico, February 1991.

[8] Yves Schabes, Anne Abeillé, and Aravind K. Joshi. Parsing strategies with 'lexicalized' grammars: Application to tree adjoining grammars. In *Proceedings of the* 12th *International Conference on Computational Linguistics (COLING'88)*, Budapest, Hungary, August 1988.

[9] Yves Schabes and Stuart Shieber. An alternative conception of tree-adjoining derivation. In 20th *Meeting of the Association for Computational Linguistics (ACL'92)*, 1992.

[10] Yves Schabes and Richard C. Waters. A normal form for context-free grammars which preserves derivations. Technical Note 93-02, Mitsubishi Electric Research Laboratories, Cambridge Research Center, 201 Broadway. Cambridge MA 02139, 1993.

[11] J. W. Thatcher. Characterizing derivations trees of context free grammars through a generalization of finite automata theory. *Journal of Computer and System Sciences*, 5:365–396, 1971.

[12] K. Vijay-Shanker. *A Study of Tree Adjoining Grammars*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 1987.