Grammar Compaction and Computation Sharing in Automaton-based Parsing

John Carroll, Nicolas Nicolov, Olga Shaumyan, Martine Smets & David Weir
School of Cognitive & Computing Sciences
University of Sussex
Brighton BN1 9HQ, UK
{johnca,nicolas,olgas,martines,davidw}@cogs.susx.ac.uk

Abstract

Wide-coverage grammars in Lexicalised Tree-Adjoining Grammar (LTAG) and related formalisms are structurally complex, containing many hundreds of elementary trees. In the context of the development of a full-scale LTAG-like grammar and parsing system, we have investigated the claim that because many of these trees have a great deal of structure in common, a parser that manipulates trees individually performs a considerable amount of redundant computation. This claim has been used to motivate a parsing technique that encodes trees as finite state automata and captures overlapping computation through automata minimization. Our preliminary results show that this technique leads to considerable computation sharing.

1 Introduction

The paper presents work that forms part of the ongoing LexSys project[1]. Our overall aim is to bring together and test in practice a variety of current NLP techniques, including the organisation of grammars into inheritance hierarchies for compact representation, exploitation of diverse precompilation techniques for efficient parsing, and use of statistical analysis to disambiguate parse results. In conjunction with this we are using several existing tools and resources, such as the lexicon developed in the Alvey Natural Language Tools project (Briscoe et al., 1987), lexical frequency information from the SPARKLE project[2], and an established lexical knowledge representation language DATR (Evans and Gazdar, 1996) to represent the grammar.

We use the Lexicalised D-Tree Grammar (LDTG) formalism (Rambow, Vijay-Shanker, and Weir, 1995), which is related to the Lexicalised Tree Adjoining Grammar (LTAG) formalism. In LTAG, LDTG, and other related formalisms, grammars contain a finite set of basic structures and these are referred to as the **elementary structures** of the grammar. The elementary structures of LDTG are called **d-trees** which are combined with various composition operations to create phrase-structure trees. Grammars are lexicalised in the sense that each elementary structure is associated with at least one lexical items. Further details of LDTG are given in Section 2.

Several advantages arise from the use of LDTG for parsing: (i) elementary structures have an extended domain of locality and allow certain kinds of dependencies to be stated directly, which in other frameworks can only be enforced by a mechanism of percolation of

1. This work is supported by UK EPSRC project GR/K97400 and by an EPSRC Advanced Fellowship to the first author. We would like to thank Roger Evans, Gerald Gazdar & K. Vijay-Shanker for helpful discussions.

2. CEC Telematics Applications Programme project LE1-2111 "SPARKLE: Shallow PARsing and Knowledge extraction for Language Engineering" http://www.ilc.pi.cnr.it/sparkle.html.

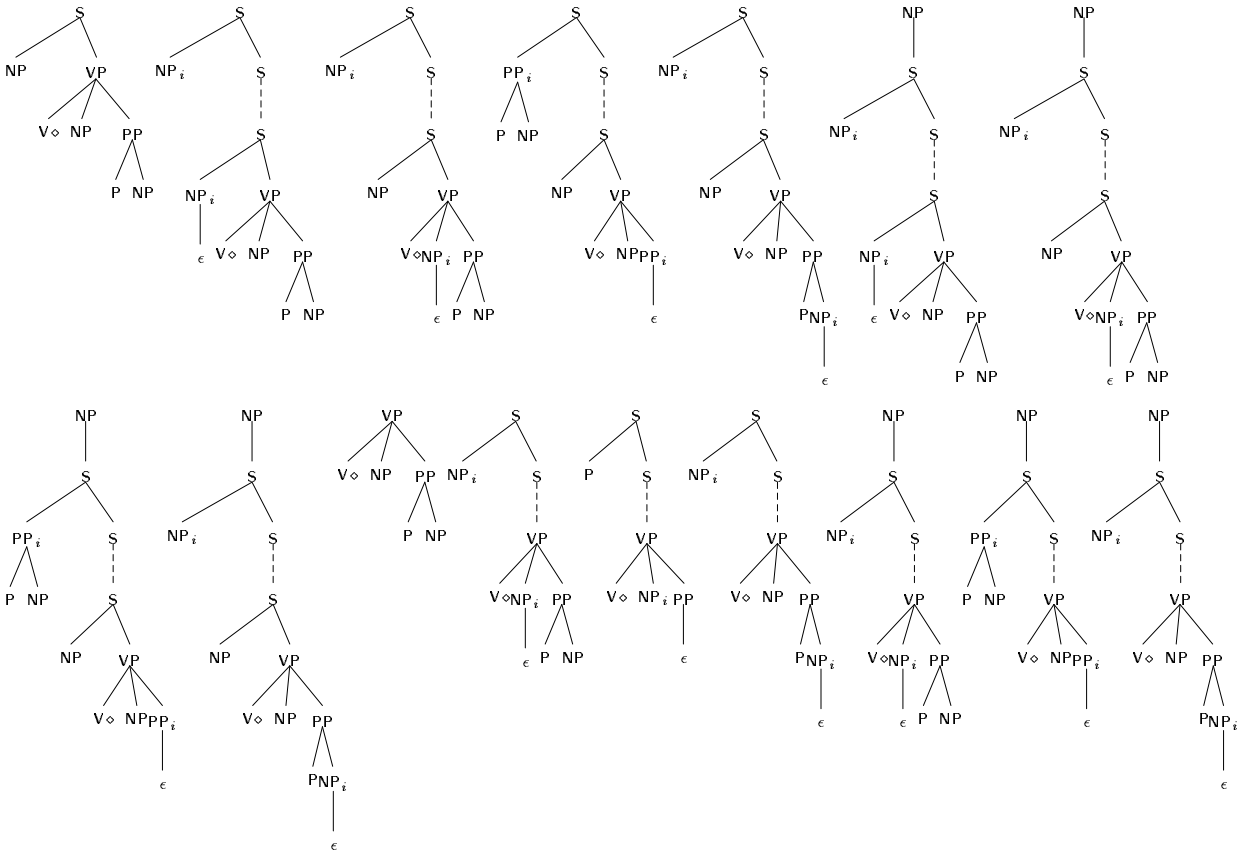

Figure 1: Selected d-trees from the `V_NP_PP_to` family

feature values; (ii) a LDTG parser need only consider the lexicalised d-trees corresponding to the input words; (iii) LDTG derivation structures directly reflect predicate argument structure due to the uniform treatment of complementation and modification.

Despite these advantages, building an efficient LDTG or LTAG parser for a *wide-coverage* grammar represents a challenge. Each word in the input string introduces a large number of elementary structures (d-trees) into the parse table: one for each of its possible alternative readings.

In the current grammar the words *come*, *break* and *give* anchor around 130, 180 and 340 d-trees, respectively. In fact, if we include d-trees for all alternative feature values, these figures rise by about one order of magnitude. The d-trees in Figure 1 represent a small sample of the d-trees that the word *give* can anchor[3]. As can be seen from Figure 1, there can be substantial overlap in structure among the d-trees associated with a given input word (see for example, the similarity between the d-trees for relative clause and wh-extraction). Existing LTAG parsing algorithms (such as Schabes (1988) and Vijay-Shanker and Weir (1993)) treat each tree as independent, which results in considerable duplication of processing of common structure during parsing. Evans and Weir (1997; 1998) describe a technique aimed at making LTAG parsing more efficient (empirically if not formally), by exploiting the possibility of shared processing that derives from shared structure.

The work presented in this paper uses the wide-coverage LDTG developed as part of the LexSys project to present a practical evaluation and justification of a claim made by Evans and Weir (1997; 1998). In particular, we quan-

3. The figure contains the base d-tree for ditransitive verbs; 4 d-trees for various cases of *wh*-extraction; 4 d-trees for various relative clauses; a VP complement d-tree; and finally, 3 *wh*-d-trees and 3 relative clause d-trees derived from the VP complement d-tree.

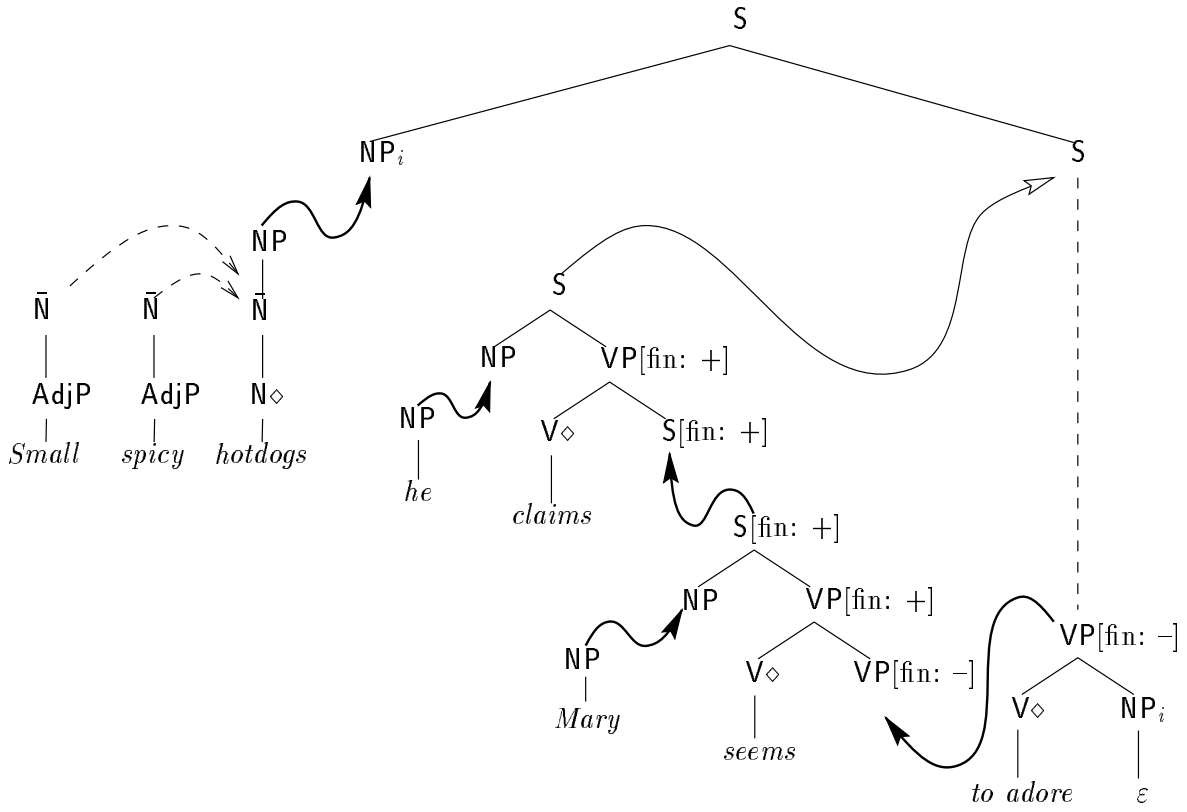

Figure 2: Example derivation

tify the extent to which the parser's processing is duplicated due to the common structure among the d-trees that the LexSys grammar associates with a word. Before presenting our results, in Section 5, we describe the LDTG formalism in Section 2 and the LexSys architecture in Section 3. In Section 4 we describe a technique introduced by Evans and Weir (1997; 1998) aimed at reducing the number of overlapping computations due to similarities among the alternative d-trees associated with words in the input.

2 Brief introduction to LDTG

The primitive elements of LDTG are called elementary d-trees and are combined together to form larger structures during a derivation. Although, for convenience, we present d-trees graphically as though they were conventional trees, they are more correctly thought of as expressions in a tree description logic (Rogers and Vijay-Shanker, 1992). These expressions *partially* describe trees by asserting various relationships between nodes: parenthood, domination, precedence (indicating that one node is to the left of another), equality and inequality.

When a d-tree is written graphically, we use two types of edges: domination edges (**d-edges**) and parent edges (**p-edges**). D-edges (represented graphically with a broken line) and p-edges (represented with a solid line) are not distributed arbitrarily in d-trees. For each internal node, either all of its daughters are linked by p-edges or it has a single daughter that is linked to it by a d-edge. Each node is labelled with a terminal symbol, a nonterminal symbol or the empty string. A d-tree containing n d-edges can be decomposed into $n + 1$ **components** containing only p-edges. Several d-trees are shown in Figure 2. The d-tree with anchor *to adore*, for example, has two components: the lower component contains the anchor of the d-tree; and the upper component

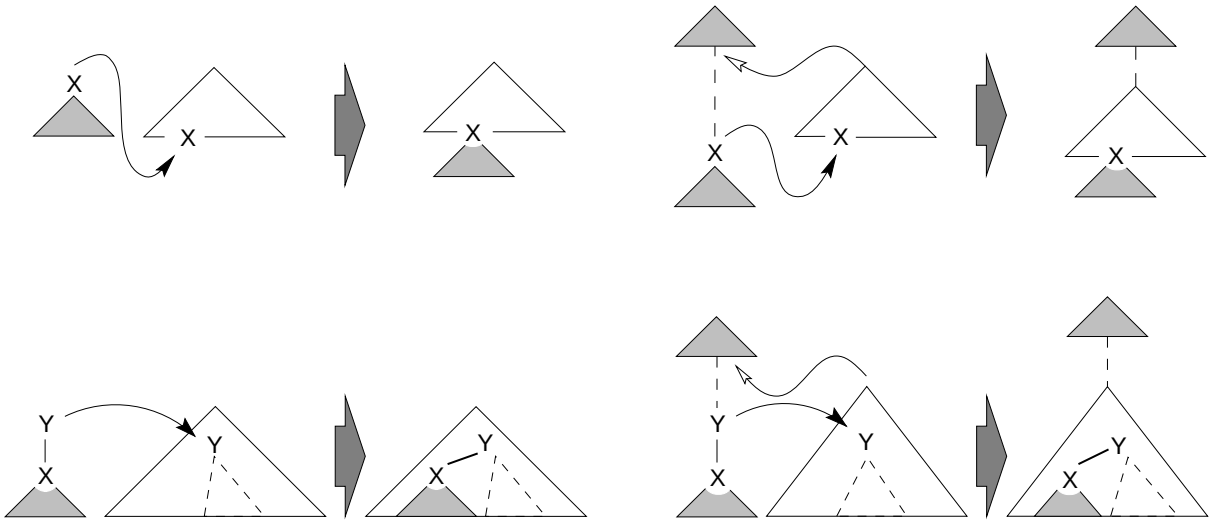

Figure 3: Composition operations

contains a frontier node for the extracted NP (which in this case will be filled with a d-tree for *hotdogs*).

There are two substitution-like operations for composing d-trees, both of which involve combining two descriptions while equating exactly one node from each description. One of the operations is always used to add complements and involves equating a frontier node (in the d-tree that is getting the complement) with the root of some component (in the d-tree that is providing the complement), such that the two nodes being equated are compatible. Two schematic examples of this operation are shown at the top of Figure 3. These are the two cases that appear in our grammar for English[4]: at the top left is the case in which the entire complement d-tree appears below the point of substitution; the top right gives the case in which the complement involves extraction where the extracted component is placed at the top of the d-tree. Several concrete examples are shown in Figure 2. For example, the d-tree for *to adore* is composed with the d-tree for *seems* by equating the two nodes labelled VP[fin: –]. The top component of the *to adore* d-tree can then be fitted into the resulting d-tree in a way that is compatible with the result of combining the two descriptions and equating the two VP[fin: –] nodes. In this particular derivation, this is not done until after the *claims* d-tree has been used. The nodes labelled S[fin: +] are equated when the *seems to adore* d-tree is composed with the *claims* d-tree; the two other S nodes are then related by a d-edge and can be equated. However, in general, equating nodes related by a d-edge takes place at any point in the derivation and need not be thought of as being part of the composition operation. We view it as being part of the process of relating the partial tree description produced in a derivation with the "minimal" trees that it partially describes.

A second operation is used to add modifiers. In terms of tree descriptions, this operation is similar to the complement-adding operation since it also involves combining two d-trees while equating a pair of nodes. In this case, however, it involves equating an *internal* node (in the d-tree that is getting the modifier) with the root of some component (in the d-tree that is providing the modifier), such that the two nodes being equated are compatible. Two schematic examples are shown at the bottom of Figure 3. As in the case of the complement-adding operation, these are the two cases that

4. The general case is explained in Rambow, Vijay-Shanker and Weir (1995).

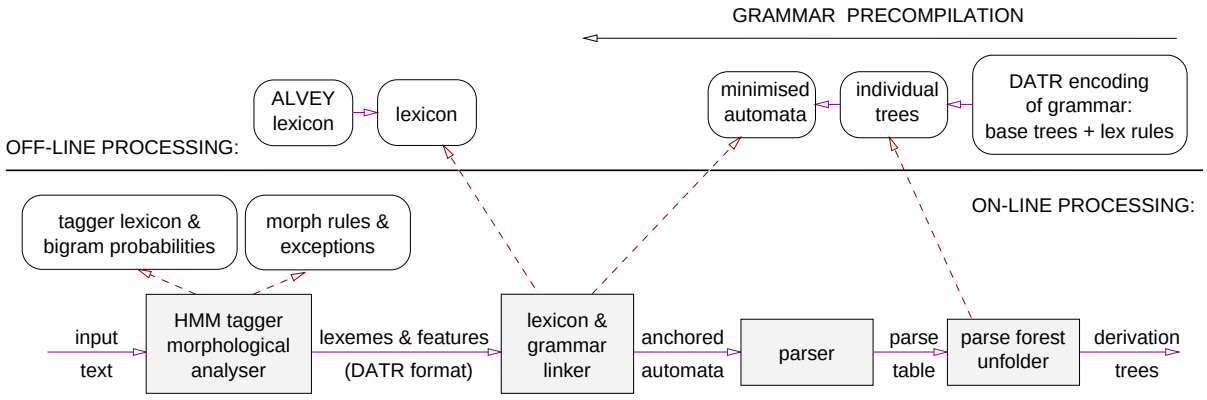

Figure 4: System architecture

appear in our grammar for English: at the bottom left is the case in which the entire modifying d-tree appears below the point of modification; the bottom right gives the case in which the modifier involves extraction, where the extracted component is placed at the top of the d-tree[5]. Two examples are shown in Figure 2 where d-trees for *Small* and *spicy* are composed with the d-tree for *hotdogs*.

We now describe the architecture of LexSys including a discussion of the wide-coverage LDTG for English.

3 LexSys architecture

The overall architecture of LexSys is shown in Figure 4. In the following subsections we describe each of the major components.

3.1 The morphological analyser

We first *tokenise* the text, and apply a *sentence-splitter* to determine likely sentence boundaries. We then assign extended part-of-speech (PoS) labels to each word in the resulting sentences using a first-order HMM *tagger* (Elworthy, 1994) trained on the SUSANNE corpus (Sampson, 1995). We employ three techniques to improve tagger robustness and accuracy: (1) the lexicon is augmented with open-class words from the LOB corpus; (2) the tagger incorporates a part-of-speech guesser that empirically achieves around 85% label assignment accuracy for unknown words; and (3) for each word the tagger returns multiple label hypotheses, but filters out any whose probabilities are below a preset factor of the most probable. We thus attempt to minimise the effect of incorrect tagging on subsequent parser components by allowing label ambiguities, but control the increase in indeterminacy and concomitant decrease in subsequent processing efficiency by applying the thresholding technique[6].

After tagging, a *lemmatiser* finds the lemma, or base form, corresponding to each word-label pair[7]. Here we use an enhanced version of the GATE project stemmer (Cunningham, Gaizauskas, and Wilks, 1995). Finally, the lemma and PoS label for each word in the sentence is combined with syntactic information derived from the word's morphological form (e.g. number for nouns).

3.2 The grammar

In a lexicalised grammar, linguistic generalisation is integrated with the lexical items. The

5. In the examples shown at the bottom of Figure 3 the modifier d-tree is placed to the left of the subtree it modifies. It is also possible for modification to take place on the right.

6. On SUSANNE, retagging allowing only a single label per word results in a 97.9% label/word assignment accuracy, whereas multi-label tagging with our thresholding scheme results in 99.5% accuracy.

7. In English this process is deterministic except for a very small number of exceptions.

LEXSYS lexicon contains information about idiosyncratic combinability of particular words and is described in Section 3.3. In this section we describe the grammar which contains higher linguistic generalisation about the syntactic functionality of part of speech classes and wordforms.

The grammar used in LEXSYS is encoded using the lexical knowledge representation language DATR. The encoding scheme we use is based on that first proposed for LTAG by Evans, Gazdar and Weir (1995). It achieves a compact encoding through the use of lexical rules and inheritance.

Inheritance. D-trees are encoded in an anchor-up fashion as illustrated on the right of Figure 5. D-trees rooted in the same node share many structural and syntactic features: this allows the grammar to be represented as an inheritance hierarchy. For example all finite verbs are of category V, have VP as their parent, S as the root node of their d-tree, have a subject (except for imperatives), and so on. These features are specified in the definition of the d-tree for intransitive verbs, and inherited by all verbs. Exceptions can be easily formulated using DATR non-monotonic inheritance mechanism.

Lexical Rules. We use the term "lexical rules" to refer to all types of morphosyntactic alternation. Lexical rules relate d-trees belonging to the same d-tree family (e.g. the transitive verb d-tree family includes amongst others the basic d-tree, passive d-tree, wh-question d-tree, etc.)[8]. Rules themselves are organised in a hierarchy, which obviates the need to duplicate information common to several rules. For example, at the top of the hierarchy for movement rules (which include *wh*-questions, relative clauses and topicalisation) is the *topic* rule, which specifies the top structure of the derived d-tree (where the 'extracted' element is localised). The rule *wh* inherits from the rule *topic* and further specifies that the 'extracted' element is of type *wh*-word. The rules *whsubj*, *whobj1* and *whobj2* inherit from the rule *wh* and specify the null category coindexed with the 'extracted' element: the subject of the clause, the first complement, and the second complement, respectively. Rules can apply to basic d-trees by themselves or in combination. Thus the *passive* rule and the *whsubj* rule can apply successively to the basic transitive d-tree (as in *What was seen?*). Possible rule combinations are inferred automatically from the set of ordered rules defined for each basic d-tree.

3.3 The lexicon

The current lexicon is a reworked version of the Alvey Natural Language Tools lexicon (Carroll and Grover, 1989) (itself derived semi-automatically from the machine-readable version of the Longman Dictionary of Contemporary English). In accordance with our grammar requirements we have transformed the lexicon file containing the (GPSG-based) category and feature assignments into DATR notation, which is also used for encoding the grammar and the results of morphological analysis. In theory, this uniform notation would permit the lexicon to form the leaf nodes in the grammar hierarchy and so inherit automatically any of the syntactic information (such as default feature assignment) contained there. However, at the moment, the lexicon interfaces with the grammar via a module which uses unification to link up lexical entries with d-trees of the grammar.

The lexicon contains only lexemes, with wordform information supplied by the morphological analyser. A preliminary analysis of the representative lexicon (chosen to contain a sample of words exhibiting different selectional behaviour) shows that, the number of d-tree families that a word anchors is on average: nouns 2.04; adjectives 2.96; verbs 6.36 (where a d-tree family may contain in excess of 100 d-trees). It should be noted that the morphological form of a linguistic datum affects how much of a family is selected: so the *ing* form of the verb will not inherit all of the d-trees associated with the base form of the verb

8. The approach to lexical rules that we have adopted was proposed by Roger Evans (personal communication), and is an alternative to the approach described by Evans et al. (1995).

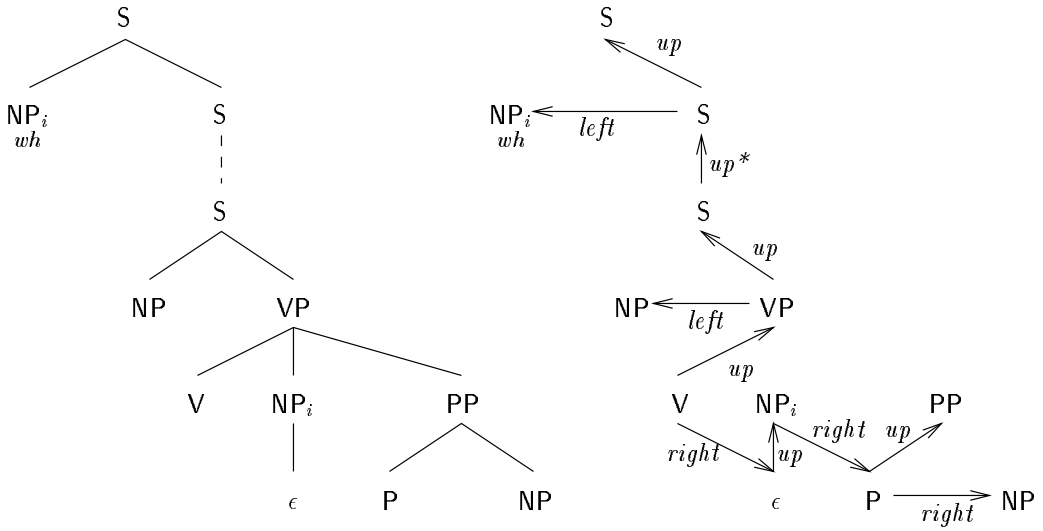

Figure 5: A d-tree and its anchor-up encoding

(the passive alternation, for example) but, in addition, will select the N-bar d-trees.

3.4 The parser

In this section we complete the description of LexSys by outlining a simple bottom-up parsing algorithm that we have implemented and currently use in our grammar development. We do not give details of the parsing algorithm because we limit ourselves to material that is needed in later sections of the paper.

The anchor-up encoding of the elementary d-tree illustrated in Figure 5 mirrors the computation performed by a bottom-up parser. In recognizing a d-tree the parser simulates a traversal of this d-tree. As in the encoding shown in Figure 5, this traversal begins at the anchor node with the parser working outwards (on both the left and right) as it moves upwards towards the root of the d-tree. When visiting nodes during this traversal, the parser must perform various actions. Which particular action is required at each node is determined by the type of node (e.g., whether it is a frontier or internal node) and its position relative to the anchor (whether it is to the right or left of the anchor). For example, in order to recognise the d-tree in Figure 5 the parser must first find evidence that a P and then an NP complement appear to the right; allow for any number of modifications at the PP and VP nodes; find an NP complement to the left; suspend traversal of the d-tree at an S node; resume traversal of the d-tree at an S node; and finally allow for any number of modifications at the S node. We refer to each step in this sequence as a **parser action** and to a sequence of parser actions associated with a d-tree, as an **elementary computation** of that d-tree.

As shown in Figure 4, prior to parsing, each word of the input is associated with each of the d-trees that it can potentially anchor. Each elementary d-tree in the grammar can be precompiled into a (flat) sequence of parser actions. These sequences, rather than the d-trees themselves, are the objects that the parser manipulates during parsing. The parser fills a 2-dimensional table (where each cell corresponds to a substring of the input) by advancing through these parser action sequences as actions are executed. In addition to action sequences, the items in cells contain multisets that hold suspended action sequences. In LTAG, adjunction has the effect of embedding one elementary d-tree within another, and a stack can be used by a parser to control the unbounded nesting of elementary d-trees that can occur

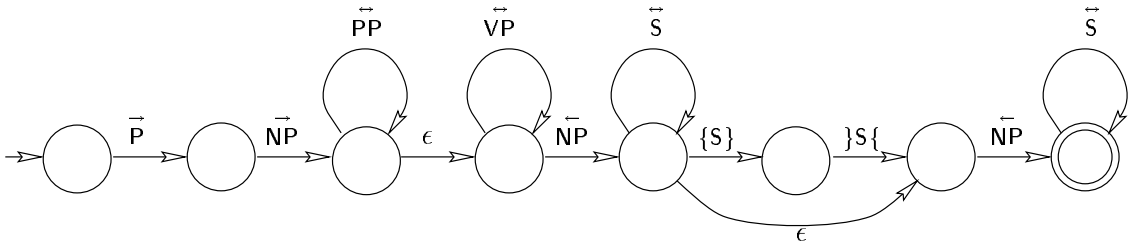

Figure 6: Automaton for d-tree in Figure 5

in derivations. LDTG also allows embedding of d-trees; however, multisets rather than stacks are used to control this embedding. This difference is due to the limited control provided by LDTG over the relative positioning of the inserted components of two composed d-trees. Multisets are manipulated by parser actions at d-edges since that is where components can be inserted. Our use of multisets is similar to that presented in Vijay-Shanker, Weir and Rambow (1995).

4 Merging computations

The sequence of parser actions associated with an elementary d-tree can be represented as a finite state machine. An example for the d-tree in Figure 5 is shown in Figure 6. The input symbols of an automaton (the edge labels) denote parser actions, and the strings accepted by an automaton are elementary computations of the d-tree from which the automaton is produced. The actions $\vec{A}$ and $\overleftarrow{A}$ indicate composition with a complement d-tree for category A to the left or right, respectively; $\overleftrightarrow{A}$ indicates composition of a modifying d-tree for category A (note that since this operation can be performed zero or more times at a node, we have a loop for this action in the automaton); and the actions $\{A\}$ and $\}A\{$ indicate that traversal of the d-tree at a node labelled A should be suspended and resumed, respectively. The ϵ-transition makes it possible to skip the $\{A\}$ and $\}A\{$ actions, and corresponds to equating the two A nodes.

In the introduction we claimed that by observing the d-trees in Figure 1 it should be obvious that similarities among d-trees would give rise to overlapping computation by the parser. If d-trees associated with a word in the input are going to cause the parser to perform repetitious processing, this will show up in the d-trees' elementary computations, since they capture elementary d-trees from the parser's point of view at a suitable level of abstraction. Evans and Weir (1997; 1998) propose that a significant amount of overlapping among elementary computations can be pre-compiled out by performing the following steps: (1) compile sets of elementary d-trees that can be associated with input words into sets of automata; (2) for each of these sets, merge the automata in the set into a single automaton by introducing a new initial state with ϵ-transitions from this new state to the initial states of the old automata; (3) minimise the number of states in the merged automaton (using standard techniques); and (4) rather than a set of d-trees, associate a single minimised automaton with each input word and parse as normal.

5 Preliminary evaluation

Although the approach described in the previous section does not improve the worst-case complexity of the parser, it may improve performance in practice. Having developed a wide-coverage LDTG, we are in the process of testing this empirically and, in this section, present some preliminary results. In particular, we have explored how much overlapping computation is captured by the pre-compilation process described in Section 4.

Table 1 shows the amount of compaction

word	*number of d-trees*	*automaton*	*number of states*	*number of transitions*	*average d-trees per state*
come	133	merged	898	1130	1
		minimised	50	130	11.86
break	177	merged	1240	1587	1
		minimised	68	182	12.13
give	337	merged	2494	3177	1
		minimised	83	233	20.25

Table 1: Results

for sets of d-trees anchoring *come*, *break* and *give*. Column 2 gives the number of trees used in the experiment. These numbers reflect the sets of trees selected by the base form of the actual lexical entries for these verbs, without regard for different possible feature value assignments. For each word there are two automata: the merged automaton (the automaton created by combining the individual automata for each d-tree) and the minimized automaton. Columns 4 and 5 give the number of states and transitions, respectively, in the merged and minimized automaton. In order to make this a fair comparison, we minimized each of the individual automata prior to merging them. In the final column we give the average number of different d-trees that share a state. Every state (except the initial state) in the merged automaton corresponds to a state in exactly one individual d-tree's automata, whereas, states in the minimised automaton are produced by combining equivalent states from the merged automaton.

The figures given in Table 1 understate the value of the computation sharing technique. The number of trees that a parser would have to consider for each of these words would be far greater: multiple instances of each d-tree would be included due to alternative settings of various features (as specified in the lexicon). For example, in the case of *give* the total number rises to in excess of 2000 d-trees. However, the feature values that are involved are not shared between different nodes of a d-tree, merely causing a choice of transitions for each possible value, but no additional states.

6 Conclusions

The number of elementary d-trees in the LexSys grammar is considerably greater than in the wide-coverage LTAG that is used in the XTAG system (XTAG-Group, 1995). One reason for this is that we include VP complement d-trees that are not found in XTAG. Another significant issue is that in designing our grammar we have tried to resist making compromises just because they would result in a smaller grammar that would be more efficient to parse with. Instead, our aim has been that the design of the grammar be primarily determined by linguistic rather than computational concerns. This has been made feasible, from a representational point of view, by our use of DATR to compactly encode the grammar.

The strategy of limiting the influence that computational efficiency places on grammar design puts a considerable burden on the technique of computation sharing discussed in this paper. We have presented some preliminary results indicating that our approach is viable. However, conclusive proof will only come from analysis of the computation sharing parser's performance in practice.

References

Briscoe, Edward, Claire Grover, Branimir Boguraev, and John Carroll. 1987. A formalism and environment for the development of a large grammar of English. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence*, pages 703–708, Milan, Italy.

Carroll, John and Claire Grover. 1989. The derivation of a large computational lexicon of English from LDOCE. In B. Boguraev and E. Briscoe, editors, *Computational Lexicography for Natural Language Processing*. Longman.

Cunningham, Hamish, Robert Gaizauskas, and Yorick Wilks. 1995. A general architecture for text engineering (GATE) — A new approach to language R&D. Research memo CS-95-21, Department of Computer Science, University of Sheffield, UK.

Elworthy, David. 1994. Does Baum-Welch re-estimation help taggers? In *Proceedings of the 4th ACL Conference on Applied Natural Language Processing (ANLP'94)*, Stuttgart, Germany.

Evans, Roger and Gerald Gazdar. 1996. DATR: A Language for Lexical Knowledge Representation. *Computational Linguistics*, 22(2):167–247.

Evans, Roger, Gerald Gazdar, and David Weir. 1995. Encoding lexicalized tree adjoining grammars with a nonmonotonic inheritance hierarchy. In *Proceedings of the 33rd Meeting of the Association for Computational Linguistics (ACL'95)*, pages 77–84.

Evans, Roger and David Weir. 1997. Automaton-based parsing for lexicalized grammars. In *Proceedings of the 5th International Workshop on Parsing Technologies (IWPT'5)*, pages 66–76.

Evans, Roger and David Weir. 1998. A structure-sharing parser for lexicalized grammars. Technical Report ITRI-98-02, ITRI, University of Brighton. Available at ftp://ftp.itri.brighton.ac.uk/pub/reports/ITRI-98-02.ps.

Rambow, Owen, K. Vijay-Shanker, and David Weir. 1995. D-Tree Grammars. In *33rd Meeting of the Association for Computational Linguistics (ACL'95)*, pages 151–158.

Rogers, James and K. Vijay-Shanker. 1992. Reasoning with descriptions of trees. In *30th Meeting of the Association for Computational Linguistics (ACL'92)*, pages 72–80.

Sampson, Geoffrey. 1995. *English for the Computer*. Oxford University Press, Oxford, UK.

Schabes, Yves and Aravind Joshi. 1988. An Earley-type parsing algorithm for tree adjoining grammars. In *26th Meeting of the Association for Computational Linguistics (ACL'88)*.

Vijay-Shanker, K. and David Weir. 1993. The use of shared forests in TAG parsing. In *Sixth Conference of the European Chapter of the Association for Computational Linguistics (EACL'93)*, pages 384–393, Utrecht.

Vijay-Shanker, K., David Weir, and Owen Rambow. 1995. Parsing D-Tree Grammars. In *International Workshop on Parsing Technologies*, pages 252–259.

XTAG-Group, The. 1995. A lexicalized tree adjoining grammar for English. Technical Report IRCS Report 95-03, The Institute for Research in Cognitive Science, University of Pennsylvania.