Finite-state Approximation of Constraint-based Grammars using Left-corner Grammar Transforms

Mark Johnson*
Cognitive and Linguistic Sciences, Box 1978
Brown University
Mark_Johnson@Brown.edu

Abstract

This paper describes how to construct a finite-state machine (FSM) approximating a 'unification-based' grammar using a left-corner grammar transform. The approximation is presented as a series of grammar transforms, and is exact for left-linear and right-linear CFGs, and for trees up to a user-specified depth of center-embedding.

1 Introduction

This paper describes a method for approximating 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. For example, the implementation described here translates a DCG version of the example grammar given by Pereira and Wright (1991) directly into a FSM without constructing an approximating CFG.

Left-corner based techniques are natural for this kind of application because (with the simple optimization described below) 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. We provide a simple tree-geometric description of the configurations that cause an increase in a left corner parser's stack depth below.

The rest of this paper is structured as follows. The remainder of this section outlines the "grammar transform" approach, summarizes the top-down parsing algorithm and discusses how finite state approximations of top-down parsers can be constructed. The fact that this approximation is not exact for left linear grammars (which define finite-state languages) motivates a finite-state approximation based on the left-corner parsing algorithm (which is presented as a grammar transform in section 2). In its standard form the approximation based on the left-corner parsing algorithm suffers from the complementary problem to the top-down approximation: it is not exact for right-linear grammars, but the "optimized" variants presented in section 3 overcome this deficiency, resulting in finite-state CFG approximations which are exact for left-linear and right-linear grammars. Section 4 discusses how these techniques can be combined in an implementation.

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.

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 t in Figure 1 will be used as an example throughout this paper.

* This research was supported by NSF grant SBR526978. I began this research while I was on sabbatical at the Xerox Research Centre in Grenoble, France. I would like to thank them and my colleages at Brown for their support.

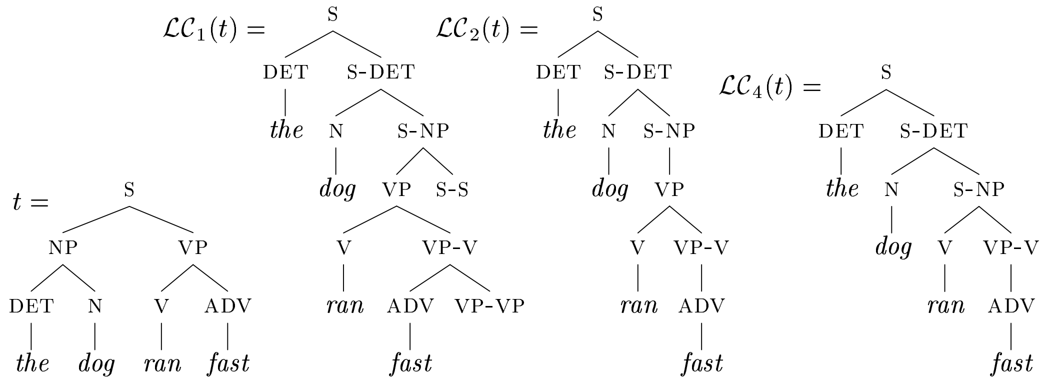


Figure 1: The analysis tree t used as a running example below, and its left-corner transforms $\mathcal{LC}_i(t)$. Note that the phonological forms are treated here as annotations on the nodes drawn above them, rather than independent nodes. That is, DET (annotated with *the*) is a terminal node.

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 $G = (N, T, P, S)$, a (top-down) *stack state* is a sequence of terminals and nonterminals. Let $Q = (N \cup T)^*$ 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 a terminal or epsilon into a set of states. It is the smallest function δ that satisfies the following conditions:

$\gamma \in \delta(a\gamma, a) : a \in T, \gamma \in (N \cup T)^*$.
$\beta\gamma \in \delta(A\gamma, \epsilon) : A \in N, \gamma \in (N \cup T)^*, A \to \beta \in P$.

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. Extending this top-down parsing algorithm to a 'unification-based' grammar is straight-forward, and described in many textbooks, such as Pereira and Shieber (1987).

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. It is easy to check that a top-down parser requires a stack of depth 3 to construct the tree t depicted in Figure 1.

1.4 Finite-state approximations

We obtain a finite-state approximation to a top-down parser by restricting attention to only a finite number of possible stack states. The system implemented here imposes a stack depth restriction, i.e., the transition function is modified so that there are no transitions to any stack state whose size is larger than some user-specified limit.[1] This restriction ensures 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.

The situation becomes more complicated when we move to 'unification-based' grammars, since there may be an unbounded number of different categories appearing in the accessible stack states. In the system implemented here we used *restriction* (Shieber, 1985) on the stack states to restrict attention to a finite number of distinct stack states for any given stack depth. Since the restriction operation maps a stack state to a more general one, it produces a finite-state approximation which accepts a *superset* of the language generated by the original unification grammar. Thus for general constraint-based grammars the language accepted by our finite-state approximation is not guaranteed to be either a superset or a subset of the language generated by the input grammar.

2 The left-corner transform

While conceptually simple, the top-down parsing algorithm presented in the last section suffers from a number of drawbacks for a finite-state approximation. For example, the number of distinct accessible stack states is unbounded if the grammar is left-recursive, yet left-linear grammars always generate regular languages. This section presents

[1] With the optimized left-corner transforms described below we obtain acceptable approximations with a stack size limit of 5 or less. In many useful cases, including the example grammar provided by Pereira and Wright (1991), this stack bound is never reached and the system reports that the FSA it returns is exact.

the standard left-corner grammar transformation (Rosenkrantz and Lewis II, 1970; Aho and Ullman, 1972); these references should be consulted for proofs of correctness. This transform serves as the basis for the further transforms described in the next section; these transforms have the property that the output grammar induces a finite number of distinct accessible stack states if their input is a left-recursive left-linear grammar.

Given an input grammar G with nonterminals N and terminals T, these transforms $\mathcal{LC}_i$ produce grammars with an enlarged set of nonterminals $N' = N \cup (N \times (N \cup T))$. The new "pair" categories in $N \times (N \cup T)$ are written A–X, where A is a nonterminal of G and X is either a terminal or nonterminal of G. It turns out that if $A \Rightarrow^*_G X\gamma$ then $A\text{–}X \Rightarrow^*_{\mathcal{LC}_1(G)} \gamma$, i.e., a non-terminal A–X in the transformed grammar derives *the difference* between A and X in the original grammar, and the notation is meant to be suggestive of this.

The *left-corner transform* of a CFG $G = (N, T, P, S)$ is a grammar $\mathcal{LC}_1(G) = (N', T, P_1, S)$, where P_1 contains all productions of the form (1.a–1.c). This paper assumes that $N \cap T = \emptyset$, as is standard. To save space we assume that P does not contain any epsilon productions (but it is straightforward to deal with them).

$$A \to a\, A\text{–}a : A \in N, a \in T. \quad \text{(1.a)}$$
$$A\text{–}X \to \beta\, A\text{–}B : A \in N, B \to X\,\beta \in P. \quad \text{(1.b)}$$
$$A\text{–}A \to \epsilon : A \in N. \quad \text{(1.c)}$$

Informally, the productions (1.a) start the left-corner recognition of A by recognizing a terminal a as a possible left-corner of A. The actual left-corner recognition is performed by the productions (1.b), 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 (1.c) terminate the recognition of A when this left-corner construction process has constructed an A.

The left-corner transform preserves the number of parses of a string, so it defines an isomorphism from analysis trees (i.e., parse trees with respect to G) to parse trees with respect to $\mathcal{LC}_1(G)$. If t is a parse tree with respect to G then (abusing notation) $\mathcal{LC}_1(t)$ is the corresponding parse tree with respect to $\mathcal{LC}_1(G)$. Figure 1 shows the effect of this mapping on a simple tree. The transformed tree is considerably more complex: it has double the number of nodes of the original tree. In a top-down parse of the tree $\mathcal{LC}_1(t)$ in Figure 1 the maximum stack depth is 3, which occurs at the recognition of the terminals *ran* and *fast*.

2.1 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 productions). While $\mathcal{LC}_1(G)$ can be pruned using standard algorithms, given the observation about the relationship between the pair non-terminals in $\mathcal{LC}_1(G)$ and non-terminals in G, it is clear that certain productions can be discarded immediately as useless. Define the *left-corner* relation $\lhd \subseteq (N \cup T) \times N$ as follows:

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

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 (1.a–1.c) without affecting the language (strongly) generated to those that only contain pair categories A–X where $X \lhd^* A$.

2.2 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 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). In contrast, extending the techniques described here to complex feature based "unification" grammar is straight-forward.

The main complication is the filter on useless nonterminals and productions just discussed. 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. van Noord (1997) provides a detailed discussion of methods for using such a "left-corner filter" in unification-grammar parsing, and the methods he discusses are used in the implementation described below.

3 Extended left-corner transforms

This section presents some simple extensions to the basic left-corner transform presented above. The 'tail-recursion' optimization permits bounded-stack parsing of both left and right linear constructions. 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 no 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. 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 non-left-corner nonterminal in the analysis tree.

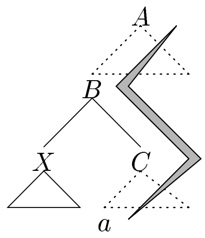
The 'tail recursion' or 'composition' optimization (Abney and Johnson, 1991; Resnik, 1992) permits right-branching structures to be parsed with bounded stack depth. It is the result of epsilon removal applied to the output of $\mathcal{LC}_1$, and can be described in terms of resolution or partial evaluation of the transformed grammar with respect to productions (1.c). In effect, the schema (1.b) is split into two cases, depending on whether or not the rightmost nonterminal A–B is expanded by the epsilon rules produced by schema (1.c). This expansion yields a grammar $\mathcal{LC}_2(G) = (N', T, P_2, S)$, where P_2 contains all productions of the form (2.a–2.c). (In these schemata $A, B \in N$; $a \in T$; $X \in N \cup T$ and $\beta \in (N \cup T)^*$).

$$A \rightarrow a\, A\text{–}a \qquad (2.a)$$
$$A\text{–}X \rightarrow \beta\, A\text{–}B : B \rightarrow X\, \beta \in P. \qquad (2.b)$$
$$A\text{–}X \rightarrow \beta : A \rightarrow X\, \beta \in P. \qquad (2.c)$$

Figure 1 shows the effect of the transform $\mathcal{LC}_2$ on the example tree. The maximum stack depth required for this tree is 2. 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 $\triangleleft^+$ be the transitive closure of the left-corner relation $\triangleleft$ defined above. 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 $X \triangleleft^+ A$. Thus we can restrict the productions in (2.a–2.b) without affecting the language (strongly) generated to just those that only contain pair categories A–X where $X \triangleleft^+ A$.

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 are unary or binary. In this situation, transformation $\mathcal{LC}_2(G)$ can be more explicitly written as $\mathcal{LC}_3(G) = (N', T, P_3, S)$, where P_3 contains all instances of the production schemata (3.a–3.e). (In these schemata, $a \in T$; $A, B \in N$ and $X, Y \in N \cup T$).



A–$X \rightarrow a\, C$–$a\, A$–B (4.f)

Figure 2: The highly distinctive "zig-zag" or "lightning bolt" configuration of nodes in the analysis tree characteristic of the use of production schema (4.f) 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 $\mathcal{LC}_4$.

$$A \rightarrow a\, A\text{–}a. \qquad (3.a)$$
$$A\text{–}X \rightarrow A\text{–}B : B \rightarrow X \in P. \qquad (3.b)$$
$$A\text{–}X \rightarrow \epsilon : A \rightarrow X \in P. \qquad (3.c)$$
$$A\text{–}X \rightarrow Y\, A\text{–}B : B \rightarrow X\, Y \in P. \qquad (3.d)$$
$$A\text{–}X \rightarrow Y : A \rightarrow X\, Y \in P. \qquad (3.e)$$

Productions (3.b–3.c) and (3.d–3.e) correspond to unary and binary productions respectively in the original grammar. Now, note that nonterminals from N only appear in the right hand sides of productions of type (3.d) and (3.e). Moreover, any such nonterminals must be immediately expanded by a production of type (3.a). Thus these non-terminals are eliminable by resolving them with (3.a); the only remaining nonterminal is the start symbol S. This expansion yields a new transform $\mathcal{LC}_4$, where $\mathcal{LC}_4(G) = (\{S\} \cup (N \times (N \cup T)), T, P_4, S)$. P_4, defined in (4.a–4.g), still contains productions of type (3.a), but these only expand the start symbol, as all occurences of nonterminals in N have been resolved away. (In these schemata $a \in T$; $A, B, C, D \in N$ and $X \in N \cup T$).

$$S \rightarrow a\, S\text{–}a. \qquad (4.a)$$
$$A\text{–}X \rightarrow A\text{–}B : B \rightarrow X \in P. \qquad (4.b)$$
$$A\text{–}X \rightarrow \epsilon : A \rightarrow X \in P. \qquad (4.c)$$
$$A\text{–}X \rightarrow a\, A\text{–}B : B \rightarrow X\, a \in P. \qquad (4.d)$$
$$A\text{–}X \rightarrow a : A \rightarrow X\, a \in P. \qquad (4.e)$$
$$A\text{–}X \rightarrow a\, C\text{–}a\, A\text{–}B : B \rightarrow X\, C \in P. \qquad (4.f)$$
$$A\text{–}X \rightarrow a\, C\text{–}a : A \rightarrow X\, C \in P. \qquad (4.g)$$

In the production schemata defining $\mathcal{LC}_4$, (4.a–4.c) are copied directly from (3.a–3.c) respectively. The schemata (4.d–4.e) are obtained by instantiating Y in (3.d–3.e) to a terminal $a \in T$, while the other two schemata (4.f–4.g) are obtained by instantiating Y in (3.d–3.e) with the right hand sides of (3.a). Figure 1 shows the result of applying the transformation $\mathcal{LC}_4$ to the example analysis tree t.

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 on the parser's stack, saving an epsilon transition associated with a stack push and an immediately following stack pop with respect to the standard left-corner algorithm. Productions (4.a) and (4.d–4.g) 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.

Note that the right hand sides of all of these productions except for schema (4.f) are right-linear. Thus instances of this schema are the only productions that can increase the stack size in a top-down parse with $\mathcal{LC}_4(G)$, and the stack depth required to parse an analysis tree is the maximum number of "zig-zag" patterns in the path in the analysis tree from any terminal node to the root. Figure 2 sketches the configuration of nodes in the analysis trees in which instances of schemata (4.f) would be used in a parse using $\mathcal{LC}_4(G)$. This highly distinctive "zig-zag" or "lightning bolt" pattern does not occur at all in the example tree t in Figure 1, so the maximum required stack depth is 2. (Recall that in a traditional top-down parser terminals are pushed onto the stack and popped later, so initialization productions (4.a) cause two symbols to be pushed onto the stack). It follows that this finite state approximation is exact for left-linear and right-linear CFGs. Indeed, analysis trees that consist simply of a left-branching subtree followed by a right-branching subtree, such as the example tree t, are transformed into strictly right-branching trees by $\mathcal{LC}_4$.

4 Implementation

This section provides further details of the finite-state approximator implemented in this research. The approximator is written in Sicstus Prolog. It takes a user-specifier Definite Clause Grammar G (without Prolog annotations) as input, which it binarizes and then applies transform $\mathcal{LC}_4$ to.

The implementation annotates each transition with the production it corresponds to (represented as a pair of a $\mathcal{LC}_4$ schema number and a production number from G), so the finite-state approximation actually defines a transducer which transduces a lexical input to a sequence of productions which specify a parse of that input with respect to $\mathcal{LC}_4(G)$. A following program inverts the tree transform $\mathcal{LC}_4$, returning a corresponding parse tree with respect to G. This parse tree can be checked by performing complete unifications with respect to the original grammar productions if so desired. Thus the finite-state approximation provides an efficient way of determining if an analysis of a given input string with respect to a unification grammar G exists, and if so, it can be used to suggest such analyses.

5 Conclusion

This paper surveyed the issues arising in the construction of finite-state approximations 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. It derived the various forms of the left-corner parsing algorithms in terms of grammar transformations from the original left-corner grammar transform.

References

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

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

Yuji Matsumoto, 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.

Tsuneko Nakazawa. 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.

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

Fernando C. N. Pereira 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.

Philip Resnik. 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.

Stanley J. Rosenkrantz 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.

Stuart M. Shieber. 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.

Gertjan van Noord. 1997. An efficient implementation of the head-corner parser. *Computational Linguistics*, 23(3):425–456.