Technical Report No. 2006-514

All Or Nothing?
Finding Grammars That Are More Than Context-Free, But Not Fully Context-Sensitive

Sarah-Jane Whittaker

August, 2006

Abstract

This survey paper describes several extensions of context-free grammars as defined in the Chomsky hierarchy. These augmentations use a variety of different strategies to create grammars that generate all context-free and some context-sensitive languages. The basic concepts of each of these extensions will be examined, as well as whether the enhanced grammars inherited some of the desirable properties normally associated with context-free grammars. In addition, a brief examination of non-Chomsky-related augmentations is included to show that the definition of "extension" is open to interpretation.

Contents

List of Figures

1 Introduction

Noam Chomsky is nothing if not influential. His language hierarchy has both enticed and disappointed those working with natural languages for quite some time. While context-free grammars and languages offer tantalizingly sweet properties such as parse trees, their restrictive nature leaves a sour taste. In contrast, context-sensitive grammars and languages are so broad that they require biting off more than can be chewed while lacking the same luscious flavour. Neither provides any real sustenance for a starving linguist.

Perhaps then there is reason to consider a graft: create fruit with sweetness and size enough to feed a researcher. If context-free grammars could be extended to generate some languages that are context-sensitive, their status may be upgraded from “frustrating” to “useful”. This is exactly what at least seven researchers over at the past 38 years have strived to do. The result is a wealth of options, some more complex than others, but all achieving the basic requirement of extending their territory across a defined border.

Of course, the definition of the word “extension” is not exclusive to the Chomsky hierarchy. Some have interpreted this word in much broader terms, resulting in language and grammar augmentations which do not necessarily eschew Chomsky but instead seek to address different challenges in the realm of natural languages. Although this report is primarily concerned with Chomsky-related extensions, it would be biased to not consider a few different approaches.

Thus, this survey will proceed as follows. Section 2 will introduce the reader to or remind the reader of the basics of Chomsky grammars. Sections 3 and 4 contain heavily distilled summaries of seven “standard” extensions of context-free grammars. Finally, Section 5 concludes the body of this report with brief examinations of two “nonstandard” extensions which attempt to solve problems that are not considered by the previous extensions.

2 Background

The definitions and examples in this section were either culled from [12] and [15] or are original work unless otherwise stated.

2.1 Chomsky Grammar

A *Chomsky grammar* [2], [3] is defined as a four-tuple $G = (T, N, R, S)$ where T is an alphabet of terminals, N is an alphabet of non-terminals, R is a set of rewriting rules (otherwise known as productions) and S is the start symbol, which is also a non-terminal. The rules contained in R are of the form $LHS \rightarrow RHS$, where LHS and RHS are mixed strings of terminals and non-terminals. A grammar G produces its associated language $L(G) \subseteq T^*$ through a series of *derivations*. Beginning with the start symbol S, rules from R are repeatedly applied by replacing a substring of the form LHS with a substring of the form RHS. Note that only strings composed entirely of terminals are members of the associated language; strings composed of terminals and non-terminals are only generated at intermediate steps. A string is produced *ambiguously* if there exist two or more different derivations that yield it. A grammar is considered to be ambiguous if one or more of its strings can be generated ambiguously.

Example Consider grammar $G = (\{a, b\}, \{S, A, B\}, R, S)$ where R contains the following rules:

- $S \rightarrow AB$
- $A \rightarrow ABA$ and $A \rightarrow a$
- $B \rightarrow BAB$ and $B \rightarrow b$

The string ab is generated by two separate derivations: $S \rightarrow AB \rightarrow aB \rightarrow ab$ and $S \rightarrow AB \rightarrow Ab \rightarrow ab$. Thus, the grammar G is ambiguous.

2.2 Chomsky Hierarchy

Depending on the form of its rules, each Chomsky grammar will fall into one of four categories which form the *Chomsky hierarchy*. These types are defined as follows:

(0) Unrestricted

(1) Context-sensitive - LHS must be longer than RHS

(2) Context-free - LHS must be a single non-terminal

(3) Regular - Either left-linear or right-linear; LHS must be a single non-terminal and

  (LL) RHS must be a single non-terminal followed by a single terminal

  (RL) RHS must be a single terminal followed by a single non-terminal

Example Consider grammar $G_1 = (\{a, b, c\}, \{S, B\}, R, S)$ containing rules

- $S \rightarrow aSBc$ and $S \rightarrow abc$
- $cB \rightarrow Bc$
- $bB \rightarrow bb$

Grammar G_1 is context-sensitive and generates the language $\{a^n b^n c^n \mid n \geq 1\}$. In contrast, $G_2 = (\{a, b\}, \{S\}, \{S \rightarrow aSb, S \rightarrow \epsilon\}, S)$ is context-free and generates the language $\{a^n b^n \mid n \geq 0\}$. Finally, $G_3 = (\{a\}, \{S\}, \{S \rightarrow a, S \rightarrow aS\}, S)$ is regular (right-linear, specifically) and generates the language $\{a^n \mid n \geq 1\}$.

2.3 Language Properties

The languages generated by each level of the Chomsky hierarchy have very desirable set properties. The language sets associated with each type are closed under $\cup, \cdot$ and $*$ while all types **but** context-free are closed under $\cap$. However, context-free languages have some impressive features which the other languages in the hierarchy do not possess. A context-free derivation can be represented in tree form, yielding a *parse tree*. The rules associated with this type of grammar can be simplified so that they yield the same language yet satisfy a *normal form* which can also provide properties for proving theorems. Finally, determining whether or not a given string is generated by a particular context-free grammar (known as the *membership problem*) is decidable in polynomial time. One of the better-known algorithms for this purpose is the Cocke-Younger-Kasami (CYK) algorithm which has an average time of n^3 for a string of length n ([23], verified using [9]).

Example None of the grammars seen thus far in this section are in Chomsky normal form, which is one of the most popular; the rule set $\{S \rightarrow a, S \rightarrow SS\}$ is. The parse tree for the string *aabb* generated by grammar G_2 appears in Figure 1.

2.4 What About Natural Languages?

Although the previous subsection may have given the impression that context-free grammars are truly sweet, they sour a bit when it comes to applying them to natural languages. Type 2 grammars are simply too restrictive; by not allowing context (in the

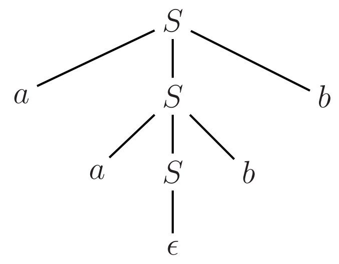

Figure 1: Example - Parse Tree

form of a single non-terminal as the LHS of a rule), any attempt at a context-free representation of a natural language would exclude certain valid phrases, or include nonsense phrases, or both. In contrast, context-sensitive languages are actually too inclusive. The advantage of generating a larger syntax comes at the price of desirable context-free properties; for example, the membership problem for Type 1 grammars is decidable in exponential time, not polynomial. This leads to an obvious question: is there a gray area somewhere between these two types? Is there a recipe for a cake (which contains all context-free and some context-sensitive languages) that can be eaten (via closure and other desirable properties) too?

3 Extensions, Part One

3.1 Indexed Grammars

3.1.1 Definition

Indexed grammars [1] were first proposed by Alfred Aho in 1968. Such a grammar is defined as a five-tuple $G = (N, T, F, P, S)$ where N is an alphabet of non-terminals, T is an alphabet of terminals, F is a set of flags or indices, P is a set of productions and S is the start symbol. This definition is similar to that of a "standard" Chomsky grammar with the exception of the flag set which affects how rules are applied.

Productions for an indexed grammar operate much like those for a standard grammar, except now the RHS can be composed of terminals and non-terminals followed by any number of flags; formally, $(NF^* \cup T)^*$. It is vital that productions generate flags next to non-terminals because flags work like context-sensitive placeholders for other rules where the RHS has the form $(N \cup T)^*$. The non-terminal directly to the left of the flag determines which of the index's rules will be applied when the flag is *consumed*.

The other significant property of note with respect to flags is the manner in which they "stick" to non-terminals when productions are applied and other indices are consumed. The best method for illustrating how this works is to use an example; a formal definition of the process can be found in [1].

Consider the word $w = abA\mathbf{jk}Bba$ where $\mathbf{j}$ and $\mathbf{k}$ are indices and the production $A \rightarrow D\mathbf{f}cC$ can be applied. When A is replaced, $\mathbf{f}$ will take precedence and stick to the non-terminals D and C with the flags $\mathbf{j}$ and $\mathbf{k}$ following. The result in this particular case is $w' = abD\mathbf{fjk}cC\mathbf{jk}Bba$. The same thing happens when a index is consumed. Consider the word $w = abA\mathbf{fjk}Bba$ where $\mathbf{f} = \{B \rightarrow b, A \rightarrow cDC\}$. When $\mathbf{f}$ is consumed along with the non-terminal A to its left, the result is $w' = abcD\mathbf{jk}C\mathbf{jk}Bba$.

Example Let $G = (\{S, A, B\}, \{a, b, c\}, \{f, g\}, P, S)$ be an indexed grammar with the following production and flag definitions:

- $S \rightarrow aA\mathbf{f}c$
- $A \rightarrow aA\mathbf{g}c$
- $A \rightarrow B$
- $\mathbf{f} = \{B \rightarrow b\}$
- $\mathbf{g} = \{B \rightarrow bB\}$

This grammar generates the language $\{a^n b^n c^n | n \geq 1\}$. The string $aabbcc$ is produced via the following derivation:

$$\begin{array}{rcll} S & \rightarrow & aA\mathbf{f}c & S \rightarrow aA\mathbf{f}c \\ & \rightarrow & aaA\mathbf{gf}cc & A \rightarrow aA\mathbf{g}c \\ & \rightarrow & aaB\mathbf{gf}cc & A \rightarrow B \\ & \rightarrow & aaB\mathbf{f}bcc & \mathbf{g} : B \rightarrow bB \\ & \rightarrow & aabbcc & \mathbf{f} : B \rightarrow b \end{array}$$

3.1.2 Properties

Indexed grammars boast virtually all of the desirable properties normally associated with context-free grammars. There exist both "reduced" and normal forms, although the RHS now allows for flags as well as terminals and non-terminals. The languages produced by indexed grammars are closed under $\cup$, $\cdot$ and $*$ with themselves and are closed under $\cap$ with regular languages. Note that this is even an improvement over context-free grammars, as they are not closed under $\cap$ with any other language. Although an algorithm is given in [1] to solve the membership problem, it is unfortunately exponential in the number of non-terminals in the grammar as opposed to polynomial in the length of the input string. Indexed grammars also provide parse trees, the only difference being that internal nodes can now be comprised of non-terminals and flags. Finally, and most importantly, indexed grammars can generate all context-free languages and some, but not all, context-sensitive languages.

3.2 Programmed Grammars

3.2.1 Definition

Programmed grammars [13] were introduced by Daniel Rosenkrantz in 1969, very shortly after indexed grammars appeared. This grammar is defined as a five-tuple $G = (N, T, J, P, S)$ where N is an alphabet of non-terminals, T is an alphabet of terminals, J is a set of production labels, P is a set of productions and S is the start symbol. Again, this definition is nearly identical to that of a Chomsky grammar except for the production labels and how they affect derivations.

In addition to the normal $LHS \rightarrow RHS$ form of a rule, each production also possesses an integer label and two sets of *success* and *failure* labels. Beginning with rule #1, if the LHS is present in the current derivation then the production rule is applied and is consider a success; otherwise it is deemed a failure. The next production is then selected from the success or failure sets; which one is purely the choice of the user. If the set is empty, the derivation halts.

As the name suggests, a derivation in a programmed grammar is rather similar to programming in a language with numbered lines. The success and failure options even operate much like an `if-else` statement. However, representing a derivation is slightly different due to these differences. Formally, a derivation would be composed of word-label pairs (w, j) and steps of the following form: $(w, j) \rightarrow (w', r)$ for a successful application and rule selection r or $(w, j) \rightarrow (w, q)$ for a failure and rule selection q.

Example Let $G = (\{S, A, B, C\}, \{a, b, c\}, J, P, S)$ be an indexed grammar with the following labeled productions:

- $S \rightarrow ABC\{2, 5\}\{\}$
- $A \rightarrow aA\{3\}\{\}$
- $B \rightarrow bB\{4\}\{\}$
- $C \rightarrow cC\{2, 5\}\{\}$
- $A \rightarrow a\{6\}\{\}$
- $B \rightarrow b\{7\}\{\}$
- $C \rightarrow c\{\}\{\}$

This grammar generates the language $\{a^n b^n c^n | n \geq 1\}$. The string *aabbcc* is produced via the following derivation:

$$\begin{array}{rcll}
S & \rightarrow & ABC & S \rightarrow ABC,\ \{2,5\},\ \text{select } 2 \\
 & \rightarrow & aABC & A \rightarrow aA,\ \{3\} \\
 & \rightarrow & aAbBC & B \rightarrow bB,\ \{4\} \\
 & \rightarrow & aAbBcC & C \rightarrow cC,\ \{2,5\},\ \text{select } 5 \\
 & \rightarrow & aabBcC & A \rightarrow a,\ \{6\} \\
 & \rightarrow & aabbcC & B \rightarrow b,\ \{7\} \\
 & \rightarrow & aabbcc & C \rightarrow c,\ \{\}
\end{array}$$

3.2.2 Properties

Unfortunately, [13] does not focus as intently on the properties of the languages generated by programmed grammars. It is given that programmed grammars are closed under $\cup$, $\cdot$ and $*$ with themselves but are not closed under $\cap$ with any other language. Rosenkrantz also states that programmed grammars can generate all context-free languages and a subset of context-sensitive languages. However, there is no information with respect to normal forms, the membership problem or parse trees. It is possible (and perhaps likely) that others have examined these topics, but a moderate search proved fruitless.

3.2.3 A Further Extension

In [4], Mariusz Flasiński and Janusz Jurek created an extension for an extension, resulting in *dynamically programmed LL(k)* context-free grammars and automata. Note that only a brief overview will be included here—please refer to the cited source for formal definitions.

This grammar is like its original counterpart with a few modifications: every production possesses an input tape (like that for a Turing machine) and may perform the operations *add*, *read* and/or *move*. A separate success or failure condition based on these tape operations is also given; it is no longer dependent on whether or not the core rule can be applied. Derivations must also satisfy conditions that make them deterministic and limit "recursive steps".

With this extension in mind, a dynamically programmed LL(k) automaton is then constructed from specialized automata, tapes, tables and stacks to perform the grammar's function. A diagram of this structure can be found in Figure 2. A dynamically

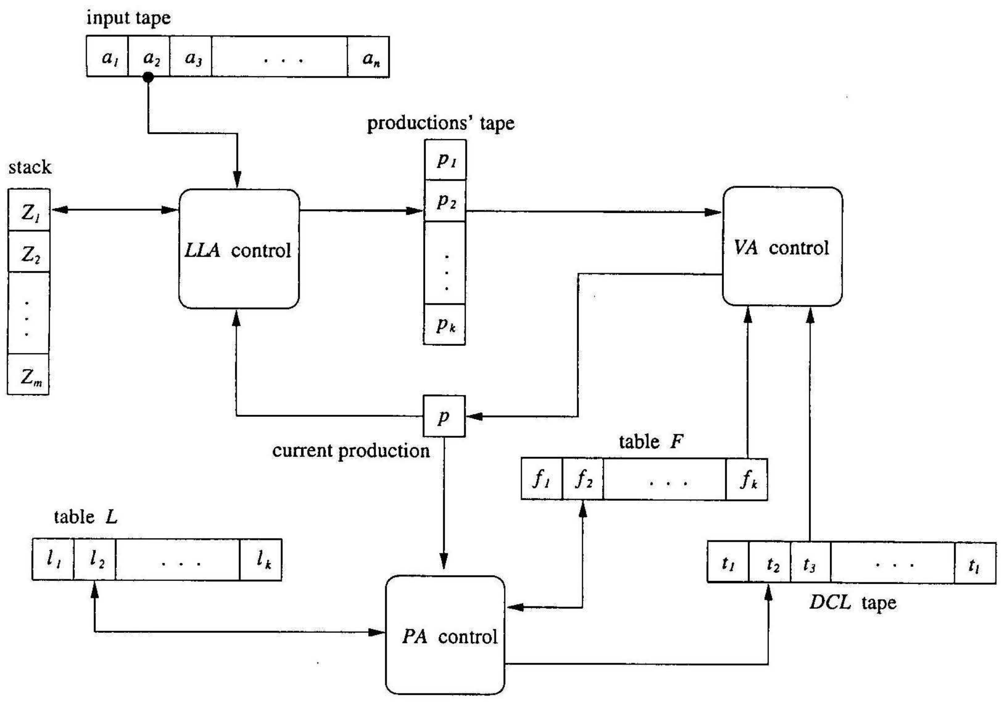

Figure 2: Dynamically Programmed LL(k) Automaton [4]

programmed LL(k) automaton has three primary components that drive the system. The first is the LL(k) sub-automaton (LLA), which reads from an input tape and propose possible productions for the current stage of the derivation. Next, the validating automaton (VA) chooses a production, followed by the programming automaton (PA) performing the actual production and writing the result to the derivation control tape (DCL). Direction is then passed back to the LLA automaton and the process described above repeats until the derivation is complete.

Dynamically programmed LL(k) grammars do not generate the same set of languages that "standard" programmed grammars do, although it is stated that the associated language set is "comparable". Having said that, dynamically programmed LL(k) automata have very desirable complexity properties: in relation to the length of the input, the computation time is polynomial and the storage space required is linear. There's also an interesting application for dynamically programmed LL(k) grammars and automata. The research that yielded these concepts was done in relation to expert systems and has been applied as a data diagnostic tool for the ZEUS detector, which monitors the energy and direction of particles in the HERA particle accelerator.

3.3 Boolean Grammars

3.3.1 Definition

Boolean grammars [9] were created by Alexander Okhotin for his Ph.D. thesis in 2003. This grammar is defined as a five-tuple $G = (\Sigma, N, P, S)$ where Σ is an alphabet of terminals, N is an alphabet of non-terminals, J is a set of production labels, P is a set of productions and S is the start symbol. This definition is identical to that of a Chomsky grammar, but the RHS of each production is now a formula of terminals, non-terminals and Boolean operators. Since the language theory behind Boolean grammars is somewhat involved and difficult to summarize, the following explanation is somewhat loose in its definitions. For the formal aspects of Boolean grammars, please refer to the cited source.

Consider a non-terminal A from any grammar G. If A were made the start symbol of G, you could remove any of the terminals, non-terminals and rules that are "inaccessible", *i.e.*, never encountered in any possible derivation starting from A. Let G_A represent this new grammar and let $L(G_A)$ denote the "language of A within G" or $L_G(A)$. This notation could be expanded for strings of terminals and non-terminals using the concatenation operation. For example, $L_G(aAbB) = \{a\} \cdot L_G(A) \cdot \{b\} \cdot L_G(B)$.

A Boolean language formula has three components: $\alpha, \beta \in (\Sigma \cup N)^*$ and Boolean operators & and $\neg$. These formulae are interpreted as follows: $\neg\alpha = \Sigma^* \backslash L_G(\alpha)$ and $\alpha \& \beta = L_G(\alpha) \cap L_G(\beta)$. Productions for Boolean grammars have (of course) a single non-terminal on the LHS and a Boolean language formula on the RHS.

Example Let $G = (\{a, b, c\}, \{S, A, P, Q, C\}, P, S)$ be a Boolean grammar with the following productions:

- $S \rightarrow AP\&QC$
- $A \rightarrow Aa \mid \epsilon$
- $P \rightarrow bPc \mid \epsilon$
- $Q \rightarrow aQb \mid \epsilon$
- $C \rightarrow Cc \mid \epsilon$

The non-terminal A generates strings with any number of as while P generates strings with equal numbers of bs and cs. In contrast, Q generates strings with equal numbers of as and bs while C generates strings with any number of cs. By taking the conjunction (or intersection) of the concatenated languages associated with AP and QC, this grammar generates the language $\{a^n b^n c^n | n \geq 1\}$.

3.3.2 Properties

Boolean grammars possess some very useful properties normally associated with context-free grammars. A binary normal form is defined in a manner that is very similar to Chomsky normal form, except it allows for Boolean formulae on the RHS of every rule. The languages produced by Boolean grammars are closed under $\cup$, $\cdot$, $*$ **and** $\cap$ with themselves. Note that this is an improvement over context-free grammars, as CFGs are not closed under $\cap$ with any other language. Okhotin also includes a modified version of the CYK algorithm for membership that retains cubic-time complexity. Unfortunately, parse trees for Boolean grammars are somewhat difficult as negative conjuncts cannot really be represented, although positive conjuncts can be through &-labeled links. Perhaps surprisingly though, Boolean grammars generate the set of deterministic context-sensitive languages, *i.e.*, the set of languages recognized by Turing machines with a limited number of states and a limited length of input tape. This is a much larger set than those produced by indexed or programmed grammars yet seemingly without sacrificing many of context-free's valuable properties.

4 Extensions, Part Two

4.1 Combinatory Categorial Grammars

4.1.1 Definition

Combinatory categorial grammars (CCGs) [16] [17] were introduced by Mark Steedman in 1988. Such a grammar is defined as a five-tuple $G = (N, T, S, f, R)$ where N is an alphabet of non-terminals, T is an alphabet of terminals, S is the start symbol, f is the terminal function and R is a set of combinatory rules. Differences between CCGs and CFGs include the function f which maps terminals to categories (more on those briefly) and a group of combinatory rules replacing the standard set of productions. As with Boolean grammars, the theory behind CCGs is somewhat complex. Hence, the brief explanation here is neither complete nor formal. For the full technical definition, please refer to the cited sources.

Consider a "standard" grammar rule $S \to AB$; this could be expressed differently as an equation $S = A \cdot B$. Continuing with this new format, the non-terminal A can be expressed as the start symbol with the non-terminal B removed, or $A = S/B = (A/B) \cdot B$. The same can be said for non-terminal B with $B = S\backslash B = A \cdot (B\backslash A)$. The trick with this "equation method" is preserving the ordering associated with the original rule. Since A precedes B, a forward slash is used as an ordered divisor while for B a backward slash is used to show that B follows A.

Categories are defined over non-terminals ($c \in \mathit{cat}(N)$) and, at their lowest level, are simply non-terminals. However, these can be expanded using either the forward or backward divisors discussed in the previous paragraph, *e.g.*, A and $(S/B\backslash A)$ are both categories. Every category has a non-terminal as its *target*, defined recursively as $target(A) = A$, $target(c/c') = c$ or $target(c\backslash c') = c$. In addition, variables can be defined to stand in for all or specific categories. The variable $y \in \mathit{cat}(N)$ can represent any possible category while the *target-restricted* variable x^A requires the category to have A as its target.

The combinatory rules possess the familiar structure $LHS \to RHS$, but the LHS is composed of two categories (possibly represented by variables) c and c' while the RHS is a single resulting category c''. This may seem odd, as the rules actually simplify, not expand. It does makes sense from an equation perspective though, as equations are typically simplified as they are solved. Therefore, to actually create strings using a derivation, begin with the start symbol S and apply the rules backwards to expand out to a series of categories.

The last key in the derivation is the terminal function $f : T \to \mathit{cat}(N)$ which maps terminals to various categories. Again, this may seem odd as in a derivation it is desirable to generate terminals, not replace them. The solution lies in going backwards here as well; by applying the inverse of this function, terminals may be substituted for categories in the derivation, thus generating strings.

Example Let $G = (\{S, T, A, B, D\}, \{a, b, c, d\}, S, f, R)$ be a combinatory categorial grammar with the following combinatory rules:

- $(x^S/T)\,(T\backslash A/T\backslash B) \rightarrow (x^S\backslash A/T\backslash B)$
- $(A/D)\,(x^S\backslash A) \rightarrow (x^S/D)$
- $(x^S/y)\,y \rightarrow x^S$
- $y\,(x^S\backslash y) \rightarrow x^S$

The terminal function f contains the following definitions:

- $f(a) = \{(A/D)\}$
- $f(b) = \{B\}$
- $f(c) = \{(T\backslash A/T\backslash B)\}$
- $f(d) = \{D\}$
- $f(\epsilon) = \{(S/T), T\}$

This grammar generates the language $\{a^n b^n c^n d^n | n \geq 1\}$. The string abc is produced via the following derivation:

$$\begin{array}{lll}
S & \rightarrow & (S/D)\,D & (x^S/y)\,y \rightarrow x^S \\
 & \rightarrow & (A/D)\,(S\backslash D)\,D & (A/D)\,(x^S\backslash A) \rightarrow (x^S/D) \\
 & \rightarrow & (A/D)\,(S\backslash D/T)\,T\,D & y\,(x^S\backslash y) \rightarrow x^S \\
 & \rightarrow & (A/D)\,B\,(S/T)\,(S\backslash D/T\backslash B)\,T\,D & (x^S/T)\,(T\backslash A/T\backslash B) \rightarrow (x^S\backslash A/T\backslash B) \\
 & \rightarrow & (A/D)\,B\,(S\backslash D/T\backslash B)\,D & f(\epsilon) = \{(S/T), T\} \\
 & \rightarrow & a\,B\,(S\backslash D/T\backslash B)\,D & f(a) = \{(A/D)\} \\
 & \rightarrow & ab\,(S\backslash D/T\backslash B)\,D & f(b) = \{B\} \\
 & \rightarrow & abc\,D & f(c) = \{(T\backslash A/T\backslash B)\} \\
 & \rightarrow & abcd & f(d) = \{D\}
\end{array}$$

4.1.2 Properties

As is the case with Rosenkrantz and programmed grammars, Steedman does not detail in [16] which context-free properties combinatory categorial grammars provide. There are, however, others who have broached the subject: [21] describes a specific "simplified" form for proof purposes while [20] proves that the membership problem for CCGs is polynomial-time solvable, specifically n^6. With respect to the language set generated by CCGs, it is known to contain all context-free languages but remain a proper subset of context-sensitive languages. Unfortunately, no precise information could be located with respect to parse trees or closure, although personal correspondence [18] with Dr. Steedman did reveal his suspicion that the languages generated by CCGs are closed under union.

4.2 Linear Indexed Grammars

4.2.1 Definition

Linear indexed grammars (LIGs) [5] were introduced by Gerald Gazder in 1988, around the same time CCGs appeared. This grammar is defined as a five-tuple $G = (N, T, I, S, P)$ where N is an alphabet of non-terminals, T is an alphabet of terminals, I is a set of indices, S is the start symbol and P is a set of productions. If this definition seems identical to that of an indexed grammar, that suspicion is accurate. The difference is in the way productions are handled.

Flags still "stick" to non-terminals in LIGs, but here they do so via a stack. Every non-terminal has an associated index stack represented as $A[...f]$ or $A[]$ if the stack is empty. As a result, there are three basic forms for rules in a linear indexed grammar. The LHS is now a single non-terminal with its associated stack and the RHS may push or pop a flag from that stack and/or replace the non-terminal with another. In addition, the RHS may also introduce further non-terminals (with stacks) and terminals into the derivation, which always begins with $S[]$.

Example Let $G = (\{S, T\}, \{a, b, c, d\}, \{l\}, S, P)$ be a linear indexed grammar with the following productions:

- $S[\ldots] \to aS[\ldots l]d$
- $S[\ldots] \to T[\ldots]$
- $T[\ldots l] \to bT[\ldots]c$
- $T[] \to \epsilon$

This grammar generates the language $\{a^n b^n c^n d^n | n \geq 1\}$. The string *aabbccdd* is produced via the following derivation:

$$\begin{array}{rlll} S & \to & aS[l]d & S[\ldots] \to aS[\ldots l]d \\ & \to & aaS[ll]dd & S[\ldots] \to aS[\ldots l]d \\ & \to & aaT[ll]dd & S[\ldots] \to T[\ldots] \\ & \to & aabT[l]cdd & T[\ldots l] \to bT[\ldots]c \\ & \to & aabbT[]ccdd & T[\ldots l] \to bT[\ldots]c \\ & \to & aabbccdd & T[] \to \epsilon \end{array}$$

4.2.2 Properties

Happily, there is some information available about the properties of languages generated by linear indexed grammars. There exists at least one simplified form, again given in [21] for proof purposes, and the membership problem is solvable with a modified CYK-type algorithm. Gazder himself provides samples of parse trees, which appear nearly identical to context-free parse trees except that non-terminal nodes also feature their associated stack. Regrettably, no information was found on the topic of closure for the languages generated by LIGs.

4.3 Tree Adjoining Grammars

4.3.1 Definition

Tree adjoining grammars (TAGs) [6] were introduced by Aravind Joshi, Leon Levy and Masako Takahashi in 1975. This grammar is defined as a five-tuple $G = (N, T, C, A, S)$ where N is an alphabet of non-terminals, T is an alphabet of terminals, C is a set of labeled center trees, A is a set of labeled adjunct trees and S is the start symbol. This grammar perhaps bears the least resemblance to the definition of a Chomsky grammar as any other described in this report; instead of productions, there are two set of label trees which act as rules within a derivation.

The nodes of center and adjunct trees are composed of terminals and non-terminals. They also share one structural property: the internal nodes of both types are of the form $A^{\{l,m,\ldots\},o}$ where A is a non-terminal, $\{l, m, \ldots\}$ is a set of adjunct tree labels and o is a Boolean indicating whether adjunction (more on that later) is obligatory for this node or not. Each center tree has $S^{\{l,m,\ldots\},o}$ as its root and terminals as its leaves. In contrast, the root of an adjunct tree is $A^{\{l,m,\ldots\},o}$ and one leaf is also designated $A^{\{n,\ldots\},o}$. Note that although the root and single leaf share the same non-terminal, their tree labels may differ. All other leaves of an adjunct tree are terminals.

A TAG derivation begins by selecting a center tree as a start point. The derivation is then expanded by performing adjunctions with adjunct trees. This process is actually quite simple. First, a non-terminal node with a nonempty set of adjunct labels is selected and one of those tree labels is chosen, *e.g.*, A and l. The non-terminal node A is the then replaced with the root of adjunct tree l, which is also A if the grammar is well-formed. The children of A are then attached to the A-leaf of l and the replacement is then complete. Note that since a center tree is used as the initial tree and adjunctions are only performed on internal non-terminal nodes, the leaves of the current tree are always terminals. Also note that no derivation is complete unless all non-terminal nodes carry $false$ in their superscript; otherwise, further adjunctions are obligatory. To extract the string from any tree, simply concatenate the terminal leaves in a breadth-first manner.

Example Let $G = (\{S\}, \{a, b, c, d\}, \{\alpha\}, \{\beta\}, S)$ be a tree adjoining grammar with the tree structures shown in Figure 3.

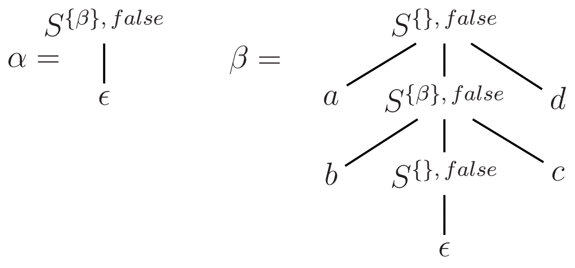

Figure 3: Example - TAG Tree Definitions

This grammar generates the language $\{a^n b^n c^n d^n | n \geq 1\}$. The string $aabbccdd$ is produced via the derivation given in Figure 4.

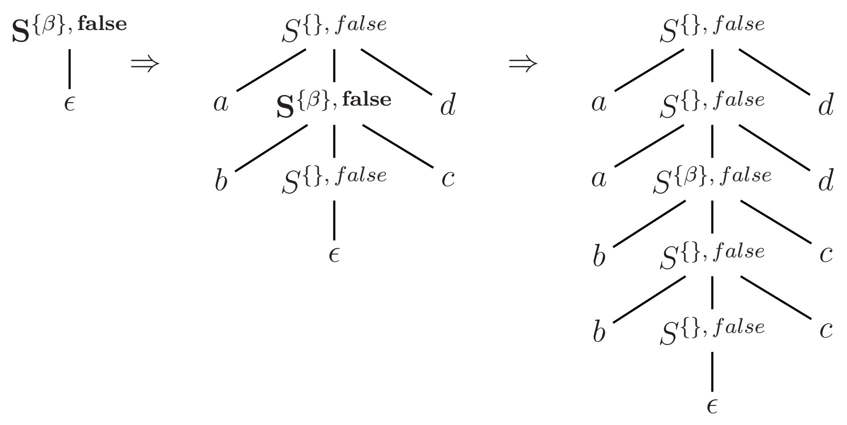

Figure 4: Example - TAG Tree Derivation

4.3.2 Properties

Fortunately, there is a wealth of information available about which context-free properties tree adjoining grammars possess. A normal form is given in the appendix of [21] and used in a proof in the body of the paper. The languages produced by TAGs are closed under $\cup, \cdot$ and $*$ with themselves and are closed under $\cap$ with regular languages [19]. The membership problem is solvable in polynomial time, again with an adaptation of the CYK algorithm [21]. And, as expected, the language set generated by tree adjoining grammars contains all context-free languages but is a proper subset of context-sensitive languages. No specific mention of parse trees was found, likely due to fact that derivations are already in tree form.

4.4 Head Grammars

4.4.1 Definition

Head grammars [10] [11] were introduced by Carl Pollard in his 1984 Ph.D. thesis. This grammar is defined as a four-tuple $G = (N, T, S, P)$ where N is an alphabet of non-terminals, T is an alphabet of terminals, S is the start symbol and P is a set of productions. This grammar may initially appear to be identical to a Chomsky grammar, but there are two core differences: the *head* $\uparrow$, which divides the current derivation string $w \in T^*$ into two parts $u \uparrow v$, and the structure of productions.

There are two types of operations that can be performed using the head. The first is *concatenation* C_n which joins n head-divided words in order and inserts a new head in the string. The position of this new head depends on the value of n. If it is even, the new head is simply inserted between the two "middle" strings, *e.g.*, $C_2(u_1 \uparrow v_1, u_2 \uparrow v_2) = u_1v_1 \uparrow u_2v_2$. Otherwise, the head is placed in its original position in the "center" word, *e.g.*, $C_3(u_1 \uparrow v_1, u_2 \uparrow v_2, u_3 \uparrow v_3) = u_1v_1u_2 \uparrow v_2u_3v_3$. The second operation is *wrapping* which inserts one word into another based on the head positions, *e.g.*, $W(u_1 \uparrow v_1, u_2 \uparrow v_2) = u_1u_2 \uparrow v_2v_1$.

The productions in a head grammar follow the standard context-free form $LHS \rightarrow RHS$ where the LHS is a single non-terminal, but the RHS is one of the concatenation or wrapping operations. The other difference is that each rule contains precisely two non-terminals: one is given as an operation parameter on the RHS and represents the current derivation string, while the other on the LHS is the "next" non-terminal to represent the derivation string.

Example Let $G = (\{S, T\}, \{a, b, c, d\}, S, P)$ be a head grammar with the following productions:

- $S \rightarrow C_1(\epsilon \uparrow \epsilon)$
- $S \rightarrow C_3(a \uparrow \epsilon, T, d \uparrow \epsilon)$
- $T \rightarrow W(S, b \uparrow c)$

This grammar generates the language $\{a^nb^nc^nd^n | n \geq 1\}$. The string *aabbccdd* is produced via the following derivation:

$$\begin{array}{lll}
S & \rightarrow \epsilon \uparrow \epsilon & S \rightarrow C_1(\epsilon \uparrow \epsilon) \\
 & \rightarrow \epsilon b \uparrow c\epsilon & T \rightarrow W(S, b \uparrow c) \\
 & \rightarrow a\epsilon b \uparrow cd\epsilon & S \rightarrow C_3(a \uparrow \epsilon, T, d \uparrow \epsilon) \\
 & \rightarrow abb \uparrow ccd & T \rightarrow W(S, b \uparrow c) \\
 & \rightarrow a\epsilon abb \uparrow ccdd\epsilon & S \rightarrow C_3(a \uparrow \epsilon, T, d \uparrow \epsilon) \\
 & \Rightarrow aabbccdd &
\end{array}$$

4.4.2 Properties

Fortunately, there is some information available detailing which desirable context-free properties head grammars possess. The membership problem is solvable, again with an adaptation of the CKY algorithm [21]. Head grammars also have the same closure properties as TAGs [22]: closed under $\cup, \cdot$ and $*$ with themselves and closed under $\cap$ with regular languages. Parse tree representation is possible for a head grammar derivation, but in a *packed* form that "avoids exponential explosion" [8]. Finally, as to be expected at this point, head grammars can generate all context-free languages and some, but not all, context-sensitive languages. No mention of simplified or normal forms was found, possibly because the rules are already in a somewhat reduced form with a single operation on the RHS and only two nonterminals in use at each step.

4.5 Surprise!

In 1994, K. Vijay-Shankar and David Wier proved that combinatory categorial grammars, linear indexed grammars, tree adjoining grammars and head grammars are all weakly equivalent, *i.e.*, they all produce the same class of languages. The proof itself is a series of language subset proofs: $CCL \subseteq LIL$, $LIL \subseteq HL$, $HL \subseteq TAL$ and $TAL \subseteq CCL$. What is really amazing, however, is that these four extensions with (seemingly) very different definitions were devised by six people over 15 years and yet these grammars **still** generate the same set of languages.

5 Alternatives

The extensions seemed so far in this report were all created with (at least) this goal in mind: producing a grammar that behaves like it's context-free but generates like it's (almost) context-sensitive. However, that's actually a narrow definition restricted to the Chomsky hierarchy. There are many other ways of "extending" a grammar or a language and this section of the report will briefly examine two of them.

5.1 Fuzzy Languages and Grammars

A *fuzzy language* [7] L is much like any language defined over an alphabet or a set of terminals, except that each word has an associated *membership grade* assigned by a *grade function* $\mu_L : L \rightarrow [0, 1]$. You could look at the membership grade as differentiating between precise and imprecise members, much like words or sentences in natural languages which may not be spoken correctly. For example, "I seen" is a grammatically incorrect form of "I've seen" that is frequently used and understood in the English language.

Example Let $L = \{(a, 1.0), (b, 0.8), (aa, 1.0), (ab, 0.4), (ba, 0.3), (bb, 0.7)\}$ be a fuzzy language, represented here as a set of $(w, \mu_L(w))$ pairs. In this case, strings composed of the same terminal are graded higher then those strings composed of a mix of terminals.

The standard set operations union, intersection, concatenation and Kleene star are defined similarly for fuzzy languages but with consideration for the grade function. For two languages L_1 and L_2, the grade function for $L_1 \cup L_2$ or $L_1 \cap L_2$ simply takes the maximum or the minimum values available from μ_1 and μ_2, respectively. Concatenation works a bit differently in that it takes the maximum of the possible grades for a string s in $L_1 \cdot L_2$; a possible grade for s is the minimum of its prefix grade in L_1 and postfix grade in L_2. As for Kleene star, it's defined as a combination of union and concatenation operations: $L^* = \epsilon \cup L \cup LL \cup \ldots$.

Example Let $L_1 = \{(a, 1.0), (b, 0.8), (ab, 0.4)\}$ and $L_2 = \{(a, 0.5), (b, 0.9), (ba, 0.3)\}$ be two fuzzy languages. The following are the results of performing the first three set operations described above on L_1 and L_2:

- $L_1 \cup L_2 = \{(a, 1.0), (b, 0.9), (ab, 0.4), (ba, 0.3)\}$
- $L_1 \cap L_2 = \{(a, 0.5), (b, 0.8)\}$
- $L_1 \cdot L_2 = \{(aa, 0.5), (ab, 0.9), (aba, 0.3), (ba, 0.5), (bb, 0.8), (bba, 0.3)$ $(abb, 0.4), (abba, 0.3)\}$

A *fuzzy grammar* [7] is defined as a four-tuple $G = (N, T, S, P)$ with all the usual members. What makes G fuzzy is its grade function μ which is now defined using P; formally, $\mu : P \rightarrow [0, 1]$. Interestingly, fuzzy grammars mirror the hierarchy for Chomsky grammars. Fuzzy context-free grammars can also be put into a normal form using an algorithm much like that for CNF but modified to handle and adjust the grade function for the new rules.

Performing a derivation with a fuzzy grammar is identical to the process for a Chomsky grammar: begin with the start symbol and apply the rules until a string composed entirely of terminals is reached. The only trick is calculating the membership grade for the resulting string. Like concatenation, this involves taking "the maximum of the minima". The grade of a derivation is the minimum grade of all the rules applied in that derivation. The membership grade of any string in L_G is thus the maximum grade of any of its derivations.

Example Consider the fuzzy grammar $G = (\{S, A, B\}, \{a, b\}, P, S)$ where P contains the following productions and membership grades:

- $\mu(S \rightarrow A) = 0.8$
- $\mu(S \rightarrow B) = 0.9$
- $\mu(A \rightarrow a) = 0.5$
- $\mu(B \rightarrow a) = 0.2$

The string a is generated by two separate derivations: $S \rightarrow A \rightarrow a$ with a grade value of 0.5 and $S \rightarrow B \rightarrow a$ with a grade value of 0.2. Therefore, $\mu_{L(G)}(a) = 0.5$.

5.2 Imperfect Strings

Take any context-free grammar $G = (N, T, S, P)$ and consider any string $s \in L(G)$. This string is precisely generated by G; if a membership algorithm were executed with s and G as parameters, the result would be positive. But what about strings that are close to s but not exactly s? In natural languages, words that are spelled incorrectly or sentences that are awkwardly constructed are often recognized and understood. Is it possible to adjust a context-free grammar to do the same thing?

The concept of *imperfect strings* [14] is designed to tackle this exact problem. Given any terminal α, there are three types of operations which create new rules for G so that the grammar can recognize words that are flawed with respect to α. These new productions are all based on G's existing rules. The operators are informally defined as follows:

- **Change** - $CH(\alpha)$ substitutes another terminal for α in existing productions
- **Delete** - $DE(\alpha)$ substitutes ϵ for α in existing productions
- **Insert** - $IN(\alpha)$ places another terminal before or after α in existing productions

Example Consider the context-free grammar $G = (\{S\}, \{a, b, c\}, \{S \rightarrow abc\}, S)$. If each of the imperfect string operations were applied to G with a as input, the following new rules would result:

- $CH(a) = \{S \rightarrow bbc, S \rightarrow cbc\}$
- $DE(a) = \{S \rightarrow bc\}$
- $IN(a) = \{S \rightarrow babc, S \rightarrow abbc, S \rightarrow cabc, S \rightarrow acbc\}$

Note the fact that these operators are designed to handle single imperfections; that is, a single change, deletion, or insertion of/around a particular terminal. To handle multiple imperfections, it would be necessary to apply each operator several additional times. The concept of imperfect strings has also been extended to embrace fuzzy grammars with equations for defining the membership grade for each new rule.

6 Conclusion

To a linguistic novice, extending context-free grammars might seem like catering to a niche market of particular and eccentric researchers who are consumed with theory and not with practical application. On the contrary, many of the authors who introduced the extensions described in this report also include an analysis of how their particular grammars related to a natural language such as French or German.

There is certainly no lack of options if one were looking for an extension of context-free grammars and it is extremely likely one could find a specific extension that precisely suits their tastes and their needs. There is a wide variety available; one grammar may appear completely different than another, but they all fulfill many or all of the desirable properties normally associated with context-free grammars. Of interest is the fact that some of the more "complicated" grammars do not possess any more generative power than their "simpler" counterparts, *e.g.*, combinatory categorial grammars produce the exact same set of languages as linear indexed grammars. Also of note is that the definition of the extension truly depends on one's point of view. Although many (and arguably most) would immediately think of the Chomsky hierarchy and context-sensitive grammars, some would take and have taken a different approach. Their augmentations may be different but they are no less valid with respect to recognizing natural languages. It will be interesting to see what further extensions arise, as there is no doubt research on this particular topic will continue to thrive.

References

[1] Alfred V. Aho. Indexed grammars - an extension of context-free grammars. *Journal of the Association for Computing Machinery*, 15(4):647–671, Oct. 1968.

[2] Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, 1956.

[3] Noam Chomsky. On certain formal properties of grammars. *Information and Control*, 2:137–167, 1959.

[4] Mariusz Flasiński and Janusz Jurek. Dynamically programmed automata for quasi context-sensitive languages as a tool for inference support in pattern recognition-based real-time control expert systems. *Pattern Recognition*, 32:671–690, 1999.

[5] Gerald Gazdar. Applicability of indexed grammars to natural languages. In U. Reyle and C. Rohrer, editors, *Natural Language Parsing and Linguistic Theories*, pages 69–94. D. Reidel Publishing Company, Englewood Cliffs, NJ, 1988.

[6] Aravind K. Joshi, Leon S. Levy, and Masako Takahashi. Tree adjunct grammars. *Journal of Computer and System Sciences*, 10:136–163, 1975.

[7] E. T. Lee and L. A. Zadeh. Notes on fuzzy languages. *Information Sciences*, 1:421–434, 1969.

[8] Yusuke Miyao and Jun'ichi Tsujii. Probabilistic disambiguation models for wide-coverage HPSG parsing. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics*, pages 83–90, Ann Arbor, Michigan, June 2005. Association for Computational Linguistics.

[9] Alexander Okhotin. Boolean grammars. *Information and Computation*, 194:19–48, 2004.

[10] Carl Pollard. *Generalized Phrase Structure Grammars, Head Grammars and Natural Language*. PhD thesis, Stanford University, 1984.

[11] Carl Pollard and Ivan A. Sag. *Information-Based Syntax and Semantics*, volume 1. Center For the Study of Language and Information, Stanford, CA, 1987.

[12] Queen's University CISC 366 course notes: Languages. Available at http://www.cs.queensu.ca/home/cisc366/ as of Apr. 23, 2006.

[13] Daniel J. Rosenkrantz. Programmed grammars and classes of formal languages. *Journal of the Association for Computing Machinery*, 16(1):107–131, Jan. 1969.

[14] M. Schneider, H. Lim, and W. Shoaff. The utilization of fuzzy sets in the recognition of imperfect strings. *Fuzzy Sets and Systems*, 49:331–337, 1992.

[15] Michael Sipser. *Introduction to the Theory of Computation.* PWS Publishing Company, Boston, MA, 1997.

[16] Mark Steedman. Combinators and grammars. In Richard T. Oehrle et al., editor, *Categorial Grammars and Natural Language Structures*, pages 417–439. D. Reidel Publishing Company, Englewood Cliffs, NJ, 1988.

[17] Mark Steedman. Categorial grammar, 1998. Draft of entry in *The MIT Encyclopedia of Cognitive Sciences*; available at http://groups.inf.ed.ac.uk/ccg/publications.html as of Mar. 24, 2006.

[18] Mark Steedman. Email correspondence, March 30 - Apr. 2, 2006.

[19] K. Vijay-Shanker and A. K. Joshi. Some computational properties of tree adjoining grammars. In *Proceedings of the 23rd Meeting of the Association for Computational Linguistics*, pages 82–93, 1985.

[20] K. Vijay-Shanker and David Weir. Polynomial time parsing of combinatory categorial grammars. In *Proceedings of the 28th Annual Meeting of the Association for Computational Linguistics, Pittsburgh*, pages 1–8, San Francisco, CA, 1990. Morgan Kaufmann.

[21] K. Vijay-Shanker and David Weir. The equivalence of four extensions of context-free grammars. *Mathematical Systems Theory*, 27:511–546, 1994.

[22] K. Vijay-Shanker, David J. Weir, and Aravind K. Joshi. Tree adjoining and head wrapping. In *In Proceedings of the 11th International Conference on Computational Linguistics*, pages 202–207, 1986.

[23] Wikipedia entry: CYK algorithm. Available at http://en.wikipedia.org/wiki/CYK_algorithm as of Apr. 23, 2006.