Finite-State Approximations of Grammars

Fernando Pereira

AT&T Bell Laboratories
600 Mountain Ave., Murray Hill, NJ 07974

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 grammars, 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, but it should contain as much as possible of the constraints built into the interpretation grammar. However, if the two grammars are built independently, those constraints are difficult to maintain. For this reason, we have developed a method for constructing automatically a finite-state approximation for an APSG. Since the purpose of the approximation is to serve as a filter in a speech-recognition front-end to the real parser, the approximation language is a superset of the language accepted by the APSG. The term "approximation" will always be used in this sense in what follows.

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. Clearly, this is not what is required. 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 regular productions. 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.

We have not yet proved that our method satisfies the above exactness criterion, but some experiments have shown that the method is exact for a variety of interesting grammars.

The Algorithm

Our approximation method applies to any context-free grammar (CFG), or any unification grammar that can be fully expanded into a context-free grammar.[1] The resulting FSA accepts a superset of the sentences accepted by the input grammar.

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. It is clear that 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 $S' \rightarrow S\cdot$ where S is the start symbol of G and S' is a new auxiliary start symbol.
- The final state is $S' \rightarrow S\cdot$.

[1] Unification grammars not in this class must first be weakened using techniques such as Shieber's restrictor (Shieber, 1985).

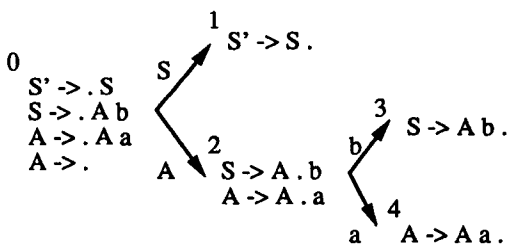

Figure 1: Characteristic Machine for G_1

- 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.

$\mathcal{M}(G)$ can be seen as a finite state control for a nondeterministic shift-reduce pushdown recognizer 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 $\langle s, tx\rangle$. 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.

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. However, as we will see below, flattening the characteristic machine recognizer directly will lead to poor approximations in many interesting cases. Instead, the characteristic machine must be *unfolded* into a larger machine whose states carry information about the possible shift-reduce stacks at states of the characteristic machine. 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

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

$\mathcal{M}(G_1)$ 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 nonterminal A, to state 2, from the state that

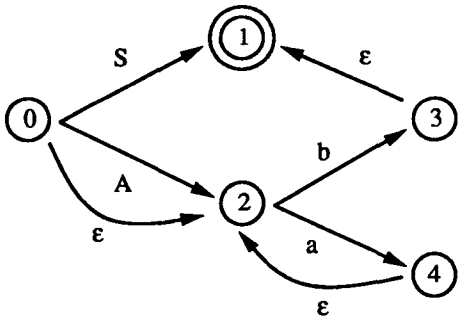

Figure 2: Flattened FSA

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. Doing that and simplifying, we get finally the FSA in Figure 3. which is the minimal FSA for the input left-linear grammar.

If flattening were 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 stack contents. 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

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

which accepts the two strings *aca* and *bcb*. Flattening $\mathcal{M}(G_2)$ will produce an FSA that will also accept *acb* and *bca*, clearly 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 the characteristic recognizer.

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 unfolded acceptor 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 par-

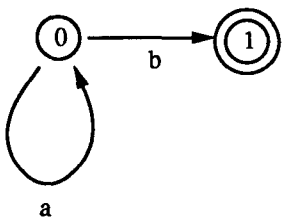

Figure 3: Minimal Acceptor

tition 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.[2] Clearly, 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.

Soundness of the Algorithm

We will show here that the approximation method described informally in the previous section is *sound*, in the sense that the approximating FSA will always accept a superset of the language accepted by the input CFG.

In what follows, G is a fixed CFG with terminal vocabulary Σ, nonterminal vocabulary N and start symbol S. $\mathcal{M}$ is the characteristic machine for G, with state set Q, start state s_0, final states F, and transition function $\delta : S \times (\Sigma \cup N) \to S$. 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. 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 terminal or nonterminal 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: $\langle s, \sigma\tau, w \rangle \vdash \langle s', \sigma\langle s'', A\rangle, w \rangle$ if $\delta(s'', A) = s'$ and 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 $c_0 = \langle s_0, \epsilon, w \rangle$, $c_m = \langle s, \langle s_0, S\rangle, \epsilon \rangle$ for some final state p, and $c_{i-1} \vdash c_i$ for $1 \le i \le n$.

Let s be a state. The set Stacks(s) contains each sequence $\langle s_0, X_0\rangle \ldots \langle s_k, X_k\rangle$ such that $s_i = \delta(s_{i-1}, X_{i-1}), 1 \le i \le k$ 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 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 of a state and stack equivalence class at that state. The initial state is $\langle s_0, [\epsilon]_{s_0}\rangle$, and the final states are all $\langle s, [\sigma]_s\rangle$ with $s \in F$. The transition function $\delta_\equiv$ of the unfolded recognizer is defined by

$$\delta_\equiv(\langle s, [\sigma]_s\rangle, X) = \langle \delta(s, X), [\sigma\langle s, X\rangle]_{\delta(s,X)}\rangle$$

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.

For any unfolded state p, let Pop(p) be the set of states reachable from p by a reduce transition. More precisely, 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 nodeterministic 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 $p' \in$ 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 $p_i = \langle q_i, [\sigma_i]_{p_i}\rangle$. 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 $\sigma_{i-1} = \sigma\langle r_1, X_1\rangle \ldots \langle r_n, X_n\rangle$, $\sigma_i = \sigma\langle r_1, A\rangle$, $\delta\langle r_1, A\rangle = q_i$, and $\delta(r_j, X_j) = r_{j+1}$ for $1 \le j < n$. By definition of stack congruence, we will then have

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

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 $\delta_\equiv(\langle r_1, [\sigma]_{r_1}\rangle, A) = p_i$. Therefore, $p_i \in$ 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.

We have thus shown how to construct from a derivation of a string in $\mathcal{R}$ an accepting path for the same string in $\mathcal{F}_\equiv$. This proves that every string in $L(G)$ is accepted by $\mathcal{F}_\equiv$, that is, that our construction is sound.

Finally, we should show that the stack collapsing equivalence informally described earlier is indeed a stack congruence. A stack τ is a *loop* if $\tau =$

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

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

$\langle s_1, X_1 \rangle \ldots \langle s_k, X_k \rangle$ 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. Clearly, this equivalence relation is closed under suffixing, therefore it is a stack congruence.

A Complete Example

The example grammar in the appendix shows an APSG for a small fragment of English, written in the notation accepted by the curent 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). The example grammar accepts sentences such as

I give a cake to Tom
Tom sleeps
I eat every nice cake

but rejects ill-formed inputs such as

I sleeps
I eats a cake
I give
Tom eat

The current grammar compiler factors out terminal productions to make the approximation algorithm independent of vocabulary size; transitions are labeled by automatically generated preterminal symbols instead of terminal symbols. After this factoring, the full instantiation of the example grammar has 181 rules, its characteristic machine 222 states and 922 transitions, the unfolded and flattened FSA 3417 states and 4255 transitions, and the determinized and minimized final DFA 18 states and 67 transitions. The compilation time is 123.19 seconds on a Sun SparcStation 1, with the determinizer and minimizer written in C and the rest of the compiler in Quintus Prolog. Most of the time is spent in the unfolding and flattening phases (90.62 seconds unfolding and 17.33 flattening). It is hoped that recoding these phases in C using carefully tuned data structures will speed them up by between one and two orders of magnitude.

Substantially larger grammars, with thousands of instantiated rules, have been developed for a speech-to-speech translation project. Compilation times range from the very reasonable (around 10 minutes) to the very high (10 hours). Very long compilations are caused by a combinatorial explosion in the unfolding of right recursions that will be discussed further in the next section.

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

Informal Analysis

The present algorithm has not yet been analyzed sufficiently to determine the class of context-free grammars generating regular languages for which it is exact. However, it is exact for in a variety of interesting cases, including the examples of Church and Patil (Church and Patil, 1982), which illustrate how typical attachment ambiguities arise as structural ambiguities on regular string sets.

For example, the left-linear grammar

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

and the right-linear grammar

$$S \rightarrow aS \mid b$$

both of which generate the regular set a^*b, are mapped by the algorithm into the FSA in Figure 3.

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

[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.

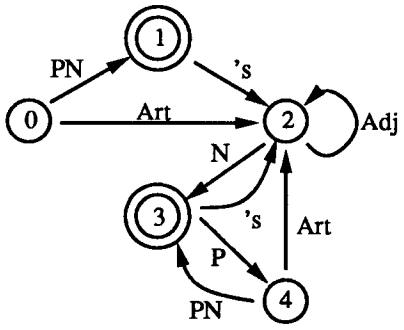

Figure 4: Acceptor for Noun Phrases

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 in a pushdown acceptor for the CFG.

As noted earlier, right recursion is rather bad for the present unfolding scheme. It is easy to see that the number of unfolded states for a grammar of the form

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

is exponential in n. However, the problem can be circumvented by left factoring the grammar as follows:

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

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.

Related Work and Conclusions

Our work can be seen as an algorithmic realization of suggestions of Church and Patil (Church, 1980; Church and Patil, 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 speeech 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 interpretation algorithms are generally expressed in terms of phrase structures described by that grammar, not in terms of the states of the FSA.

The current algorithm can be combinatorially explosive in two places: the instantiation of unification grammar rules to derive an equivalent CFG, and the unfolding of the characteristic machine, in particular for right-recursive rules. The former problem can 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. This can also help decrease the right-recursion unfolding explosion discussed earlier. As for the cost of unfolding, preliminary experiments suggest that dividing the grammar into non-mutually-recursive components and applying the LR(0) construction and unfolding separately to those components could lead to much smaller unfolded automata.

Acknowledgments

Thanks are due to Mark Liberman for suggesting that finite-state approximations might be worth investigating to David Roe and Pedro Moreno for using the grammar compiler prototype and patiently putting up with its bugs and inefficiencies.

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.

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.

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

Stuart M. Shieber. 1985. Using restriction to extend parsing algorithms for complex-feature-based formalisms. In *23rd Annual Meeting of the Association*

for Computational Linguistics, pages 145–152, Morristown, New Jersey. Association for Computational Linguistics.

Appendix

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 the form

$$f = v$$

for a single value v or

$$f = (v_1, \ldots, v_n)$$

for several alternative 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 can be used to enforce feature agreement, 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

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

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.
```