Finite-State Approximation of Phrase-Structure Grammars

Fernando C. N. Pereira
Rebecca N. Wright
AT&T Research
600 Mountain Ave., Murray Hill, NJ 07974

July 24, 2011

Abstract

Phrase-structure grammars are effective models for important syntactic and semantic aspects of natural languages, but can be computationally too demanding for use as language models in real-time speech recognition. Therefore, finite-state models are used instead, even though they lack expressive power. To reconcile those two alternatives, we designed an algorithm to compute finite-state approximations of context-free grammars and context-free-equivalent augmented phrase-structure grammars. The approximation is exact for certain context-free grammars generating regular languages, including all left-linear and right-linear context-free grammars. The algorithm has been used to build finite-state language models for limited-domain speech recognition tasks.

1 Motivation

Grammars for spoken language systems are subject to the conflicting requirements of language modeling for recognition and of language analysis for sentence interpretation. For efficiency reasons, most current recognition systems rely on finite-state language models. These models, however, are inadequate for language interpretation, since they cannot express the relevant syntactic and semantic regularities. Augmented phrase structure grammar (APSG) formalisms, such as unification grammars [15], can express many of those regularities, but they are computationally less suitable for language modeling because of the inherent cost of computing state transitions in APSG parsers.

The above conflict can be alleviated by using separate grammars for language modeling and language interpretation. Ideally, the recognition grammar should not reject sentences acceptable by the interpretation grammar and as far as possible it should enforce the constraints built into the interpretation grammar.

However, if the two grammars are built independently, those goals are difficult to maintain. For that reason, we have developed a method for approximating APSGs with finite-state acceptors (FSAs). Since such an approximation is intended to serve as language model for a speech-recognition front-end to the real parser, we require it to be *sound* in the sense that the approximation accepts all strings in the language defined by the APSG. Without qualification, the term "approximation" will always mean here "sound approximation."

If no further requirements were placed on the closeness of the approximation, the trivial algorithm that assigns to any APSG over the alphabet Σ the regular language Σ^* would do, but of course this language model is useless. One possible criterion for "goodness" of approximation arises from the observation that many interesting phrase-structure grammars have substantial parts that accept regular languages. That does not mean that grammar rules are in the standard forms for defining regular languages (left-linear or right-linear), because syntactic and semantic considerations often require that strings in a regular set be assigned structural descriptions not definable by left- or right-linear rules. An ideal criterion would thus be that if a grammar generates a regular language, the approximation algorithm yields an acceptor for that regular language. In other words, one would like the algorithm to be *exact* for all APSGs yielding regular languages. However, we will see later that no such general algorithm is possible, that is, any approximation algorithm will be inexact for some APSGs yielding regular languages. Nevertheless, we will show that our method is exact for left-linear and right-linear grammars, and for certain useful combinations thereof.

2 The Approximation Method

Our approximation method applies to any context-free grammar (CFG), or any constraint-based grammar [15, 6] that can be fully expanded into a context-free grammar.[1] The resulting FSA accepts all the sentences accepted by the input grammar, and possibly some non-sentences as well.

The implementation takes as input unification grammars of a restricted form ensuring that each feature ranges over a finite set. Clearly, such grammars can only generate context-free languages, since an equivalent CFG can be obtained by instantiating features in rules in all possible ways.

2.1 The Basic Algorithm

The heart of our approximation method is an algorithm to convert the LR(0) *characteristic machine* $\mathcal{M}(G)$ [2, 3] of a CFG G into an FSA for a superset of the language $L(G)$ defined by G. The characteristic machine for a CFG G is an

[1] Unification grammars not in this class must first be weakened using techniques such as Shieber's restrictor [16].

FSA for the *viable prefixes* of G, which are just the possible stacks built by the standard shift-reduce recognizer for G when recognizing strings in $L(G)$.

This is not the place to review the characteristic machine construction in detail. However, to explain the approximation algorithm we will need to recall the main aspects of the construction. The states of $\mathcal{M}(G)$ are sets of *dotted rules* $A \to \alpha \cdot \beta$ where $A \to \alpha\beta$ is some rule of G. $\mathcal{M}(G)$ is the determinization by the standard subset construction [2] of the FSA defined as follows:

- The initial state is the dotted rule $S' \to \cdot S$ where S is the start symbol of G and S' is a new auxiliary start symbol.
- The final state is $S' \to S\cdot$.
- The other states are all the possible dotted rules of G.
- There is a transition labeled X, where X is a terminal or nonterminal symbol, from $A \to \alpha \cdot X\beta$ to $A \to \alpha X \cdot \beta$.
- There is an ϵ-transition from $A \to \alpha \cdot B\beta$ to $B \to \cdot\gamma$, where B is a nonterminal symbol and $B \to \gamma$ is a rule in G.

$\mathcal{M}(G)$ can be seen as the finite state control for a nondeterministic shift-reduce pushdown recognizer $\mathcal{R}(G)$ for G. A state transition labeled by a terminal symbol x from state s to state s' licenses a *shift* move, pushing onto the stack of the recognizer the pair $\langle s, x\rangle$. Arrival at a state containing a *completed dotted rule* $A \to \alpha\cdot$ licenses a *reduction* move. This pops from the stack $|\alpha|$ elements, checking that the symbols in the pairs match the corresponding elements of α, takes the transition labeled by A from the state s in the last pair popped, and pushes $\langle s, A\rangle$ onto the stack. (Full definitions of those concepts are given in Section 3.)

The basic ingredient of our approximation algorithm is the *flattening* of a shift-reduce recognizer for a grammar G into an FSA by eliminating the stack and turning reduce moves into ϵ-transitions. It will be seen below that flattening $\mathcal{R}(G)$ directly leads to poor approximations in many interesting cases. Instead, $\mathcal{M}(G)$ must first be *unfolded* into a larger machine whose states carry information about the possible shift-reduce stacks of $\mathcal{R}(G)$. The quality of the approximation is crucially influenced by how much stack information is encoded in the states of the unfolded machine: too little leads to coarse approximations, while too much leads to redundant automata needing very expensive optimization.

The algorithm is best understood with a simple example. Consider the left-linear grammar G_1

$$\begin{array}{l} S \to Ab \\ A \to Aa \mid \epsilon \end{array} \quad .$$

$\mathcal{M}(G_1)$ is shown on Figure 1. Unfolding is not required for this simple example,

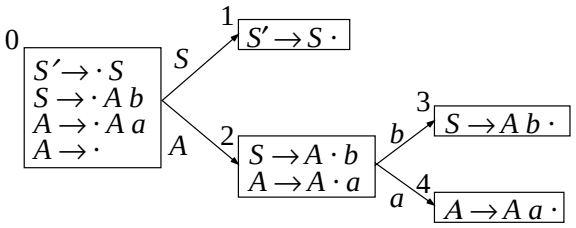

Figure 1: Characteristic Machine for G_1

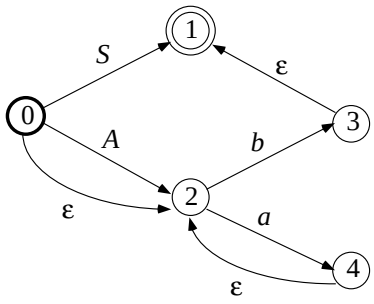

Figure 2: Flattened Canonical Acceptor for $L(G_1)$

so the approximating FSA is obtained from $\mathcal{M}(G_1)$ by the flattening method outlined above. The reducing states in $\mathcal{M}(G_1)$, those containing completed dotted rules, are states 0, 3 and 4. For instance, the reduction at state 3 would lead to a $\mathcal{R}(G_1)$ transition on nonterminal S to state 1, from the state that activated the rule being reduced. Thus the corresponding ϵ-transition goes from state 3 to state 1. Adding all the transitions that arise in this way we obtain the FSA in Figure 2. From this point on, the arcs labeled with nonterminals can be deleted, and after simplification we obtain the deterministic finite automaton (DFA) in Figure 3, which is the minimal DFA for $L(G_1)$.

If flattening were always applied to the LR(0) characteristic machine as in the example above, even simple grammars defining regular languages might be

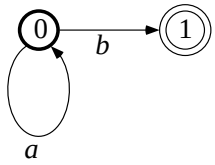

Figure 3: Minimal Acceptor for $L(G_1)$

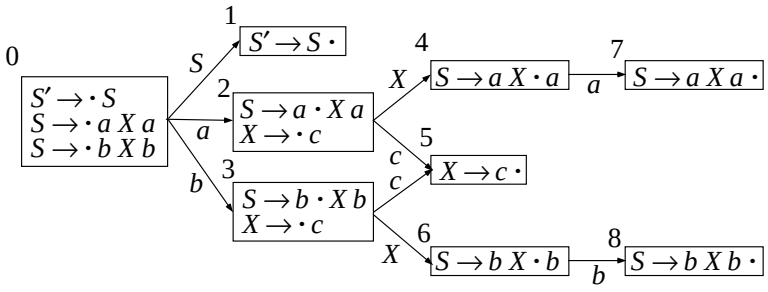

Figure 4: Minimal Acceptor for $L(G_2)$

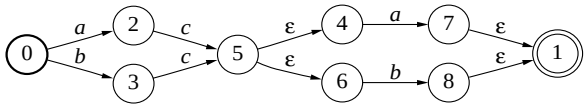

Figure 5: Flattened Acceptor for $L(G_2)$

inexactly approximated by the algorithm. The reason for this is that in general the reduction at a given reducing state in the characteristic machine transfers to different states depending on stack contents. In other words, the reducing state might be reached by different routes which use the result of the reduction in different ways. The following grammar G_2

$$S \rightarrow aXa \mid bXb$$
$$X \rightarrow c$$

accepts just the two strings *aca* and *bcb*, and has the characteristic machine $\mathcal{M}(G_2)$ shown in Figure 4. However, the corresponding flattened acceptor shown in Figure 5 also accepts *acb* and *bca*, because the ϵ-transitions leaving state 5 do not distinguish between the different ways of reaching that state encoded in the stack of $\mathcal{R}(G_2)$.

Our solution for the problem just described is to unfold each state of the characteristic machine into a set of states corresponding to different stacks at that state, and flattening the corresponding recognizer rather than the original one. Figure 6 shows the resulting acceptor for $L(G_2)$, now exact, after determinization and minimization.

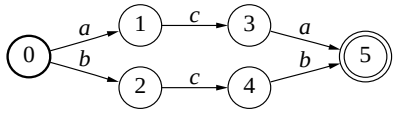

Figure 6: Exact Acceptor for $L(G_2)$

In general the set of possible stacks at a state is infinite. Therefore, it is necessary to do the unfolding not with respect to stacks, but with respect to a finite partition of the set of stacks possible at the state, induced by an appropriate equivalence relation. The relation we use currently makes two stacks equivalent if they can be made identical by *collapsing loops*, that is, removing in a canonical way portions of stack pushed between two arrivals at the same state in the finite-state control of the shift-reduce recognizer, as described more formally and the end of section 3.1. The purpose of collapsing a loop is to "forget" a stack segment that may be arbitrarily repeated.[2] Each equivalence class is uniquely defined by the shortest stack in the class, and the classes can be constructed without having to consider all the (infinitely) many possible stacks.

2.2 Grammar Decomposition

Finite-state approximations computed by the basic algorithm may be extremely large, and their determinization, which is required by minimization [1], can be computationally infeasible. These problems can be alleviated by first decomposing the grammar to be approximated into *subgrammars* and approximating the subgrammars separately before combining the results.

Each subgrammar in the decomposition of a grammar G corresponds to a set of nonterminals that are involved, directly or indirectly, in each other's definition, together with their defining rules. More precisely, we define a directed graph $\mathrm{conn}(G)$ whose nodes are G's nonterminal symbols, and which has an arc from X to Y whenever Y appears in the right-hand side of one of G's rules and X in the left-hand side. Each strongly connected component of this graph [1] corresponds to a set of mutually recursive nonterminals.

Each nonterminal X of G is in exactly one strongly connected component $\mathrm{comp}(X)$ of $\mathrm{conn}(G)$. Let $\mathrm{prod}(X)$ be the set of G rules with left-hand sides in $\mathrm{comp}(X)$, and $\mathrm{rhs}(X)$ be the set of right-hand side nonterminals of $\mathrm{comp}(X)$. Then the *defining subgrammar* $\mathrm{def}(X)$ of X is the grammar with start symbol X, nonterminal symbols $\mathrm{comp}(X)$, terminal symbols $\Sigma \cup (\mathrm{rhs}(X) - \mathrm{comp}(X))$ and rules $\mathrm{prod}(X)$. In other words, the nonterminal symbols not in $\mathrm{comp}(X)$ are treated as *pseudoterminal* symbols in $\mathrm{def}(X)$.

Each grammar $\mathrm{def}(X)$ can be approximated with our basic algorithm, yielding an FSA $\mathrm{aut}(X)$. To see how to merge together each of these subgrammar approximations to yield an approximation for the whole of G, we observe first that the notion of strongly connected component allows us to take each $\mathrm{aut}(X)$ as a node in a directed acyclic graph with an arc from $\mathrm{aut}(X)$ to $\mathrm{aut}(X')$ whenever X' is a pseudoterminal of $\mathrm{def}(X)$. We can then replace each occurrence of a pseudoterminal X' by its definition. More precisely, each transition labeled by a pseudoterminal X' from some state s to state s' in $\mathrm{aut}(X)$ is replaced

[2] Since possible stacks can be shown to form a regular language, loop collapsing has a direct connection to the pumping lemma for regular languages.

by ϵ-transitions from s to the initial state of a separate copy of aut(X') and ϵ-transitions from the final states of the copy of aut(X') to s'. This process is then recursively applied to each of the newly created instances of aut(X') for each pseudoterminal in def(X). Since the subautomata dependency graph is acyclic, the replacement process must terminate.

3 Formal Properties

We will show now that the basic approximation algorithm described informally in the previous section is sound for arbitrary CFGs and is exact for left-linear and right-linear CFGs. From those results, it will be easy to see that the extended algorithm based on decomposing the input grammar into strongly connected components is also sound, and is exact for CFGs in which every strongly connected component is either left linear or right linear.

In what follows, G is a fixed CFG with terminal vocabulary Σ, nonterminal vocabulary N and start symbol S, and $V = \Sigma \cup N$.

3.1 Soundness

Let $\mathcal{M}$ be the characteristic machine for G, with state set Q, start state s_0, final states F, and transition function $\delta : S \times V \to S$. As usual, transition functions such as δ are extended from input symbols to input strings by defining $\delta(s, \epsilon) = s$ and $\delta(s, \alpha\beta) = \delta(\delta(s, \alpha), \beta)$. The shift-reduce recognizer $\mathcal{R}$ associated to $\mathcal{M}$ has the same states, start state and final states as $\mathcal{M}$. Its *configurations* are triples $\langle s, \sigma, w\rangle$ of a state, a stack and an input string. The stack is a sequence of pairs $\langle s, X\rangle$ of a state and a symbol. The transitions of the shift-reduce recognizer are given as follows:

Shift: $\langle s, \sigma, xw\rangle \vdash \langle s', \sigma\langle s, x\rangle, w\rangle$ if $\delta(s, x) = s'$

Reduce: $\langle s, \sigma\tau, w\rangle \vdash \langle \delta(s', A), \sigma\langle s', A\rangle, w\rangle$ if either (1) $A \to \cdot$ is a completed dotted rule in s, $s' = s$ and τ is empty, or (2) $A \to X_1 \ldots X_n\cdot$ is a completed dotted rule in s, $\tau = \langle s_1, X_1\rangle \cdots \langle s_n, X_n\rangle$ and $s' = s_1$.

The *initial* configurations of $\mathcal{R}$ are $\langle s_0, \epsilon, w\rangle$ for some input string w, and the *final* configurations are $\langle s, \langle s_0, S\rangle, \epsilon\rangle$ for some state $s \in F$. A *derivation* of a string w is a sequence of configurations $c_0, \ldots, c_m$ such that $c_0 = \langle s_0, \epsilon, w\rangle$, c_m is final, and $c_{i-1} \vdash c_i$ for $1 \le i \le n$.

Let s be a state. We define the set Stacks(s) to contain every sequence $\langle q_0, X_0\rangle \ldots \langle q_k, X_k\rangle$ such that $q_0 = s_0$ and $q_i = \delta(q_{i-1}, X_{i-1}), 1 \le i \le k$ and $s = \delta(q_k, X_k)$. In addition, Stacks(s_0) contains the empty sequence ϵ. By construction, it is clear that if $\langle s, \sigma, w\rangle$ is reachable from an initial configuration in $\mathcal{R}$, then $\sigma \in$ Stacks(s).

A *stack congruence* on $\mathcal{R}$ is a family of equivalence relations $\equiv_s$ on Stacks(s) for each state $s \in \mathcal{S}$ such that if $\sigma \equiv_s \sigma'$ and $\delta(s, X) = s'$ then $\sigma\langle s, X\rangle \equiv_{s'}$

$\sigma\langle s, X\rangle$. A stack congruence $\equiv$ partitions each set Stacks(s) into equivalence classes $[\sigma]_s$ of the stacks in Stacks(s) equivalent to σ under $\equiv_s$.

Each stack congruence $\equiv$ on $\mathcal{R}$ induces a corresponding *unfolded recognizer* $\mathcal{R}_{\equiv}$. The states of the unfolded recognizer are pairs $\langle s, [\sigma]_s\rangle$, notated more concisely as $[\sigma]^s$, of a state and stack equivalence class at that state. The initial state is $[\epsilon]^{s_0}$, and the final states are all $[\sigma]^s$ with $s \in F$ and $\sigma \in$ Stacks(s). The transition function $\delta_{\equiv}$ of the unfolded recognizer is defined by

$$\delta_{\equiv}([\sigma]^s, X) = [\sigma\langle s, X\rangle]^{\delta(s,X)} \qquad .$$

That this is well-defined follows immediately from the definition of stack congruence.

The definitions of dotted rules in states, configurations, shift and reduce transitions given above carry over immediately to unfolded recognizers. Also, the characteristic recognizer can also be seen as an unfolded recognizer for the trivial coarsest congruence.

Unfolding a characteristic recognizer does not change the language accepted:

Proposition 1 *Let G be a CFG, $\mathcal{R}$ its characteristic recognizer with transition function δ, and $\equiv$ a stack congruence on $\mathcal{R}$. Then $\mathcal{R}_{\equiv}$ and $\mathcal{R}$ are equivalent.*

Proof: We show first that any string w accepted by $\mathcal{R}_{\equiv}$ is accepted by $\mathcal{R}$. Let d be configuration of $\mathcal{R}_{\equiv}$. By construction, $d = \langle [\rho]^s, \sigma, u\rangle$, with $\sigma = \langle\langle q_0, e_0\rangle, X_0\rangle \cdots \langle\langle q_k, e_k\rangle, X_k\rangle$ for appropriate stack equivalence classes e_i. We define $\hat{d} = \langle s, \hat{\sigma}, u\rangle$, with $\hat{\sigma} = \langle q_0, X_0\rangle \cdots \langle q_k, X_k\rangle$. If $d_0, \ldots, d_m$ is a derivation of w in $\mathcal{R}_{\equiv}$, it is easy to verify that $\hat{d}_0, \ldots, \hat{d}_m$ is a derivation of w in $\mathcal{R}$.

Conversely, let $w \in L(G)$, and let $c_0, \ldots, c_m$ be a derivation of w in $\mathcal{R}$, with $c_i = \langle s_i, \sigma_i, u_i\rangle$. We define $\bar{c}_i = \langle [\sigma_i]^{s_i}, \bar{\sigma}_i, u_i\rangle$, where $\bar{\epsilon} = \epsilon$ and $\overline{\sigma\langle s, X\rangle} = \bar{\sigma}\langle [\sigma]^s, X\rangle$.

If $c_{i-1} \vdash c_i$ is a shift move, then $u_{i-1} = xu_i$ and $\delta(s_{i-1}, x) = s_i$. Therefore,

$$\begin{aligned} \delta_{\equiv}([\sigma_{i-1}]^{s_{i-1}}, x) &= [\sigma_{i-1}\langle s_{i-1}, x\rangle]^{\delta(s_{i-1},x)} \\ &= [\sigma_i]^{s_i} \qquad . \end{aligned}$$

Furthermore,

$$\bar{\sigma}_i = \overline{\sigma_{i-1}\langle s_{i-1}, x\rangle} = \bar{\sigma}_{i-1}\langle [\sigma_{i-1}]^{s_{i-1}}, x\rangle \qquad .$$

Thus we have

$$\begin{aligned} \bar{c}_{i-1} &= \langle [\sigma_{i-1}]^{s_{i-1}}, \bar{\sigma}_{i-1}, xu_i\rangle \\ \bar{c}_i &= \langle [\sigma_i]^{s_i}, \bar{\sigma}_{i-1}\langle [\sigma_{i-1}]^{s_{i-1}}, x\rangle, u_i\rangle \end{aligned}$$

with $\delta_{\equiv}([\sigma_{i-1}]^{s_{i-1}}, x) = [\sigma_i]^{s_i}$. Thus, by definition of shift move, $\bar{c}_{i-1} \vdash \bar{c}_i$ in $\mathcal{R}_{\equiv}$.

Assume now that $c_{i-1} \vdash c_i$ is a reduce move in $\mathcal{R}$. Then $u_i = u_{i-1}$ and we have a state s in $\mathcal{R}$, a symbol $A \in N$, a stack σ and a sequence τ of state-symbol pairs such that

$$\begin{aligned} s_i &= \delta(s, A) \\ \sigma_{i-1} &= \sigma\tau \\ \sigma_i &= \sigma\langle s, A\rangle \end{aligned}$$

and either

(a) $A \to \cdot$ is in s_{i-1}, $s = s_{i-1}$ and $\tau = \epsilon$, or

(b) $A \to X_1 \cdots X_n\cdot$ is in s_{i-1} , $\tau = \langle q_1, X_1\rangle \cdots \langle q_n, X_n\rangle$ and $s = q_1$.

Let $\bar{s} = [\sigma]^s$. Then

$$\begin{aligned} \delta_{\equiv}(\bar{s}, A) &= [\sigma\langle s, A\rangle]^{\delta(s,A)} \\ &= [\sigma_i]^{s_i} \end{aligned}$$

We now define a pair sequence $\bar{\tau}$ to play the same role in $\mathcal{R}_{\equiv}$ as τ does in $\mathcal{R}$. In case (a) above, $\bar{\tau} = \epsilon$. Otherwise, let $\tau_1 = \epsilon$ and $\tau_i = \tau_{i-1}\langle q_{i-1}, X_{i-1}\rangle$ for $2 \le i \le n$, and define $\bar{\tau}$ by

$$\bar{\tau} = \langle [\sigma]^{q_1}, X_1\rangle \cdots \langle [\sigma\tau_i]^{q_i}, X_i\rangle \cdots \langle [\sigma\tau_n]^{q_n}, X_n\rangle$$

Then

$$\begin{aligned} \bar{\sigma}_{i-1} &= \overline{\overline{\sigma\tau}} \\ &= \overline{\sigma\langle q_1, X_1\rangle \cdots \langle q_{n-1}, X_{n-1}\rangle}\langle [\sigma\tau_n]^{q_n}, X_n\rangle \\ &= \overline{\sigma\langle q_1, X_1\rangle \cdots \langle q_{i-1}, X_{i-1}\rangle}\langle [\sigma\tau_i]^{q_i}, X_i\rangle \cdots \langle [\sigma\tau_n]^{q_n}, X_n\rangle \\ &= \bar{\sigma}\bar{\tau} \\ \bar{\sigma}_i &= \overline{\sigma\langle s, A\rangle} \\ &= \bar{\sigma}\langle [\sigma]^s, A\rangle \\ &= \bar{\sigma}\langle \bar{s}, A\rangle \qquad . \end{aligned}$$

Thus

$$\begin{aligned} \bar{c}_i &= \langle \delta_{\equiv}(\bar{s}, A), \bar{\sigma}\langle \bar{s}, A\rangle, u_i\rangle \\ \bar{c}_{i-1} &= \langle [\sigma_{i-1}]^{s_{i-1}}, \bar{\sigma}\bar{\tau}, u_{i-1}\rangle \end{aligned}$$

which by construction of $\bar{\tau}$ immediately entails that $\bar{c}_{i-1} \vdash \bar{c}_i$ is a reduce move in $\mathcal{R}_{\equiv}$. □

For any unfolded state p, let $\mathrm{Pop}(p)$ be the set of states reachable from p by a reduce transition. More precisely, $\mathrm{Pop}(p)$ contains any state p' such that there is a completed dotted rule $A \to \alpha\cdot$ in p and a state p'' containing $A \to \cdot\alpha$ such that $\delta_{\equiv}(p'', \alpha) = p$ and $\delta_{\equiv}(p'', A) = p'$. Then the *flattening* $\mathcal{F}_{\equiv}$ of $\mathcal{R}_{\equiv}$ is an NFA with the same state set, start state and final states as $\mathcal{R}_{\equiv}$ and nondeterministic transition function $\phi_{\equiv}$ defined as follows:

- If $\delta_{\equiv}(p, x) = p'$ for some $x \in \Sigma$, then $p' \in \phi_{\equiv}(p, x)$
- If $p' \in \mathrm{Pop}(p)$ then $p' \in \phi_{\equiv}(p, \epsilon)$.

Let $c_0, \ldots, c_m$ be a derivation of string w in $\mathcal{R}$, and put $c_i = \langle q_i, \sigma_i, w_i \rangle$, and $p_i = [\sigma_i]^{p_i}$. By construction, if $c_{i-1} \vdash c_i$ is a shift move on x ($w_{i-1} = xw_i$), then $\delta_{\equiv}(p_{i-1}, x) = p_i$, and thus $p_i \in \phi_{\equiv}(p_{i-1}, x)$. Alternatively, assume the transition is a reduce move associated to the completed dotted rule $A \to \alpha\cdot$. We consider first the case $\alpha \neq \epsilon$. Put $\alpha = X_1 \ldots X_n$. By definition of reduce move, there is a sequence of states $r_1, \ldots, r_n$ and a stack σ such that $\sigma_{i-1} = \sigma\langle r_1, X_1\rangle \ldots \langle r_n, X_n\rangle$, $\sigma_i = \sigma\langle r_1, A\rangle$, r_1 contains $A \to \cdot\alpha$, $\delta(r_1, A) = q_i$, and $\delta(r_j, X_j) = r_{j+1}$ for $1 \leq j < n$. By definition of stack congruence, we will then have

$$\delta_{\equiv}([\sigma\tau_j]^{r_j}, X_j) = [\sigma\tau_{j+1}]^{r_{j+1}}$$

where $\tau_1 = \epsilon$ and $\tau_j = \langle r_1, X_1\rangle \ldots \langle r_{j-1}, X_{j-1}\rangle$ for $j > 1$. Furthermore, again by definition of stack congruence we have $\delta_{\equiv}([\sigma]^{r_1}, A) = p_i$. Therefore, $p_i \in \mathrm{Pop}(p_{i-1})$ and thus $p_i \in \phi_{\equiv}(p_{i-1}, \epsilon)$. A similar but simpler argument allows us to reach the same conclusion for the case $\alpha = \epsilon$. Finally, the definition of final state for $\mathcal{R}_{\equiv}$ and $\mathcal{F}_{\equiv}$ makes p_m a final state. Therefore the sequence $p_0, \ldots, p_m$ is an accepting path for w in $\mathcal{F}_{\equiv}$. We have thus proved

Proposition 2 *For any CFG G and stack congruence $\equiv$ on the canonical* LR(0) *shift-reduce recognizer $\mathcal{R}(G)$ of G, $L(G) \subseteq L(\mathcal{F}_{\equiv}(G))$, where $\mathcal{F}_{\equiv}(G)$ is the flattening of $\mathcal{R}(G)_{\equiv}$.*

To complete the proof of soundness for the basic algorithm, we must show that the stack collapsing equivalence described informally earlier is indeed a stack congruence. A stack τ is a *loop* if $\tau = \langle s_1, X_1\rangle \ldots \langle s_k, X_k\rangle$ and $\delta(s_k, X_k) = s_1$. A stack τ is a *minimal* loop if no prefix of τ is a loop. A stack that contains a loop is *collapsible*. A collapsible stack σ *immediately collapses* to a stack σ' if $\sigma = \rho\tau\upsilon$, $\sigma' = \rho\upsilon$, τ is a minimal loop and there is no other decomposition $\sigma = \rho'\tau'\upsilon'$ such that ρ' is a proper prefix of ρ and τ' is a loop. By these definitions, a collapsible stack σ immediately collapses to a unique stack $C(\sigma)$. A stack σ *collapses* to σ' if $\sigma' = C^n(\sigma)$. Two stacks are equivalent if they can be collapsed to the same uncollapsible stack. This equivalence relation is closed under suffixing, therefore it is a stack congruence. Each equivalence class has a canonical representative, the unique uncollapsible stack in it, and clearly there are finitely many uncollapsible stacks.

We compute the possible uncollapsible stacks associated with states as follows. To start with, the empty stack is associated with the initial state. Inductively, if stack σ has been associated with state s and $\delta(s, X) = s'$, we associate $\sigma' = \sigma\langle s, X\rangle$ with s' unless σ' is already associated with s' or s' occurs in σ, in which case a suffix of σ' would be a loop and σ' thus collapsible. Since there are finitely many uncollapsible stacks, the above computation is must terminate.

When the grammar G is first decomposed into strongly connected components $\mathrm{def}(X)$, each approximated by $\mathrm{aut}(X)$, the soundness of the overall construction follows easily by induction on the partial order of strongly connected components and by the soundness of the approximation of $\mathrm{def}(X)$ by $\mathrm{aut}(X)$, which guarantees that each G sentential form over $\Sigma \cup (\mathrm{rhs}(X) - \mathrm{comp}(X)$ accepted by $\mathrm{def}(X)$ is accepted by $\mathrm{aut}(X)$.

3.2 Exactness

While it is difficult to decide what should be meant by a "good" approximation, we observed earlier that a desirable feature of an approximation algorithm is that it be exact for a wide class of CFGs generating regular languages. We show in this section that our algorithm is exact for both left-linear and right-linear CFGs, and as a consequence for CFGs that can be decomposed into independent left and right linear components. On the other hand, a theorem of Ullian's [17] shows that there can be no partial algorithm mapping CFGs to FSAs that terminates on every CFG yielding a regular language L with an FSA accepting exactly L.

The proofs that follow rely on the following basic definitions and facts about the LR(0) construction. Each LR(0) state s is the *closure* of a set of a certain set of dotted rules, its *core*. The closure $[R]$ of a set R of dotted rules is the smallest set of dotted rules containing R that contains $B \to \cdot\gamma$ whenever it contains $A \to \alpha \cdot B\beta$ and $B \to \gamma$ is in G. The core of the initial state s_0 contains just the dotted rule $S' \to \cdot S$. For any other state s, there is a state s' and a symbol X such that s is the closure of the set core consisting of all dotted rules $A \to \alpha X \cdot \beta$ where $A \to \alpha \cdot X\beta$ belongs to s'.

3.2.1 Left-Linear Grammars

A CFG G is left-linear if each rule in G is of the form $A \to B\beta$ or $A \to \beta$, where $A, B \in N$ and $\beta \in \Sigma^*$.

Proposition 3 *Let G be a left-linear CFG, and let $\mathcal{F}$ be the FSA derived from G by the basic approximation algorithm. Then $L(G) = L(\mathcal{F})$.*

Proof: By Proposition 2, $L(G) \subseteq L(\mathcal{F})$. Thus we need only show $L(\mathcal{F}) \subseteq L(G)$.

Since $\mathcal{M}(G)$ is deterministic, for each $\alpha \in V^*$ there is at most one state s in $\mathcal{M}(G)$ reachable from s_0 by a path labeled with α. If s exists, we define $\bar{\alpha} = s$. Conversely, each state s can be identified with a string $\hat{s} \in V^*$ such that every dotted rule in s is of the form $A \to \hat{s} \cdot \alpha$ for some $A \in N$ and $\alpha \in V^*$. Clearly, this is true for $s_0 = [S' \to \cdot S]$, with $\hat{s}_0 = \epsilon$. The core $\dot{s}$ of any other state s will by construction contain only dotted rules of the form $A \to \alpha \cdot \beta$ with $\alpha \neq \epsilon$. Since G is left linear, β must be a terminal string, thus $s = \dot{s}$. Therefore every dotted rule $A \to \alpha \cdot \beta$ in s results from dotted rule $A \to \cdot\alpha\beta$ in s_0 by a unique

transition path labeled by α (since $\mathcal{M}(G)$ is deterministic). This means that if $A \to \alpha \cdot \beta$ and $A' \to \alpha' \cdot \beta'$ are in s, it must be the case that $\alpha = \alpha'$.

To go from the characteristic machine $\mathcal{M}(G)$ to the FSA $\mathcal{F}$, the algorithm first unfolds $\mathcal{M}(G)$ using the stack congruence relation, and then flattens the unfolded machine by replacing reduce moves with ϵ-transitions. However, the above argument shows that the only stack possible at a state s is the one corresponding to the transitions given by $\hat{s}$, and thus there is a single stack congruence state at each state. Therefore, $\mathcal{M}(G)$ will only be flattened, not unfolded. Hence the transition function ϕ for the resulting flattened automaton $\mathcal{F}$ is defined as follows, where $\alpha \in N\Sigma^* \cup \Sigma^*, a \in \Sigma$, and $A \in N$:

(a) $\phi(\bar{\alpha}, a) = \{\overline{\alpha a}\}$

(b) $\phi(\bar{\alpha}, \epsilon) = \{\bar{A} \mid A \to \alpha \in G\}$

The start state of $\mathcal{F}$ is $\bar{\epsilon}$. The only final state is $\bar{S}$.

We will establish the connection between $\mathcal{F}$ derivations and G derivations. We claim that if there is a path from $\bar{\alpha}$ to $\bar{S}$ labeled by w then either there is a rule $A \to \alpha$ such that $w = xy$ and $S \stackrel{*}{\Rightarrow} Ay \Rightarrow \alpha xy$, or $\alpha = S$ and $w = \epsilon$. The claim is proved by induction on $|w|$.

For the base case, suppose $|w| = 0$ and there is a path from $\bar{\alpha}$ to $\bar{S}$ labeled by w. Then $w = \epsilon$, and either $\alpha = S$, or there is a path of ϵ-transitions from $\bar{\alpha}$ to $\bar{S}$. In the latter case, $S \stackrel{*}{\Rightarrow} A \Rightarrow \epsilon$ for some $A \in N$ and rule $A \to \epsilon$, and thus the claim holds.

Now, assume that the claim is true for all $|w| < k$, and suppose there is a path from $\bar{\alpha}$ to $\bar{S}$ labeled w', for some $|w'| = k$. Then $w' = aw$ for some terminal a and $|w| < k$, and there is a path from $\overline{\alpha a}$ to $\bar{S}$ labeled by w. By the induction hypothesis, $S \stackrel{*}{\Rightarrow} Ay \Rightarrow \alpha a x' y$, where $A \to \alpha a x'$ is a rule and $x'y = w$ (since $\alpha a \neq S$). Letting $x = ax'$, we have the desired result.

If $w \in L(\mathcal{F})$, then there is a path from $\bar{\epsilon}$ to $\bar{S}$ labeled by w. Thus, by the claim just proved, $S \stackrel{*}{\Rightarrow} Ay \Rightarrow xy$, where $A \to x$ is a rule and $w = xy$ (since $\epsilon \neq S$). Therefore, $S \stackrel{*}{\Rightarrow} w$, so $w \in L(G)$, as desired. □

3.2.2 Right-Linear Grammars

A CFG G is right linear if each rule in G is of the form $A \to \beta B$ or $A \to \beta$, where $A, B \in N$ and $\beta \in \Sigma^*$.

Proposition 4 *Let G be a right-linear CFG and $\mathcal{F}$ be the FSA derived from G by the basic approximation algorithm. Then $L(G) = L(\mathcal{F})$.*

Proof: As before, we need only show $L(\mathcal{F}) \subseteq L(G)$.

Let $\mathcal{R}$ be the shift-reduce recognizer for G. The key fact to notice is that, because G is right-linear, no shift transition may follow a reduce transition. Therefore, no terminal transition in $\mathcal{F}$ may follow an ϵ-transition, and after

any ϵ-transition, there is a sequence of ϵ-transitions leading to the final state $[S' \to S\cdot]$. Hence $\mathcal{F}$ has the following kinds of states: the start state, the final state, states with terminal transitions entering and leaving them (we call these *reading* states), states with ϵ-transitions entering and leaving them (*prefinal* states), and states with terminal transitions entering them and ϵ-transitions leaving them (*crossover* states). Any accepting path through $\mathcal{F}$ will consist of a sequence of a start state, reading states, a crossover state, prefinal states, and a final state. The exception to this is a path accepting the empty string, which has a start state, possibly some prefinal states, and a final state.

The above argument also shows that unfolding does not change the set of strings accepted by $\mathcal{F}$, because any reduction in $\mathcal{R}_{\equiv}$ (or ϵ-transition in $\mathcal{F}$), is guaranteed to be part of a path of reductions (ϵ-transitions) leading to a final state of $\mathcal{R}_{\equiv}$ ($\mathcal{F}$).

Suppose now that $w = w_1 \dots w_n$ is accepted by $\mathcal{F}$. Then there is a path from the start state s_0 through reading states $s_1, \dots, s_{n-1}$, to crossover state s_n, followed by ϵ-transitions to the final state. We claim that if there there is a path from s_i to s_n labeled $w_{i+1} \dots w_n$, then there is a dotted rule $A \to x \cdot yB$ in s_i such $B \stackrel{*}{\Rightarrow} z$ and $yz = w_{i+1} \dots w_n$, where $A \in N, B \in N \cup \Sigma^*, y, z \in \Sigma^*$, and one of the following holds:

(a) x is a nonempty suffix of $w_1 \dots w_i$,

(b) $x = \epsilon$, $A'' \stackrel{*}{\Rightarrow} A$, $A' \to x' \cdot A''$ is a dotted rule in s_i, and x' is a nonempty suffix of $w_1 \dots w_i$, or

(c) $x = \epsilon$, $s_i = s_0$, and $S \stackrel{*}{\Rightarrow} A$.

We prove the claim by induction on $n - i$. For the base case, suppose there is an empty path from s_n to s_n. Because s_n is the crossover state, there must be some dotted rule $A \to x\cdot$ in s_n. Letting $y = z = B = \epsilon$, we get that $A \to x \cdot yB$ is a dotted rule of s_n and $B = z$. The dotted rule $A \to x \cdot yB$ must have either been added to s_n by closure or by shifts. If it arose from a shift, x must be a nonempty suffix of $w_1 \dots w_n$. If the dotted rule arose by closure, $x = \epsilon$, and there is some dotted rule $A' \to x' \cdot A''$ such that $A'' \stackrel{*}{\Rightarrow} A$ and x' is a nonempty suffix of $w_1 \dots w_n$.

Now suppose that the claim holds for paths from s_i to s_n, and look at a path labeled $w_i \dots w_n$ from s_{i-1} to s_n. By the induction hypothesis, $A \to x \cdot yB$ is a dotted rule of s_i, where $B \stackrel{*}{\Rightarrow} z$, $uz = w_{i+1} \dots w_n$, and (since $s_i \neq s_0$), either x is a nonempty suffix of $w_1 \dots w_i$ or $x = \epsilon, A' \to x' \cdot A''$ is a dotted rule of s_i, $A'' \stackrel{*}{\Rightarrow} A$, and x' is a nonempty suffix of $w_1 \dots w_i$.

In the former case, when x is a nonempty suffix of $w_1 \dots w_i$, then $x = w_j \dots w_i$ for some $1 \leq j < i$. Then $A \to w_j \dots w_i \cdot yB$ is a dotted rule of s_i, and thus $A \to w_j \dots w_{i-1} \cdot w_i yB$ is a dotted rule of s_{i-1}. If $j \leq i - 1$, then $w_j \dots w_{i-1}$ is a nonempty suffix of $w_1 \dots w_{i-1}$, and we are done. Otherwise, $w_j \dots w_{i-1} = \epsilon$, and so $A \to \cdot w_i yB$ is a dotted rule of s_{i-1}. Let $y' = w_i y$. Then

Symbol	Category	Features
`s`	sentence	`n` (number), `p` (person)
`np`	noun phrase	`n`, `p`, `c` (case)
`vp`	verb phrase	`n`, `p`, `t` (verb type)
`args`	verb arguments	`t`
`det`	determiner	`n`
`n`	noun	n
`pron`	pronoun	`n`, `p`, `c`
`v`	verb	`n`, `p`, `t`

Table 1: Categories of Example Grammar

$A \to \cdot y'B$ is a dotted rule of s_{i-1}, which must have been added by closure. Hence there are nonterminals A' and A'' such that $A'' \stackrel{*}{\Rightarrow} A$ and $A' \to x' \cdot A''$ is a dotted rule of s_{i-1}, where x' is a nonempty suffix of $w_1 \ldots w_{i-1}$.

In the latter case, there must be a dotted rule $A' \to w_j \ldots w_{i-1} \cdot w_i A''$ in s_{i-1}. The rest of the conditions are exactly as in the previous case.

Thus, if $w = w_1 \ldots w_n$ is accepted by $\mathcal{F}$, then there is a path from s_0 to s_n labeled by $w_1 \ldots w_n$. Hence, by the claim just proved, $A \to x \cdot yB$ is a dotted rule of s_n, and $B \stackrel{*}{\Rightarrow} z$, where $yz = w_1 \ldots w_n = w$. Because the s_i in the claim is s_0, and all the dotted rules of s_i can have nothing before the dot, and x must be the empty string. Therefore, the only possible case is case 3. Thus, $S \stackrel{*}{\Rightarrow} A \to yz = w$, and hence $w \in L(G)$. The proof that the empty string is accepted by $\mathcal{F}$ only if it is in $L(G)$ is similar to the proof of the claim. □

3.3 Decompositions

If each def(X) in the strongly-connected component decomposition of G is left-linear or right-linear, it is easy to see that G accepts a regular language, and that the overall approximation derived by decomposition is exact. Since some components may be left-linear and others right-linear, the overall class we can approximate exactly goes beyond purely left-linear or purely right-linear grammars.

4 Implementation and Example

The example in the appendix is an APSG for a small fragment of English, written in the notation accepted by our grammar compiler. The categories and features used in the grammar are described in Tables 1 and 2 (categories without features are omitted). The example grammar accepts sentences such as

```
i give a cake to tom
```

Feature	Values
`n` (number)	`s` (singular), `p` (plural)
`p` (person)	`1` (first), `2` (second), `3` (third)
`c` (case)	`s` (subject), `o` (nonsubject)
`t` (verb type)	`i` (intransitive), `t` (transitive), `d` (ditransitive)

Table 2: Features of Example Grammar

```
tom sleeps
i eat every nice cake
```

but rejects ill-formed inputs such as

```
i sleeps
i eats a cake
i give
tom eat
```

It is easy to see that the each strongly-connected component of the example is either left-linear or right linear, and therefore our algorithm will produce an equivalent FSA. Grammar compilation is organized as follows:

1. Instantiate input APSG to yield an equivalent CFG.
2. Decompose the CFG into strongly-connected components.
3. For each subgrammar def(X) in the decomposition:
   (a) approximate def(X) by aut(X);
   (b) determinize and minimize aut(X);
4. Recombine the aut(X) into a single FSA using the partial order of grammar components.
5. Determinize and minimize the recombined FSA.

For small examples such as the present one, steps 2, 3 and 4 can be replaced by a single approximation step for the whole CFG. In the current implementation, instantiation of the APSG into an equivalent CFG is written in Prolog, and the other compilation steps are written in C, for space and time efficiency in dealing with potentially large grammars and automata.

For the example grammar, the equivalent CFG has 78 nonterminals and 157 rules, the unfolded and flattened FSA 2615 states and 4096 transitions, and the determinized and minimized final DFA shown in Figure 7 has 16 states and

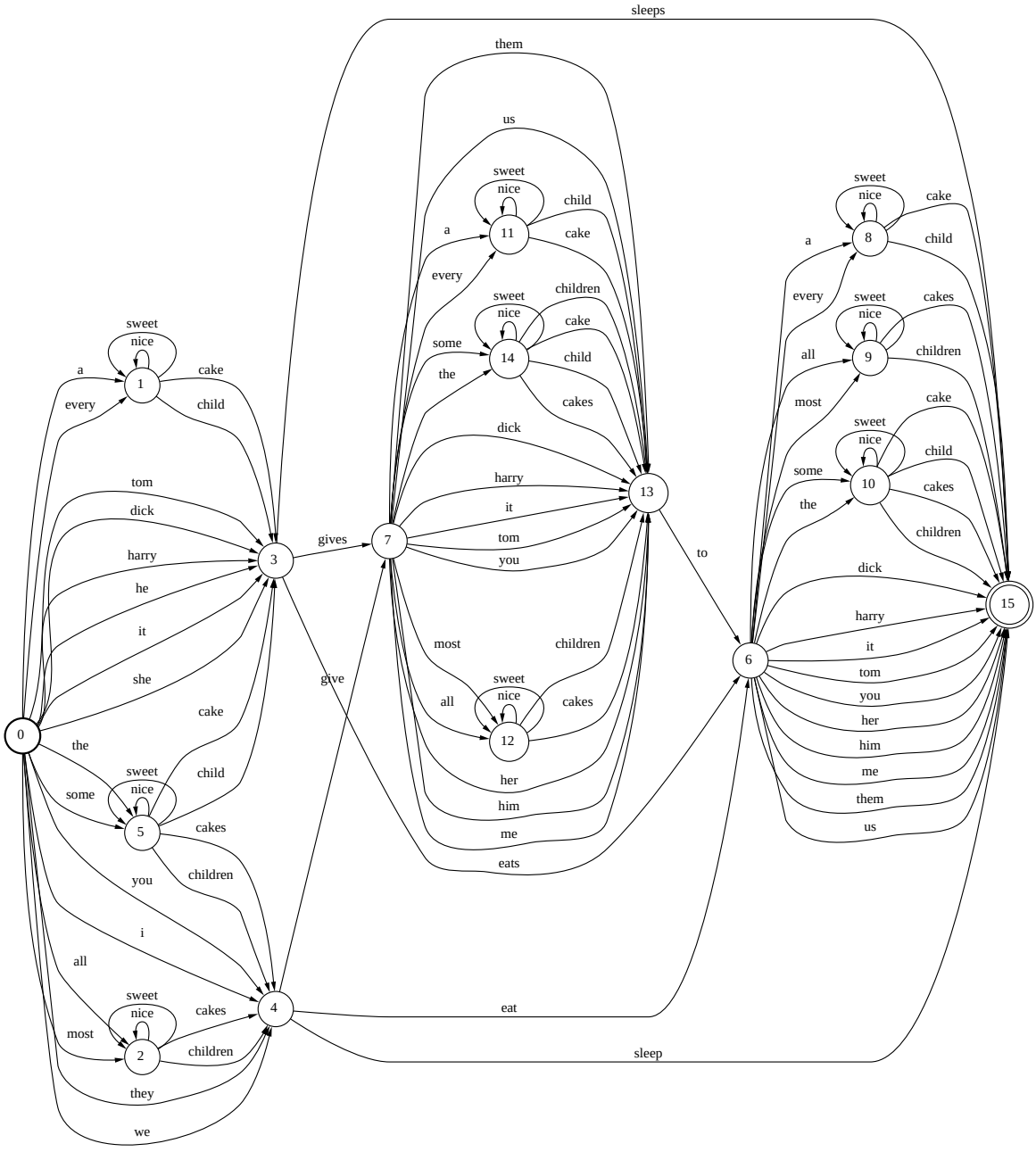

Figure 7: Approximation for Example Grammar

97 transitions. The runtime for the whole process is 1.78 seconds on a Sun SparcStation 20.

Substantially larger grammars, with thousands of instantiated rules, have been developed for a speech-to-speech translation project [14]. Compilation times vary widely, but very long compilations appear to be caused by a combinatorial explosion in the unfolding of right recursions that will be discussed further in the next section.

5 Informal Analysis

In addition to the cases of left-linear and right-linear grammars and decompositions into those cases discussed in Section 3, our algorithm is exact in a variety of interesting cases, including the examples of Church and Patil [8], which illustrate how typical attachment ambiguities arise as structural ambiguities on regular string sets.

The algorithm is also exact for some self-embedding grammars[3] of regular languages, such as

$$S \rightarrow aS \mid Sb \mid c$$

defining the regular language a^*cb^*.

A more interesting example is the following simplified grammar for the structure of English noun phrases:

NP → Det Nom | PN
Det → Art | NP 's
Nom → N | Nom PP | Adj Nom
PP → P NP

The symbols Art, Adj, N, PN and P correspond to the parts of speech article, adjective, noun, proper noun and preposition, and the nonterminals Det, NP, Nom and PP to determiner phrases, noun phrases, nominal phrases and prepositional phrases, respectively. From this grammar, the algorithm derives the exact DFA in Figure 8. This example is typical of the kinds of grammars with systematic attachment ambiguities discussed by Church and Patil [8]. A string of parts-of-speech such as

Art N P Art N P Art N

is ambiguous according to the grammar (only some constituents shown for simplicity):

Art [$_{\text{Nom}}$N [$_{\text{PP}}$P[$_{\text{NP}}$Art [$_{\text{Nom}}$ N [$_{\text{PP}}$P [$_{\text{NP}}$Art N]]]]]]
Art [$_{\text{Nom}}$[$_{\text{Nom}}$N [$_{\text{PP}}$P [$_{\text{NP}}$Art N]]] [$_{\text{PP}}$P [$_{\text{NP}}$Art N]]]

[3] A grammar is self-embedding if and only if licenses the derivation $X \stackrel{*}{\Rightarrow} \alpha X \beta$ for nonempty α and β. A language is regular if and only if it can be described by some non-self-embedding grammar.

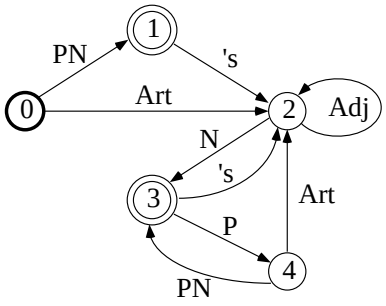

Figure 8: Acceptor for Noun Phrases

However, if multiplicity of analyses are ignored, the string set accepted by the grammar is regular and the approximation algorithm obtains the correct DFA. However, we have no characterization of the class of CFGs for which this kind of exact approximation is possible.

As an example of inexact approximation, consider the self-embedding CFG

$$S \rightarrow aSb \mid \epsilon$$

for the nonregular language $a^n b^n, n \geq 0$. This grammar is mapped by the algorithm into an FSA accepting $\epsilon \mid a^+ b^+$. The effect of the algorithm is thus to "forget" the pairing between a's and b's mediated by the stack of the grammar's characteristic recognizer.

Our algorithm has very poor worst-case performance. First, the expansion of an APSG into a CFG, not described here, can lead to an exponential blow-up in the number of nonterminals and rules. Second, the subset calculation implicit in the LR(0) construction can make the number of states in the characteristic machine exponential on the number of CF rules. Finally, unfolding can yield another exponential blow-up in the number of states.

However, in the practical examples we have considered, the first and the last problems appear to be the most serious.

The rule instantiation problem may be alleviated by avoiding full instantiation of unification grammar rules with respect to "don't care" features, that is, features that are not constrained by the rule.

The unfolding problem is particularly serious in grammars with subgrammars of the form

$$S \rightarrow X_1 S \mid \cdots \mid X_n S \mid Y \qquad . \tag{1}$$

It is easy to see that the number of unfolded states in the subgrammar is exponential in n. This kind of situation often arises indirectly in the expansion of an APSG when some features in the right-hand side of a rule are unconstrained and thus lead to many different instantiated rules. However, from the proof

of Proposition 4 it follows immediately that unfolding is unnecessary for right-linear grammars. Therefore, if we use our grammar decomposition method first and test individual components for right-linearity, unnecessary unfolding can be avoided. Alternatively, the problem can be circumvented by left factoring (1) as follows:

$$S \rightarrow ZS \mid Y$$
$$Z \rightarrow X_1 \mid \cdots \mid X_n$$

6 Related Work and Conclusions

Our work can be seen as an algorithmic realization of suggestions of Church and Patil [7, 8] on algebraic simplifications of CFGs of regular languages. Other work on finite state approximations of phrase structure grammars has typically relied on arbitrary depth cutoffs in rule application. While this may be reasonable for psycholinguistic modeling of performance restrictions on center embedding [12], it does not seem appropriate for speech recognition where the approximating FSA is intended to work as a filter and not reject inputs acceptable by the given grammar. For instance, depth cutoffs in the method described by Black [4] lead to approximating FSAs whose language is neither a subset nor a superset of the language of the given phrase-structure grammar. In contrast, our method will produce an exact FSA for many interesting grammars generating regular languages, such as those arising from systematic attachment ambiguities [8]. It is important to note, however, that even when the result FSA accepts the same language, the original grammar is still necessary because interpretation algorithms are generally expressed in terms of phrase structures described by that grammar, not in terms of the states of the FSA.

Several extensions of the present work may be worth investigating.

As is well known, speech recognition accuracy can often be improved by taking into account the probabilities of different sentences. If such probabilities are encoded as rule probabilities in the initial grammar, we would need a method for transferring them to the approximating FSA. Alternatively, transition probabilities for the approximating FSA could be estimated directly from a training corpus, either by simple counting in the case of a DFA or by an appropriate version of the Baum-Welch procedure for general probabilistic FSAs [13].

Alternative pushdown acceptors and stack congruences may be considered with different size-accuracy tradeoffs. Furthermore, instead of expanding the APSG first into a CFG and only then approximating, one might start with a pushdown acceptor for the APSG class under consideration [10], and approximate it directly using a generalized notion of stack congruence that takes into account the instantiation of stack items. This approach might well reduce the explosion in grammar size induced by the initial conversion of APSGs to CFGs, and also make the method applicable to APSGs with unbounded feature sets, such as general constraint-based grammars.

We do not have any useful quantitative measure of approximation quality. Formal-language theoretic notions such as the rational index of a language [5] capture a notion of language complexity but it is not clear how it relates to the intuition that an approximation is "worse" than another if it strictly contains it. In a probabilistic setting, a language can be identified with a probability density function over strings. Then the Kullback-Leibler divergence [9] between the approximation and the original language might be a useful measure of approximation quality.

Finally, constructions based on finite-state transducers may lead to a whole new class of approximations. For instance, CFGs may be decomposed into the composition of a simple fixed CFG with given approximation and a complex, varying finite-state transducer that needs no approximation.

Acknowledgments

We thank Mark Liberman for suggesting that we look into finite-state approximations, Richard Sproat, David Roe and Pedro Moreno trying out several prototypes supplying test grammars, and Mehryar Mohri, Edmund Grimley-Evans and the editors of this volume for corrections and other useful suggestions. This paper is a revised and extended version of the 1991 ACL meeting paper with the same title [11].

References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Massachusetts, 1976.

[2] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design*. Addison-Wesley, Reading, Massachusetts, 1977.

[3] Roland C. Backhouse. *Syntax of Programming Languages—Theory and Practice*. Series in Computer Science. Prentice-Hall, Englewood Cliffs, New Jersey, 1979.

[4] Alan W. Black. Finite state machines from feature grammars. In Masaru Tomita, editor, *International Workshop on Parsing Technologies*, pages 277–285, Pittsburgh, Pennsylvania, 1989. Carnegie Mellon University.

[5] Luc Boasson, Bruno Courcelle, and Maurice Nivat. The rational index: a complexity measure for languages. *SIAM Journal of Computing*, 10(2):284–296, 1981.

[6] Bob Carpenter. *The Logic of Typed Feature Structures*. Number 32 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, England, 1992.

[7] Kenneth W. Church. On memory limitations in natural language processing. Master's thesis, M.I.T., 1980. Published as Report MIT/LCS/TR-245.

[8] Kenneth W. Church and Ramesh Patil. Coping with syntactic ambiguity or how to put the block in the box on the table. *Computational Linguistics*, 8(3–4):139–149, 1982.

[9] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. Wiley-Interscience, New York, New York, 1991.

[10] Bernard Lang. Complete evaluation of Horn clauses: an automata theoretic approach. Rapport de Recherche 913, INRIA, Rocquencourt, France, November 1988.

[11] Fernando C. N. Pereira and Rebecca N. Wright. Finite-state approximation of phrase-structure grammars. In *29th Annual Meeting of the Association for Computational Linguistics*, pages 246–255, Berkeley, California, 1991. University of California at Berkeley, Association for Computational Linguistics, Morristown, New Jersey.

[12] Steven G. Pulman. Grammars, parsers, and memory limitations. *Language and Cognitive Processes*, 1(3):197–225, 1986.

[13] Lawrence R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[14] David B. Roe, Pedro J. Moreno, Richard W. Sproat, Fernando C. N. Pereira, Michael D. Riley, and Alejandro Macarrón. A spoken language translator for restricted-domain context-free languages. *Speech Communication*, 11:311–319, 1992.

[15] Stuart M. Shieber. *An Introduction to Unification-Based Approaches to Grammar*. Number 4 in CSLI Lecture Notes. Center for the Study of Language and Information, Stanford, California, 1985. Distributed by Chicago University Press.

[16] Stuart M. Shieber. Using restriction to extend parsing algorithms for complex-feature-based formalisms. In *23rd Annual Meeting of the Association for Computational Linguistics*, pages 145–152, Chicago, Illinois, 1985. Association for Computational Linguistics, Morristown, New Jersey.

[17] Joseph S. Ullian. Partial algorithm problems for context free languages. *Information and Control*, 11:90–101, 1967.

Appendix—APSG Formalism and Example

Nonterminal symbols (syntactic categories) may have features that specify variants of the category (eg. singular or plural noun phrases, intransitive or transitive verbs). A category *cat* with feature constraints is written

$$cat\texttt{\#[}c_1, \ldots, c_m\texttt{]}\,.$$

Feature constraints for feature f have one of the forms

$$f \;\texttt{=}\; v \tag{2}$$
$$f \;\texttt{=}\; c \tag{3}$$
$$f \;\texttt{=}\; \texttt{(}c_1, \ldots, c_n\texttt{)} \tag{4}$$

where v is a variable name (which must be capitalized) and $c, c_1, \ldots, c_n$ are feature values.

All occurrences of a variable v in a rule stand for the same unspecified value. A constraint with form (2) specifies a feature as having that value. A constraint of form (3) specifies an actual value for a feature, and a constraint of form (4) specifies that a feature may have any value from the specified set of values. The symbol "!" appearing as the value of a feature in the right-hand side of a rule indicates that that feature must have the same value as the feature of the same name of the category in the left-hand side of the rule. This notation, as well as variables, can be used to enforce feature agreement between categories in a rule, for instance, number agreement between subject and verb.

It is convenient to declare the features and possible values of categories with category declarations appearing before the grammar rules. Category declarations have the form

$$\begin{aligned}\texttt{cat}\ cat\texttt{\#[}\ \ f_1 \;\texttt{=}\; & \texttt{(}v_{11}, \ldots, v_{1k_1}\texttt{)},\\ & \ldots,\\ f_m \;\texttt{=}\; & \texttt{(}v_{m1}, \ldots, v_{mk_m}\texttt{)}\ \ \texttt{]}\,.\end{aligned}$$

giving all the possible values of all the features for the category.

The declaration

$$\texttt{start}\ cat\texttt{.}$$

declares *cat* as the start symbol of the grammar.

In the grammar rules, the symbol "'" prefixes terminal symbols, commas are used for sequencing and "|" for alternation.

```
start s.

cat s#[n=(s,p),p=(1,2,3)].
```

```
cat np#[n=(s,p),p=(1,2,3),c=(s,o)].
cat vp#[n=(s,p),p=(1,2,3),type=(i,t,d)].
cat args#[type=(i,t,d)].

cat det#[n=(s,p)].
cat n#[n=(s,p)].
cat pron#[n=(s,p),p=(1,2,3),c=(s,o)].
cat v#[n=(s,p),p=(1,2,3),type=(i,t,d)].

s => np#[n=!,p=!,c=s], vp#[n=!,p=!].

np#[p=3] => det#[n=!], adjs, n#[n=!].
np#[n=s,p=3] => pn.
np => pron#[n=!, p=!, c=!].

pron#[n=s,p=1,c=s] => `i.
pron#[p=2] => `you.
pron#[n=s,p=3,c=s] => `he | `she.
pron#[n=s,p=3] => `it.
pron#[n=p,p=1,c=s] => `we.
pron#[n=p,p=3,c=s] => `they.
pron#[n=s,p=1,c=o] => `me.
pron#[n=s,p=3,c=o] => `him | `her.
pron#[n=p,p=1,c=o] => `us.
pron#[n=p,p=3,c=o] => `them.

vp => v#[n=!,p=!,type=!], args#[type=!].

adjs => [].
adjs => adj, adjs.

args#[type=i] => [].
args#[type=t] => np#[c=o].
args#[type=d] => np#[c=o], `to, np#[c=o].

pn => `tom | `dick | `harry.

det => `some| `the.
det#[n=s] => `every | `a.
det#[n=p] => `all | `most.

n#[n=s] => `child | `cake.
n#[n=p] => `children | `cakes.
```

```
adj => ‘nice | ‘sweet.

v#[n=s,p=3,type=i] => ‘sleeps.
v#[n=p,type=i] => ‘sleep.
v#[n=s,p=(1,2),type=i] => ‘sleep.

v#[n=s,p=3,type=t] => ‘eats.
v#[n=p,type=t] => ‘eat.
v#[n=s,p=(1,2),type=t] => ‘eat.

v#[n=s,p=3,type=d] => ‘gives.
v#[n=p,type=d] => ‘give.
v#[n=s,p=(1,2),type=d] => ‘give.
```