June 1988

Upper Bounds on Recognition of a Hierarchy of Non-Context-Free Languages

Michael Palis
University of Pennsylvania

Sunil Shende
University of Pennsylvania

Upper Bounds on Recognition of a Hierarchy of Non-Context-Free Languages

Abstract

Control grammars, a generalization of context-free grammars recently introduced for use in natural language recognition, are investigated. In particular, it is shown that a hierarchy of non-context-free languages, called the *Control Language Hierarchy* (CLH), generated by control grammars can be recognized in polynomial time. Previously, the best known upper bound was exponential time. It is also shown that CLH is in $NC^{(2)}$ the class of languages recognizable by uniform boolean circuits of polynomial size and $O(\log^2 n)$ depth.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-88-56.

UPPER BOUNDS ON RECOGNITION OF A HIERARCHY OF NON-CONTEXT-FREE LANGUAGES

Michael Palis and
Sunil Shende

MS-CIS-88-56
LINC LAB 122

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104

July 1988

Acknowledgements: This research was supported in part by DARPA grant NOOO14-85-K-0018, NSF grants MCS-8219196-CER, MCS-82-07294, DCR-84-10413, MCS 83-05221, IRI84-10413-AO2 and U.S. Army grants DAA29-84-K-0061, DAA29-84-9-0027.

Upper Bounds on Recognition of a Hierarchy of Non-Context-Free Languages[*]

Michael Palis Sunil Shende[†]

July 5, 1988

Abstract

Control grammars, a generalization of context-free grammars recently introduced for use in natural language recognition, are investigated. In particular, it is shown that a hierarchy of non-context-free languages, called the *Control Language Hierarchy* (**CLH**), generated by control grammars can be recognized in polynomial time. Previously, the best known upper bound was exponential time. It is also shown that **CLH** is in $\mathbf{NC}^{(2)}$, the class of languages recognizable by uniform boolean circuits of polynomial size and $O(\log^2 n)$ depth.

Categories and Subject Descriptors: F.1.2. **[Computation by Abstract Devices]:** Modes of Computation - *alternation and nondeterminism, parallelism*; F.4.1. **[Mathematical Logic and Formal Languages]:** Formal Languages - *context-free grammars, control grammars*; I.2.7. **[Artificial Intelligence]:** Natural Language Processing - *language parsing.*

[*]Research funded in part by ARO grant DAA29-84-9-0027, NSF grants MCS-8219116-CER, MCS-82-07294, DCR-84-10413, MCS-83-05221, and DARPA grant N00014-85-K-0018.

[†]Both authors affiliated to the Dept. of Computer and Information Sciences, University of Pennsylvania, Philadelphia, PA 19104.

1 Introduction

A fundamental problem in computational linguistics is the development of grammatical models for natural language that are not only linguistically adequate but also amenable to efficient processing. To date, a large number of natural language grammars have been proposed in the literature, most of which are provably more powerful (in terms of generative capacity) than general context-free grammars (see, e.g. [1]). A recent addition to this list are control grammars [15], which generalize context-free grammars in an interesting way. Informally, a control grammar is a pair $\{G, C\}$ where G is an ordinary context-free grammar whose productions are each assigned a unique label from some finite set V_L. C, called the control set, is a set of strings over V_L. A derivation in a control grammar is similar to that in an ordinary context-free grammar except that the control set C is used to further constrain the set of "valid" derivations. In particular, if one views a derivation as a tree, then (in a manner to be described later) each edge in such a tree is given a label from V_L according to the production associated with the edge. The derivation tree is considered valid iff certain paths in the tree (called control paths) correspond to strings which are in the control set C. The language generated by the control grammar is then the set of strings having at least one valid derivation tree in the sense just described.

In essence, the control set C provides a way of limiting the set of valid derivation trees to those which have some predetermined "structure". For instance, C can be pre-selected as belonging to a particular language class, e.g., regular, context-free, or even one that is generated by another control grammar.

In [15], Weir introduced a hierarchy of language classes which are generated by control grammars in the following way: (1) the first class consists of all languages generated by control grammars whose control sets are context-free languages; (2) the k-th language class consists of all languages generated by control grammars whose control sets are members of the $(k-1)$-st class. This hierarchy has interesting properties, for instance, Weir has shown that every class in the hierarchy is a full **AFL** [1] and contains only semilinear sets (hence, all members are included among the context-sensitive languages). These classes can also be characterized in terms of automata which are interesting generalizations of (nondeterministic) pushdown automata (see [14, 15]).

An open problem posed by Weir is whether the language classes in his hierarchy are polynomial-time recognizable. One way of proving this is by using the following inductive argument. Suppose that every language in the k-th level of the hierarchy is polynomial-time recognizable. Then, one can construct a recognition algorithm for each language L in the $(k+1)$-st level as follows: Let $\{G, C\}$ be the control grammar generating L. Then, given an input string, first obtain all derivation trees of the string (if any) that are generated by G; then test whether at least one of these derivation trees has every control path in the control set C, where C is a language in the k-th level. That is, one simply uses the recognizer for C as a "subroutine". However, the inductive step fails since the input string may have exponentially many distinct derivation trees so that recognizing the language L may take exponential time.

In this paper, we prove that every language in Weir's hierarchy is indeed polynomial-time recognizable. In particular, we show that if L is a k-th level language, $k \geq 1$, then L can be recognized in $O(n^{3*2^k})$ time. The proof of this result is based on the observation that the recognizer for the control set can instead be used as a "coroutine" of the recognizer for L. This way, partial derivations which cannot possibly lead to valid derivation trees can be detected immediately and removed from further consideration. Using the recognition algorithm, we also show that every language class in Weir's hierarchy is in **LOGCFL**, the class of languages log-space reducible to context-free languages. Thus, using the result in [9], we get the corollary that every language in the hierarchy is in $\mathbf{NC}^{(2)}$, the class of languages recognized by uniform boolean circuits of polynomial size and $(\log^2 n)$ depth.

The paper is organized as follows. In Section 2, we define control grammars and the control language hierarchy (**CLH**) of Weir. Section 3 describes the recognition algorithms for languages in **CLH** and Section

[1] A (Full) **AFL** (Abstract Family of Languages) is a family closed under the operations of union, concatenation, Kleene star, (arbitrary) ϵ -free homomorphism, and intersection with regular languages, e.g., see [3]

4 proves containment of the language classes in **LOGCFL**. Section 5 ends the paper with some concluding remarks.

2 Control Grammars

An important result in formal language theory is the characterization of the *paths* in derivation trees of a context-free grammar. It was shown in [13] that the set of all such paths in derivation trees of any context-free grammar is a *regular language*. Control Grammars are defined by extending this idea of paths in two ways - by restricting our attention to a certain, well-defined subset of derivation paths in derivation trees of the context-free grammar and associating strings with the paths in a uniform way, and secondly, by prescribing a language (also called *the control set*) to which these strings must belong. In particular, the control set can be a language of arbitrary complexity, e.g. a context-free language. The following definition is adapted from [15], where Control Grammars were introduced [2].

Definition 2.1 A *Control Grammar* (henceforth **CG**) $\mathcal{G}$, is a pair $\{G, C\}$, where $G = (V_N, V_T, V_L, Z, P, Label)$ and $C \subseteq {V_L}^+$. The first component, G, of the control grammar is, by itself, called a Labeled, Distinguished Context-free grammar (or LDCFG) in [15]. V_N and V_T are, respectively, finite sets of ***nonterminals*** and ***terminals*** of the LDCFG G, with $Z \in V_N$ the ***start symbol*** of G. The set of ***grammar symbols***, $V_N \cup V_T$, is denoted by V. P is a finite set of ***distinguished*** productions of the form $(X \rightarrow X_1 \ldots X_n,\ i)$, where $X \rightarrow X_1 \ldots X_n$ can be viewed as a standard context-free production with $X \in V_N$ and the right-hand side $X_1 \ldots X_n$ belongs to V^*. In addition, i is an integer (with $1 \leq i \leq n$) that identifies exactly one symbol X_i on the right-hand side as being ***distinguished***. V_L is a finite set of ***production labels*** and *Label* is a one-to-one function from P to V_L, which assigns a unique label to every production. For the sake of clarity, we will write a distinguished production $p = (X \rightarrow X_1 \ldots X_n,\ i)$ with $Label(p) = l$ as

$$p = l : X \rightarrow X_1 \ldots \check{X}_i \ldots X_n$$

The set $C \subseteq {V_L}^+$ is called the *control set* of the grammar $\mathcal{G}$; each string in ${V_L}^+$ is referred to as a *control string* or *control word*. We say that grammar G is *controlled* by control set C. Note that by definition, C does not include the *empty string* ϵ.

An example of a control grammar is shown in Figure 1.

$$\begin{array}{ll} l_1 : Z_1 \rightarrow a\check{Z}_1 & \\ l_2 : Z_1 \rightarrow \check{Z}_1 c & \\ l_3 : Z_1 \rightarrow b\check{Z}_1 & \\ l_4 : Z_1 \rightarrow \check{b} & C = \{(l_1 l_2)^n {l_3}^{n-1} l_4 \mid n \geq 1\} \end{array}$$

LDCFG Productions Control Set

Figure 1: Control Grammar

Derivations and derivation trees of control grammars are very similar to those of standard context-free grammars, and are defined inductively as follows. $A \overset{0}{\underset{G}{\Longrightarrow}} A$ is a derivation in 0 steps of G, for every nonterminal $A \in V_N$. The set of derivation trees corresponding to this derivation is the singleton consisting of a tree with a single node labeled A, and is denoted by $TreeSet(A \overset{0}{\underset{G}{\Longrightarrow}} A)$.

[2]Other authors have described formalisms which have a somewhat different notion of *control*, but are also called control grammars; for example, see [7]. Throughout this paper, a "control grammar" satisfies definition 2.1 above.

Inductively, let $Y \overset{k}{\underset{G}{\Longrightarrow}} \alpha X \beta$ denote a derivation of G in k or fewer steps, with its associated derivation trees, $T = TreeSet(Y \overset{k}{\underset{G}{\Longrightarrow}} \alpha X\ \beta)$. Then for every production $p = l : X \rightarrow X_1 \ldots \check{X}_i \ldots X_n$ of G, we say that $Y \overset{k+1}{\underset{G}{\Longrightarrow}} \alpha X_1 \ldots X_i \ldots X_n \beta$. For every tree $\Delta \in T$, let δ be the the leaf node labeled X which corresponds to the instance of X used on the left-hand side of the production. Let Δ_1 be the tree obtained from Δ by adding new leaf nodes labeled $X_1, \ldots, X_i, \ldots, X_n$, and new undirected edges from to all the new leaf nodes *except* the one labeled X_i. For this node, we add an directed edge to it from δ, and label the edge with the production label l. All such trees Δ_1 are included in $TreeSet(Y \overset{k+1}{\underset{G}{\Longrightarrow}} \alpha X_1 \ldots X_i \ldots X_n\ \beta)$.

Following standard terminology, we say that $A \overset{*}{\underset{G}{\Longrightarrow}} \alpha$, if $A \overset{k}{\underset{G}{\Longrightarrow}} \alpha$ for some finite $k \geq 0$. Likewise, $TreeSet(A \overset{*}{\underset{G}{\Longrightarrow}} \alpha)$ is the set of all derivation trees for derivations $A \overset{*}{\underset{G}{\Longrightarrow}} \alpha$. Figure 2 shows a derivation tree in $TreeSet(Z_1 \overset{*}{\underset{G}{\Longrightarrow}} aabbcc)$ of the grammar G in Figure 1. Note that from every node in the tree, there is a unique, directed path to some leaf node in the tree.

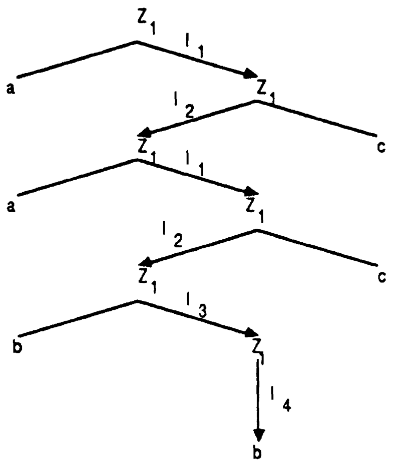

Figure 2: Derivation Tree associated with $Z_1 \overset{*}{\underset{G}{\Longrightarrow}} aabbcc$

For any derivation tree $\Gamma \in TreeSet(X \overset{*}{\underset{G}{\Longrightarrow}} \alpha)$, we shall call the unique directed, labeled path from the root node to a leaf node as $Spine(\Gamma)$ (or simply, the spine if Γ is clear from the context). Thus, $l_1 l_2 l_1 l_2 l_3 l_4$ is the spine in Figure 2. The (unique) leaf node which terminates $Spine(\Gamma)$ is denoted as the *foot node* of Γ. Finally, $ControlWords(\Gamma)$ is the set of all *maximal directed, labeled paths* in Γ (such a path begins at a node which is either the root or one which is connected to its parent by an *undirected edge*; the path ends at a leaf node). In particular, $Spine(\Gamma) \in ControlWords(\Gamma)$.

Definition 2.2 The Control Language $L(\mathcal{G})$, generated by CG $\mathcal{G} = \{G, C\}$, with start symbol Z of G, is

$$L(\mathcal{G}) = \{a_1 \ldots a_n \in V_T{}^* \mid \text{ there is a derivation tree } \Gamma \in TreeSet(Z \overset{*}{\underset{G}{\Longrightarrow}} a_1 \ldots a_n), \text{ and } ControlWords(\Gamma) \subseteq C\}.$$

Let $\mathcal{C}$ be any family of languages over a finite alphabet. We say that a language L is *controlled in family* $\mathcal{C}$ iff there is a control grammar $\mathcal{G} = \{G, C\}$ such that $L = L(\mathcal{G})$ and $C \in \mathcal{C}$.

For instance, it may be verified that the control language generated by the grammar in Figure 1 is the context-sensitive language $\{a^n b^n c^n \mid n \geq 1\}$. with the context-free control set $\{(l_1 l_2)^n l_3{}^{n-1} l_4 \mid n \geq 1\}$. The control language generated is, therefore, controlled in the family **CFL** of context-free languages, but is itself not a context-free language.

2.1 The Control Language Hierarchy

Following [15], we define a countable hierarchy of language classes, such that the 0-th family in the hierarchy is exactly the family of context-free languages, and every language in the $(i+1)$-th family is generated by a control grammar whose control set is a language in the i-th family.

Definition 2.3 The Control Language Hierarchy (**CLH**) is defined as follows:

- $\mathbf{CLH_0} = \{L \mid L = L(G)$, where G is a standard context-free grammar $\}$; i.e. $\mathbf{CLH_0} = \mathbf{CFL}$, the family of context-free languages.
- for all $k \geq 1$,
  $\mathbf{CLH_k} = \{L \mid$ there exists a context-free grammar G_0, and a sequence of LDCFGs $G_1, G_2, \ldots, G_k$ such that
  1. $C_0 = L(G_0)$,
  2. for all $1 \leq j < k$, $C_j = L(\{G_j, C_{j-1}\})$, and
  3. $L = L(\{G_k, C_{k-1}\})\}$.

  We say that G_0 and the sequence of LDCFGs $G_1, G_2, \ldots, G_k$ *defines* L.
- $\mathbf{CLH} = \cup_k \mathbf{CLH_k}$, for all countable $k \geq 0$.

A language L is said to be *ϵ-free* iff L does not contain the empty string. It is well known that every ϵ-free context-free language can be generated by a context-free grammar in Chomsky Normal Form (CNF), i.e, one whose productions are of the form $A \rightarrow BC$ or $A \rightarrow a$, where A, B and C are nonterminal symbols and a is a terminal symbol [6]. An LDCFG G is said to be in CNF iff for every production $l : X \rightarrow \alpha$ of G, the corresponding unlabeled, non-distinguished context-free production $X \rightarrow \alpha$ is in CNF. The following lemma states an analogous result for ϵ-free languages in **CLH**.

Lemma 2.1 (Chomsky Normal Form) Let L be an ϵ-free language in the family $\mathbf{CLH_k}$, $k > 0$. Then there is a context-free grammar G_0 and a sequence of LDCFGs $G_1, G_2, \ldots, G_k$ defining L such that G_0 and *every* LDCFG G_j, $1 \leq j \leq k$, in the sequence is in CNF.

The proof of Lemma 2.1 is deferred to the Appendix; it utilizes techniques similar to the conversion of a standard context-free grammar into CNF (as discussed, e.g., in [6]), and also the property that every family $\mathbf{CLH_k}$, $k > 1$, forms a full **AFL**.

In the next section, we shall describe a family of recognition algorithms for languages in the hierarchy. These algorithms are essentially motivated by the well-known Cocke-Kasami-Younger (CKY) recognition algorithm for context-free grammars [5], and like the CKY algorithm, require that every grammar in the sequence of grammars defining a particular control language be in CNF. Lemma 2.1 provides such a sequence.

3 Family of Recognition Algorithms for CLH

We generalize the Cocke-Kasami-Younger recognition algorithm [5, 6] for context-free languages, to a family of algorithms $Recognizer_k$, $k \geq 0$, where the kth algorithm recognizes any ϵ-free language in $\mathbf{CLH_k}$.

We know that any language $L \in \mathbf{CLH_k}$ is defined by a context-free grammar G_0 and a sequence of LDCFGs G_i, $1 \leq i \leq k$. Given a string whose membership in L is in question, it should be intuitively clear that we must check whether there is a complete derivation tree for the string, and whether all the control strings (over the terminal alphabet T_{k-1}) in the tree belong to the language $L(\mathcal{G}_{k-1})$. Each control string should, therefore, have a derivation tree of $\mathcal{G}_{k-1}$ such that all the control strings (now over T_{k-2}) in that tree belong to the language $L(\mathcal{G}_{k-2})$; this process is carried out till the entire sequence of grammars is unraveled and we can finally decide the context-freeness of some collection of control strings over T_0. Unfortunately, there may be many derivation trees for strings at any level (i.e. the input string at level k, and control strings at lower levels) and it should be apparent that our algorithm must be able to represent multiple derivations without ***explicitly storing control strings in these derivations.***

In order to get around this problem, we extend the idea used by the CKY recognizer of *implicitly* encoding potentially unbounded information contained in derivation trees, in a bounded collection of objects, which we call *items* of the appropriate grammars. The CKY algorithm makes use of the CNF property of the context-free grammar by creating for every input string $\alpha = a_1a_2\ldots a_n$ of length n, a 2-dimensional recognition matrix M such that any matrix entry $M(i,j)$ contains exactly all the *nonterminals* which derive the substring $a_i \ldots a_j$ of input α. Note that a nonterminal in $M(i,j)$ may derive $a_i \ldots a_j$ in many different ways none of which are explicitly represented by the algorithm. In the next section, we pursue this idea further by defining a data structure, for a language in family $\mathbf{CLH_k}$, which is analogous to a nonterminal in a context-free grammar. We shall see that these *items* encode derivations compactly, and can be combined to produce new items by using information in the productions of the sequence of grammars defining the language.

3.1 Data Structures and Operations

For $k > 0$, let L be any ϵ-free language in $\mathbf{CLH_k}$. Then by Lemma 2.1, there is a context-free grammar $G_0 = (N_0, T_0, P_0, Z_0)$ in CNF, and a sequence of control grammars $\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_k$, such that

$$\begin{gathered}\mathcal{G}_1 = \{G_1, L(G_0)\}\\ \mathcal{G}_2 = \{G_2, L(\mathcal{G}_1)\}\\ \vdots\\ \mathcal{G}_k = \{G_k, L(\mathcal{G}_{k-1})\},\end{gathered}$$

and $L = L(\mathcal{G}_k)$, where each of the LDCFGs G_i for $1 \le i \le k$ is defined as $G_i = (N_i, T_i, P_i, Z_i, T_{i-1}, Label_i)$ and is in Chomsky Normal Form. For notational convenience, we shall occasionally refer to G_0 as control grammar $\mathcal{G}_0$, with the understanding that $L(G_0) = L(\mathcal{G}_0)$. We shall also denote by C_i, $0 \le i < k$, the control set $L(\mathcal{G}_i)$ in the definition above.

For $0 \le i \le k$, let $x \in T_i^*$ be a string over the terminal alphabet of G_i. Then every 2^i tuple of strings

$$(u_1, u_2, \ldots, u_{2^{i-1}}, v_{2^{i-1}}, \ldots, v_2, v_1), \quad u_j, v_j \in T_i^*$$

such that $x = u_1u_2\ldots u_{2^{i-1}}v_{2^{i-1}}\ldots v_2v_1$ is called an *(i)-factorization* of x. We shall denote this as

$$x = \#(u_1, u_2, \ldots, u_{2^{i-1}}, v_{2^{i-1}}, \ldots, v_2, v_1).$$

As a special case, if $i = 0$ then $u_1 = x$ is the unique (0)-factorization of x.

Before we define the data structures and operations used by the kth recognition algorithm, we describe a restricted kind of derivation tree of any LDCFG G_i, $1 \le i \le k$. A derivation tree Γ of G_i is called *simple* iff all the leaves of Γ, except possibly the foot node (i.e. the node which terminates the directed labeled path from the root node), are labeled by *terminal* symbols in T_i. As a special case, a single node labeled by a grammar symbol in $(N_i \cup T_i)$ (which represents both the root node and the foot node) is also a simple tree of G_i.

A simple tree Γ of G_i with root node labeled $A \in N_i$ and foot node labeled $B \in N_i$ is said to *yield a pair* (u, v), $u, v \in T_i^*$, iff $\Gamma \in TreeSet(A \underset{G_i}{\overset{*}{\Longrightarrow}} uBv)$ and every string in $ControlWords(\Gamma)$, *except possibly* $Spine(\Gamma)$, is in the control set C_{i-1}. If the foot node is labeled by a terminal symbol $B = a \in T_i$, then Γ yields both (u, av) and (ua, v). As a special case, if Γ is a single node labeled $A \in (N_i \cup T_i)$, then Γ yields (ϵ, ϵ), and $Spine(\Gamma) = \epsilon$ (see Figures 3 (a) and (b)).

Definition 3.1 For $0 \le i \le k$, an *(i)-item* is defined inductively as follows:

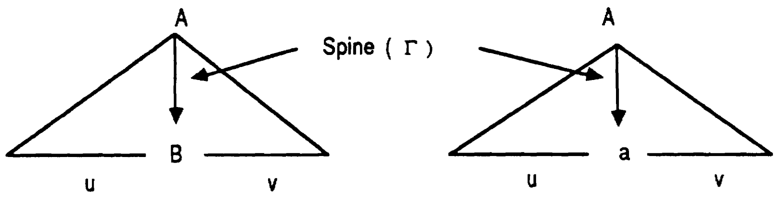

Figure 3: Derivation trees yielding a pair of terminal strings

- Every nonterminal symbol $A \in N_0$ is a (0)-item.
- For $1 \leq i \leq k$, an (i)-item is a tuple of the form $[(A_1, B_1, A_2, B_2, \ldots, A_{2^{i-1}}, B_{2^{i-1}});\ I\ ]$, such that
  1. $A_1 \in N_i$,
  2. $B_1, A_2, B_2, \ldots, A_{2^{i-1}},\ B_{2^{i-1}} \in (N_i \cup T_i)$,
  3. if A_j (for $2 \leq j \leq 2^{i-1}$), or B_l (for $1 \leq l \leq 2^{i-1}$) is the *first* terminal symbol, say a, in the sequence of symbols in 2 above, then all the symbols following A_j (i.e., $B_j, \ldots, B_{2^{i-1}}$), or following B_l (i.e., $A_{l+1}, \ldots, B_{2^{i-1}}$), are equal to A_j or B_l respectively, and
  4. I is an $(i-1)$-item.

We denote the set of (i)-items as $\mathcal{I}_i$, $0 \leq i \leq k$. For example, for $i = 2$, if $A,\ B\ \in N_2$, $a\ \in T_2$, and I is a (1)-item, then $[(A, B, A, B);\ I]$, $[(A, B, A, a);\ I]$, $[(A, B, a, a);\ I]$ and $[(A, a, a, a);\ I]$ are (2)-items.

Definition 3.2 For $0 \leq i \leq k$, we inductively define the notion of an (i)-item being *valid* for an (i)-factorization.

- A (0)-item A is *valid* for $u \in {T_0}^+$ iff $A \overset{*}{\underset{G_0}{\Longrightarrow}} u$.
- For $i > 0$, an (i)-item

$$[(A_1, B_1, A_2, B_2, \ldots, A_{2^{i-1}}, B_{2^{i-1}});\ I]$$

  is *valid* for an (i)-factorization

$$(u_1, u_2, \ldots, u_{2^{i-1}}, v_{2^{i-1}}, \ldots, v_2, v_1), \text{iff}$$

  1. there exists a sequence of simple trees $\Gamma_1,\ \Gamma_2, \ldots, \Gamma_{2^{i-1}}$ such that for $1 \leq j \leq 2^{i-1}$, the tree Γ_j has root and foot nodes labeled A_j and B_j respectively, and Γ_j yields (u_j, v_j), and
  2. I is valid for the $(i-1)$-factorization $(\mathit{Spine}(\Gamma_1), \mathit{Spine}(\Gamma_2), \ldots, \mathit{Spine}(\Gamma_{2^{i-1}}))$.

Note that an (i)-item ***represents*** a set of sequences of simple trees where any sequence read from the left to the right, can be depicted from top to bottom as a sequence of disjoint simple trees. These trees share a common "thread ", viz. their respective *spines* represent disjoint substrings of a single control string. This intuitive picture of (i)-items will be extensively used to illustrate the various operations below. For example, Figure 4 above shows an item valid for an appropriately sized tuple of strings, as a sequence of simple trees. By definition, all maximal paths (i.e., control words) in any of these trees (except possibly their spines) are already in the control set. We now define the following predicates on (i)-items, which will be used in the sequel.

Definition 3.3 For all $1 \leq i \leq k$:

- for $1 \leq j \leq 2^{i-1}$, $\mathit{StartItem}_{i,j}(I)$ is true iff I is an (i)-item of the form

$$[(A_1, B_1, \ldots, A_j, B_j, \ldots, A_{2^{i-1}}, B_{2^{i-1}});\ I']$$

  where for all $1 \leq m \neq j \leq 2^{i-1}$, $A_m\ =\ B_m$ (A_j may or may not be equal to B_j).

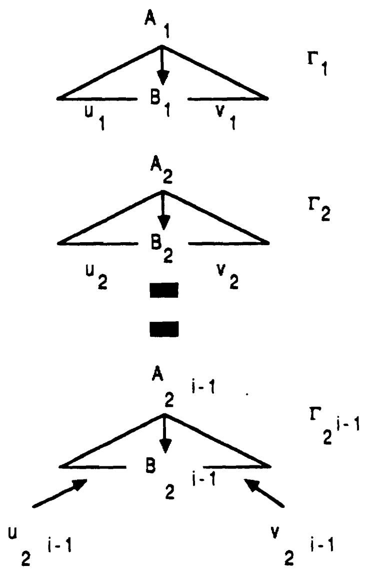

Figure 4: An (i)-item valid for $(u_1, \ldots, u_{2^{i-1}}\ v_{2^{i-1}}, \ldots, v_1)$

- *SourceItem*$_1(I)$ holds iff I is an item of the form $[(A, a);\ Z_0]$, where $a \in T_1$ and Z_0 is the start symbol of grammar G_0.
  For $1 < i \leq k$, *SourceItem*$_i(I)$ is true iff I is an (i)-item of the form
  $$[(A_1, B_1, \ldots, A_j, B_j, \ldots, A_{2^{i-1}}, a);\ I']$$
  where $A_j = B_{j-1}$ for all $2 \leq j \leq 2^{i-1}$, $a \in T_i$, I' is of the form $[(Z_{i-1}, \ldots);\ I'']$ with Z_{i-1} being the start symbol of G_{i-1}, and *SourceItem*$_{i-1}(I')$ holds recursively for item I'.
- let $I_1 = [(A_1, B_1, \ldots, A_j, B_j, \ldots, A_{2^{i-1}}, B_{2^{i-1}});\ I_1']$ and $I_2 = [(C_1, D_1, \ldots, C_j, D_j, \ldots, C_{2^{i-1}}, D_{2^{i-1}});\ I_2']$ be any two (i)-items of $\mathcal{G}_i$. Then *Compatible*$_i(I_1, I_2)$ is true iff $A_j = D_j$ for all even j, and $B_j = C_j$ for all odd j.

Note that *StartItem*$_{i,j}$, *SourceItem*$_i$ and *Compatible*$_i$ are only syntactic restrictions on items, and do not imply any notion of validity.

Definition 3.4 The set of wrapping operations, $\{W_i \mid 0 \leq i \leq k\}$, is inductively defined for (i)-items I_1, I_2 as follows:

1. For $i = 0$, let $I_1 = Y$ and $I_2 = Z$, where $Y,\ Z \in N_0$. Then
   $$W_0(Y,\ Z) = \{X \in N_0 \mid X \rightarrow YZ \text{ is a production in } P_0\}$$
2. for $1 \leq i \leq k$, let $I_1 = [(A_1, B_1, \ldots, A_j, B_j, \ldots, A_{2^{i-1}}, B_{2^{i-1}});\ I_1']$ and $I_2 = [(C_1, D_1, \ldots, C_j, D_j, \ldots, C_{2^{i-1}}, D_{2^{i-1}});\ I_2']$ be (i)-items of $\mathcal{G}_i$. Then
   $$W_i(I_1,\ I_2) = \left\{[(X_1, Y_1, \ldots, X_j, Y_j, \ldots, X_{2^{i-1}}, Y_{2^{i-1}});\ I'] \;\middle|\; \begin{array}{l} \textit{Compatible}_i(I_1, I_2) \text{ holds and} \\ X_j = \begin{cases} A_j & \text{for all odd } j, \\ C_j & \text{for all even } j \end{cases} \\ Y_j = \begin{cases} D_j & \text{for all odd } j, \\ B_j & \text{for all even } j \end{cases} \\ \text{and } I' \in W_{i-1}(I_1',\ I_2') \end{array}\right\}$$

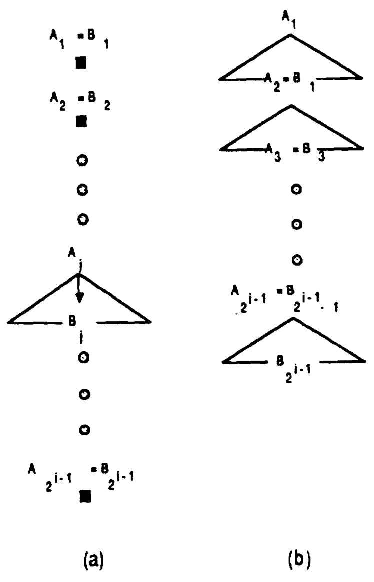

Figure 5: (a) *StartItem*$_{i,j}$ (b) *SourceItem*$_i$

Intuitively, W_i is the binary *wrapping* operation which combines two ***compatible*** (i)-items into a single one (see Figure 6). It can be extended in a straightforward way to the case when the arguments are sets of (i)-items, e.g.

$$W_i(S_1, S_2) = \cup_{I_1 \in S_1, I_2 \in S_2} W_i(I_1, I_2).$$

In the sequel, we shall assume that other operations are similarly extended to sets without ambiguity.

An easy consequence of the above definition is the following:

Proposition 3.1 For any given i, let I_1 and I_2 be (i)-items respectively valid for factorizations $(x_1, x_2, \ldots, x_{2^{i-1}}, y_{2^{i-1}}, \ldots, y_2, y_1)$ and for $(w_1, w_2, \ldots, w_{2^{i-1}}, z_{2^{i-1}}, \ldots, z_2, z_1)$.
If $W_i(I_1, I_2)$ is defined, then every item $I \in W_i(I_1, I_2)$ is valid for the factorization

$$(u_1, u_2, \ldots, u_{2^{i-1}}, v_{2^{i-1}}, \ldots, v_2, v_1)$$

where
for all $1 \leq j \leq 2^{i-1}$: $(u_j,\ v_j) = \begin{cases} (x_j w_j,\ z_j y_j) & \text{if } j \text{ is odd} \\ (w_j x_j,\ y_j z_j) & \text{if } j \text{ is even} \end{cases}$

We now define some other operations which construct new items. It is intuitively easier to understand these operations as though they are applied to simple tree sequences (represented as items). Thus, the subscript i stands for the fact that the corresponding operation is applied to (i)-items, whereas the subscript j identifies the appropriate tree in the simple tree sequence represented by the constructed item.

Definition 3.5 The operations $Init_{i,j}$, $LC_{i,j}$ and $RC_{i,j}$ are defined inductively as follows:

- For $i = 0$, let $a \in T_0$.
  1. $Init_{0,1}(a) = \{X \mid X \rightarrow a \in P_0\}$.
  2. $LC_{0,1}(I) = RC_{0,1}(I) = \{I\}$, for any (0)-item I.
- For $1 \leq i \leq k$, and $1 \leq j \leq 2^{i-1}$:

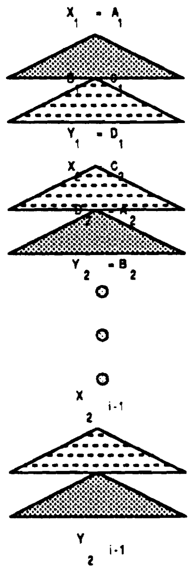

Figure 6: Operation W_i

1. let $a \in T_i$. Then

$$
\begin{aligned}
Init_{i,j}(a) \;=\;& \{I = [(A_1, B_1, \ldots, A_j, a, \ldots, a);\; I'] \;\in\; \mathcal{I}_i \;\mid \\
& StartItem_{i,j}(I) \;=\; true, \text{ and , there is a labeled production} \\
& l:\; A_j \rightarrow \breve{a} \;\in P_i, \;\text{ such that the } (i-1) \text{ item} \\
& I' \;\in \begin{cases} Init_{i-1,j}(l) \cup LC_{i-1,j}(Init_{i-1,1}(l)) & \text{if } 1 \le j \le 2^{i-2} \\ Init_{i-1,2^{i-1}-j+1}(l) \cup RC_{i-1,2^{i-1}-j+1}(Init_{i-1,1}(l)) & \text{otherwise}\} \end{cases}
\end{aligned}
$$

(note that if $i = 1$, then $I' \in LC_{0,1}(Init_{0,1}(l)) = RC_{0,1}(Init_{0,1}(l)) = Init_{0,1}(l)$)

2. Let $I_1 = [(C_1, D_1, \ldots, C_{2^{i-1}}, D_{2^{i-1}});\; I']$ be any (i)-item such that $SourceItem_i(I_1)$ is true. Then

$$
\begin{aligned}
LC_{i,j}(I_1) \;=\;& \{I = [(A_1, B_1, \ldots, A_j, B_j, \ldots, A_{2^{i-1}}, B_{2^{i-1}});\; I''] \;\in\; \mathcal{I}_i \;\mid \\
& StartItem_{i,j}(I) \;=\; true, \text{ and there is a labeled} \\
& \text{production } l:\; A_j \rightarrow C_1 \breve{B}_j \;\in P_i \text{ such that} \\
& I'' \;\in \begin{cases} LC_{i-1,j}(Init_{i-1,1}(l)) & \text{if } 1 \le j \le 2^{i-2} \\ RC_{i-1,2^{i-1}-j+1}(Init_{i-1,1}(l)) & \text{otherwise}\} \end{cases}
\end{aligned}
$$

(if $i = 1$, then $I'' \in Init_{0,1}(l)$)

3. Let $I_1 = [(C_1, D_1, \ldots, C_{2^{i-1}}, D_{2^{i-1}});\; I']$ be any (i)-item such that $SourceItem_i(I_1)$ is true. Then

$$
\begin{aligned}
RC_{i,j}(I_1) \;=\;& \{I = [(A_1, B_1, \ldots, A_j, B_j, \ldots, A_{2^{i-1}}, B_{2^{i-1}});\; I''] \;\in\; \mathcal{I}_i \;\mid \\
& StartItem_{i,j}(I) \;=\; true, \text{ and there is a labeled} \\
& \text{production } l:\; A_j \rightarrow \breve{B}_j C_1 \;\in P_i \text{ such that} \\
& I'' \;\in \begin{cases} LC_{i-1,j}(Init_{i-1,1}(l)) & \text{if } 1 \le j \le 2^{i-2} \\ RC_{i-1,2^{i-1}-j+1}(Init_{i-1,1}(l)) & \text{otherwise}\} \end{cases}
\end{aligned}
$$

(if $i = 1$, then $I'' \in Init_{0,1}(l)$))

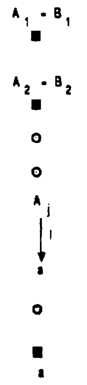

Figure 7: $Init_{i,j}(l)$

Intuitively, $Init_{i,j}(a)$ produces an initial set of (i)-items valid for a terminal symbol a. $LC_{i,j}(I_1)$ and $RC_{i,j}(I_1)$ *concatenate* the "degenerate" sequence represented by I_1 to, respectively, the left and the right in the resulting simple tree sequence (see Figures 7 and 8). The definition ensures the following proposition analogous to proposition 3.1. The proof is inductive as before, and is left to the reader.

Proposition 3.2 1. If $I \in Init_{i,j}(a)$, then I is valid for the factorization

$$(u_1, u_2, \ldots, u_{2^{i-1}}, v_{2^{i-1}}, \ldots, v_2, v_1)$$

where for all $1 \leq m \neq j \leq 2^{i-1}$, $u_m = v_m = \epsilon$ and

- either $u_j = \epsilon$ and $v_j = a$, or
- $u_j = a$ and $v_j = \epsilon$.

If $i = 0$ then I is simply valid for a.

2. Let I_1 be valid for the factorization $(x_1, x_2, \ldots, x_{2^{i-1}}, y_{2^{i-1}}, \ldots, y_2, y_1)$ of string x. Then
   - every item $I \in LC_{i,j}(I_1)$ is valid for the factorization $(u_1, u_2, \ldots, u_{2^{i-1}}, v_{2^{i-1}}, \ldots, v_2, v_1)$ of x where $x = u_j$ (hence all other u_i's and v_i's are empty).
   - every item $I \in RC_{i,j}(I_1)$ is valid for the factorization $(u_1, u_2, \ldots, u_{2^{i-1}}, v_{2^{i-1}}, \ldots, v_2, v_1)$ of x where $x = v_j$ (hence all other u_i's and v_i's are empty).

(Observe that if $i = 0$ then both I_1 and I are valid for x).

3.2 Soundness and Completeness of the Operations

We now state a lemma which asserts the soundness and completeness of the operations with respect to items and the factorizations that they are valid for. We shall use this result to prove the correctness of algorithm *Recognizer$_k$*.

Lemma 3.3 For all $0 \leq i \leq k$, the following holds:
Let $(u_1, \ldots, u_{2^{i-1}}, v_{2^{i-1}}, \ldots, v_1)$ be a factorization of a nonempty string $x \in T_i{}^+$, let I be any (i)-item. Then

I is valid for $(u_1, \ldots, u_{2^{i-1}}, v_{2^{i-1}}, \ldots, v_1)$ iff
 (A) $i = 0$, I is valid for $x = u_1$, and either
  (1) $u_1 = a \in T_0$, and $I \in Init_{0,1}(a)$, or
  (2) $u_1 = x_1 y_1$ such that both x_1 and y_1 are nonempty, there are items I_1 and I_2 valid respectively for x_1 and y_1, and $I \in W_0(I_1, I_2)$
 (B) $i \geq 1$, $I = [(X_1, Y_1, \ldots, X_{2^{i-1}} Y_{2^{i-1}});\ I']$, and either

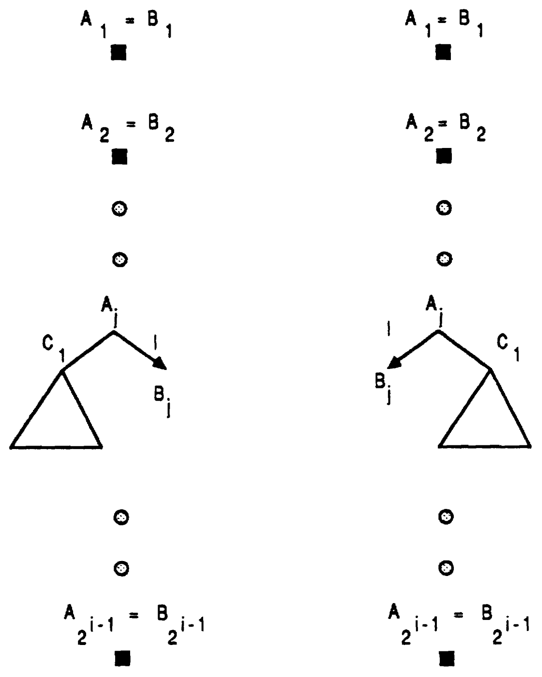

Figure 8: Operations $LC_{i,j}$ and $RC_{i,j}$

(1) there is a terminal symbol $a \in T_i$, such that $x = a$, and $I \in Init_{i,j}(a)$ for some j: $1 \leq j \leq 2^{i-1}$, or

(2) there is some $1 \leq j \leq 2^{i-1}$ such that $x = u_j$ (or $x = v_j$), and an item I_1 for which $SourceItem_i(I_1)$ is true, where $I \in LC_{i,j}(I_1)$ (respectively, $I \in RC_{i,j}(I_1)$), and I_1 is valid for a factorization $(y_1, y_2, \ldots, y_{2^i})$ of u_j (respectively, v_j).

(3) there are (i)-items I_1 and I_2 respectively valid for factorizations $(x_1, x_2, \ldots, x_{2^{i-1}}, y_{2^{i-1}}, \ldots, y_2, y_1)$ and for $(w_1, w_2, \ldots, w_{2^{i-1}}, z_{2^{i-1}}, \ldots, z_2, z_1)$, where

for all $1 \leq j \leq 2^{i-1}$: $(u_j,\ v_j) \ = \begin{cases} (x_j w_j,\ z_j y_j) & \text{if } j \text{ is odd} \\ (w_j x_j,\ y_j z_j) & \text{if } j \text{ is even} \end{cases}$

and $I \in W_i(I_1, I_2)$.

Proof Sketch: The reverse direction of the lemma follows from the Propositions 3.1 and 3.2.

The proof in the forward direction proceeds by induction on i; statement (A) forms the basis of the inductive assertion and is a direct consequence of the fact that G_0 is a context-free grammar in Chomsky Normal Form.

Consider an (i)-item $I = [(X_1, Y_1, \ldots, X_{2^{i-1}} Y_{2^{i-1}}); I']$ which is valid for the (i)-factorization $(u_1, \ldots, u_{2^{i-1}}, v_{2^{i-1}}, \ldots, v_1)$ of a nonempty string x. By definition, there is a sequence of simple trees $\Gamma_1, \Gamma_2, \ldots, \Gamma_{2^{i-1}}$ such that for all $1 \leq j \leq 2^{i-1}$, Γ_j has root node labeled X_j, foot node labeled Y_j, and yields (u_j, v_j). Moreover, I' is valid for the $(i-1)$-factorization $(Spine(\Gamma_1), \ldots, Spine(\Gamma_{2^{i-1}}))$ of the control string $y = \#(Spine(\Gamma_1), \ldots, Spine(\Gamma_{2^{i-1}}))$.

We shall illustrate the proof for the forward direction when $i = 1$, since it provides the basic argument used for the rest of the cases where $i > 1$. By definition, I is an item of the form $[(X_1, Y_1);\ I']$ and represents the simple tree Γ_1 which yields (u_1, v_1), where $\#(u_1, v_1) = x$. Also, I' is a single nonterminal of G_0 such that $I' \overset{+}{\underset{G_0}{\Longrightarrow}} y$. Consider the following different cases:

1. $\#(u_1, v_1) = a$, a single terminal symbol in T_1.
   It is easily verified as a consequence of the CNF property of G_1, that the tree Γ_1 must be one of the trees in Figure 9. Let $u_1 = a$. Now $I' \in Init_0(l)$ and hence $\in RC_{0,1}(Init_0(l))$ by definition; therefore

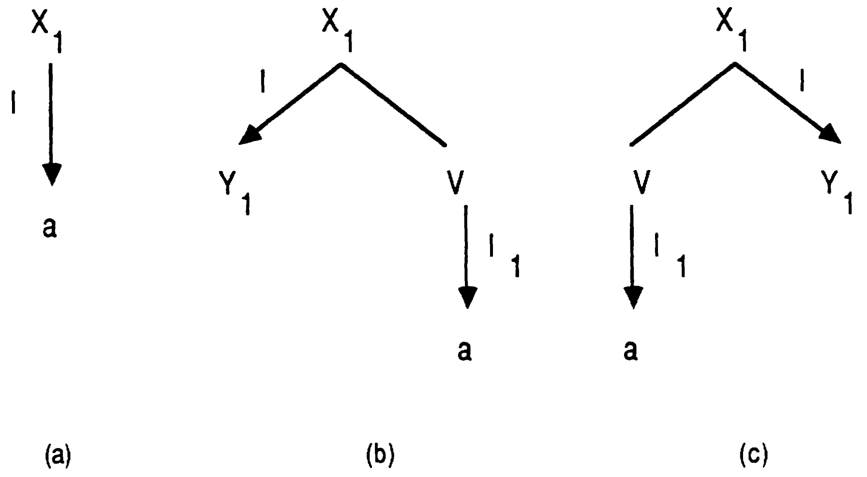

Figure 9: Simple tree corresponding to $x = a$

   for the tree (a) in the Figure, we must have that $I \in Init_1(a)$ by definition (note that $StartItem_{1,1}(I)$ is true). Hence I is valid for (a, ϵ) and satisfies part (1) of case (B) of the lemma. A similar argument disposes of the case when I is valid for (ϵ, a).

   Now consider tree (b) in the Figure, where $u_1 = a$. Since every control word in the tree other than the spine must be in the control set, we must have that Z_0, the start symbol of G_0 is valid for l'. Also, $I' \in Init_0(l)$ and hence $\in LC_{0,1}(Init_0(l))$. But then there is an item $I_0 = [(V_1, a);\ Z_0]$ such that $I \in Con1, 1, left(I_0)$ and I_0 is valid for the factorization (a, ϵ) (or (ϵ, a)) of $\#(u_1, v_1)$. This satisfies part (2) of case (B) of the lemma. The same holds of tree (c) in the figure except that $v_1 = a$ and $I \in RC_{1,1}(I_0)$.

2. $(u_1, v_1) = (a_1 \ldots a_p, a_{p+1} \ldots a_q)$. Now, there are three possible subcases; viz. $p = 0$, $p = q$, and $1 \leq p < q$. The first two subcases have similar proofs so we shall only illustrate the instance when $p = 0$.

   In particular, I is valid for $(\epsilon, a_1 \ldots a_q)$ and represents the different trees shown in Figure 10. The situation with respect to tree (a) is similar to the one encountered before; thus we have that there is an item $I_0 = [(V_1, a_r);\ Z_0]$ which is valid for the factorization $(a_1 \ldots a_r, a_{r+1} \ldots a_q)$ (or, the factorization $(a_1 \ldots a_{r-1}, a_r \ldots a_q)$), such that $I \in RC_{1,1}(I_0)$. Part (2) of case (B) of the lemma holds and we are done.

   For the tree (b), it can be verified that the spine y of the tree must have length greater than 1. By part (A) of the lemma, and the fact that I' is valid for y, it must be true that $y = x_1 y_1$, and there are (0)-items Q_1 and Q_2 respectively valid for x_1 and y_1 such that $I' \in W_0(Q_1, Q_2)$. From the figure and the previous definitions, it should be clear that $I \in W_1(I_1, I_2)$ where $I_1 = [(X_1, V_m);\ Q_1]$ and $I_2 = [(V_m, Y_1);\ Q_2]$ are items respectively valid for the factorizations (ϵ, u_2) and (ϵ, v_2). Thus I satisfies part (3) of case (B) of the lemma.

   Finally, consider the case when $1 \leq p < q$. We have only one possible form of simple tree shown in Figure 11. Again the argument in the previous paragraph works; this should be evident from the figure and the foregoing explanation.

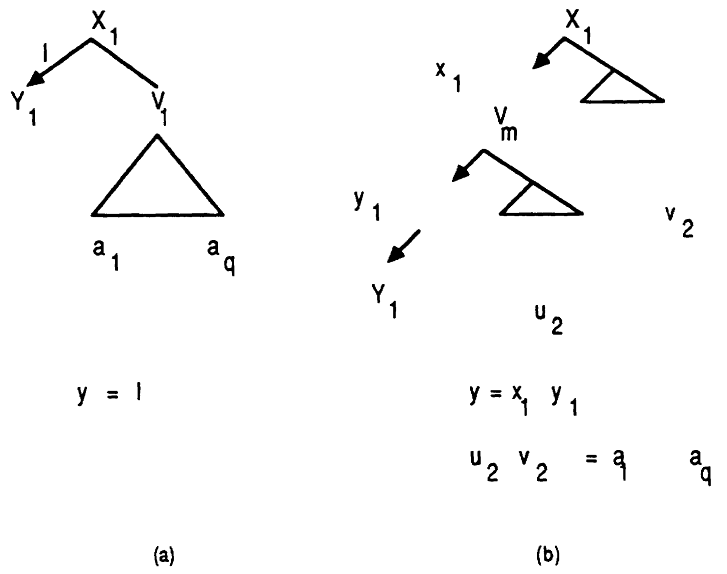

Figure 10: Simple tree corresponding to $x = \#(\epsilon, a_1a_2 \ldots a_q)$

For $i > 1$, the proof essentially follows the same lines except that we use the three cases in statement (B) for our inductive assertions about I' and y. The details are fairly tedious and are omitted here. The reader may work his way through them by using the appropriate definitions introduced previously. □

3.3 The Recognition Algorithm

In the discussion so far, we have explicitly talked about sequences of strings over T_i and their relationships to (i)-items. The algorithm, however, uses a recognition matrix whose entries are indexed according to the specific input string. We therefore, provide some additional notation which relates factorizations of strings to matrix indices.

Definition 3.6 Given $n, i \geq 0$, *Indices*(n,i) is the set of tuples, $(p_1, p_2, \ldots, p_{2^i+1})$, of natural numbers $0 \leq p_1 \leq p_2 \leq \ldots \leq p_{2^i+1} \leq n$, such that there exists an m, $1 \leq m \leq 2^i$, with $(p_{2m} - p_{2m-1}) > 0$. The *size* of $(p_1, p_2, \ldots, p_{2^i+1}) \in$ *Indices*(n,i) is given by

$$\sum_{j=1}^{2^i}(p_{2j} - p_{2j-1}).$$

Whenever the value of n is understood from the context, elements of *Indices*(n,i) will simply be denoted as (i)-indices. Note that the size of every (i)-index is greater than 0.

Let $\alpha = a_1, \ldots, a_n \in T_k{}^+$, the input string, and n, its length, be considered fixed in the subsequent discussion. Let ${}_i\alpha_j$, $0 \leq i \leq j \leq n$, stand for the substring $a_{i+1} \ldots a_j$ of α. In particular, if $i = j$ then ${}_i\alpha_j$ denotes the empty string ϵ.

Given input string $\alpha \in T_k{}^+$, the kth recognition algorithm creates a 2^{k+1}-dimensional matrix M_k, with each dimension indexed from 0 through n (inclusive), which is accessed by (k)-indices. Its entries contain (k)-items and satisfy the following invariant:

Lemma 3.4 Let *Recognizer*$_k$ be the k-th recognition recognition algorithm. Then given an input string $\alpha = a_1a_2 \ldots a_n$ of length n, a (k)-index $(i_1, i_2, \ldots, i_{2^k+1})$, and a (k)-item $[(A_1, B_1, \ldots, A_j, B_j, \ldots, A_{2^i-1}, B_{2^i-1});\ I']$, *Recognizer*$_k$ satisfies the invariant condition

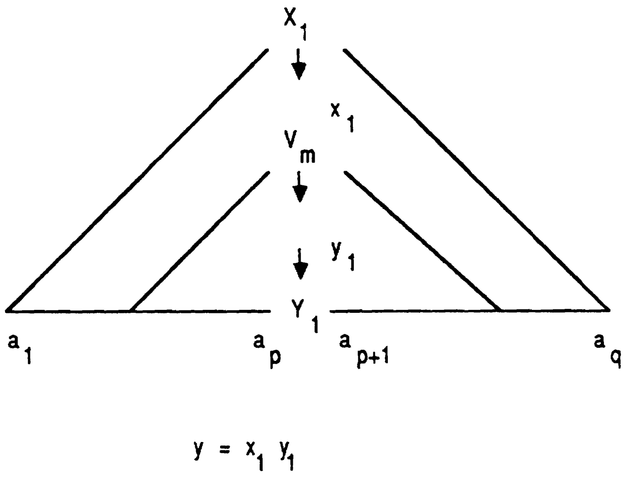

Figure 11: Simple tree corresponding to $x = \#(a_1 \ldots a_p, a_{p+1} \ldots a_q)$

$[(A_1, B_1, \ldots, A_j, B_j, \ldots, A_{2^{i-1}}, B_{2^{i-1}});\ I'] \in M_k(i_1, i_2, \ldots, i_{2^{k+1}})$ iff
$[(A_1, B_1, \ldots, A_j, B_j, \ldots, A_{2^{i-1}}, B_{2^{i-1}});\ I']$ is valid for the tuple
of strings $({}_{i_1}\alpha_{i_2}, \ldots, {}_{i_{2^{k+1}-1}}\alpha_{i_{2^{k+1}}})$ of the input string α.

The algorithm is a *dynamic* programming algorithm, i.e., it starts out with items constructed by $Init_{k,j}$ for $1 \leq j \leq 2^{k-1}$, and then applies the operations W_k, $LC_{k,j}$ and $RC_{k,j}$ on items in appropriately indexed matrix entries which are *already computed.* The resulting (k)-items are inserted into the entry being currently computed. We simply need to ensure that entries of the matrix are accessed in the correct order so as to preserve the invariant.

Observe first that the definitions of the predicates $StartItem_{k,j}$, $SourceItem_k$ and $Compatible_k$ on (k)-items can be translated into corresponding definitions for (k)-indices, as follows:

Definition 3.7 For all (k)-indices $arg_1 = (p_1, p_2, \ldots, p_{2^{k+1}})$, $arg_2 = (q_1, q_2, \ldots, q_{2^{k+1}})$, and $current = (i_1, i_2, \ldots, i_{2^{k+1}})$,

a For any $1 \leq j \leq 2^k$, we say that $StartIndex_{k,j}(current)$ is true iff $(i_{2m} - i_{2m-1}) = 0$ for all $1 \leq m \neq j \leq 2^k$.
Note that the size of *current* is equal to $(i_{2j} - i_{2j-1})$.

b $SourceIndex_k(current)$ is true iff the size of $current = (i_{2^{k+1}} - i_1)$.

c Indices arg_1 and arg_2 are *compatible with current*, denoted $ICompatible_k(arg_1, arg_2, current)$, iff either

- $k = 0$, $p_2 = q_1$, and $[p_1, q_2] = [i_1, i_2]$, or
- $k \geq 1$ and for all $1 \leq m \leq 2^{k-1}$, arg_1 and arg_2 satisfy:
  1. if m is odd, then
     - $[p_{2m}, p_{2^{k+1}-2m+1}] = [q_{2m-1}, q_{2^{k+1}-2m+2}]$, and
     - $[p_{2m-1}, q_{2m}, q_{2^{k+1}-2m+1}, p_{2^{k+1}-2m+2}] = [i_{2m-1}, i_{2m}, i_{2^{k+1}-2m+1}, i_{2^{k+1}-2m+2}]$.
  2. if m is even, then
     - $[q_{2m}, q_{2^{k+1}-2m+1}] = [p_{2m-1}, p_{2^{k+1}-2m+2}]$, and
     - $[q_{2m-1}, p_{2m}, p_{2^{k+1}-2m+1}, q_{2^{k+1}-2m+2}] = [i_{2m-1}, i_{2m}, i_{2^{k+1}-2m+1}, i_{2^{k+1}-2m+2}]$.

Note that the size of $current =$ the size of arg_1 + the size of arg_2.

With these auxiliary definitions, it is easy to see that $Init_{k,j}$ should initialize entries whose indices have size 1 and satisfy either $StartIndex_{k,j}$ or $StartIndex_{k,2^k-j+1}$. Similarly, operation $LC_{k,j}$ (respectively,

$RC_{k,j}$) takes arguments from entries whose indices are of size m for some $m \leq n$, and satisfy $SourceIndex_k$. The resulting items satisfy the predicate $StartItem_{k,j}$ *but do not satisfy* $SourceItem_k$. These items are placed in entries whose indices are of the same size m but satisfy $StartIndex_{k,j}$ (respectively, satisfy $StartIndex_{k,2^k-j+1}$). Finally, W_k is applied to entries indexed by arg_1 and arg_2 and the result placed in *current*, if $ICompatible(arg_1, arg_2, current)$ is true.

The foregoing discussion implies that it suffices to access the matrix in increasing order of the size of its (k)-indices, with no further restriction on the order for (k)-indices *of the same size.*

Algorithm 3.1 (*Recognizer*$_k$) For an input string $\alpha = a_1 \ldots a_n$ of length n, the algorithm creates a 2^{k+1}-dimensional recognition matrix M_k such that matrix entry $M(i_1, i_2, \ldots, i_{2^{k+1}})$ is referenced iff $(i_1, i_2, \ldots, i_{2^{k+1}})$ belongs to $\mathcal{I}_k$.

Initialization:

```
For all ($k$)-indices $arg = (i_1, i_2, \ldots, i_{2^{k+1}})$ such that $arg$ has size 1,
    for all $0 \leq i \leq (n-1)$ if $StartIndex_{k,j}(arg)$ is true for some $j$
        and $i_{2j} = (i+1)$, $i_{2j-1} = i$, do
        perform
            $M_k(arg) := Init_{k,j}(a_{i+1})$;
```

Main Loop:

```
For all ($k$)-indices $current$ of size = 1, 2, ..., $n$ do
    begin
        for all ($k$)-indices $arg_1$ and $arg_2$ such that
        $ICompatible_k(arg_1, arg_2, current)$, do
        perform
              (I) $M_k(current) := M_k(current) \cup W_k(M_k(arg_1), M_k(arg_2))$;
        for all $1 \leq j \leq 2^k$
            if $StartIndex_{k,j}(current)$ then
              begin
                    if $j \leq 2^{k-1}$ then
                      for all ($k$)-indices $arg$ such that $SourceIndex_k(arg)$ is true
                      and size of $arg$ = size of $current$, do
                      perform
                          (II) $M_k(current) := M_k(current) \cup LC_{k,j}(M_k(arg))$;
                    else  % $j > 2^{k-1}$
                      for all ($k$)-indices $arg$ such that $SourceIndex_k(arg)$ is true
                      and size of $arg$ = size of $current$, do
                      perform
                          (III) $M_k(current) := M_k(current) \cup RC_{k,j}(M_k(arg))$;
              end
    end
```

Recognition Condition:

```
If there exists a ($k$)-index $arg$ such that
        (a) size of $arg = n$,
        (b) $SourceIndex_k(arg)$ is true, and
        (c) $M_k(arg)$ contains some item $I = [(Z_k, \ldots); I']$ satisfying $SourceItem_k(I)$
then declare string $\alpha$ **accepted**
else **reject** $\alpha$
```

It may be observed that *Recognizer*$_0$ is simply the CKY algorithm. The instance of the invariant stated in Lemma 3.4 for $k = 0$ is the familiar invariant satisfied by the CKY algorithm, viz. a nonterminal A is in $M_0(i,j)$ iff $A \underset{G_0}{\overset{*}{\Longrightarrow}} a_{i+1} \ldots a_j$. The correctness of the algorithm immediately follows from the Lemma, which can be proved by making use of the soundness and completeness of the operations with respect to items and factorizations of the input string.

The reader will observe that the set $\mathcal{I}_k$ is bounded in size by a constant

$$Q_k \;=\; |\, N_k \,|^{2^k} |\, N_{k-1} \,|^{2^{k-1}} \ldots |\, N_0 \,|$$

which depends only on the sequence of grammars $G_0,\ G_1,\ \ldots,\ G_k$. Consequently, for any set (or, any pair of sets if the operation used is W_k) of (k)-items, the results of applying the operations are sets of (k)-items of constant size $O(Q_k)$ and can be computed in time at most $O({Q_k}^2)$. Moreover, the algorithm uses a constant number of operations.

Now, the main loop of the algorithm is executed $O(n^{2^{k+1}})$ times, once for each (k)-index. Statements (I), (II) and (III) are respectively executed in secondary loops, each of which take $O(n^{2^k})$ time within a main loop iteration, thus giving an overall time complexity of $O(n^{2^k+2^{k+1}}) \;=\; O(n^{3*2^k})$ for the execution of the main loop. The initialization and recognition condition can be implemented in $O(n^{2^k})$ and $O(n^{2^k-1})$ time respectively. Hence,

Corollary 3.1 For any $k \geq 0$, and any control grammar $\mathcal{G}$ generating language L in the family $\mathbf{CLH_k}$, there is a constant Q_k which depends on $\mathcal{G}$ such that *Recognizer*$_k$ accepts L in polynomial time $O(T(n))$ and polynomial space $O(S(n))$ where $T(n) \;=\; {Q_k}^2 n^{3*2^k}$ and $S(n) \;=\; Q_k n^{2^{k+1}}$, for an input string of length n.

4 Parallel Recognition of Languages in CLH

In [9] it was shown that $\mathbf{CFL} = \mathbf{CLH_0}$ is in $\mathbf{NC^{(2)}}$, the class of languages recognizable by simultaneous (log n)-space bounded and (log^2 n)-time bounded alternating Turing machines (ATMs), or equivalently, by uniform boolean circuits of polynomial size and (log^2 n) depth [8]. We generalize this result to the following: the class of languages $\mathbf{CLH_k}$, for any fixed $k \;\geq\; 0$, is in $\mathbf{NC^{(2)}}$. In fact, we prove a stronger theorem, namely,

Theorem 4.1 The class of languages $\mathbf{CLH_k}$, for any fixed $k \;\geq\; 0$, is in **LOGCFL**.

LOGCFL is the class of languages log-space reducible to context-free languages. In [9] it was shown that **LOGCFL** is in $\mathbf{NC^{(2)}}$; thus, we have

Corollary 4.1 The class of languages $\mathbf{CLH_k}$, for any fixed $k \;\geq\; 0$, is in $\mathbf{NC^{(2)}}$.

The proof of Theorem 4.1 uses the well-known characterization of **LOGCFL** in terms of ATMs, namely, **LOGCFL** is exactly the class of languages accepted by simultaneously (log n)-space bounded and polynomial tree-size bounded ATMs.

An alternating Turing machine (ATM) [2, 9, 8] is a generalization of a nondeterministic TM whose state set is partitioned into "universal" and "existential" states. As with a nondeterministic TM, one can view the computation of an ATM as a tree of configurations. A configuration is called universal (existential) if the state associated with the configuration is universal (existential). A computation tree of an ATM M on input w is a tree whose nodes are labeled by configurations of M on w, such that the root is the initial configuration and the children of any non-leaf node labeled by a universal (existential) configuration include all (one) of the immediate successors of that configuration. A computation tree is accepting iff it is finite and all the leaves are accepting configurations. M accepts w if there is an accepting computation tree for M on input w. Note that nondeterministic TMs are essentially ATMs with only existential states. We assume that ATMs have a read-only input tape with endmarkers. We use a variant of an ATM, called an indexing ATM [9], which allows sublinear time bounds. An indexing ATM has a special "index tape"; whenever an integer i is written on the index tape, the i-th symbol of the input is immediately accessible to the ATM. Thus, in $\log n$ steps, it can read any position on the input tape.

A language L is accepted by an ATM M within time $T(n)$ (space $S(n)$) if for every string w in L of length n, there is an accepting computation tree for M on w of height at most $T(n)$ (each of whose nodes is labeled by a configuration using space at most $S(n)$). Similarly, L is accepted by M within tree-size bound $Z(n)$ if for every string in L of length n, there is an accepting computation tree of size (number of nodes) at most $Z(n)$.

Proof of Theorem 4.1: Let L be language in $\mathbf{CLH_k}$ defined by a sequence G_0, G_1, ..., G_k, where G_0 is a context-free grammar in CNF and G_i, $1 \leq j \leq k$ is an LDCFG in CNF. We construct an ATM M which for a given string α checks that α is in L by essentially executing the recursive version of algorithm *Recognizer$_k$* discussed in the previous section. M does this by splitting at a universal state and performing steps I and II below:

I. Guess the length n of the input string α and verify by checking that the $(n+1)$-st symbol of the input tape is the endmarker.

II. Guess a (k)-index P such that size$(P) = n$ and $SourceIndex_k(P)$ holds. Guess a (k)-item $I = [(Z_k, ...); I']$, where Z_k is the start symbol of G_k and $SourceItem_k(I)$ holds. Accept iff $Verify_k(I,P)$ accepts.

Informally, given a (k)-item I and a (k)-index $P = (i_1, i_2, ..., i_{2^{k+1}})$, procedure $Verify_k(I,P)$ accepts iff I is valid for the tuple of strings $({}_{i_1}\alpha_{i_2}, ..., {}_{i_{2^{k+1}-1}}\alpha_{i_{2^{k+1}}})$.

procedure $Verify_k(I,P)$:

1. If size$(P) > 1$ then guess an $r \in \{2,3\}$ and go to step r. Otherwise, if $StartIndex_{k,j}(P)$ holds for some j and $i_{2j} = (i+1)$, then if $I \in Init_{k,j}(a_{i+1})$, accept and halt; else go to step 3.

2. [I obtained via W_k] Guess (k)-indices P_1 and P_2 such that $ICompatible_k(P_1, P_2, P)$ holds. Guess (k)-items I_1 and I_2 such that $I \in W_k(I_1, I_2)$. Accept iff $Verify_k(I_1, P_1)$ and $Verify_k(I_2, P_2)$ both accept.

3. [I obtained via $LC_{k,j}$ or $RC_{k,j}$] If $StartIndex_{k,j}(P)$ does not hold for any j, reject and halt. Otherwise, let j be such that $StartIndex_{k,j}(P)$ holds. If If $j \leq 2^{k-1}$ go to step 3.1; else go to step 3.2.

3.1 Guess a (k)-index P_1 such that $SourceIndex_k(P_1)$ holds and size(P_1) = size(P). Guess a (k)-item I_1 such that $SourceItem_k(I_1)$ holds and $I \in LC_{k,j}(I_1)$. Accept iff $Verify_k(I_1, P_1)$ accepts.

3.2 Guess a (k)-index P_1 such that $SourceIndex_k(P_1)$ holds and size(P_1) = size(P). Guess a (k)-item I_1 such that $SourceItem_k(I_1)$ holds and $I \in RC_{k,j}(I_1)$. Accept iff $Verify_k(I_1, P_1)$ accepts.

end $Verify_k$.

The proof of correctness of the above procedure follows from the proof of correctness of procedure *Recognizer$_k$* and is left to the reader. That the ATM M uses $O(\log\ n)$ space is easily seen from the fact that it stores a constant number of (k)-items and (k)-indices; a (k)-item requires constant space and a (k)-index requires $2^{k+1} \log\ n = O(\log\ n)$ space.

We now show that an accepting computation tree of M on an input of length n has size polynomial in n. Consider the first call to $Verify_k$ in step II. Clearly, the second argument of this call is a (k)-index whose size is n. The execution of this call results in further recursive calls to $Verify_k$ in either steps 2, 3.1 or 3.2. The recursion ends when step 1 is executed, which happens when the argument (k)-index has size 1.

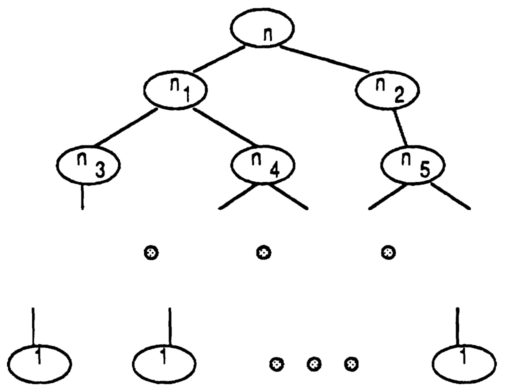

Figure 12: Tree Δ

The sequence of recursive calls can be viewed as binary tree Δ whose nodes are labeled by the sizes of the (k)-indices that appear as arguments in the calls, as illustrated in Figure 12. In tree Δ, a node with two children represents the two recursive calls to $Verify_k$ in step 2, and a node with one child represents the single recursive call in either step 3.1 or step 3.2. A leaf represents the execution of step 1, with argument (k)-index of size 1. (Since we are considering an accepting computation tree, all leaves represent "acceptance" in step 1).

If a node in Δ represents a (k)-index of size m and it has two children representing (k)-indices of sizes m_1 and m_2, then m_1, $m_2 > 0$ and $m_1 + m_2 = m$. On the other hand, if the node has only one child representing a (k)-index P_1 of length m_1, then $m_1 = m$. Now, the (k)-item I_1 corresponding to P_1 should be such that $SourceItem_k(I_1)$ holds. If this is the case, one can verify from the definitions that I_1 cannot be in $LC_{k,j}(I_2)$ or $RC_{k,j}(I_2)$ for any I_2. This implies that if a recursive call to $Verify_k$ results in the execution of either step 3.1 or 3.2, then the next recursive call cannot result in the execution of either of these steps and still guarantee acceptance. Thus, in tree Δ, if node m has only one child m_1, then m_1 must either have two children or be a leaf node.

Let $T(n)$ be the number of nodes in tree Δ whose root represents a (k)-index of size n. Then, for $n > 1$,

$$T(n) \leq \max_{1 \leq j \leq (n-1)} (2 + T(n-j) + T(j)),$$

where the constant 2 represents a chain of at most 2 nodes in the tree, the first of which has the second one as its only child, and the second of which has two children (representing the term $T(n-j) + T(j)$). Since $T(1) = 1$, the solution to the above recurrence is easily seen by induction to be

$$T(n) \leq (3n - 2) = O(n).$$

Finally, we note that the portion of the accepting computation tree Γ of the ATM M whose root corresponds to the first call to $Verify_k$ in step II is "isomorphic" to the tree Δ except that each node in Δ would correspond to $O(\log\ n)$ nodes in Γ to take into account the steps carried out by the ATM in writing the (k)-indices on its worktapes, and in the case of step 1 of $Verify_k$, in looking at a symbol on the input tape. Together with the $O(\log\ n)$ nodes required by step I, the size of the accepting computation tree is thus $O(n\ \log\ n)$.

5 Conclusions

We have shown that a hierarchy of non-context-free language classes generated by control grammars can be recognized in polynomial time, settling an open problem posed in [15]. Previously, the best known upper bound was exponential time. We have also shown that every language class in this hierarchy is in $\mathbf{NC}^{(2)}$, generalizing Ruzzo's result [9] that the class of context-free languages is in $\mathbf{NC}^{(2)}$.

An interesting question that we have not addressed is the following: suppose that the control set of a given control grammar is a language not generated by some grammatical family but instead a language from some general complexity class, say $DSPACE(\log n)$ or $PTIME$, what can be said about the complexity of the language generated by this control grammar? In other words, a control grammar can be thought of as the grammatical analog of an oracle Turing machine, with the control set taking the role of an oracle. It would be interesting to investigate whether such control grammars give rise to complexity hierarchies similar to the logspace and polynomial-time hierarchies defined in [2, 10, 11].

A Chomsky Normal Form for CLH

We prove the Chomsky Normal Form lemma 2.1; i.e. the existence of a defining sequence of grammars for a language L in the family $\mathbf{CLH_k}$, where each grammar in the sequence is in CNF as defined in an earlier section.

Let G be an arbitrary LDCFG. Productions in G of the form $l \;:\; X \rightarrow \check{\epsilon}$ and $l \;:\; X \rightarrow \check{Y}$ for nonterminals X, Y are respectively called ϵ-productions and *chain*-productions of G. The grammar is said to be in *two-normal-form* if every labeled production of G either has exactly two nonterminal symbols on the right-hand side, or has a single terminal symbol or ϵ on the right-hand side. G is said to be in Chomsky Normal Form (abbreviated as CNF) iff it is in two-normal-form and furthermore, does not contain any ϵ-productions.

For the basis of the proof, it suffices to know that any context-free language can be generated by a context-free grammar in CNF, cf. [5]. Assume that every language in $\mathbf{CLH_{k-1}}$ is definable by a sequence of grammars in CNF (see definition 2.3).

Consider $L \in \mathbf{CLH_k}$ for some k, and let $L \;=\; L(\{G, C\})$ for some LDCFG G and a control set $C \in \mathbf{CLH_{k-1}}$. Let V_N and V_T denote the nonterminals and terminals of G respectively. In an intermediate step, we produce from G and C, a control grammar $\{H, D\}$ such that H is in two-normal-form, $D \in \mathbf{CLH_{k-1}}$, and $L \;=\; L(\{H, D\})$.

A.1 Two Normal Form

The construction involves the following two stages:

Stage I

- For every terminal symbol a of G, we introduce a new nonterminal V_a, a new label l_a and a new production $l_a \;: V_a \rightarrow \check{a}$. Now, consider a non-CNF production $p \;=\; l : X \rightarrow X_1 \ldots \check{X}_i \ldots X_n$ of G. This production is transformed to $p \;=\; l : X \rightarrow Y_1 \ldots \check{Y}_i \ldots Y_n$, where $Y_j \;=\; X_j$ if X_j is a nonterminal symbol, or else $Y_j \;=\; V_a$ if $X_j \;=\; a$, where a is terminal. Simultaneously, we define the substitution

$$\Theta_1(l) \;=\; \begin{cases} l & \text{if } X_i \text{ is a nonterminal.} \\ l \cdot l_a & if\; X_i \;=\; a. \end{cases}$$

  Let G_1 be the new LDCFG with the foregoing additions and transformations to G.

- We define the new control set C_1 to be

$$C_1 \;=\; \{l_a \;|\, a \;\in V_T\} \cup \Theta_1(C)$$

It is very easy to show that $L \;=\; L(\{G, C\}) \;=\; L(\{G_1, C_1\}$; the detailed proof is left to the reader.

Stage II

- Let V_ϵ be a new nonterminal, and $l_\epsilon :\; V_\epsilon \rightarrow \check{\epsilon}$ be a new production. For every *chain*-production $l :\; X \rightarrow \check{Y}$, we transform it to the new production $l :\; X \rightarrow \check{Y} V_\epsilon$.
- For every production $p \;=\; l : X \rightarrow X_1 \ldots \check{X}_i \ldots X_n$ of G_1, with $n > 2$, we discard p and create

instead the following set of new productions (see Figure 13):

$$\begin{array}{lll}
l_1 & : & X \rightarrow X_1 \check{Y}_1, \\
\vdots & & \\
l_{i-1} & : & Y_{i-2} \rightarrow X_{i-1} \check{Y}_{i-1}, \\
l_i & : & Y_{i-1} \rightarrow \check{X}_i Y_i, \\
l_{i+1} & : & Y_i \rightarrow X_{i+1} \check{Y}_{i+1}, \\
\vdots & & \\
l_{n-2} & : & Y_{n-3} \rightarrow X_{n-2} \check{Y}_{n-2}, \\
l_{n-1} & : & Y_{n-2} \rightarrow X_{n-1} \check{X}_n .
\end{array}$$

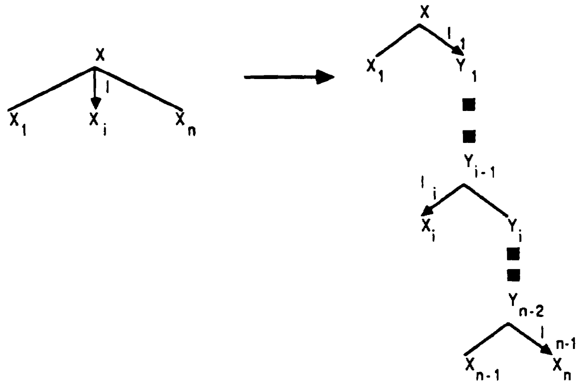

Figure 13: Expansion of a production to two-normal-form productions.

Simultaneously, define the substitution Θ_2 such that $\Theta(l) = l_1 l_2 \ldots l_i$, and include the string $l_{i+1} l_{i+2} \ldots l_{n-1}$ to a set *Prefixes*. Let H be the new LDCFG so obtained from G_1.

- Let the new control set be given by

$$D = \Theta_2(C_1 \cup \mathit{Prefixes} \cdot C_1) \cup \{l_\epsilon\}$$

where $\cdot$ (dot) represents concatenation of languages.

Note that the substitutions Θ_1 and Θ_2 behave like the identity substitutions on labels not specified above. Clearly, H is an LDCFG in two-normal-form. A straightforward inductive proof can be constructed to show that control grammars $\{G_1, C_1\}$ and $\{H, D\}$ generate the same languages. Now every family in **CLH** is a full AFL [15], and C_1 and D are obtained in stages I and II by applying the operations of regular substitution, union and concatenation. Since AFLs are closed under these operations, it follows that C_1 and D are both languages in $\mathbf{CLH_{k-1}}$. Applying the inductive hypothesis to D, we see that L is defined by combining a sequence of grammars in CNF (defining D), with LDCFG H which is in two-normal-form. Let the entire sequence of grammars defining L be $(H_0, H_1, \ldots, H_{k-1}, H_k = H)$, where each component grammar H_j, $0 \leq k-1$ is in CNF, and $H_k = H$ is in two-normal-form. We now use this sequence to eliminate ϵ-productions.

A.2 Elimination of ϵ-productions

Given an LDCFG H with control set D, we say that a nonterminal A of H is *valid* for string w iff there is a derivation tree Γ in $\mathit{TreeSet}(A \overset{*}{\underset{H}{\Longrightarrow}} w)$ such that $\mathit{ControlWords}(\Gamma) \in D$. Let T be the set of nonterminals of H which are valid for ϵ.

T is obtained from H in a somewhat complicated way, because verifying that all the control words in a derivation tree in $TreeSet(X \overset{*}{\underset{H}{\Longrightarrow}} \epsilon)$ essentially forces us to use the recognizer at lower levels, i.e. CLH_{k-1}, CLH_{k-2} etc. This is done in the following way. Let

$$S_0 = \bigcup_{1 \leq j \leq 2^{k-1}} Init_{k,j}(\epsilon)$$

Inductively, for $i \geq 0$, let

$$S^1{}_{i+1} = S_i \bigcup_{1 \leq j \leq 2^{k-1}} (LC_{k,j}(S_i) \cup RC_{k,j}(S_i)), \text{ and}$$

$$S_{i+1} = S^1{}_{i+1} \cup W_k(S^1{}_{i+1}, S^1{}_{i+1})$$

We terminate this iterative process as soon as $S_{i+1} = S_i$ for some $i \geq 0$. It is left to the reader to verify that that the number of iterations i is bounded by the number of possible (k)-items of the grammar sequence $H_0, \ldots, H_k$. We now claim that:

Claim A.1 Let $S = \{I \mid I$ is a (k)-item such that I is valid for the 2^{k+1}-tuple of strings $(\epsilon, \epsilon, \ldots, \epsilon)\}$. Then $S_i = S_{i+1}$ for some $i \geq 0$ if and only if $S_i = S$.

One direction is trivial, i.e. $S_i = S$ implies $S_i = S_{i+1}$. For the forward implication, it should be clear from the definitions that $S_i \subseteq S$ for all $i \geq 0$. Assume to the contrary that $S_i = S_{i+1}$ but $S_i \neq S$. Then there is an item I in the set $S - S_i \neq \emptyset$. By definition of S, there are possibly many sequences of simple trees represented by I (which is valid for the 2^{k+1}-tuple of empty strings). Let us denote these sequences by $MT(I)$. From among all sequences in $\cup_{I \in (S-S_i)} MT(I)$, we shall choose the smallest sequence, i.e the one with minimum total depth, and with minimum total spine length [3]. Let this sequence be represented by item $I' \in (S - S_i)$.

If one recalls lemma 3.3, it is easy to show that either $I' \in Init_{k,j}(\epsilon)$, or there must be an item I_1 such that $I' = LC_{k,j}(I_1)$ for some j (or, $I' = RC_{k,j}(I_1)$), or there are items I_1 and I_2 such that $I' = W_k(I_1, I_2)$. If $I' \in Init_{k,j}(\epsilon)$ then $I' \in S_0 \subseteq S_i$, by definition, and hence cannot be in $S - S_i$, a contradiction. Otherwise, if I_1 (for the first two cases) or I_1 and I_2 (in the final case involving W_k) are in S_i, then the construction of S_{i+1} guarantees that I' is in S_{i+1}. But $S_i = S_{i+1}$, thus yielding a contradiction to our assumption that $I' \notin S_i$. The only remaining possibility, viz. I_1 (respectively, I_1 or I_2 or both) is (are) not in S_i, violates the minimality of our choice of I'. It follows then that our assumption must be wrong, i.e. $S_i = S$.

It is easy to see that

$$T = \{X \mid I = [(X, \ldots, \epsilon);\ \ldots] \in S, \text{ and } SourceItem_k(I) = true\}.$$

Also, consider the set $T' \subseteq T$ defined by:

$$T' = \{X \mid [(X = A_1, B_1, \ldots, A_{2^{k-1}}, B_{2^{k-1}} = \epsilon);\ \ldots] \in S, \text{ and } B_j = A_{j+1} \text{ for all } 1 \leq j \leq 2^{k-1} - 1\}$$

Clearly, both T and T' can be computed syntactically from S.

Then, for every labeled production $l : X \rightarrow X_1\check{X}_2$ (the other possibility where symbol X_1 is distinguished is handled in a similar way), we consider the following two cases:

1. $X_1 \in T$. Then, a new production $p_l : X \rightarrow \check{X}_2$, is introduced into the grammar, with the substitution $\Theta(l) = \{l, p_l\}$.

[3]The total depth of a simple tree sequence is the sum of the depths of the simple trees in the sequence. The total spine length is similarly the sum of the lengths of the spines of the simple trees in the sequence.

2. $X_2 \in T'$. In this case, we create a new production $q_l : X \rightarrow \check{X}_1$. Let C_l denote the language $l.Label_k^*$.

All ϵ-productions in H are now eliminated, and the new control set is specified by the expression [4]:

$$D_1 = \bigcup_{l \in Label_k} \Theta([D/C_l]^* D)$$

The new LDCFG is denoted by H_1. It is easy to see that if $w \in L(\{H, D\})$ then $w \in L(\{H_1, D_1\})$ by induction on the number of productions used in a proper derivation of w from the start symbol of H (which remains unchanged in H_1). The basic idea is that subtrees of the derivation tree which derive ϵ are pruned, and either the substitution Θ is applied, or a new control path is below. These cases are shown in Figures 14 and 15.

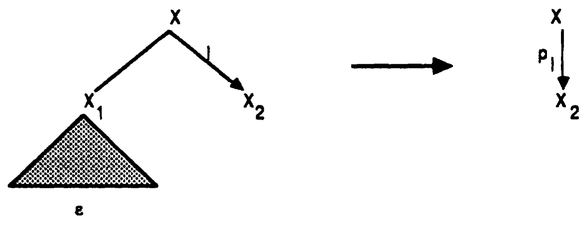

Figure 14: $X_1 \in T$.

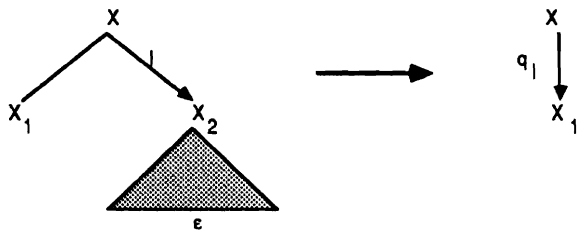

Figure 15: $X_2 \in T'$.

The converse is slightly more tricky, but essentially runs along the same lines as the previous paragraph. Hence if $w \in L(\{H_1, D_1\})$ then we undo the transformations and the substitution wherever applied in the derivation tree of $\{H_1, D_1\}$ to get a derivation tree of $\{H, D\}$ for w. It is crucial to recognize that without specifying set T, the transformation outlined above will not work (the standard procedure for eliminating ϵ-productions from a CFG requires specifying only the set of nonterminals which derive the empty string; our procedure must take into account the fact that a derivation tree may contain invalid paths not in the control set, which cannot be eliminated in the transformation).

A.3 Elimination of chain-productions

To get back to matters at hand, we have now produced a control grammar with no ϵ-productions, but which may contain *chain*-productions. These can be eliminated as follows. Let $V_N = (V_1, \ldots, V_p$ be the set of nonterminals of a control grammar free of ϵ-productions. Let the set $Chain(V_j)$ is the set of nonterminals V_i such that $V_i \overset{*}{\underset{G}{\Rightarrow}} V_j$. This set can be computed by standard methods.

[4]The operator / denotes *right-quotient of languages*. Thus, L/C is the language $\{x \mid$ there is a string $y \in C$ such that $xy \in L\}$. Full AFLs are closed under right-quotient with regular sets.

Remark: The language

$$L_{chain}(V_i, V_j) \;=\; \{\omega \mid V_i \overset{*}{\underset{G}{\Longrightarrow}} V_j \text{ via a derivation tree with spine } \omega\}$$

is *regular*.

Now, for every non-chain production in the original LDCFG of the form $l: \; X \rightarrow X_1\check{X}_2$ (or with X_1 distinguished), we create a new production:

$$l_{V_i} : \; V_i \rightarrow X_1\check{X}_2$$

for every nonterminal $V_i \in Chain(X)$. Simultaneously, let Θ be a substitution where $\Theta(l_{V_i}) \;=\; L_{chain}(V_i, X)$.

The new LDCFG is obtained by removing all chain-productions from the original one and incorporating the new productions described above. The new control set is simply $\Theta^{-1}(D)$, where D is the original control-set. Note that all the operations described so far are full AFL operations [5]; consequently the new control set is still a member of $\mathbf{CLH_{k-1}}$. It is clear that the resulting LDCFG is also in CNF. The inductive hypothesis now applies to the new control set, thus proving the normal-form lemma.

[5]Full AFLs are closed under inverse regular substitutions; for a comprehensive discussion, see [6]

References

[1] G. E. Barton, R. C. Berwick, and E. S. Ristad. *Computational Complexity and Natural Languages.* MIT Press, Cambridge, MA, 1987.

[2] A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer. Alternation. *J. ACM*, 28:114–122, 1981.

[3] S. Ginsburg and S. A. Greibach. Abstract families of languages. *Mem. Am. Math. Soc.*, 87(1):1–32, 1969.

[4] S. Ginsburg and E. H. Spanier. AFL with semilinear properties. *J. Comput. Syst. Sci.*, 5:365–396, 1971.

[5] M. A. Harrison. *Introduction to Formal Language Theory.* Addison-Wesley, Reading, MA, 1978.

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

[7] N. A. Khabbaz. A geometric hierarchy of languages. *J. Comput. Syst. Sci.*, 8:142–157, 1974.

[8] W. L. Ruzzo. On uniform circuit complexity. *J. Comput. Syst. Sci.*, 22:365–383, 1981.

[9] W. L. Ruzzo. Tree-size bounded alternation. *J. Comput. Syst. Sci.*, 21:218–235, 1980.

[10] W. L. Ruzzo, J. Simon, and M. Tompa. Space-bounded hierarchies and probabilistic computations. *J. Comput. Syst. Sci.*, 28:216–230, 1984.

[11] L. Stockmeyer. The polynomial-time hierarchy. *Theoretical Comput. Sci.*, 3:1–22, 1977.

[12] I. H. Sudborough. On the tape complexity of deterministic context-free languages. *J. ACM*, 25(3):405–414, July 1978.

[13] J. W. Thatcher. Tree automata: An informal survey. In A. V. Aho, editor, *Currents in the Theory of Computing*, pages 143–172, Prentice Hall Inc., Englewood Cliffs, NJ, 1973.

[14] K. Vijay-Shanker. *A Study of Tree Adjoining Grammars.* PhD thesis, University of Pennsylvania, Philadelphia, Pa, 1987.

[15] D. J. Weir. *Context-Free Grammars to Tree Adjoining Grammars and Beyond.* Technical Report, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, 1987.