FINITE-STATE APPROXIMATION OF PHRASE STRUCTURE GRAMMARS

Fernando C. N. Pereira
AT&T Bell Laboratories
600 Mountain Ave.
Murray Hill, NJ 07974

Rebecca N. Wright
Dept. of Computer Science, Yale University
PO Box 2158 Yale Station
New Haven, CT 06520

Abstract

Phrase-structure grammars are an effective representation for important syntactic and semantic aspects of natural languages, but are computationally too demanding for use as language models in real-time speech recognition. An algorithm is described that computes finite-state approximations for context-free grammars and equivalent augmented phrase-structure grammar formalisms. 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 construct 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. Current recognition algorithms can most directly use finite-state acceptor (FSA) language models. However, these models are inadequate for language interpretation, since they cannot express the relevant syntactic and semantic regularities. Augmented phrase structure grammar (APSG) formalisms, such as unification-based grammars (Shieber, 1985a), 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 problems might be circumvented by using separate grammars for language modeling and language interpretation. Ideally, the recognition grammar should not reject sentences acceptable by the interpretation grammar and it should contain as much as reasonable of the constraints built into the interpretation grammar. However, if the two grammars are built independently, those goals are difficult to maintain. For this reason, we have developed a method for constructing automatically a finite-state approximation for an APSG. Since the approximation serves as language model for a speech-recognition front-end to the real parser, we require it to be *sound* in the sense that the it accepts all strings in the language defined by the APSG. Without qualification, the term "approximation" will always mean here "sound approximation."

If no further constraints were placed on the closeness of the approximation, the trivial algorithm that assigns to any APSG over 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 the 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. A useful criterion is thus 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 APSGs yielding regular languages.[1] While we have not proved that in general our method satisfies the above exactness criterion, we show in Section 3.2 that the method is exact for left-linear and right-linear grammars, two important classes of context-free grammars generating regular languages.

[1] At first sight, this requirement may be seen as conflicting with the undecidability of determining whether a CFG generates a regular language (Harrison, 1978). However, note that the algorithm just produces an approximation, but cannot say whether the approximation is exact.

2 The Algorithm

Our approximation method applies to any context-free grammar (CFG), or any unification-based grammar (Shieber, 1985a) that can be fully expanded into a context-free grammar.[2] The resulting FSA accepts all the sentences accepted by the input grammar, and possibly some non-sentences as well.

The current implementation accepts as input a form of unification grammar in which features can take only atomic values drawn from a specified finite set. Such grammars can only generate context-free languages, since an equivalent CFG can be obtained by instantiating features in rules in all possible ways.

The heart of our approximation method is an algorithm to convert the LR(0) *characteristic machine* $\mathcal{M}(G)$ (Aho and Ullman, 1977; Backhouse, 1979) 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 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* $A \rightarrow \alpha \cdot \beta$ where $A \rightarrow \alpha\beta$ is some rule of G. $\mathcal{M}(G)$ is the determinization by the standard subset construction (Aho and Ullman, 1977) of the FSA defined as follows:

- The initial state is the dotted rule $S' \rightarrow \cdot S$ where S is the start symbol of G and S' is a new auxiliary start symbol.
- The final state is $S' \rightarrow S \cdot$.
- 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 dotted rule $A \rightarrow \alpha \cdot X\beta$ to $A \rightarrow \alpha X \cdot \beta$.
- There is an ϵ-transition from $A \rightarrow \alpha \cdot B\beta$ to $B \rightarrow \cdot\gamma$, where B is a nonterminal symbol and $B \rightarrow \gamma$ a rule in G.

[2] Unification-based grammars not in this class would have to be weakened first, using techniques akin to those of Sato and Tamaki (1984), Shieber (1985b) and Haas (1989).

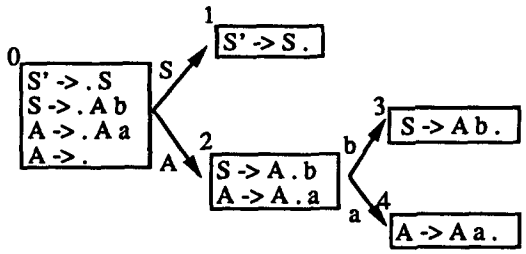

Figure 1: Characteristic Machine for G_1

$\mathcal{M}(G)$ 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* $A \rightarrow \alpha \cdot$ licenses a *reduction* move. This pops from the stack as many pairs as the symbols in α, checking that the symbols in the pairs match the corresponding elements of α, and then takes the transition out of the last state popped s labeled by A, pushing $\langle s, A \rangle$ 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 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

$$S \rightarrow Ab$$
$$A \rightarrow Aa \mid \epsilon \quad .$$

$\mathcal{M}(G_1)$ is shown on Figure 1. Unfolding is not required for this simple example, 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 4 would lead to a transition on nonter-

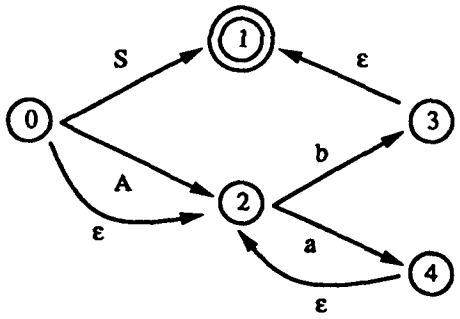

Figure 2: Flattened FSA

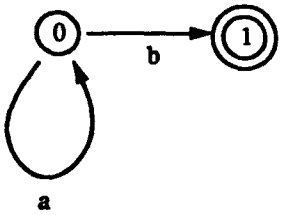

Figure 3: Minimal Acceptor

minal A, to state 2, from the state that activated the rule being reduced. Thus the corresponding ϵ-transition goes from state 4 to state 2. 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 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 context. In other words, the reducing state might be reached by different routes which use the result of the reduction in different ways. Consider for example the grammar G_2

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

which accepts just the two strings *aca* and *bcb*. Flattening $\mathcal{M}(G_2)$ will produce an FSA that will also accept *acb* and *bca*, an undesirable outcome. The reason for this is that the ϵ-transitions leaving the reducing state containing $X \rightarrow c\cdot$ do not distinguish between the different ways of reaching that state, which are encoded in the stack of $\mathcal{R}(G_2)$.

One way of solving the above problem 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. However, the set of possible stacks at a state is in general 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 portions of stack pushed between two arrivals at the same state in the finite-state control of the shift-reduce recognizer. The purpose of collapsing loops is to "forget" stack segments that may be arbitrarily repeated.[3] 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.

3 Formal Properties

In this section, we will show here that the approximation method described informally in the previous section is sound for arbitrary CFGs and is exact for left-linear and right-linear CFGs.

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

3.1 Soundness

Let $\mathcal{M}$ be the characteristic machine for G, with state set Q, start state s_0, set of final states F, and transition function $\delta : S \times V \rightarrow S$. 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. 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) $A \rightarrow \cdot$ is a completed dotted rule

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

in s, $s'' = s$ and τ is empty, or (2) $A \rightarrow X_1 \ldots X_n \cdot$ 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 $c_0 = \langle s_0, \epsilon, w \rangle$, $c_m = \langle s, \langle s_0, S \rangle, \epsilon \rangle$ for some final state s, and $c_{i-1} \vdash c_i$ for $1 \leq i \leq n$.

Let s be a state. We define the set Stacks(s) to contain every sequence $\langle s_0, X_0 \rangle \ldots \langle s_k, X_k \rangle$ such that $s_i = \delta(s_{i-1}, X_{i-1}), 1 \leq i \leq k$ and $s = \delta(s_k, X_k)$. 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

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

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 the unfolded recognizer $\mathcal{R}_{\equiv}$ and $\mathcal{R}$ are equivalent recognizers.*

Proof: We show first that any string w accepted by $\mathcal{R}_{\equiv}$ is accepted by $\mathcal{R}$. Let $d_0, \ldots, d_m$ be a derivation of w in $\mathcal{R}_{\equiv}$. Each d_i has the form $d_i = \langle [\rho_i]^{s_i}, \sigma_i, u_i \rangle$, and can be mapped to an $\mathcal{R}$ configuration $\hat{d}_i = \langle s_i, \hat{\sigma}_i, u_i \rangle$, where $\hat{\epsilon} = \epsilon$ and $\widehat{\langle s, C \rangle, X} = \hat{\sigma}\langle s, X \rangle$. It is straightforward to verify that $\hat{d}_0, \ldots, \hat{d}_m$ is a derivation of w in $\mathcal{R}$.

Conversely, let $w \in \mathrm{L}(G)$, and $c_0, \ldots, c_m$ 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 $u_{i-1} = x u_i$ and $\delta(s_{i-1}, x) = s_i$. Therefore,

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

Furthermore,

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

Thus we have

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

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

(b) $A \rightarrow X_1 \cdots X_n \cdot$ 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 $2 \leq i \leq n$, 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

$$\begin{aligned} \bar{\sigma}_{i-1} &= \overline{\sigma\tau} \\ &= \overline{\sigma\langle q_1, X_1 \rangle \cdots \langle q_{n-1}, X_{n-1} \rangle} \end{aligned}$$

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

Thus

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

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'' such that $\delta_{\equiv}(p'', \alpha) = p$ and $\delta_{\equiv}(p'', A) = p'$. Then the *flattening* $\mathcal{F}_{\equiv}$ of $\mathcal{R}_{\equiv}$ is a nondeterministic FSA 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 $\sigma_{i-1} = \sigma\langle r_1, X_1\rangle \ldots \langle r_n, X_n\rangle$, $\sigma_i = \sigma\langle r_1, A\rangle$, $\delta(r_1, A) = q_i$, 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 of $\mathcal{R}(G)_{\equiv}$.*

Finally, we should 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 σ *collapses* to a stack σ' if $\sigma = \rho\tau\upsilon$, $\sigma' = \rho\upsilon$ and τ is a loop. Two stacks are equivalent if they can be collapsed to the same stack. This equivalence relation is closed under suffixing, therefore it is a stack congruence.

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 would be that it be exact for a wide class of CFGs generating regular languages. We show in this section that our algorithm is exact both for left-linear and for right-linear context-free grammars, which as is well-known generate regular languages.

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'.

3.3 Left-Linear Grammars

In this section, we assume that the CFG G is left-linear, that is, 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 produced by the approximation algorithm from G. Then $L(G) = L(\mathcal{F})$.*

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

The proof hinges on the observation that each state s of $\mathcal{M}(G)$ 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 $s_0 = [S' \rightarrow \cdot S]$, with $\hat{s}_0 = \epsilon$. The core $\hat{s}$ of any other state s will by construction contain only dotted rules of the form $A \rightarrow \alpha \cdot \beta$ with $\alpha \neq \epsilon$. Since G is left linear, β must be a terminal string, ensuring that $s = [\hat{s}]$. Therefore, every dotted rule $A \rightarrow \alpha \cdot \beta$ in s must result from dotted rule $A \rightarrow \cdot\alpha\beta$ in s_0 by the sequence of transitions determined by α (since $\mathcal{M}(G)$ is deterministic). This means that if $A \rightarrow \alpha \cdot \beta$ and $A' \rightarrow \alpha' \cdot \beta'$ are in s, it must be the case that $\alpha = \alpha'$. In the remainder of this proof, let $\bar{\alpha} = s$ whenever $\alpha = \hat{s}$.

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

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 $A \rightarrow \alpha$ 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 $A \rightarrow \epsilon$, 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 $A \rightarrow \alpha a x'$ 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 claim just proved, $S \stackrel{*}{\Rightarrow} Ay \Rightarrow xy$, where $A \rightarrow x$ is a rule and $w = xy$ (since $\epsilon \neq S$). Therefore, $S \stackrel{*}{\Rightarrow} w$, so $w \in L(G)$, as desired. □

3.4 Right-Linear Grammars

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

Proposition 4 *Let G be a right-linear CFG and $\mathcal{F}$ be the unfolded, flattened automaton produced by the approximation algorithm on input G. 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 $[S' \rightarrow S\cdot]$. Hence $\mathcal{F}$ has the following kinds of states: the start state, the final state, states with terminal transitions entering or 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 $w = w_1 \ldots w_n$ is accepted by $\mathcal{F}$. Then there is a path from the start state s_0 through reading states $s_1, \ldots, s_{n-1}$, 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 $w_{i+1} \ldots w_n$, then there is a dotted rule $A \rightarrow x \cdot yB$ in s_i such $B \stackrel{*}{\Rightarrow} z$ and $yz = w_{i+1} \ldots w_n$, 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 $w_1 \ldots w_i$,

(b) $x = \epsilon$, $A'' \stackrel{*}{\Rightarrow} A$, $A' \rightarrow x' \cdot A''$ is a dotted rule in s_i, and x' is a nonempty suffix of $w_1 \ldots w_i$, 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 $A \rightarrow x\cdot$ in s_n. Letting $y = z = B = \epsilon$, we get that $A \rightarrow x \cdot yB$ is a dotted rule of s_n and $B = z$. The dotted rule $A \rightarrow x \cdot yB$ 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 $w_1 \ldots w_n$. If the dotted rule arose by closure, $x = \epsilon$, and there is some dotted rule $A' \rightarrow x' \cdot A''$ such that $A'' \stackrel{*}{\Rightarrow} A$ and x' is a nonempty suffix of $w_1 \ldots w_n$.

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

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

4 A Complete Example

The appendix shows an APSG for a small fragment of English, written in the notation accepted by the current version of our grammar compiler. The categories and features used in the grammar are described in Tables 1 and 2 (categories without features are omitted). Features enforce person-number agreement, personal pronoun case, and a limited verb subcategorization scheme.

Grammar compilation has three phrases: (i) construction of an equivalent CFG, (ii) approximation, and (iii) determinization and minimization of the resulting FSA. The equivalent CFG is derived by finding all full instantiations of the initial APSG rules that are actually reachable in a derivation from the grammar's start symbol. In the current implementation, the construction of the equivalent CFG is is done by a Prolog program, while the approximator, determinizer and minimizer are written in C.

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 16 states and 97 transitions. The runtime for the whole process is 4.91 seconds on a Sun SparcStation 1.

Substantially larger grammars, with thousands of instantiated rules, have been developed for a speech-to-speech translation project. 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 discussed in Section 3, our algorithm is exact in a variety of interesting cases, including the examples of Church and Patil (1982), which illustrate how typical attachment ambiguities arise as structural ambiguities on regular string sets.

The algorithm is also exact for some self-embedding grammars[4] 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

[4] 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.

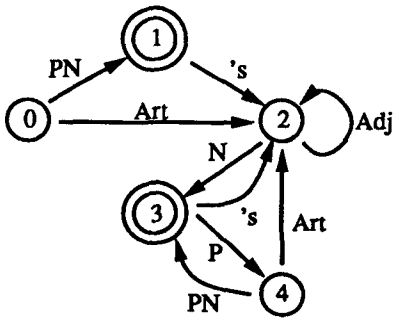

Figure 4: Acceptor for Noun Phrases

phrases:

$$\begin{aligned}&\text{NP} \rightarrow \text{Det Nom} \mid \text{PN}\\&\text{Det} \rightarrow \text{Art} \mid \text{NP 's}\\&\text{Nom} \rightarrow \text{N} \mid \text{Nom PP} \mid \text{Adj Nom}\\&\text{PP} \rightarrow \text{P NP}\end{aligned}$$

The symbols Art, N, PN and P correspond to the parts of speech article, noun, proper noun and preposition. From this grammar, the algorithm derives the DFA in Figure 4.

As an example of inexact approximation, consider the 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

$$S \rightarrow X_1 S \mid \cdots \mid X_n S \mid Y \quad . \qquad (1)$$

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. In fact, from the proof of Proposition 4 it follows immediately that unfolding is unnecessary for right-linear grammars. Ultimately, by dividing the grammar into non-mutually recursive (strongly connected) components and only unfolding center-embedded components, this particular problem could be avoided.[5] In the meanwhile, the problem can be circumvented by left factoring (1) as follows:

$$\begin{aligned}&S \rightarrow ZS \mid Y\\&Z \rightarrow X_1 \mid \cdots \mid X_n\end{aligned}$$

6 Related Work and Conclusions

Our work can be seen as an algorithmic realization of suggestions of Church and Patil (1980; 1982) 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 is reasonable for psycholinguistic modeling of performance restrictions on center embedding (Pulman, 1986), 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 (1989) 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 (Church and Patil, 1982). It important to note, however, that even when the result FSA accepts the same language, the original grammar is still necessary because interpreta-

[5] We have already implemented a version of the algorithm that splits the grammar into strongly connected components, approximates and minimizes separately each component and combines the results, but the main purpose of this version is to reduce approximation and determinization costs for some grammars.

tion algorithms are generally expressed in terms of phrase structures described by that grammar, not in terms of the states of the FSA.

Although the algorithm described here has mostly been adequate for its intended application — grammars sufficiently complex not to be approximated within reasonable time and space bounds usually yield automata that are far too big for our current real-time speech recognition hardware — it would be eventually of interest to handle right-recursion in a less profligate way. In a more theoretical vein, it would also be interesting to characterize more tightly the class of exactly approximable grammars. Finally, and most speculatively, one would like to develop useful notions of degree of approximation of a language by a regular language. Formal-language-theoretic notions such as the rational index (Boason et al., 1981) or probabilistic ones (Soule, 1974) might be profitably investigated for this purpose.

Acknowledgments

We thank Mark Liberman for suggesting that we look into finite-state approximations and Pedro Moreno, David Roe, and Richard Sproat for trying out several prototypes of the implementation and supplying test grammars.

References

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

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

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

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

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

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

Andrew Haas. 1989. A parsing algorithm for unification grammar. *Computational Linguistics*, 15(4):219–232.

Michael A. Harrison. 1978. *Introduction to Formal Language Theory.* Addison-Wesley, Reading, Massachussets.

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

Taisuke Sato and Hisao Tamaki. 1984. Enumeration of success patterns in logic programs. *Theoretical Computer Science*, 34:227–240.

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

Stuart M. Shieber. 1985b. 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. Association for Computational Linguistics, Morristown, New Jersey.

Stephen Soule. 1974. Entropies of probabilistic grammars. *Information and Control*, 25:57–74.

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

$$cat\#[c_1, \ldots, c_m].$$

Feature constraints for feature f have one of the forms

$$f = v \quad (2)$$

$$f = c \quad (3)$$

$$f = (c_1, \ldots, c_n) \quad (4)$$

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,

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

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

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

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

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

The declaration

```
start cat.
```

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.
```