Combinatory Categorial Grammar

Fabienne Sophie Eigner

Saarland University
Faculty of Natural Sciences and Technology I
Computer Science Department
fabieigner@aol.com

Advisor: Ralph Debusmann

Abstract. In this paper we will introduce Combinatory Categorial Grammars and explain the polynomial time parsing algorithm for such grammars as defined by K. Vijay-Shanker and David Weir. We will describe the recognition algorithm via an inference system that resembles the CKY algorithm, and then briefly discuss an efficient representation for the parse forest.

1 Introduction

"Peter is from England and Paul from Sweden" and "dat ik Cecilia de nijlpaarden zag voeren"(in English "that I saw Cecilia feed the hippos") are two examples (one English, one Dutch) for linguistic phenomena that cannot be captured by context free grammars (CFG). The first sentence reflects the problem of coordination, which deals with the different uses of conjunctions like "and". It is quite easy (and possible in context free grammars) to use a conjunction to combine two expressions of the same type to an expression of the same type again, e.g. "Peter is from England and Paul is from Sweden" where the "and" connects two complete sentences yielding a new complete sentence. But in the afore mentioned example the part "Paul from Sweden" is not complete by itself but rather needs the conjunction to adjoin it to another sentence which will contain the missing verb. This is not accomplishable by context free grammars. The second example shows the possibility of crossed dependencies in the Dutch language. These dependencies also occur in the so called "Copy-Language" that we will introduce later.

Languages containing one or both of these phenomena (among others) can be described using "mildly context-sensitive grammars" that were recognized in 1985 by Joshi. Examples for these grammatical formalisms are the "Tree Adjoining Grammars" (TAG), the "Linear Indexed Grammars" (LIG) and the "Combinatory Categorial Grammars" (CCG), that we will discuss in this paper. All three formalisms are eqivalent and can be parsed in polynomial time $O(n^6)$ as K. Vijay-Shanker and David Weir have shown [3, 4, 9] . Combinatory Categorial Grammars [7, 6] were introduced in the 1980's by Mark Steedman as an extension of the context free "Categorial Grammars" (that have a quite transparent semantic interpretation). CCG is the only one of the three above mentioned formalisms that deals rather intuitively with coordination.

In the following we will briefly describe the CCG formalism and explain it by examplarily defining a CCG for the Copy-Language. Then we will simplify a Cocke-Kasami-Younger (CKY)-like polynomial time recognition algorithm for CCG, that was developed by K. Vijay-Shanker and David Weir in 1990 [8], by reducing it to four inference rules. We will then show how this algorithm together with the idea of sharing can be used for efficient parsing.

2 Definition of CCG

A CCG $G = (V_T, V_N, f, S, R)$ is defined as follows:

- V_T defines the finite set of all terminals.
- V_N defines the finite set of all nonterminals. These nonterminals are also called "atomic categories" which can be combined into more complex functional categories by using the backward operator $\backslash$ or the forward operator $/$.
- The function f maps terminals to sets of categories and corresponds to the first step in bottom-up parsing.
- The unique starting symbol is denoted by $S \in V_N$.
- R describes a finite set of combinatory rules that for $n \geq 1$, $|_i \in \{\backslash, /\}$ are defined to be of the following form (where x, y and z_i are meta-categories, that can be replaced by any categories of the grammar):
  - Functional applications:
    - Forward application: $x/y \quad y \;\rightarrow\; x \quad (>)$
    - Backward application: $y \quad x\backslash y \;\rightarrow\; x \quad (<)$
  - Composition rules:
    - Forward composition:
      $x/y \quad y \,|_1\, z_1 \,|_2 \ldots |_n\, z_n \;\rightarrow\; x \,|_1\, z_1 \,|_2 \ldots |_n\, z_n \quad (B_>^{(*)})$
    - Backward composition:
      $y \,|_1\, z_1 \,|_2 \ldots |_n\, z_n \quad x\backslash y \;\rightarrow\; x \,|_1\, z_1 \,|_2 \ldots |_n\, z_n \quad (B_<^{(*)})$

  Here (r) is used to denote the application of a rule r in a derivation, and the * in $(B^{(*)})$ reflects crossing dependencies (categories contain both the backward operator $\backslash$ and the forward operator $/$) where occuring.

  Restricting the rules to applications yields a context free grammar formalism, therefore the additional power of CCG must lie in the composition rules. If we allow n to be 0 as well, we can see that in priniciple application is just a special case of composition. For future reference, we will call $x \mid y$ the "primary category" and the other category "secondary" (or argument). With "target" we denote the leftmost nonterminal of a category (e.g. x in $x \mid y$).

  Without loss of generality we will usually only consider paranthesis free categories, which are assumed to be implicitely left associative.

To define the language generated by G we define the derivation operator $\Longrightarrow$ as follows: Let τ_1 and τ_2 be strings of categories and terminals, c, c_1, c_2 be categories, and $a \in V_T$.
Then

- $\tau_1 c \tau_2 \Longrightarrow \tau_1 c_1 c_2 \tau_2$, if $c_1 c_2 \to c$ is an instance of a rule in R, and
- $\tau_1 c \tau_2 \Longrightarrow \tau_1 a \tau_2$, if $c \in f(a)$.

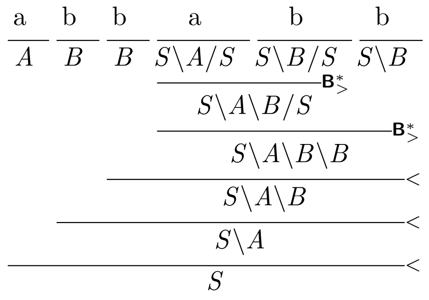
The generated language can now be described as $L(G) = \{w \in {V_T}^* \mid S \Longrightarrow^* w\}$.

2.1 The Copy-Language, an Example for CCG's

The Copy-Language is an example for crossing dependencies that cannot be described by context free grammars. It is defined as $L_{copy} = \{ww \mid w \in \{a, b\}^+\}$. We now define a CCG $G = (V_T, V_N, f, S, R)$ with $L(G) = L_{copy}$ as follows:

- $V_T = \{a, b\}$
- $V_N = \{S, A, B\}$
- function f:
  - $f(a) = \{A, S\backslash A, S\backslash A/S\}$
  - $f(b) = \{B, S\backslash B, S\backslash B/S\}$
- rules R:
  - $y \quad (x\backslash y) \to \quad x$
  - $(x/y) \quad (y\backslash z) \to (x\backslash z)$

We will now examplarily show how a word *abbabb* $\in L_{copy}$ can be recognized bottom-up: First of all we use the function f to identify the terminals by matching categories. In this example language we will always assign the respective atomic categories to the left half of the terminal word, the categories $S\backslash A$ respectively $S\backslash B$ to the rightmost terminal and $S\backslash A/S$ or $S\backslash B/S$ to all other terminals in between.



a b b a b b
A B B $S\backslash A/S$ $S\backslash B/S$ $S\backslash B$
$\mathbf{B}^*_>$
$S\backslash A\backslash B/S$
$\mathbf{B}^*_>$
$S\backslash A\backslash B\backslash B$
$<$
$S\backslash A\backslash B$
$<$
$S\backslash A$
$<$
S

Looking at the above figure from bottom to top, we have found a derivation of *abbabb* from the starting symbol S. One can already see that contrary to derivations of CFG not only the nonterminals (atomic categories) of a grammar are used in the derivation but also the more complex functional categories. These categories can be polynomially large in the length of the input string. For example in the derivation depicted above, the categories are used in a stack-like fashion, where for each read terminal of the right half of the word a corresponding nonterminal is added (pushed) to the "stack". In our example only three terminals had to be read, but, e.g. $ab^{100}ab^{100} \in L_{copy}$ would lead to a stack that temporarily contains 102 atomic categories. Once the complete half is stored as a stack, each nonterminal on the stack can be popped of it if a matching terminal is found in the left half of the word. Since eliminating a nonterminal from the stack by using functional application is only possible for the rightmost nonterminal of the primary category (meaning the last symbol pushed upon the stack) the categories do indeed fulfill the stack-property "last-in-first-out".

3 Parsing CCG's

We will now have a look at Vijay-Shanker and Weir's polynomial time parsing algorithm as introduced in [8]. In fact, the algorithm consists of three different steps:

– The recognition algorithm determines whether a string $a_1 \dots a_n$, $n \in \mathbb{N}$ is contained in a language L defined by a CCG G.
– If the recognition process was succesful the actual parsing algorithm recovers all possible parse trees of the string using the results of the recognition algorithm.
– Since there might be exponentially many different parse trees we will need an efficient representation of the parse-forest, which uses the idea of "sharing" and resembles this of [5].

3.1 Recognition Algorithm

The recognition algorithm is bottom-up, resembling the CKY-algorithm. But as described above, the length of the inner nodes of the parse tree of a string $a_1 \dots a_n$ can grow polynomially in n. Hence the recognition algorithm might have exponential running time. Until Vijay-Shanker and Weir solved this by introducing their algorithm in 1990 [8], no other solution for this problem existed and people used the basic (potentially exponential) CKY-algorithm.

The idea of an efficient polynomial time algorithm is as follows: Consider the categories

– $A/X_1/.../X_l/B \quad B/C$, with $l \in \mathbb{N}$
– $A/Y_1/.../Y_m/B \quad B/C$, with $m \in \mathbb{N}$

In both cases the application of a rule (functional application or composition) only depends on the "target" category A, the "top of stack" B and the secondary category B/C. As one can easily see, both examples above are rather similar and only differ in the X's and Y's. So we will use the idea of sharing: Instead of storing the complete categories used in bottom-up recognition, we will fix a constant upper bound b which we will later define in detail. Whenever the size of a category (stack) that we need to store is smaller than b we will simply store the entire category. For all categories exceeding b we will only store the target category, the top b_2 (definition follows) symbols of the stack and set a "link" to the remaining stack symbols that are called the tail.

The recognition algorithm uses a four-dimensional Array $L[i, j][p, q]$ to store the entries produced by an input $a_1 \dots a_n$. The indices i and j resemble those in the CKY-algorithm as defined in [5]: if an entry x is contained in $L[i, j][p, q]$ it means that from x the string $a_1 \dots a_j$ is derivable. The indices p and q are "link-indices". If the size of a category that should be entered into the array crosses the bound b, a link to the tail will be coded into these indices: the tail can then be found at an entry $L[p, q][r, s]$. Hence two types of entries are possible:

- $((A, \alpha), -) \in L[i, j][0, 0]$, where A describes the target category of the entry, α describes the whole top of stack, which is smaller than the fixed upper bound b, and no tail exists, thus the empty tail is described by "$-$" and no link-indices are needed (so p and q are both 0).
- $((A, \alpha), T) \in L[i, j][p, q]$, where A again describes the target category of the entry, α is the top of stack (maximal size b_2) and the rest of the stack (tail) is described by a symbol T and the link indices p and q that are both different from 0.

We will now give an exact definition of the fixed upper bound b of the size of categories, that are allowed to be stored as the "top of stack" and two other necessary bounds b_1, b_2, and b_3. For a given CCG $G = (V_T, V_N, f, S, R)$ we define

$$\begin{aligned}
b_1 &:= max\ \{n \in \mathbb{N} \mid\ c \in f(a),\ a \in V_T,\ c = A \mid_1 z_1 \mid_2 \dots \mid_n z_n\},\\
b_2 &:= max\ \{n \in \mathbb{N} \mid\ (x/y \quad y \mid_1 z_1 \mid_2 \dots \mid_n z_n \to x \mid_1 z_1 \mid_2 \dots \mid_n z_n) \in R \vee \\
&\qquad\qquad (y \mid_1 z_1 \mid_2 \dots \mid_n z_n \quad x\backslash y \to x \mid_1 z_1 \mid_2 \dots \mid_n z_n) \in R\},\\
b_3 &:= max\ \{b_1, b_2\},\\
b &:= b_2 + b_3.
\end{aligned}$$

It is clear, that the first step of the recognition algorithm, namely the lexical assignment, should not already result in sharing, hence we use b_1 to describe the maximal category size that can result from lexical assignment to define our upper bound. Since the recognition algorithm will always assume the secondary category of an applicable rule to be stored in its entirety within an entry of the array, the upper bound also depends on b_2 which describes the maximal length of the secondary category of a rule. We then introduced b_3 to make the definition of b more concise.

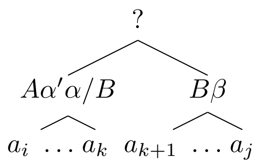
The Inference System We will now define the inference rules of the algorithm. Without loss of generality we will only consider forward application and composition, by exchanging the indices i and k with $k + 1$ and j the corresponding backward operations can be captured as well. To abbreviate, we define $|x|$ to be m such that $x \in (\{\backslash, /\}V_N)^m$.

– **Initialization**

$$\frac{c \in f(a_i) \qquad c = A\alpha}{((A,\alpha),-) \in L[i,i][0,0]}$$

– **Operations**

Apart from initialization, the inference rules model the following situation:



Given a primary and secondary category that are both already stored in the array L and coincide in the category B, we want to insert the category that results from composition or application into L.

- **Rule 1**

$$\frac{((A,\alpha/B),T) \in L[i,k][p,q] \qquad ((B,\beta),-) \in L[k+1,j][0,0]}{((A,\alpha\beta),T) \in L[i,j][p,q]}$$

if:
  * $T = -$ and $|\alpha\beta| < b$
  * $T \neq -$ and $|\beta| > 1$ and $\alpha = \epsilon$
  * $T \neq -$ and $|\beta| = 1$
  * $T \neq -$ and $|\beta| = 0$ and $\alpha \neq \epsilon$

This case occurs whenever the size of the result category derivates not much from that of the primary category, i.e., application does not need to acces the tail and composition does not surpass the bound of allowed top symbols. If that is the case, the (encoded) tail of the primary category (whether existent or not) is not affected by the operation and the resulting category differs from the primary only in its top elements.

- **Rule 2**

$$\frac{((A,\alpha/B),T) \in L[i,k][p,q] \qquad ((B,\beta),-) \in L[k+1,j][0,0]}{((A,\beta),/B) \in L[i,j][i,k]}$$

if:
  * $T = -$ and $|\alpha\beta| \geq b$
  * $T \neq -$ and $|\beta| > 1$ and $\alpha \neq \epsilon$

If composition increases the size of the primary category so much that the top of stack crosses the allowed upper bound b, we need to divide the resulting category into top and tail and set a link to the latter. This works as follows: we take the top of the secondary category as the new top of stack. Since the new category results from eliminating B from the primary category, we will memorize this fact by writing "$/B$" as the second element of the new entry and furthermore setting the link indices to i and k. This means that the new tail can be accessed by following the link indices and removing the category B from the so found stack.

- **Rule 3**

$$\frac{\begin{array}{c}((A,/B),\gamma) \in L[i,k][p,q] \\ ((B,\epsilon),-) \in L[k+1,j][0,0] \\ ((A,\beta'\gamma),T') \in L[p,q][r,s]\end{array}}{((A,\beta'),T') \in L[i,j][r,s]}$$

if:

$*$ $\gamma = T \neq -$

The last rule applies in case that the functional application reduces the primary stack so much, that the tail needs to be accessed in order to determine the result category. This tail was encoded by γ, p, and q, which means that the category it links to can be found at position $[p,q]$ and has γ as the top symbol (which, as shown in the second rule, does not belong to the tail). The resulting stack corresponds to this tail.

– A word $a_1 \dots a_n$ has been recognized if $((S,\epsilon),-) \in L[1,n][0,0]$.

Complexity The runtime of the algorithm can be determined by looking at the number of independent variables of the inference rules. Rule 3 contains seven different variables i, j, k, p, q, r, and s, ranging from 1 to n. Hence the runtime is at most $O(n^7)$.

In fact, Vijay-Shanker and Weir have developed a similar algorithm in [9], that makes it possible to parse CCG's in $O(n^6)$. This algorithm is defined for LIG and then modified to parse CCG and TAG in the same time.

3.2 Recovering the Parse Trees

Once we made sure that a string $a_1 \dots a_n$ is indeed contained in a language L_G generated by a CCG $G = (V_T, V_N, f, S, R)$, which is assured by the succesful completion of the recognition algorithm, we need to enumerate all possible parse trees. Since there might be exponentially many of those, Vijay-Shanker and Weir again use the idea of sharing for an efficient representation. To explain their idea, we first need to shortly introduce Linear Indexed Grammars LIG. LIG's were developed in 1988 by Gazdar [2] based on Indexed grammars that were introduced in 1968 by Aho [1]. They are equivalent to TAG and CCG formalisms as shown by Vijay-Shanker, Weir and Joshi in [3, 4] but contrary to the latter they are not really used in practice but are mainly interesting from a theoretical point of view: e.g., to prove properties of weakly context sensitive grammars or develop algorithms for them.

LIG's are basically CFG's extended by indices associated with the nonterminals. These indices are used to describe a stack. Hence the application of a production not only depends on the nonterminal but also on the top symbols of the stack associated with it. We will only very shortly define the grammars, for further reference please have a look at the mentioned literature.

Let A be a nonterminal, α a sequence of indices (stack symbols) and γ an index. Then the notation $A[\alpha\gamma]$ is used to describe the fact that a stack consisting of symbols α with symbol γ on top is associated with the nonterminal A. There are two different kinds of allowed productions:

– $A[\alpha] \rightarrow a$
– $A[\cdot\cdot\,\alpha] \rightarrow A_1[\alpha_1] \ldots A_{i-1}[\alpha_{i-1}]\, A_i[\cdot\cdot\,\beta]\, A_{i+1}[\alpha_{i+1}] \ldots A_n[\alpha_n]$

The first production covers the basic cases where the nonterminal A with a stack containing only the sequence α can be replaced by a terminal symbol a. The second type of production rule means that a nonterminal associated with a stack that has the sequence α on top can be rewritten as follows: one child (namely i) inherits the stack, where α is replaced by β. Each other child inherits a single element of the sequence α according to their position.

A string is contained in the language generated by a LIG if it is derivable from the start symbol associated with the empty stack.

We will now explain how LIG's can be used to represent a shared parse forest for a given input string $a_1 \ldots a_n$. The idea is to develop a LIG G_{sf} while parsing, such that the language derivable from G_{sf} consists of an encoded description of all derivations of $a_1 \ldots a_n$ in G.

This works as follows: We rely on the recognizing algorithm having succesfully accepted $a_1 \ldots a_n$ and completed the array L. To find the derivations that are used in the corresponding parse trees, we will determine why an entry $((A, \alpha), T)$ has been inserted into the array L by searching L for two categories that can be combined to $((A, \alpha), T)$. Therefore the indices that we will use in G_{sf} are of the form (A, α, i, j). The terminals of G_{sf} are names for the combinatory rules and lexical assingments that are used in the derivation of $a_1 \ldots a_n$. These names might look as follows: e.g. the lexical assignment of c to the terminal a is denoted by $\langle c, a\rangle$ and F_m is used to describe the use of a forward composition $x/y \quad y \;|_1\; z_1 \;|_2\; \ldots \;|_n\; z_n \rightarrow x \;|_1\; z_1 \;|_2\; \ldots \;|_n\; z_n$. Only one nonterminal P is needed. The set R of production rules is determined by the algorithm that we will now define. Basically the algorithm constructs all needed productions by inverting the inference rules of the recognition algorithm, using the indices of the LIG G_{sf} to describe the stack-like categories used in the recognition process.

For this algorithm,we will need the notion of "marking" entries in the array L: marking an entry $((A, \alpha), T) \in L[i, j][p, q]$ symbolizes that the category that is represented by this entry will occur in some derivation tree with root S and leaves $a_1 \ldots a_n$, and hence we need to "remember" the entry to process it later on. The algorithm now works as follows:

– First we will mark the entry $((S, \epsilon), -) \in L[1, n][0, 0]$. This entry clearly exists, since its existence is the condition for accepting the string $a_1 \ldots a_n$.
– Now for all $1 \leq i \leq n$, $i \leq j \leq n$, and all marked entries in $L[i, j][p, q]$ do the following until the system is stable: Look for two categories in L that when combined result in this marked entry. Once they are found, construct the respective production rule as follows:

- **Production I** (inverse of initialization rule)

$$\frac{\begin{array}{c}((A,\alpha),T) \in L[i,i][0,0] \text{ "marked"} \\ c \in f(a_i) \qquad c = A\alpha\end{array}}{P[(A,\alpha,i,i)] \to \langle A\alpha, a_i\rangle}$$

- **Production 1** (inverse of rule 1)

$$\frac{\begin{array}{c}((A,\alpha\beta),T) \in L[i,j][p,q] \text{ "marked"} \\ ((A,\alpha/B),T) \in L[i,k][p,q] \\ ((B,\beta),-) \in L[k+1,j][0,0]\end{array}}{P[\cdot\cdot(A,\alpha\beta,i,j)] \to F_m\ P[\cdot\cdot(A,\alpha/B,i,k)]\, P[(B,\beta,k+1,j)] \in R}$$

for $m = |\beta|$, where α can be empty.

- **Production 2** (inverse of rule 2)

$$\frac{\begin{array}{c}((A,\beta),T) \in L[i,j][i,k] \text{ "marked"} \\ ((A,\alpha/B),T') \in L[i,k][p,q] \\ ((B,\beta),-) \in L[k+1,j][0,0]\end{array}}{P[\cdot\cdot(A,\alpha/B,i,k)(A,\beta,i,j)] \to F_m\ P[\cdot\cdot(A,\alpha/B,i,k)]\, P[(B,\beta,k+1,j)] \in R}$$

for $m = |\beta|$.

- **Production 3** (inverse of rule 3)

$$\frac{\begin{array}{c}((A,\alpha),T) \in L[i,j][p,q] \text{ "marked"} \\ ((A,/B),\gamma) \in L[i,k][r,s] \\ ((A,\alpha\gamma),T) \in L[r,s][p,q] \\ ((B,\epsilon),-) \in L[k+1,j][0,0]\end{array}}{P[\cdot\cdot(A,\alpha,i,j)] \to F_0\ P[\cdot\cdot(A,\alpha\gamma,r,s)(A,/B,i,k)]\, P[(B,\epsilon,k+1,j)] \in R}$$

We add the newly generated rules to R of G_{sf}. Furthermore we mark all unmarked entries that occured in the premises of the applied generation rule.

Complexity It is easy to see, that the number of terminals and nonterminals in G_{sf} is constant. Furthermore the number of indices and productions are $O(n^5)$, since the maximal number of independent variables (between 1 and n) in a production rule is five (see Production 3: i, j, r, s, and k). Hence the grammar G_{sf} itself is only polynomially large, even though it might describe exponentially many parse tree.

Analogously to the recognition algorithm, the complexity of the algorithm that builds the shared forest is dominated by the execution of Production 3 (corresponds to Rule 3 of the recognition algorithm). Here the whole array has to be searched for entries depending on seven independent variables between 1 and n. Hence the runtime lies in $O(n^7)$.

4 Conclusion

There exist several linguistic phenomena that cannot be captured by context free grammars, e.g. coordination and crossed dependencies. Languages that include these phenomena can be described by mildly context-sensitive grammar formalisms. In this paper we introduced one mildly context-sensitive formalism, the so called Combinatory Categorial Grammars, that were developed by Mark Steedman and are especially helpful to model coordination. We have pointed out the problems of efficiently recognizing and parsing these grammars and explained Vijay-Shanker's and Weir's approach that modifies a simple bottom-up parser by including the idea of sharing. Afterwards we have described how to efficiently encode the parse forest of a string. This algorithm is based on that of Vijay-Shanker and Weir as well and again uses sharing, by describing the parse forest via a linear indexed grammar.

References

1. Alfred V. Aho. Indexed grammars - an extension of context-free grammars. *Journal of the Association for Computer Machinery*, 15(4):647–671, October 1968.
2. 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, 1988.
3. Aravind Joshi, K. Vijay-Shanker, and David Weir. The convergence of mildly context-sensitive grammar formalisms. *Foundational Issues in Natural Language Processing*, pages 31–81, 1991.
4. Aravind Joshi and David Weir. Combinatory categorial grammars: Generative power and relationship to linear context-free rewriting systems. *26th meeting Assoc. Comp. Ling.*, 1988.
5. Mark-Jan Nederhof and Giorgio Satta. Introduction to parsing algorithms for NLP. Lecture Notes, ESSLLI 2004.
6. Mark Steedman. *The Syntactic Process*. The MIT Press, 2000.
7. Mark Steedman and Jason Baldridge. Combinatory categorial grammar. Unpublished Tutorial, 2003. http://groups.inf.ed.ac.uk/ccg/publications.html.
8. K. Vijay-Shanker and David J. Weir. Polynomial time parsing of combinatory categorial grammars, 1990. http://acl.ldc.upenn.edu/P/P90/P90-1001.pdf.
9. K. Vijay-Shanker and David J. Weir. Parsing some constrained grammar formalisms, 1993. http://acl.ldc.upenn.edu/J/J93/J93-4002.pdf.