PARSING BY SUCCESSIVE APPROXIMATION

Helmut Schmid
IMS-CL, University of Stuttgart
Azenbergstr. 12, D-70174 Stuttgart, Germany
Email: schmid@ims.uni-stuttgart.de

Abstract

It is proposed to parse feature structure-based grammars in several steps. Each step is aimed to eliminate as many invalid analyses as possible as efficiently as possible. To this end the set of feature constraints is divided into three subsets, a set of context-free constraints, a set of filtering constraints and a set of structure-building constraints, which are solved in that order. The best processing strategy differs: Context-free constraints are solved efficiently with one of the well-known algorithms for context-free parsing. Filtering constraints can be solved using unification algorithms for non-disjunctive feature structures whereas structure-building constraints require special techniques to represent feature structures with embedded disjunctions efficiently. A compilation method and an efficient processing strategy for filtering constraints are presented.

1 Introduction

The efficiency of context-free parsing is well known [Younger, 1967]. Since many featured structure-based grammars either have a context-free backbone or can be transformed into a grammar with a context-free backbone, it is possible to take advantage of the efficiency of context free parsing if the parser proceeds in two steps: First a context-free parser builds the context-free part of the syntactic analysis which is then extended by the calculation of feature structures. Maxwell and Kaplan [Maxwell III and Kaplan, 1994] experimented with variants of this strategy in their LFG parser. One result of their experiments was that their grammar is processed more efficiently by their parser if the rather broad context-free categories chosen by the grammar writers are replaced by more specific categories. To this end some relevant features have been compiled manually into the context-free grammar. Of course one would prefer to have a compiler perform this task automatically. This would enable the grammar writer to use the categories which he considers appropriate, without loosing efficiency. How this can be done is shown in section 3.

Often it is useful to split the second part of processing, the evaluation of feature constraints, into two steps as well. Feature constraints which are likely to eliminate analyses (filtering constraints) are evaluated first whereas the evaluation of other constraints (structure-building constraints) which mainly serve to build some (e.g. semantic) representation is delayed. The ALEP system (Advanced Language Engineering Platform [Simpkins, 1994]) allows the user to explicitly specify features whose constraints are to be delayed. Kasper and Krieger [Kasper and Krieger, 1996] present a similar idea for HPSG parsing.

Separating filtering constraints and structure-building constraints has two advantages: Since many analyses are eliminated by the filtering constraints, the parser does not waste time on the – usually costly – evaluation of structure-building constraints for analyses which will fail anyway. As another advantage, it is possible to choose the most efficient processing strategy for each of the two constraint types independently.

Structure-building features tend to reflect the syntactic structure of a constituent. The semantic feature of a VP node e.g. would encode the attachment side of an embedded PP if there is ambiguity. In order to avoid such local ambiguities multiplying out in the semantic representation, it is necessary to have means to represent ambiguity locally; i.e. the feature structure representation must allow for embedded disjunctions. A representation in disjunctive normal form (i.e. as a set of alternative non-disjunctive feature structures) would be inefficient because the number of feature structures would grow too fast as local ambiguities multiply out. More efficient algorithms for processing disjunctive feature constraints have been presented e.g. in [Kasper, 1987], [Dörre and Eisele, 1990], [Maxwell III and Kaplan, 1996], and [Emele, 1991].

Filtering constraints, on the other hand, can be processed with standard unification algorithms and a disjunctive normal form representation for feature structures if the feature values restricted by these constraints have limited depth and therefore limited compexity. The SUBCAT and SLASH features in HPSG e.g. have this property whereas e.g. the SUBJ and OBJ features in LFG do not. Even if the depth of a feature value is unbounded it is still possible to limit the complexity of the feature structures artificially by pruning feature structures below some level of embedding. Little of the restrictive power of the constraints is lost thereby, since constraints seldom refer to deeply embedded information. However, in order to ensure the correctness of the final result, the filtering constraints have to be evaluated again in the next step together with the structure-building constraints.

In this article a parser is presented which implements the first two steps of the parsing strategy outlined above. Currently it is assumed that the grammars do not contain structure-building constraints which would require the third processing step (see also section 4.3). Section 2 provides an overview of the grammar formalism used by this parser. Section 3 describes the compilation of the grammar. Details of the parsing strategy and its current limitations are given in section 4. Section 5 presents results from experiments with the parser and section 6 closes with a summary.

2 The Grammar Formalism

The parser employs a rule-based grammar formalism. Each grammar rule has a context-free backbone. One of the daughter nodes in a rule is marked as the head with a preceding backquote. Trace nodes are marked with an asterisk after the category name. At least one daughter node has to be non-empty because rules which generate empty strings are not allowed. Associated to each node in a rule is a set of feature constraint equations which restrict the values of its features. Variables are used to express feature unification: Two features are unified by assigning the value of the same variable to both of them (e.g. f1 = v; f2 = v;). Feature structures are totally well-typed, i.e. they are typed and each feature which is appropriate for some type is present and has a value of an appropriate type. Equality is interpreted extensionally, i.e. two feature structures are considered equal if they have the same type and all of their feature values are equal. Feature structures have to be acyclic. Type hierarchies are not supported currently.

Two predefined feature types and three classes of user-defined types are available to the grammar writer. Features of the predefined type STRING accept any character string as value. Features of the predefined type FS_LIST take a list of feature structures of the class *category* (see below) as value. The user defines his own feature types of the class *enumeration type* by listing the corresponding set of possible values which have to be atomic. Another class of user-defined feature types are the *structured types* which are defined by listing the set of attributes appropriate for this type with the types of their values. *Categories* are the last class of user-defined feature types. Their definition is analogous to that of a structured type. Each node of category X has an associated feature structure of type X.

To simplify the grammar writer's task, the grammar formalism supports templates, default inheritance between the mother node and the head daughter of a rule (the value of a feature of the head daughter of a rule is inherited from the mother node if it is undefined otherwise and if the feature structure of the mother node contains a feature with the same name and type – and vice versa), automatic handling of the two features *Phon* and *HeadLex* (lexical head), and special variable types called "restrictor" types which define a subset of features which are to be unified when two *category* feature structures are (partially) unified by assigning the same variable to both of them. The last feature is needed e.g. to exclude the *Phon* feature from unification when the feature structure of a trace node is unified with the feature structure of a filler node which has been threaded through the tree.

The grammar formalism allows disjunctive value specifications in the case of features of an *enumeration type*[1]. A simple toy grammar written in this formalism is shown in the appendix.

3 Compilation

A compiler transforms the plain text representation of the grammar into a form which is appropriate for the parser and provides error reports. Compilation aims to minimize the computations required during parsing.

[1] Using a bit-vector representation, such disjunctions are easy to store and process efficiently.

The compiler expands templates, adds constraint equations for automatic features and for inherited features, flattens feature structures by replacing structured features with a set of new features corresponding to the subfeatures of the structured feature, and infers in some cases additional constraints. E.g. while compiling the rule[2]

```
VP {Subcat=[*];Subcat=r;}  ->  'V {Subcat=[ NP{}=np | r ];}  NP {}=np;
```

the following constraint equations are obtained[3] (among others):

```
r = 0.VP.Subcat           x = 0.VP.Subcat.cdr(1)
r = 1.VP.Subcat.cdr(1)    x = []        ...
```

From these constraints the compiler infers the additional constraint:

```
x = 1.VP.Subcat.cdr(2)
```

These inferences are necessary for the constraint evaluation algorithm presented in section 4.1.

Finally, the compiler replaces unified variables with a single variable, merges equations of the form *x=constant1; x=constant2* into a new equation *x=constant3*, eliminates redundant equations and generates *fixed assignments* for equations with a feature path expression on the right hand side if the variable on the left hand side is unified with an unambiguous constant in some other equation. This is e.g. the case for the third equation and the inferred equation above. The fixed assignments derived from these equations are:

```
0.VP.Subcat.cdr(1) := []      1.VP.Subcat.cdr(2) := []
```

The three equations involved are removed at this point. For each of the remaining equations with a path expression on the right hand side, the compiler generates a *variable assignment*:

```
0.VP.Subcat := r      1.VP.Subcat.cdr(1) := r
```

Equations with the same variable on the left hand side are then grouped together:

```
r:  r = 0.VP.Subcat   r = 1.VP.Subcat.cdr(1)
```

The variables representing these groups are sorted so that variables which depend on the values of other variables will follow these other variables[4]. This ordering is required by the constraint evaluation algorithm presented in section 4.1.

3.1 Generation of Context-Free Rules

The compiler supports compilation of feature constraints into the context-free backbone of the grammar in the case of features of the class *enumeration type*. Features of other types cannot be compiled because the number of possible values is infinite. The user has to specify which features are to be *incorporated* – i.e. compiled – for each category, and the compiler automatically generates all valid context-free rules with the refined categories.

The following algorithm is used for the generation of the context-free rules: First the compiler orders the incorporated features of all nodes of a given grammar rule. A sequence $f_1, f_2, \ldots, f_n$ is obtained. Then the set of permitted values for the first feature f_1 is determined. To this end, the compiler checks whether there is a fixed assignment for this feature. If one exists, the corresponding value is the only permitted value. Otherwise, the compiler checks whether there are two constraint equations of the form $v = f_1$ and $v = (c_1; c_2; \ldots; c_m)$ where $(c_1; c_2; \ldots; c_m)$ is a disjunction of constant values. In this case the set of permitted values is $\{c_1, c_2, \ldots, c_m\}$. Otherwise all values appropriate for feature f_1 are permitted features. The compiler chooses one of the permitted values and switches to the next feature.

While assigning a value to feature f_i, the compiler first checks whether f_i is unified with some feature f_k where $k < i$. This is the case if there are two equations $y = f_i$ and $y = f_k$. If there is such a feature f_k, which already got a value since it has a smaller index, then its value is assigned to feature f_i. Otherwise the set of permitted values is computed as described above and one value is selected. After the value of the last

[2]The notation `NP{}=np` means "*unify the feature structure of the node* NP *with the feature structure denoted by the variable* np *according to the definition of the restrictor type of the variable* np," i.e. unify the subset of features listed in the restrictor definition. The list notation is similar to that in Prolog, but an asterisk rather than an underscore is used to mark dummy arguments.

[3]The number in front of a path expression refers to the position of the node in the rule. The expression *cdr(1)* refers to the rest list at position 1 of a list, i.e. the list minus its first element.

[4]Dependencies arise when a feature value of type STRING is defined as the concatenation of the values of two other STRING features. The value of the *Phon* feature e.g. is defined in this way.

feature has been fixed, the corresponding context-free rule is output. The other context-free rules are obtained by backtracking.

Assuming that the feature *Number* is to be incorporated into the categories *NP*, *DT*, and *N*, the parser will generate in case of the rule

```
NP {Number=n;} -> DT {Number=n;}  ‘N {Number=n;};
```

the following two context-free rules:

```
NP_sg -> DT_sg N_sg
NP_pl -> DT_pl N_pl
```

3.2 Compression of the Lexicon

For a parser to be able to process arbitrary text it is essential to have a large lexicon with broad coverage. In order to reduce the space requirements of such a large lexicon, the compiler checks for redundancies. Most information is stored in the form of linked lists and if two lists are identical from some position up to the end, the common tail of the lists is stored only once. Also if two list elements (not necessarily of the same list) are identical, only one copy is stored. With this technique it was possible to compress a lexicon with 300,000 entries to about 18 MBytes, which is about 63 bytes per entry.

4 Parsing

The parser proper consists of two components. The first component is a context-free parser which generates a parse forest, i.e. a compact representation of a set of parse trees which stores common parts of the parse trees only once. The BCKY parser developed by Andreas Eisele[5] is used for this purpose. It is a fast bit-vector implementation of the Cocke-Kasami-Younger algorithm. The second component of the parser reads the context-free parse forest and computes the feature structures in several steps. In each step the parse forest is traversed and a new parse forest with more informative feature structures is generated. Parsing is finished when the feature structures do not change anymore. The first step is the most expensive one computationally since most analyses are typically eliminated in this step. The goal is therefore to make the first step as efficient as possible, rather than minimizing the number of steps.

The recomputation of the parse forest proceeds bottom-up and top-down in turn. During bottom-up processing, the parser first computes the feature structures of terminal nodes by evaluating the constraints associated with the lexical rules. Since the number of lexical rules for a terminal node can be larger than one, there may be more than one resulting feature structure. The new nodes with their feature structures are inserted into a new chart. If a node with the same category and feature structure already exists in the new chart, the parser just adds the new analysis (i.e. the rule number and pointers to the daughter nodes) to the list of analyses at this node. Otherwise, a new node is generated. In both cases, the parser stores a link from the old node to the new one. When the feature structure of a nonterminal node is computed, the parser checks all alternative analyses of this node one after the other. For each analysis it has to try out all combinations of the new nodes which are linked to its daughter nodes (cp. figure 1). For each consistent combination, the parser builds an updated feature structure for the mother node and inserts it into the new chart as in the case of terminal nodes. This method is analogous to the chart parsing techniques used in context-free parsing.

During top-down processing, the parser first copies all top-level nodes which cover the whole input string to the new chart and inserts them into a queue. Then the first node is retrieved from the queue and its daughter nodes are recomputed. The recomputed daughter nodes are inserted into the new chart and, if new, also inserted into the queue for recursive processing. After a traversal of the parse forest is completed, it is checked whether any node has changed. If not, parsing is finished. Otherwise the old chart is cleared, the charts are switched and the next processing step begins.

Why is it necessary to traverse the parse forest more than once? In contrast to formalisms like LFG and HPSG it is not assumed that the feature structure of the root node of a parse tree contains all relevant information[6]. Hence it is necessary to compute the feature structures of all nodes in a parse tree. If there were only one unambiguous parse tree, it would be sufficient to traverse the parse forest once. By means of value sharing it

[5]Andreas Eisele, IMS-CL, University of Stuttgart, andreas@ims.uni-stuttgart.de
[6]It is even assumed that this is not the case (cp. section 4.3).

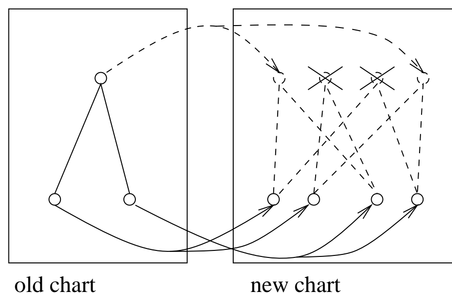

Figure 1: Recomputation of the parse forest

would be possible to update the values of unified features of different nodes in the parse tree synchronously. This is not possible in the case of parse forests, however, because cross-talk would result whenever two analyses have a common node, unless such a shared node is always copied before it is modified which is expensive and to no avail if the analysis later fails. Instead the parse forest is traversed again to update the feature structures of the non-root nodes.

The presented parser has to traverse the parse forest even more often because value sharing between different features of the same feature structure is not used, in order to keep data structures and algorithms as simple as possible. By repeated recomputation, information is properly propagated within the parse forest so that the correct result is obtained. This strategy might seem inefficient, but it turns out that the first two passes which are necessary in any case account for about three quarters of the total processing time and the number of passes seldom exceeds five. As mentioned earlier, it seems more important to speed up the first pass than to reduce the number of passes.

4.1 Constraint Evaluation

The recomputation of feature structures is carried out in four steps. First the input feature structures are specified. During top-down processing, the mother node is a node in the new chart and the daughter nodes are from the old chart. During bottom-up processing it is the mother node which is contained in the old chart and the daughter nodes are from the new chart. The parser then checks whether the fixed assignments are compatible with the input feature structures. If this is the case, the parser computes the values of the variables used in this rule by non-destructively unifying the values specified in the constraint equations for this variable (cp. section 3). The values to be unified are either values of feature paths, or constants[7], or results of string concatenation operations.

Once the values of the variables have been computed, the new feature structures are built by modifying the old feature structures according to the set of fixed assignments and variable assignments. The assignments have been sorted by the compiler so that assignments to less deeply embedded features are carried out first. A lazy copying strategy is used: Before the value of a feature is changed, all levels of the feature structure above this feature are copied unless they have been copied before. After all assignments have been made, the resulting feature structure is inserted into a hash table. If an identical feature structure is already contained in the hash table, a pointer to this feature structure is returned. Otherwise, the new feature structure is inserted. The hashing is done recursively: All embedded feature structures (i.e. elements of feature structure lists), are hashed before an embedding feature structure is hashed. Hashing simplifies the comparison of feature structures to a mere comparison of pointers.

4.2 Optimization

The parsing scheme presented so far has been modified in several ways in order to improve the speed of the parser.

1. The compatibility check for fixed assignments can be done for each node independently of all the other nodes. It is not necessary to repeat it for all combinations of daughter nodes. If a feature structure turns out to be

[7] Such constants are necessarily disjunctive values because otherwise a set of fixed assignments would have been generated.

incompatible, the number of combinations is reduced.

2. The probability that a constraint fails is not identical for all constraints in a rule. Therefore the constraints are sorted so that those constraints which are more likely to fail will be checked first. Inconsistent analyses are therefore eliminated earlier on average. The statistics are collected during parsing.

3. Sometimes it is known in advance that a recomputation of a feature structure will not change its content. This is the case if all input feature structures remained unchanged when they were recomputed the last time. In this case it is sufficient to copy the feature structures to the new chart without recomputing them.

4. Expensive computations are sometimes done repeatedly during parsing, e.g. feature structure unifications. In order to avoid this redundancy, the parser stores each unification operation with pointers to the argument feature structures and the resulting feature structure in a hash table. Before a unification of two feature structures is carried out it is checked whether the result is already in the hash table. Other expensive operations like string concatenations are stored as well.

5. The parser generates a large number of data structures dynamically. In order to avoid the overhead associated with memory allocation calls to the operating system, the parser uses its own simple memory management system which allocates memory from the operating system in large chunks and supplies it to other functions in smaller chunks as needed. Once a sentence has been parsed the allocated memory is freed in one step.

The parser and the compiler have been implemented in the C programming language.

4.3 Limitations

Only the first two processing steps discussed in section 1 – context-free parsing and processing of filtering constraints – have been implemented in the parser so far. In order to be able to build a semantic representation, it would be necessary to add another step which processes structure-building constraints efficiently. The algorithm presented in [Maxwell III and Kaplan, 1996] could be used for this purpose. An even better alternative is Dörre's algorithm [Dörre, 1997] which has polynomial complexity but only works if the constraints never fail. If alternative parse trees are scored after parsing, e.g. with a probabilistic model, semantic construction could also be confined to the best analyses.

The presented parsing method cannot immediately be used to process other grammar formalisms like LFG or HPSG. LFG has no feature typing which is essential for the compilation of the context-free grammar. A separation of filtering constraints and structure-building constraints is difficult in LFG because the SUBJ and OBJ features are used to check subcategorization and to build a simple semantic representation at the same time. The pruning strategy outlined in section 1 might help, but an additional module for the processing of structure-building constraints would still be needed. Of course, other modifications would also be necessary.

The main problem when parsing HPSG with the presented method is to obtain a rule-based grammar from the principle-based representation. Apart from this it would be necessary to emulate the type hierarchy with features. The head features could be partially compiled into the context-free grammar. It is not necessary to compute the DAUGHTERS feature because the tree structure is already represented in the chart. The computation of the features which store the semantic information would have to be done by an additional module.

5 Experimental Results

An English grammar with 290 phrase structure rules[8] has been written for the parser. A lexicon of about 300,000 entries with subcategorization information was extracted from the COMLEX lexical database [Grishman et al., 1994]. The parser has been used to parse 30,000 sentences from the Penn Treebank corpus [Marcus et al., 1993]. Missing lexical entries were automatically generated from the part-of-speech tags in the tagged version of the corpus. However, the part-of-speech tags were not used for parsing itself. Quotation marks were ignored during parsing. More than 7 words per second were parsed on average with a Sun Ultra-2

[8] About 90 rules only deal with coordination, quotation and punctuation.

workstation. Three times the parser stopped prematurely due to memory exhaustion. The calculation of the feature structures was the most time-consuming part of parsing.

For 80 percent of the sentences the parser produced at least one analysis. For 54 percent of the sentences there was at least one analysis which was compatible with the Penn Treebank analysis. An analysis was considered compatible if there were no crossing brackets. However, analyses without crossing brackets are not necessarily acceptable analyses. 100 sentences have been parsed and inspected manually to estimate how often there was an acceptable analysis. For 57 of these sentences the parser had produced a Treebank-compatible analysis, but for only 48 an acceptable one. Interpolating these results, the portion of sentences with an acceptable analysis is probably around 45 percent in the larger corpus.

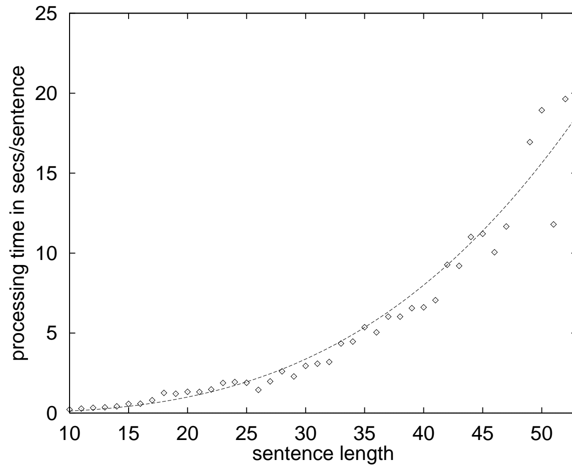

Figure 2: Empirical parsing complexity

Figure 2 shows the empirical parsing complexity which is close to n^3 (the dashed line in the diagram) where n is the sentence length[9].

strategy	25 sentences	1 complex sent.
all optimizations	65.9	180
no hashing of unifications	67.4	193
no hashing of string concatenations	79.3	244
recomputing always	67.3	236

Table 1: Parsing times for 25 randomly selected sentences and a single complex sentence

Another experiment was carried out to check the influence of some of the optimization strategies described in section 4.2 on parsing time. A randomly selected set of 25 sentences was parsed with different variants of the parser in the first part of the experiment. In the second part a single complex sentence was parsed. In each variant of the parser one optimization was switched off. Table 1 shows the results. Hashing of unifications only showed minor effects on parsing speed. Hashing of string concatenation operations was more effective. Presumably string concatenation operations are more likely to be repeated than feature structure unifications. Avoiding unnecessary recomputation of feature structures had a bigger influence on the parsing of the complex sentence than on the parsing of the simpler sentences.

The impact of the incorporation of features into the context-free grammar has also been examined. We observed in contrast to Maxwell and Kaplan [Maxwell III and Kaplan, 1994], only a marginal speedup of about 3 percent from feature incorporation. The incorporation of some features let to disastrous results because the parse forest generated by the context-free parser became very big, slowing down both context-free parsing and the calculation of the feature structures. A close relationship between the number of nodes in the context-free parse forest and parsing time has been observed.

[9] There is an outlier at (48, 36.7) which is not shown in the diagram.

The parser was also compared to a state-of-the-art parser, the XLE system developed at Rank Xerox which was available for the experiments. A corpus of 700 words which both parsers have been able to parse completely was used in this experiment. The XLE system parsed this corpus in 110 seconds whereas our parser needed 123 seconds. Of course it is very difficult to compare these figures since the parsers are too different wrt. the grammar formalisms used, the information contained in the analyses, the degree of ambiguity and other criteria.

6 Summary

A parsing strategy has been outlined which splits parsing into three steps: context-free parsing, evaluation of filtering constraints and evaluation of structure-building constraints. A parser has been presented which implements the first two of these steps. A compiler is used to transform grammar descriptions into a form which the parser is able to process efficiently. The compiler automatically refines the context-free backbone of the grammar by compiling a user-defined set of feature constraints into the context-free backbone. An iterative procedure is used to compute feature structures in disjunctive normal form after a context-free parse forest has been built. As long as the feature structures are not used to build representations which encode the structure of constituents, this parsing strategy works very well: Wall Street Journal data has been parsed at a speed of 7 words per second.

References

[Dörre, 1997] Dörre, J. (1997). Efficient construction of underspecified semantics under massive ambiguity. submitted to ACL'97.

[Dörre and Eisele, 1990] Dörre, J. and Eisele, A. (1990). Feature logic with disjunctive unification. In *Proceedings of the 13th International Conference on Computational Linguistics*, pages 100–105, Helsinki, Finland.

[Emele, 1991] Emele, M. (1991). Unification with lazy non-redundant copying. In *Proceedings of the 29th Annual Meeting of the Association for Computational Linguistics*, pages 323–330, Berkeley.

[Grishman et al., 1994] Grishman, R., Macleod, C., and Meyers, A. (1994). Comlex syntax: Building a computational lexicon. In *Proceedings of the 15th International Conference on Computational Linguistics*, Kyoto, Japan.

[Kasper, 1987] Kasper, R. T. (1987). A unification method for disjunctive feature descriptions. In *Proceedings of the 25th Annual Meeting of the ACL*, pages 235–242, Stanford, CA.

[Kasper and Krieger, 1996] Kasper, W. and Krieger, H.-U. (1996). Modularizing codescriptive grammars for efficient parsing. In *Proceedings of the 16th International Conference on Computational Linguistics*, pages 628–633, Copenhagen, Denmark.

[Marcus et al., 1993] Marcus, M. P., Santorini, B., and Marcinkiewicz, M. A. (1993). Building a large annotated corpus of English: the Penn Treebank. *Computational Linguistics*, 19(2):313–330.

[Maxwell III and Kaplan, 1994] Maxwell III, J. T. and Kaplan, R. M. (1994). The interface between phrasal and functional constraints. *Computational Linguistics*, 19(4):571–589.

[Maxwell III and Kaplan, 1996] Maxwell III, J. T. and Kaplan, R. M. (1996). Unification-based parsers that automatically take advantage of context freeness. Draft.

[Schiehlen, 1996] Schiehlen, M. (1996). Semantic construction from parse forests. In *Proceedings of the 16th International Conference on Computational Linguistics*, Copenhagen, Denmark.

[Simpkins, 1994] Simpkins, N. K. (1994). *ALEP-2 User Guide*. CEU, Luxembourg. This document is online available at http://www.anite-systems.lu/alep/doc/index.html.

[Younger, 1967] Younger, D. H. (1967). Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10:189–208.

A A Toy Grammar

```
% comments start with a percent sign

%%%%%% declarations %%%%%%%%%%%%%%%

% definition of the automatic
% feature 'Phon'
auto Phon;

% enumeration type features
enum PERSON  {1st,2nd,3rd};
enum NUMBER  {sg,pl};
enum CASE    {nom,acc};
enum VFORM   {fin,inf,bse,prp,pap,pas};
enum BOOLEAN {yes,no};

% definition of a structured feature
struct AGR {
  NUMBER  Number;
  PERSON  Person;
  CASE    Case;
};

% category definitions
category TOP {
};

category COMP {
};

category SBAR {
  BOOLEAN Wh;
};

category S {
  FS_LIST Slash;
};

category VP {
  VFORM   VForm;
  BOOLEAN Aux;
  FS_LIST Subcat;
  FS_LIST Slash;
};

% Definition of the features which
% are to be compiled into the context
%  free grammar.

VP incorporates {VForm, Aux};

category VBAR {
  VFORM   VForm;
  FS_LIST Subcat;
  FS_LIST Slash;
};

VP incorporates {VForm};

category V {
  VFORM   VForm;
  BOOLEAN Aux;
  FS_LIST Subcat;
};

VP incorporates {VForm,Aux};

category NP {
  BOOLEAN Wh;
  AGR     Agr;
};

NP incorporates {Wh};

category N {
  AGR     Agr;
};

category DT {
  BOOLEAN Wh;
  AGR     Agr;
};

DT incorporates {Wh};

category PP {
};

category P {
};

% definition of restrictor types
restrictor+ NP_R(NP) {Phon, Wh, Agr};

% In the next definition, the Phon
% feature is exempted from unification.
restrictor+ NP2_R(NP) {Wh, Agr};
restrictor+ SBAR_R(SBAR) {Phon, Wh};

% variable declarations
BOOLEAN wh;
AGR agr;
NP_R np;
NP2_R np2;
SBAR_R sbar;
FS_LIST r, r2;

%%%%%% grammar rules %%%%%%%%%%%%%%%

TOP {} ->
    `S {Slash=[];};

S {} ->
    NP {Agr.Case=nom;}=np
   `VP {Subcat=[NP{}=np];};
% The subject-NP is unified with the
% single element of the Subcat list.

VP {} ->     % All features of the two
   `VP {}    % VP nodes are unified due
    PP {};   % to feature inheritance.

VP {} ->
   `VBAR {};

VBAR {} ->
   `VBAR {}
    PP   {};

VBAR {Subcat=r;} ->
   `VBAR {Subcat=[NP{}=np|r];}
    NP   {Agr.Case=acc;}=np;
% All features of the VBAR nodes are
% unified by default feature inheritance
```

```
% excepted the Subcat features.

VBAR {Subcat=r;} ->
   ‘VBAR {Subcat=[SBAR{}=sbar|r];}
    SBAR {}=sbar;

VBAR {Slash=[];} ->
   ‘V {};

PP {} ->
   ‘P  {}
    NP {Agr.Case=acc;};

NP {Wh=wh;} ->
    DT {Wh=wh;Agr=agr;}
   ‘N  {Agr=agr;};

SBAR {Wh=no;} ->
    COMP {}
      ‘S {Slash=[];};

SBAR {Wh=yes;} ->
    NP {Wh=yes;}=np2
   ‘S  {Slash=[NP{}=np2];};
% All features of the NP node and the
% element on the Slash list are unified
% excepted the Phon feature.
% See the definition of NP2_R.

VBAR {Subcat=r;Slash=[np|r2];} ->
   ‘VBAR {Subcat=[NP{}=np|r];Slash=r2;}
    NP*  {Agr.Case=acc;}=np;
% An NP trace is generated. Information
% from the filler node is threaded via
% the Slash feature.


%%%%%% template definitions %%%%%%%%%%

N_sg   : N {Agr.Number=sg;};
PRO    : NP {Agr.Number=sg;
             Agr.Person=3rd;};
NPRO   : PRO {Wh=no;};
WHPRO  : PRO {Wh=yes;};

%%%%%% lexical entries %%%%%%%%%%%%%%%

"the"  : DT {Wh=no;Agr.Person=3rd;};
"a"    : DT {Wh=no;Agr.Number=sg;
                   Agr.Person=3rd;};
"which": DT {Wh=yes;Agr.Person=3rd;};
"man"  : N_sg {};
"pizza": N_sg {};
"restaurant": N_sg {};
"he"   : NPRO {Agr.Case=nom;};
"him"  : NPRO {Agr.Case=acc;};
"it"   : NPRO {};
"what" : WHPRO {};
"eats" : V {Subcat=[
            NP{},
            NP{Agr.Number=sg;
               Agr.Person=3rd;}];};
"at"   : P {};
"that" : COMP {};
```