Left Corner Transforms and Finite State Approximations

Mark Johnson
Rank Xerox Research Centre
Grenoble

DRAFT of 12th May, 1996

1 Introduction

This paper describes methods for approximating context-free grammars with finite state machines. Unlike the method derived from the LR(k) parsing algorithm described in Pereira and Wright (1991), these methods use grammar transformations based on the left-corner grammar transform (Rosenkrantz and Lewis II, 1970; Aho and Ullman, 1972). One advantage of the left corner methods is that they generalize straightforwardly to complex feature "unification based" grammars, unlike the LR(k) based approach.

Left-corner based techniques are natural for this kind of application because (with a simple optimization) they can parse pure left-branching or pure right-branching structures with a stack depth of one (two if terminals are pushed and popped from the stack). Higher stack depth occurs with center-embedded structures, which humans find difficult to comprehend. This suggests that we may get a finite-state approximation to human performance by simply imposing a stack depth bound, and ignoring any parses which require stack depths greater than this bound. We provide a simple tree-geometric description of the configurations that cause an increase in a left corner parser's stack depth below.

We also take this opportunity to point out some simple extensions of this technique, which can capture using pure grammar transform techniques a range of parsing strategies similiar to the *generalized left-corner parsing strategies* (Demers, 1977; Nijholt, 1980).

Finally, this paper discusses methods for using these finite state approximations in actual parsing applications, showing how the finite state machine can be used as an oracle to guide a left corner parser (and hence recover the tree structure) and how to construct a transducer that produces partial brackettings.

1.1 Parsing strategies as grammar transformations

The parsing algorithms discussed here are presented as *grammar transformations*, i.e., functions T that map a context-free grammar G into another context-free grammar $T(G)$. The transforms have the property that a top-down parse using the transformed grammar is isomorphic to some other kind of parse using the original grammar. Thus grammar transforms provide a simple, compact way of describing various parsing algorithms, as a top-down parser using $T(G)$ behaves identically to the kind of parser we want to study using G.

One way to understand this is to recognize that a top-down parser using grammar G is a one-state push-down automaton whose transitions are specified by G.

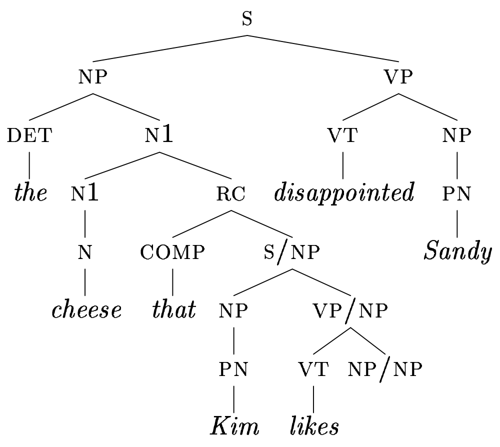

Figure 1: The analysis tree used as a running example below. Note that the phonological forms are treated here as annotations on the nodes drawn above them, rather than independent nodes. That is, DET (annotated with *the*) is a terminal node.

Thus a grammar transform is nothing more than a convenient, compact way of specifying a one-state PDA parsing algorithm.

This observation points out an inherent limitation of this method: the LR(k) parsing algorithms are essentially beyond the scope of these methods, as the LL(k) languages are a strict subset of the LR(k) languages. We will see that we can come tantalizing close, in that we can describe PLC(k) parsing in terms of grammar transforms.

An advantage of presenting left corner parsing techniques as grammar transforms is that the transformed grammars can be used with other parsing methods besides the pure top down parsing algorithms discussed here. For example, memoized left corner parsers can be constructed by using Earley's algorithm with the left corner transformed grammars.

1.2 Mappings from trees to trees

The transformations presented here can also be understood as isomorphisms from the set of parse trees of the source grammar G to parse trees of the transformed grammar which preserve terminal strings. Thus it is convenient to explain the transforms in terms of their effect on parse trees. We call a parse tree with respect to the source grammar G an *analysis tree*, in order to distinguish it from parse trees with respect to some transform of G. The analysis tree in Figure 1 will be used as an example throughout this paper.

1.3 Top-down parsers and parse trees

The "predictive" or "top-down" recognition algorithm is one of the simplest CFG recognition algorithms. Given a CFG $G = (V, P, T, S)$, a (top-down) *stack state* is a sequence $q \in (V \cup T)^*$. Let Q be the set of stack states for G. The *start state* $q_0 \in Q$ is the sequence S, and the *final state* $q_f \in Q$ is the empty sequence ϵ. The state transition function $\delta : Q \times (T \cup \{\epsilon\}) \mapsto 2^Q$ maps a state and and a terminal into a set of states. It is the smallest function δ that satisfies (1.a–1.b).

$$\gamma \in \delta(a\gamma, a) \qquad \text{for all } a \in T, \gamma \in (V \cup T)^*. \tag{1.a}$$

$$\beta\gamma \in \delta(A\gamma, \epsilon) \qquad \text{for all } A \in V, \gamma \in (V \cup T)^*, A \to \beta \in P. \qquad (1.b)$$

A string w is accepted by the top-down recognition algorithm if $q_f \in \delta^*(q_0, w)$, where δ^* is the reflexive transitive closure of δ with respect to epsilon moves, as standardly defined in the finite-state machine literature (Hopcroft and Ullman, 1979).

It is easy to read off the stack states of a top-down parser constructing a parse tree from the tree itself. For any node X in the tree, the stack contents of a top-down parser just before the construction of X consists of (the label of) X followed by the sequence of labels on the *right siblings* of the nodes encountered on the path from X back to the root. For example, the stack contents of a top-down parser immediately before the construction of the PN annotated *Kim* in the tree depicted in Figure 1 is PN VP/NP VP, and it is easy to check that a top-down parser requires a stack of depth 3 to construct this tree.

1.4 Finite state approximations

It is immediate from the definition above that a top-down parser differs from a finite-state machine (with epsilon transitions) only in that it possesses a possibly unbounded set of accessible states. Finite state machine approximations to top-down parsers can be obtained in at least two ways, both of which have the effect of bounding the number of states. (Neither seems to be of much use unless one of the left-corner transforms described below is applied to the grammar first).[1]

First, we can restrict attention to only a finite number of possible stack states. One way to do this is to impose a stack depth restriction, i.e., the top down parser's transition function is modified so that there are no transitions leaving any stack state whose size is larger than some prespecified limit. Alternatively, one can impose a bound on the number of times any given non-terminal may appear on the stack. Both of these restrictions ensure that there is only a finite number of possible stack states, and hence that the top down parser is an finite state machine. The resulting finite state machine accepts a *subset* of the language generated by the original grammar G.

Second, we can divide the infinite number of distinct stack states into a finite number of equivalence classes, and redefine the state transition function so that it operates on these equivalence classes. For example, we might choose to make two stacks equivalent if they share the same prefix of a certain length. The resulting finite state machines accept a *superset* of the language generated by the original grammar G.

1.5 Choice points in top down parsing

In general, several different top down parsing actions are possible from any given stack state and next terminal symbol. A top down parser faces a choice point whenever there is more than one production that can expand a nonterminal on top of the parser's stack (although look-ahead can sometimes rule out some of these choices). In terms of the parse tree, this means that each local tree must be identified by the parser when it has read the terminals up to the left edge of this local tree.

[1]This may not be true. Analysing the 3,267 trees of the Penn Treebank (trees containing punctuation, coordination, foreign words and headlines were discarded), the mean maximum stack depth required for a top parse was 2.9, with a standard deviation of 0.4, while for a left corner parse with the tail recursion optimization described below the mean maximum stack depth was 2.7, with a standard deviation of 0.8. This probably reflects the paucity of left-recursion in English, and suggests that it may be worthwhile attempting to develop explicit computational parsing models based on top-down parsing algorithms.

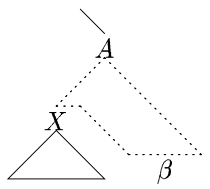

Figure 2: The relationship between category A–X in the left-corner transform grammar $\mathcal{LC}_1(G)$ and categories in the original grammar G. The dotted part of the tree is dominated by A–X.

2 The left-corner transform

This section presents the standard left-corner grammar transformation (Rosenkrantz and Lewis II, 1970; Aho and Ullman, 1972) that serves as the basis for the further transforms described in the next section. Given an input grammar G with nonterminals N and terminals T, these transforms produce grammars with an enlarged set of nonterminals $N' = V \cup (V \times (V \cup T))$. The new "pair" categories in $V \times (V \cup T)$ are written A–X, where A is a non-terminal in G and X is either a terminal or nonterminal in G. It turns out that if $A \Rightarrow^*_G X\gamma$ then $A\text{–}X \Rightarrow^*_{\mathcal{LC}_i(G)} \gamma$, i.e., a non-terminal A–X in the transformed grammar derives *the difference* between A and X in the original grammar (the dotted part of the tree in Figure 2), and the notation is meant to be suggestive of this.

The *left-corner transform* of a CFG $G = (V, P, T, S)$ is a grammar $\mathcal{LC}_1(G) = (V', P_1, T, S)$, where P_1 contains all productions of the form (2.a–2.d). (In this paper it is assumed that $V \cap T = \emptyset$, as is standard).

$$A \to a\, A\text{–}a \qquad \text{for all } A \in V, a \in T. \tag{2.a}$$

$$A \to A\text{–}C \qquad \text{for all } A \in V, C \to \epsilon \in P. \tag{2.b}$$

$$A\text{–}X \to \beta\, A\text{–}B \qquad \text{for all } A \in V, B \to X\,\beta \in P. \tag{2.c}$$

$$A\text{–}A \to \epsilon \qquad \text{for all } A \in V. \tag{2.d}$$

Figure 3 depicts the effect of the different production schemata (2.a–2.d) in terms of configurations of nodes of the analysis tree with respect to G, and Figure 4 sketches the general structural relationship between analysis trees and parse trees of $\mathcal{LC}_1(G)$. Informally, the productions (2.a) and (2.b) start the left-corner recognition of A by recognizing either a terminal a or an empty category C as a possible left-corner of A. The actual left-corner recognition is performed by the productions (2.c), which extend the left-corner from X to its parent B by recognizing β; these productions are used repeatedly to construct increasingly larger left-corners. Finally, the productions (2.c) terminate the recognition of A when this left-corner construction process has constructed an A.

The effect of this transform on the analysis tree in Figure 1 is shown in Figure 5. The transformed tree is considerably more complex: it has double the number of nodes of the original tree. In order to understand this tree, pay attention to the right-hand part of the pair categories, and notice that left-corner dominance relationships in the analysis tree corresponds to right-corner inverse dominance relationships in the transformed tree; e.g., the left-most left-corner chain in Figure 1, namely DET NP S, corresponds to the right-corner chain S S-DET S-NP S-S. In a top-down parse of the tree in Figure 5 the maximum stack depth is 6, which occurs at the recognition of the empty node NP/NP.

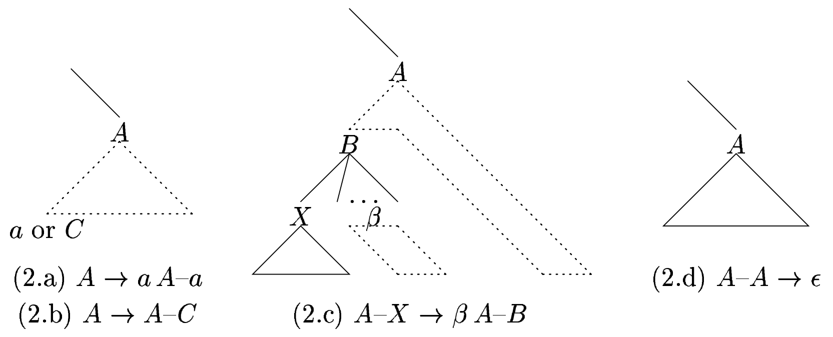
A
a or C
(2.a) $A \to a\, A\text{–}a$
(2.b) $A \to A\text{–}C$
A
B
X
. . .
β
(2.c) $A\text{–}X \to \beta\, A\text{–}B$
A
(2.d) $A\text{–}A \to \epsilon$

Figure 3: The configurations of nodes of the analysis tree in which the various production schemata of $\mathcal{LC}_1(G)$ apply. The tree fragments drawn in solid lines would already have been recognized by a top-down recognizer using $\mathcal{LC}_1(G)$ by the time this production was used, while the dotted tree fragments correspond to predictions still to be instantiated.

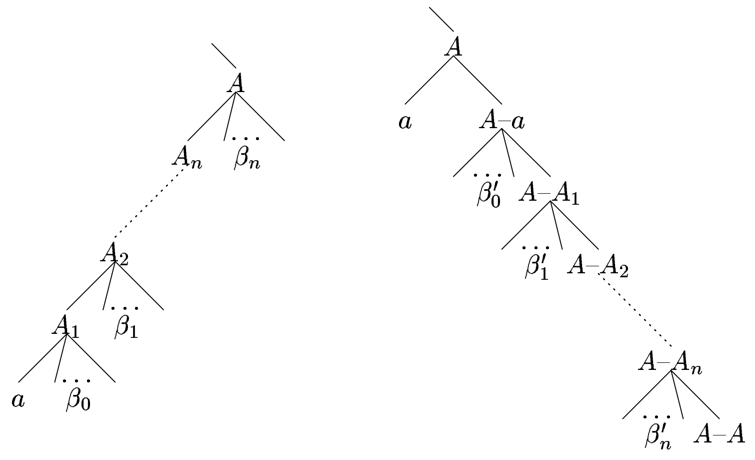

Figure 4: The structural relationship between analysis trees of G (on the left) and parse trees of $\mathcal{LC}_1(G)$ (on the right). In this diagram, the trees β'_i are obtained by applying the left-corner transform to each of the trees β_i.

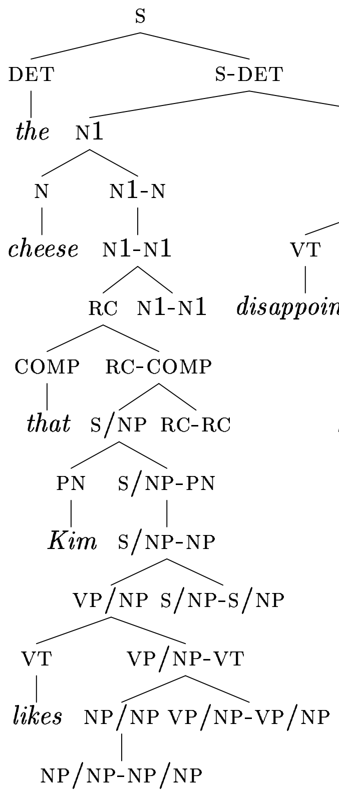

Figure 5: The result of applying the transform $\mathcal{LC}_1$ to the tree in Figure 1.

2.1 Choice points in left-corner parsing

The left corner transform changes the location in the input string of the choice points corresponding to local trees in the original parse tree. Specifically, a choice point corresponding to the different possible instantiations of a local tree is located immediately after the recognition of the leftmost subtree of that local tree. This can be significantly further to the right in the terminal string than the corresponding choice point in top down parsing, and there are grammars that are deterministically left corner parsable, but not deterministically top down parsable. (In grammar transform terms $\mathcal{LC}_1(G) \in \mathrm{LL}(k)$, i.e., is deterministically top down parsable, with lookahead k, even though G is not).

Left-corner parsing also introduces a new sort of choice point not present in top-down parsing: when the left-corner subtree X is recognized as completing the category A predicted top-down in the pair categories A–X. In standard left corner parsing this choice point is reached immediately after the entire subtree (including all of the children) corresponding to the prediction has been been constructed. In terms of top down parsing using the transformed grammar $\mathcal{LC}_1(G)$, this corresponds to the fact that the productions of schema A–$A \to \epsilon$ are used immediately after the node A has been recognized bottom up.

2.2 Filtering useless categories

In general the grammar produced by the transform $\mathcal{LC}_1(G)$ contains a large number of *useless nonterminals*, i.e., non-terminals which can never appear in any complete derivation, even if the grammar G is fully pruned (i.e., contains no useless non-terminals or productions). While the grammars $\mathcal{LC}_1(G)$ can be pruned using the standard algorithms, given the observation about the relationship between the pair non-terminals in $\mathcal{LC}_1(G)$ and non-terminals in G presented in Figure 2, it is clear that certain productions can be discarded immediately as useless. Define the *left corner* relation $\lhd \subseteq (V \cup T) \times V$ as follows:

$$X \lhd A \qquad \text{iff} \qquad \exists \beta .\ A \to X\beta \in P, \tag{3}$$

Let $\lhd^*$ be the reflexive and transitive closure of $\lhd$. It is easy to show that a category A–X is useless in $\mathcal{LC}_1(G)$ (i.e., derives no sequence of terminals) unless $X \lhd^* A$. Thus we can restrict the productions in (2.a–2.d) without affecting the language (strongly) generated to those that only contain pair categories A–X where $X \lhd^* A$.

2.3 Unification grammars

One of the main advantages of left-corner parsing algorithms over LR(k) based parsing algorithms is that they extend straight-forwardly to complex feature based "unification" grammars: the transformations listed above are applied directly to the annotated phrase structure rules, and the feature "unification" constraints with phrase structure rules are copied directly into the corresponding rules of the transformed grammar. The transformation $\mathcal{LC}_1$ itself can be encoded in several lines of Prolog (Matsumoto et al., 1983; Pereira and Shieber, 1987).

This contrasts with the LR(k) methods. In LR(k) parsing a single LR state may correspond to several items or dotted rules, so it is not clear how the feature "unification" constraints should be associated with transitions from LR state to LR state (see Nakazawa (1995) for one proposal). Pereira and Wright (1996) pose the extension of their technique for constructing finite-state approximations of CFGs using an LR-based method to unification grammars as an open problem for further research. in contrast, extending the techniques described here to complex feature based "unification" grammar is in principle straight-forward.

The main complication is the filter on useless nonterminals and productions discussed in subsection 2.2 above. Generalizing the left corner closure filter on pair categories to complex feature "unification" grammars in an efficient way is complicated, and is the primary difficulty in using left-corner methods with complex feature based grammars.

It is easy to construct unification grammars for which the closure $\triangleleft^*$ of the left corner relation $\triangleleft$ is not a recursive set. However, this set can be finitely approximated by using *restriction* (Shieber, 1985) to restrict attention to a finite subset of categories (this method is called *generalization* in the logic programming literature). One constructs a finite set F of possibly partially instantiated pair categories such that if $X \triangleleft^* A$ then there is some $A'\text{–}X' \in F$ such that $A'\text{–}X'$ subsumes $A\text{–}X$.

A naive way of applying this filter to the transformed grammar is to unify each pair category $A\text{–}X$ in the productions of the transformed category with each $A'\text{–}X' \in F$. This results in a grammar that is correct, but typically introduces massive spurious ambiguities, as in general a pair category $A\text{–}X$ occuring in a production produced by the transform can unify with several distinct categories in F, resulting in "overlapping" productions.

There seem to be at least two ways of dealing with this; neither method is a totally satisfactory solution to the problem, and both can be applied together.

First, one can apply the filter at run time during the parsing process, and merely test each freshly constructed pair category for unifiability with at least one member of F, but not actually perform the unification. This run-time compatability test is computationally expensive, and because unification is not actually performed no feature dependencies are progagated between the top-down component A and bottom-up component X in pair categories $A\text{–}X$. This can dramatically weaken the effectiveness of the filter for certain grammars.

Second, one can further weaken the left corner closure relation so that distinct pair categories in F never unify with each other. One way of doing this is to replace with their generalization $U_1 \sqcup U_2$ any two pair categories $U_1, U_2 \in F$ where U_1 and U_2 unify. This also dramatically weakens the power of the filter, but the resulting filter can be applied off-line, and it may permit at least some feature percolation through the filter.

3 Extended left-corner transforms

This section presents some simple extensions to the basic left corner transform presented above. The tail-recursion optimization discussed immediately below permits bounded-stack parsing of both left and right recursive structures. Further manipulation of this transform puts it into a form in which we can identify precisely the tree configurations in the original grammar which cause the stack size of a left corner parser to increase. These observations motivate the special binarization methods described in the next section, which minimize stack depth in grammars that contain productions of length greater than two.

3.1 A tail-recursion optimization

If G is a left-linear grammar, a top-down parser using $\mathcal{LC}_1(G)$ can recognize any string generated by G with a constant-bounded stack size. (The parse trees are as sketched in Figure 4, with all of the β_i' terminals). However, the corresponding operation with right-linear grammars requires a stack of size proportional to the length of the string, since the stack fills with paired categories $A–A$ for each nonterminal node in the analysis tree. These paired categories will only be popped at the end of the string using the epsilon productions in schema (2.d).

The "tail recursion" or "composition" optimization (Abney and Johnson, 1991; Resnik, 1992) permits right-branching structures to be parsed with bounded stack depth. It is the result of epsilon removal applied to the output of $\mathcal{LC}_1$, and can be described in terms of resolution or partial evaluation of the transformed grammar with respect to productions (2.d). In effect, each of the schemata (2.b–2.c) is split into two cases, depending on whether or not the rightmost nonterminal $A–B$ is expanded by the epsilon rules produced by schema (2.d). This expansion yields a grammar $\mathcal{LC}_2(G) = (V', P_2, T, S)$, where P_2 contains all productions of the form (4.a–4.e).

$$A \to a\, A–a \qquad \text{for all } A \in V, a \in T. \tag{4.a}$$

$$A \to A–C \qquad \text{for all } A \in V, C \to \epsilon \in P. \tag{4.b}$$

$$C \to \epsilon \qquad \text{for all } C \to \epsilon \in P. \tag{4.c}$$

$$A–X \to \beta\, A–B \qquad \text{for all } A \in V, B \to X\,\beta \in P. \tag{4.d}$$

$$A–X \to \beta \qquad \text{for all } A \to X\,\beta \in P. \tag{4.e}$$

The parse trees of $\mathcal{LC}_2(G)$ stand in the same relationship to analysis trees sketched in Figure 4, except that the empty subtree $A–A$ at the bottom right of that diagram is no longer present. Figure 6 shows the effect of the transform $\mathcal{LC}_2$ on the example parse tree. The maximum stack depth required for this tree is 3 (just as for the untransformed grammar). It seems that the stack depth required by a top-down parser on any grammar G is never less than the stack depth required by $\mathcal{LC}_2(G)$.

When this "tail recursion" optimization is applied, pair categories in the transformed grammar encode *proper* left-corner relationships between nodes in the analysis tree. This lets us strengthen the useless category filter described above as follows. Let $\lhd^+$ be the transitive closure of the left-corner relation defined in (3). It is easy to show that a category $A–X$ is useless in $\mathcal{LC}_2(G)$ (i.e., derives no sequence of terminals) unless $X \lhd^+ A$. Thus we can restrict the productions in (4.a–4.d) without affecting the language (strongly) generated to just those that only contain pair categories $A–X$ where $X \lhd^+ A$.

This transform also affects the location of the parser's choice points associated with the completion of the top down prediction. Because productions of type (2.d) have been resolved into the other productions, productions (4.c) and (4.e) have this completion operation "built in", whereas (4.b) and (4.d) can only be used if the top down prediction is incomplete. Thus the tail recursion optimization moves the completion choice point forward to just after the left corner has been recognized (instead of just after all of the children have been recognized, as in standard left corner parsing). It is easy to construct grammars G for which $\mathcal{LC}_1(G)$ is deterministically (top down) parsable, but $\mathcal{LC}_2(G)$ is not deterministically parsable.

3.2 The special case of binary productions

We can get a better idea of the properties of transformation $\mathcal{LC}_2$ if we investigate the special case where the productions of G have at most two symbols on the right hand-side (i.e., all productions are nullary, unary or binary). In this situation, transformation $\mathcal{LC}_2(G)$ can be more explicitly written as $\mathcal{LC}_3(G) = (V', P_3, T, S)$, where P_3 contains all instances of the production schemata (5.a–5.g).

$$A \to a\, A–a \qquad \text{for all } A \in V, a \in T. \tag{5.a}$$

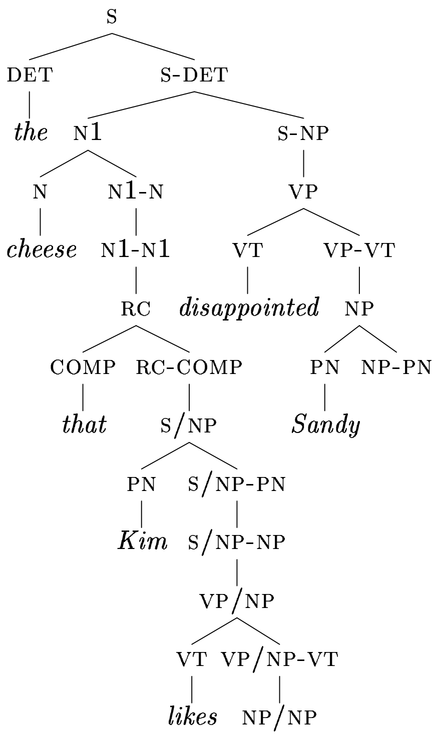

Figure 6: The result of applying the transform $\mathcal{LC}_2$ to the tree in Figure 1.

$$A \to A\text{–}C \qquad \text{for all } A \in V, C \to \epsilon \in P. \tag{5.b}$$

$$C \to \epsilon \qquad \text{for all } C \to \epsilon \in P. \tag{5.c}$$

$$A\text{–}X \to A\text{–}B \qquad \text{for all } A \in V, B \to X \in P. \tag{5.d}$$

$$A\text{–}X \to \epsilon \qquad \text{for all } A \to X \in P. \tag{5.e}$$

$$A\text{–}X \to Y\,A\text{–}B \qquad \text{for all } A \in V, B \to X\,Y \in P. \tag{5.f}$$

$$A\text{–}X \to Y \qquad \text{for all } A \to X\,Y \in P. \tag{5.g}$$

Productions (5.b–5.c) in $\mathcal{LC}_3(G)$ correspond to empty productions in the original grammar G, while (5.d–5.e) and (5.f–5.g) correspond to unary and binary productions respectively.

Now, note that nonterminals from V only appear in productions of type (5.f) and (5.g). Moreover, any such nonterminals must be immediately expanded by a production of type (5.a–5.c).

Thus the non-terminals in V in the output of $\mathcal{LC}_3$ are eliminable by resolving them with (5.a–5.c); the only remaining occurence of a nonterminal from V is the start symbol S. This expansion yields a new transform $\mathcal{LC}_4$, where $\mathcal{LC}_4(G) = (\{S\} \cup (V \times (V \cup T)), P_4, T, S)$. P_4, defined in (6.a–6.m), still contains productions of type (5.a–5.c), but these only directly expand the start symbol, as all occurences of other categories from V have been resolved away.

$$S \to a\,S\text{–}a \qquad \text{for all } a \in T. \tag{6.a}$$

$$S \to S\text{–}C \qquad \text{for all } C \to \epsilon \in P. \tag{6.b}$$

$$S \to \epsilon \qquad \text{if } S \to \epsilon \in P. \tag{6.c}$$

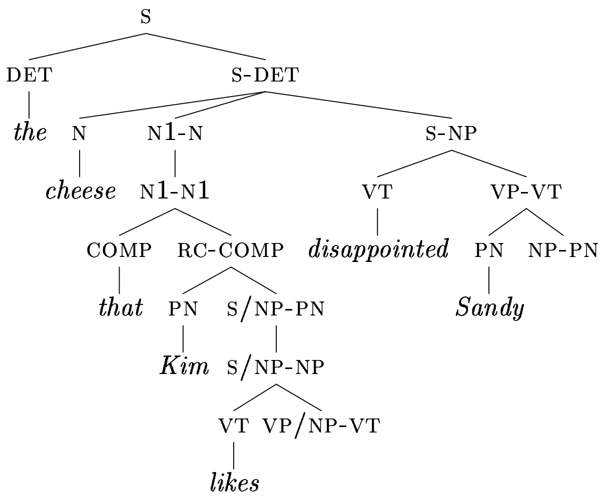

Figure 7: The result of applying the transform $\mathcal{LC}_4$ to the tree in Figure 1.

$$A\text{–}X \to A\text{–}B \qquad \text{for all } A \in V, B \to X \in P. \tag{6.d}$$

$$A\text{–}X \to \epsilon \qquad \text{for all } A \to X \in P. \tag{6.e}$$

$$A\text{–}X \to a\, A\text{–}B \qquad \text{for all } A \in V, a \in T, B \to X\, a \in P. \tag{6.f}$$

$$A\text{–}X \to a \qquad \text{for all } a \in T, A \to X\, a \in P. \tag{6.g}$$

$$A\text{–}X \to a\, C\text{–}a\, A\text{–}B \qquad \text{for all } A, C \in V, a \in T, B \to X\, C \in P. \tag{6.h}$$

$$A\text{–}X \to a\, C\text{–}a \qquad \text{for all } C \in V, a \in T, A \to X\, C \in P. \tag{6.i}$$

$$A\text{–}X \to C\text{–}D\, A\text{–}B \qquad \text{for all } A, C \in V, B \to X\, C, D \to \epsilon \in P. \tag{6.j}$$

$$A\text{–}X \to C\text{–}D \qquad \text{for all } C \in V, A \to X\, C, D \to \epsilon \in P. \tag{6.k}$$

$$A\text{–}X \to A\text{–}B \qquad \text{for all } A \in V, B \to X\, C, C \to \epsilon \in P. \tag{6.l}$$

$$A\text{–}X \to \epsilon \qquad \text{for all } A \to X\, C, C \to \epsilon \in P. \tag{6.m}$$

In the production schemata defining $\mathcal{LC}_4$, (6.a–6.e) are copied directly from (5.a–5.e) respectively. The schemata (6.f–6.g) are obtained by instantiating Y in (5.f–5.g) to a terminal $a \in T$, while the other six schemata (6.h–6.m) are obtained by instantiating Y in (5.f–5.g) with the right hand sides of (5.a–5.c), renaming variables if necessary. Figure 7 shows the result of applying the transformation $\mathcal{LC}_4$ to the analysis tree of Figure 1.

The schemata (6.a–6.c) produce productions that can be viewed as initializing the left-corner parsing process by constructing the appropriate paired category. The instances of (6.d–6.e) each correspond to a single unary productions in the original grammar. The instances of the remaining schemata (6.f–6.m) each correspond to a single binary production; the different schemata correspond to all the ways in which the local tree corresponding to this production might be related to the predicted category A and the right subtree.

The transform also simplifies the specification of finite state machine approximations. Because all terminals are introduced as the left-most symbols in their productions, there is no need for terminal symbols to appear in the parser's stack, saving an epsilon transition associated with a stack push and an immediately stack

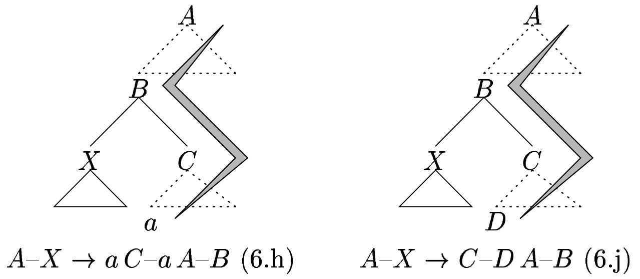
A
B
X
C
a
$A\text{–}X \to a\,C\text{–}a\,A\text{–}B$ (6.h)
A
B
X
C
D
$A\text{–}X \to C\text{–}D\,A\text{–}B$ (6.j)

Figure 8: The highly distinctive "zig-zag" or "lightning bolt" configuration of nodes in the analysis tree characteristic of the use of production schemata (6.h) and (6.j) in transform $\mathcal{LC}_4$. This is the only configuration which causes an increase in stack depth in a top-down parser using a grammar transformed with $\mathcal{LC}_2$, $\mathcal{LC}_3$ or $\mathcal{LC}_4$.

pop with respect to the standard left corner algorithm. Productions (6.a) and (6.f–6.i) can be understood as transitions over a terminal a that replace the top stack element with a sequence of other elements, while the other productions can be interpreted as epsilon transitions that manipulate the stack contents accordingly.

An interesting thing here is that the right hand sides of all of these productions except for schemata (6.h) and (6.j) are linear, i.e., contain at most one nonterminal. Thus instances these schemata are the only productions that can increase the stack size of a top-down parse with $\mathcal{LC}_4(G)$. Figure 8 sketches the configuration of nodes in the analysis trees in which instances of schemata (6.h) and (6.j) would be used in a parse using $\mathcal{LC}_4(G)$. This highly distinctive "zig-zag" or "lightning bolt" pattern occurs once in the example tree in Figure 1 (with A = S, B = NP, C = N1 and a = N), so the maximum required stack depth is 3. (Recall that in a traditional top-down parser terminals are pushed onto the stack and popped later, so initialization productions (6.a) cause two symbols to be pushed onto the stack).

4 Binarization

The observations just made about the configurations in which stack size increases suggest that it may be worthwhile reconsidering how productions in G of length greater than two are treated.

In the grammar transforms discussed above that can deal with productions of arbitrary length (namely $\mathcal{LC}_1$ and $\mathcal{LC}_2$), n-ary productions in G correspond to n-ary or $n-1$-ary productions of the transformed grammar $\mathcal{LC}(G)$. Thus in a finite-state approximation with a stack-depth bound of m, only productions of length $m+1$ or greater can possibly be used. Moreover, the length of useful productions decreases as the depth of embedding increases. This seems highly unrealistic in natural language parsing applications.

One natural way to deal with this problem is to reduce all productions of length greater than 2 to sequences of binary productions. This is often done "on the fly" in the parser, but we present binarization methods as grammar transforms here.

There are an extremely large number of ways to convert n-ary branching nodes into binary branching structures (even purely left branching or purely right branching structures), and the effects that the different methods have on the behaviour of even well-known parsing algorithms is still largely unexplored. In the context of the various left corner transforms discussed above, binarization can applied either before or after the left corner transform, yielding even more possibilities for exploration.

Standard methods for converting n-ary branching nodes into left associative or right associative binary branching structures can be presented as grammar transforms. Given an input grammar $G = (V, P, T, S)$, we define new grammars $\mathcal{L}(G)$ and $\mathcal{R}(G)$, which are G's *left binarization* and *right binarization* respectively. The non-terminals of $\mathcal{L}(G)$ are those of G together with all non-empty proper prefixes of the right hand sides of G's productions, while the non-terminals of $\mathcal{R}(G)$ are those of G together with all non-empty proper suffixes of the right hand sides of G's productions.

The left associative tranform $\mathcal{L}(G) = (V_l, P_l, T, S)$ is defined as follows:

$$V_l \;=\; V \cup \{\beta : A \to \beta\,\gamma,\ \beta, \gamma \in (V \cup T)^+\} \tag{7}$$

$$\begin{aligned} P_l \;=\; & \{A \to \beta : A \to \beta \in P,\ |\beta| \le 2\} \cup \\ & \{A \to \text{`}\beta\text{'}\, X : A \to \beta\, X \in P,\ |\beta| \ge 2\} \cup \\ & \{\text{`}\beta X\text{'} \to \text{`}\beta\text{'}\, X : X \in V \cup T,\ X \in V \cup T,\ \beta \in (V \cup T)^+\} \end{aligned} \tag{8}$$

In (8) and below, 'β' notates a single nonterminal in a transformed grammar which consists of a sequence of categories from the source grammar; note thar 'X' and X are the same if $X \in V \cup T$.

The right associative tranform $\mathcal{R}(G) = (V_r, P_r, T, S)$ essentially a mirror image of $\mathcal{L}(G)$, as would be expected.

$$V_r \;=\; V \cup \{\gamma : A \to \beta\,\gamma,\ \beta, \gamma \in (V \cup T)^+\} \tag{9}$$

$$\begin{aligned} P_r \;=\; & \{A \to \beta : A \to \beta \in P,\ |\beta| \le 2\} \cup \\ & \{A \to X\,\text{`}\beta\text{'} : A \to X\,\beta \in P,\ |\beta| \ge 2\} \cup \\ & \{\text{`}X\beta\text{'} \to X\,\text{`}\beta\text{'} : X \in V \cup T,\ X \in V \cup T,\ \beta \in (V \cup T)^+\} \end{aligned} \tag{10}$$

The left associative binarization transform $\mathcal{L}$ has the property that the transformed grammar when used with a standard left corner parser behaves as a PLC parser (Nijholt, 1980), i.e., the choice point for the category of node in the analysis tree is delayed in a top-down parse using $\mathcal{LC}_1(\mathcal{L}(G))$ until its left edge has been reached in the input stream. (In standard left corner parsing an entire production is chosen once its left corner has been recognized, which may entail prediction of many categories to the right). Because this left binarization transform delays some of the choice points, there are grammars G such that $\mathcal{LC}_1(\mathcal{L}(G))$ is deterministically top down parsable, even though $\mathcal{LC}_1(G)$ is not. Thus the left associative binarization transform is a reasonable choice if the goal is to delay choice points as long as possible (i.e., to make parsing as deterministic as possible).

Interestingly, neither uniform right nor uniform left association seems to be optimal in terms of minimizing stack depth requirements.

Figure 9 shows the result of the two different binarization strategies of a local tree consisting of a parent labelled A itself on a left branch and children $X_0\, b_1\, X_2$, with $b_1 \in T$. As that figure shows, right association in such a situation causes an instance of the "zig-zag" pattern that is not present in the tree produced by left association that indicates an increase in stack depth.

The situation is reversed if the local tree being binarized lies on a right branch and the second subtree is a non-empty nonterminal. As Figure 10 shows, left association in such a situation induces the "zig-zag" pattern here that signals an increase in stack depth.

Thus the optimal binarization strategy seems to be to use a left associative binarization strategy on subtrees lying on a left branch, and a right associative binarization strategy on subtrees lying on a right branch. It is straight-forward, but tedious, to define a grammar transform that achieves this.

Finally, it is worth remembering that simply minimizing stack depth is not an appropriate goal in isolation: stack depth can be reduced by an arbitrary constant

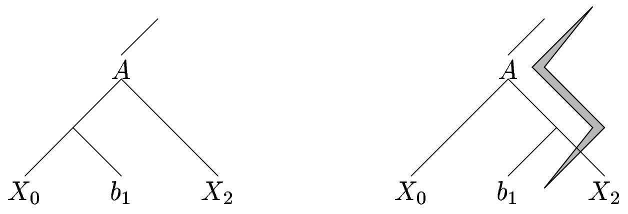

Figure 9: The result of left association and right association applied to a local tree with parent labelled A, itself lying on a *left branch*, and children $X_0\, b_1\, X_2$, where b_1, is a *terminal.* Right association induces an instance of the “zig-zag” pattern not present in the tree produced by left association.

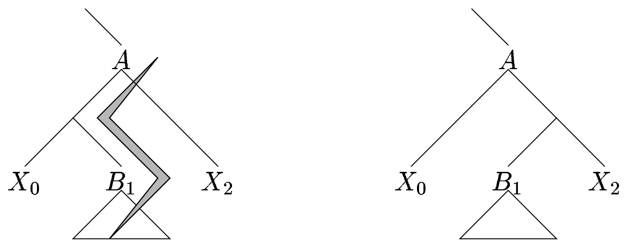

Figure 10: The result of left association and right association applied to a local tree with parent labelled A, itself lying on a *right branch*, and whose second child, B_1, is a *non-empty non-terminal.* Here, left association induces an instance of the “zig-zag” pattern not present in the tree produced by right association.

factor by simply collapsing adjacent sequences of stack symbols into a single complex symbol. If the goal is the construction of finite state approximations to CFGs, then the primarily practical limitation is that the number of different accessible stack configurations that satisfy the stack depth bound may be astronomical, and it might be more reasonable to choose a binarization strategy that reduces the total number of accessible stack configurations, even at the price of larger stack depths.

5 Reporting parse trees

This paper has concentrated so far on recognition algorithms. This section discusses how information about the structure of the input can be obtained by these techniques as well. This research is still in progress, and this section should be read as a progress report. It should be noted that finite state approximation are still useful even in the absence of techniques for extracting structure; e.g., Pereira and Wright (1996) use similar finite state approximations purely as a filter in a pre-processing stage before parsing proper begins.

5.1 Constructing analysis trees from left corner derivations

In a non-deterministic stack based implementation a parsing algorithm can be obtained by constructing an analysis tree in tandem with the recognition process. A simple way to do this is to express the grammar as a Definite Clause Grammar, and add an additional argument to each category that encodes the analysis tree rooted at the node that instantiates this category, e.g., in the manner described by Pereira and Shieber (1987). At the end of the recognition process this argument will be instantiated to the parse tree generated by the original grammar, even if the recognizer itself applies one or more of the grammar transforms discussed here.

However, this method for extracting analysis trees cannot be used directly if the CFG is to be compiled into a finite state approximation: a finite number of states cannot encode the infinite number of distinct analysis trees. This section considers ways of recovering analysis trees from finite state analyses.

Kay (1996) suggests the construction of a finite state transducer that maps the string to be analysed into a sequence of symbols (say, the stack states) recording the steps that a left corner parser would take analysing this string. This transducer's output is then used as an *oracle* for a conventional left corner parser which actually produces the analysis trees that are the output of the parsing process.

Now, because the number of transducer states is finite, a chart-like graph representing all the possible transductions can be constructed in time linear in the length of the input string. The standard method for computing this transduction involves a forward "construction" phase and a backward "pruning" phase, both of which may require time proportional to the number of transducer states. Since this number of states is proportional to the number of distinct bounded stack states, it may be quite large, and although a constant factor, might never the less dominate parsing time.

There is a simple extension of Kay's idea, based on the idea of a *sequential bimachine* (Reutenauer and Schutzenberger, 1991) that avoids this constant factor. One reverses, determinizes and then minimizes a finite state *machine* approximation of the left corner parser constructed as described in subsection 1.4. Now, note that the states of this machine can be interpreted as *sets of stack states* of the original left corner parser, and that we can construct a table that identifies whether a given parser stack state is a member of the set of stack states associated with a state of the finite state automaton.

Moreover, running this automaton on a *reversed* input string associates each string position in that string with one of the finite state machine's states, and therefore a set of stack states of the left-corner parser. By construction, the left corner parser stack states associated with each string position have at least one non-deterministic path over the remaining input that leads to a final parser state. Thus the single states assigned to each string position by the reversed, minimized automaton together with the table mapping automaton states to sets of left corner parser stack states serves as a quickly constructed oracle that restricts the left corner parser to successful parses.

However both Kay's original proposal and the extension just proposed suffer from another difficulty: the number of different bounded stack left corner parses can grow exponentially with the length of the string to be parsed. It is possible to pack this set of left corner derivations into a chart-like graph whose size is linear in the length of the input string, but the problem of *interpreting* it still remains.

One can map this packed representation of left-corner parses into a standard chart representation of (standard) parse trees. Unfortunately, it is easy to show that the set of standard parse trees corresponding to such a packed representation of left corner derivations cannot be exactly represented by a standard chart representation, as the left corner stack depth bound can introduce non-local constraints in parse trees. However, it is possible to construct a standard chart representation from the packed left corner parses which includes all of the standard parse trees corresponding to the left corner parses, and for which the "extra" standard parse trees present in the standard chart that do not correspond to any left corner parse are in fact well formed parses with respect to the original grammar. This chart's size is a quadratic function of input sentence length, and its construction from the packed left corner derivations can be done in cubic time, i.e., it has the same time requirements as parsing the original input string with the original grammar. However, the constant factors may be much better here than in standard CFG parsing algorithms, since all parse-time search has been eliminated.

5.2 A left corner bracketting transduction

The previous subsection concentrated on retrieving standard analysis trees from left corner derivations. This subsection shows how a transducer can be constructed from a left corner parser or a finite state approximation thereof that produces partially bracketted representations of standard analysis trees that can be extended to standard fully bracketted analysis trees as described below.

First, note that no finite state transducer can produce fully bracketted representations of standard parse trees for the full range of bounded stack left corner parsable grammars. For example, a recursive right linear grammar (i.e., one that generates purely right branching structures) is bounded stack left corner parsable (and hence defines a finite state language), yet no finite state transducer can produce fully bracketted analysis trees for such a grammar, as that involves producing $|w|$ closing brackets at the end of the input string w.

Thus we must content ourselves with a weaker bracketted representation. The bracketting transduction proposed here produces at least either an opening or a closing bracket for each constituent. It is based on the subset of production schemata (6.a) and (6.f–6.i) of $\mathcal{LC}_4$ that deal with only binary productions: extending it to non-binary productions requires further research. The bracketting scheme is based on the idea of "superbrackets" sometimes used in Lisp programs, which can close off an arbitrary number of ordinary brackets.

Given a grammar $G = (V, P, T, S)$ where all productions in P are binary, we specify a new grammar $\mathcal{LC}_b(G) = (\{S\} \cup (V \times (V \cup T)), P_b, T_b, S)$ as follows. The terminal symbols T_b are pairs $w : a$, where $a \in T$ and $w \in (V \cup T \cup \{\langle, [, \langle\!\langle, \rangle,]\})^+$.

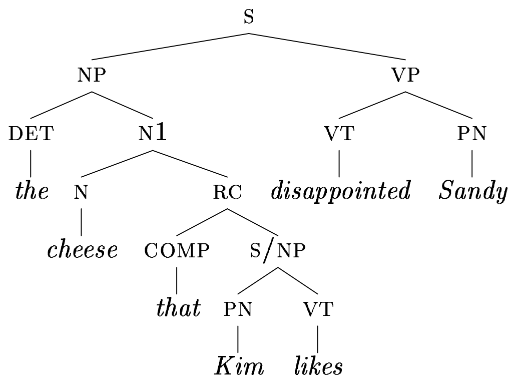

Figure 11: A pure binary branching tree used to demonstrate the left corner representation

A terminal $w : a$ should be viewed as mapping the symbol a in the input to the string w, which constitutes part of the bracketted representation of the input. This representation contains three different kinds of left brackets, viz., '$\langle$', '[' and '$\langle\!\langle$', and two kinds of right brackets '$\rangle$' and ']'. Their interpretation in terms of standard analysis trees will be defined below.

The productions P_b are the instances of the production schemata (11.a–11.e). These are exactly the same as (6.a) and (6.f–6.i), except that the terminal a in these latter production schemata has been replaced with a terminal of the form $w : a$. Thus the stack states of the transducer $\mathcal{LC}_b(G)$ are exactly the same as for the recognizer $\mathcal{LC}_4(G)$.

$$S \to \text{`}[S\text{'}\!: a\; S\text{–}a \qquad \text{for all } a \in T. \tag{11.a}$$

$$A\text{–}X \to \text{`}X\rangle\langle a\text{'}\!: a\; A\text{–}B \qquad \text{for all } A \in V, a \in T, B \to X\, a \in P. \tag{11.b}$$

$$A\text{–}X \to \text{`}X\rangle\langle a]\text{'}\!: a \qquad \text{for all } a \in T, A \to X\, a \in P. \tag{11.c}$$

$$A\text{–}X \to \text{`}X\rangle[C\text{'}\!: a\; C\text{–}a\; A\text{–}B \qquad \text{for all } A, C \in V, a \in T, B \to X\, C \in P. \tag{11.d}$$

$$A\text{–}X \to \text{`}X\rangle\langle\!\langle C\text{'}\!: a\; C\text{–}a \qquad \text{for all } C \in V, a \in T, A \to X\, C \in P. \tag{11.e}$$

In the transduction, a '[' indicates a stack push, i.e., an increase in the left corner stack depth, while a ']' indicates a stack pop, or a decrease in stack depth. The '$\langle$' introduces a terminal lying on a right branch, while the '$\langle\!\langle$' introduces a non-terminal lying on a right branch. Finally, the '$\rangle$' closes a node lying on a left branch (not necessarily a terminal).

We demonstrate the left corner representation using the tree in Figure 11 (which is just the result of unary and nullary rule removal applied to the tree in Figure 1). The output given the terminal string of this tree produced by the transduction defined by $\mathcal{LC}_b$ is shown in (12).

$$\begin{array}{l} [_{\mathrm{S}}\ \mathsf{the}\ _{\mathrm{DET}}\rangle\ [_{\mathrm{N1}}\ \mathsf{cheese}\ _{\mathrm{N}}\rangle\ \langle\!\langle_{\mathrm{RC}}\ \mathsf{that}\ _{\mathrm{COMP}}\rangle\ \langle\!\langle_{\mathrm{S/NP}}\ \mathsf{Kim}\ _{\mathrm{PN}}\rangle \\ \langle_{\mathrm{VT}}\ \mathsf{likes}\,]\ _{\mathrm{NP}}\rangle\ \langle\!\langle_{\mathrm{VP}}\ \mathsf{disappointed}\ _{\mathrm{VT}}\rangle\ \langle_{\mathrm{PN}}\ \mathsf{Sandy}\,] \end{array} \tag{12}$$

This left corner bracketting contains sufficient information to recover the standard analysis tree. The following operations show how to obtain a standard analysis tree bracketted only with '[' and ']'.

1. A right ‘]’ bracket closes off the matching left ‘[’ bracket, plus any number of open left ‘⟪’ brackets. Changing all ‘⟪’ brackets into ‘[’ brackets and adding matching ‘]’ brackets converts (12) into (13).

$$\begin{array}{l}[_{\mathrm{S}}\, \textsf{the}\,_{\mathrm{DET}}\rangle\; [_{\mathrm{N1}}\, \textsf{cheese}\,_{\mathrm{N}}\rangle\; [_{\mathrm{RC}}\, \textsf{that}\,_{\mathrm{COMP}}\rangle\; [_{\mathrm{S/NP}}\, \textsf{Kim}\,_{\mathrm{PN}}\rangle \\ \langle_{\mathrm{VT}}\, \textsf{likes}\,]\,]\,]\,_{\mathrm{NP}}\rangle\; [_{\mathrm{VP}}\, \textsf{disappointed}\,_{\mathrm{VT}}\rangle\; \langle_{\mathrm{PN}}\, \textsf{Sandy}\,]\,]\end{array} \tag{13}$$

2. Right ‘⟩’ brackets take as wide a scope as possible, consistent with the ‘[’, ‘]’ bracketting just established. (In terms of the original bracketting, left ‘[’ and ‘⟪’ brackets both close an arbitrary number of ‘⟩’ brackets). To make the display more readable, the nonterminal immediately before the right‘⟩’ bracket can be moved to subscript the newly introduced left ‘[’ bracket. Closing off ‘⟩’ brackets in this way converts (13) into (14).

$$\begin{array}{l}[_{\mathrm{S}}[_{\mathrm{NP}}[_{\mathrm{DET}}\, \textsf{the}\,]\; [_{\mathrm{N1}}[_{\mathrm{N}}\, \textsf{cheese}\,]\; [_{\mathrm{RC}}[_{\mathrm{COMP}}\, \textsf{that}\,]\; [_{\mathrm{S/NP}}[_{\mathrm{PN}}\, \textsf{Kim}\,] \\ \langle_{\mathrm{VT}}\, \textsf{likes}\,]]]]\; [_{\mathrm{VP}}[_{\mathrm{VT}}\, \textsf{disappointed}]\; \langle_{\mathrm{PN}}\, \textsf{Sandy}\,]]\end{array} \tag{14}$$

3. Finally, ‘⟨’ brackets are closed off taking as wide a scope as possible, consistent with the ‘[’, ‘]’ bracketting already established. This converts (14) into (15).

$$\begin{array}{l}[_{\mathrm{S}}[_{\mathrm{NP}}[_{\mathrm{DET}}\, \textsf{the}\,]\; [_{\mathrm{N1}}[_{\mathrm{N}}\, \textsf{cheese}\,]\; [_{\mathrm{RC}}[_{\mathrm{COMP}}\, \textsf{that}\,]\; [_{\mathrm{S/NP}}[_{\mathrm{PN}}\, \textsf{Kim}\,] \\ [_{\mathrm{VT}}\, \textsf{likes}\,]]]]]\; [_{\mathrm{VP}}[_{\mathrm{VT}}\, \textsf{disappointed}]\; [_{\mathrm{PN}}\, \textsf{Sandy}\,]]]\end{array} \tag{15}$$

6 Conclusion

This paper surveyed the issues arising in the construction of finite state approximations of various kinds of left corner parsers. The different kinds of parsers were presented as grammar transforms, which let us abstract away from the algorithmic details of parsing algorithms themselves. We derived the various forms of the left corner parsing algorithms in terms of grammar transformations from the original left corner grammar transform. The last section of the paper discussed how finite state approximations might be extended to provide a partial bracketting representing the analysis trees.

References

Abney, Stephen and Mark Johnson. 1991. Memory requirements and local ambiguities of parsing strategies. *Journal of Psycholinguistic Research*, 20(3):233–250.

Aho, Alfred V. and Jeffery D. Ullman. 1972. *The Theory of Parsing, Translation and Compiling; Volume 1: Parsing.* Prentice-Hall, Englewood Cliffs, New Jersey.

Demers, A. 1977. Generalized left-corner parsing. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, 1977 ACM SIGACT/SIGPLAN*, pages 170–182.

Hopcroft, John E. and Jeffrey D. Ullman. 1979. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley.

Kay, Martin. 1996. Limited parsing in linear time. Technical report, Xerox PARC.

Matsumoto, Yuji, Hozumi Tanaka, Hideki Hirakawa, Hideo Miyoshi, and Hideki Yasukawa. 1983. BUP: A bottom-up parser embedded in Prolog. *New Generation Computing*, 1(2):145–158.

Nakazawa, Tsuneko. 1995. Construction of LR parsing tables for grammars using feature-based syntactic categories. In Jennifer Cole, Georgia M. Green, and Jerry L. Morgan, editors, *Linguistics and Computation*, number 52 in CSLI Lecture Notes Series, pages 199–219, Stanford, California. CSLI Publications.

Nijholt, Anton. 1980. *Context-free Grammars: Covers, Normal Forms, and Parsing*. Springer Verlag, Berlin.

Pereira, Fernando C. N. and Rebecca N. Wright. 1991. Finite state approximation of phrase structure grammars. In *The Proceedings of the 29th Annual Meeting of the Association for Computational Linguistics*, pages 246–255.

Pereira, Fernando C. N. and Rebecca N. Wright. 1996. Finite state approximation of phrase structure grammars. Appeared as cmp-lg/9603002.

Pereira, Fernando C.N. and Stuart M. Shieber. 1987. *Prolog and Natural Language Analysis*. Number 10 in CSLI Lecture Notes Series. Chicago University Press, Chicago.

Resnik, Philip. 1992. Left-corner parsing and psychological plausibility. In *The Proceedings of the fifteenth International Conference on Computational Linguistics, COLING-92*, volume 1, pages 191–197.

Reutenauer, Christophe and Marcel-Paul Schutzenberger. 1991. Minimization of rational word functions. *SIAM Journal of Computing*, 20(4):669–685.

Rosenkrantz, Stanley J. and Philip M. Lewis II. 1970. Deterministic left corner parser. In *IEEE Conference Record of the 11th Annual Symposium on Switching and Automata*, pages 139–152.

Shieber, Stuart M. 1985. Using Restriction to extend parsing algorithms for unification-based formalisms. In *Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics*, pages 145–152, Chicago.