A structure-sharing parser for lexicalized grammars

Roger Evans
Information Technology Research Institute
University of Brighton
Brighton, BN2 4GJ, UK
Roger.Evans@itri.brighton.ac.uk

David Weir
Cognitive and Computing Sciences
University of Sussex
Brighton, BN1 9QH, UK
David.Weir@cogs.susx.ac.uk

Abstract
In wide-coverage lexicalized grammars many of the elementary structures have substructures in common. This means that in conventional parsing algorithms some of the computation associated with different structures is duplicated. In this paper we describe a precompilation technique for such grammars which allows some of this computation to be shared. In our approach the elementary structures of the grammar are transformed into finite state automata which can be merged and minimised using standard algorithms, and then parsed using an automaton-based parser. We present algorithms for constructing automata from elementary structures, merging and minimising them, and string recognition and parse recovery with the resulting grammar.

1 Introduction
It is well-known that fully lexicalised grammar formalisms such as LTAG (Joshi and Schabes, 1991) are difficult to parse with efficiency. Each word in the parser's input string introduces an elementary tree into the parse table for each of its possible readings, and there is often a substantial overlap in structure between these trees. A conventional parsing algorithm (Vijay-Shanker and Joshi, 1985) views the trees as independent, and so is likely to duplicate the processing of this common structure. Parsing could be made more efficient (empirically if not formally), if the shared structure could be identified and processed only once.

Recent work by Evans and Weir (1997) and Chen and Vijay-Shanker (1997) addresses this problem from two different perspectives. Evans and Weir (1997) outline a technique for compiling LTAG grammars into automata which are then merged to introduce some sharing of structure. Chen and Vijay-Shanker (1997) use underspecified tree descriptions to represent sets of trees during parsing. The present paper takes the former approach, but extends our previous work by:

• showing how merged automata can be minimised, so that they share as much structure as possible;
• showing that by precompiling additional information, parsing can be broken down into recognition followed by parse recovery;
• providing a formal treatment of the algorithms for transforming and minimising the grammar, recognition and parse recovery.

In the following sections we outline the basic approach, and describe informally our improvements to the previous account. We then give a formal account of the optimisation process and a possible parsing algorithm that makes use of it.

2 Automaton-based parsing
Conventional LTAG parsers (Vijay-Shanker and Joshi, 1985; Schabes and Joshi, 1988; Vijay-Shanker and Weir, 1993) maintain a parse table, a set of items corresponding to complete and partial constituents. Parsing proceeds by first seeding the table with items anchored on the input string, and then repeatedly scanning the table for parser actions. Parser actions introduce new items into the table licensed by one or more items already in the table. The main types of parser actions are:

1. extending a constituent by incorporating a complete subconstituent (on the left or

1However, due to lack of space, no proofs and only minimal informal descriptions are given in this paper.
right);
2. extending a constituent by adjoining a
surrounding complete auxiliary constituent;
3. predicting the span of the foot node of an
auxiliary constituent (to the left or right).

Parsing is complete when all possible parser ac-
tions have been executed.

In a completed parse table it is possible to
trace the sequence of items corresponding to the
recognition of an elementary tree from its lexi-
cal anchor upwards. Each item in the sequence
corresponds to a node in the tree (with the se-
quence as a whole corresponding to a complete
traversal of the tree), and each step corresponds
to the parser action that licensed the next item,
given the current one. From this perspective,
parser actions can be restated relative to the
items in such a sequence as:
1. substitute a complete subconstituent (on
the left or right);
2. adjoin a surrounding complete auxiliary
constituent;
3. predict the span of the tree’s foot node (to
the left or right).

The recognition of the tree can thus be viewed
as the computation of a finite state automaton,
whose states correspond to a traversal of the
tree and whose input symbols are these rela-
tivised parser actions.

This perspective suggests a re-casting of the
conventional LTAG parser in terms of such au-
tomata\(^2\). For this automaton-based parser, the
grammar structures are not trees, but automata
 corresponding to tree traversals whose inputs
are strings of relativised parser actions. Items
in the parse table reference automaton states
instead of tree addresses, and if the automa-
ton state is final, the item represents a complete
constituent. Parser actions arise as before, but
are executed by relativising them with respect to
the incomplete item participating in the ac-
tion, and passing this relativised parser action
as the next input symbol for the automaton re-
 ferenced by that item. The resulting state of
that automaton is then used as the referent of
the newly licensed item.

On a first pass, this re-casting is exactly that: it
does nothing new or different from the original

parser on the original grammar. However there
are a number of subtle differences\(^3\):

- the automata are more abstract than the
trees: the only grammatical information
they contain are the input symbols and the
root node labels, indicating the category of
the constituent the automaton recognises;
- automata for several trees can be merged
together and optimised using standard
well-studied techniques, resulting in a sin-
gle automaton that recognises many trees
at once, sharing as many of the common
parser actions as possible.

It is this final point which is the focus of this
paper. By representing trees as automata, we
can merge trees together and apply standard
optimisation techniques to share their common
structure. The parser will remain unchanged,
but will operate more efficiently where struc-
ture has been shared. Additionally, because
the automata are more abstract than the trees,
capturing precisely the parser’s view of the
trees, sharing may occur between trees which
are structurally quite different, but which hap-
pen to have common parser actions associated
with them.

3 Merging and minimising automata
Combining the automata for several trees can
be achieved using a variety of standard algo-
rithms (Huffman, 1954; Moore, 1956). How-
ever any transformations must respect one im-
portant feature: once the parser reaches a fi-
nal state it needs to know what tree it has just
recognised\(^4\). When automata for trees with
different root categories are merged, the resulting
automaton needs to somehow indicate to the
parser what trees are associated with its final
states.

In Evans and Weir (1997), we combined au-
tomata by introducing a new initial state with
\(\epsilon\)-transitions to each of the original initial states,

\(^2\)Evans and Weir (1997) provides a longer informal
introduction to this approach.

\(^3\)A further difference is that the traversal encoded
in the automaton captures part of the parser’s control
strategy. However for simplicity we assume here a fixed
parser control strategy (bottom-up, anchor-out) and do
not pursue this point further – Evans and Weir (1997)
offers some discussion.

\(^4\)For recognition alone it only needs to know the root
category of the tree, but to recover the parse it needs to
identify the tree itself.
and then determining the resulting automaton to induce some sharing of structure. To recover trees, final automaton states were annotated with the number of the tree the final state is associated with, which the parser can then readily access.

However, the drawback of this approach is that differently annotated final states can never be merged, which restricts the scope for structure sharing (minimisation, for example, is not possible since all the final states are distinct). To overcome this, we propose an alternative approach as follows:

- each automaton transition is annotated with the set of trees which pass through it: when transitions are merged in automaton optimisation, their annotations are unioned;
- the parser maintains for each item in the table the set of trees that are valid for the item: initially this is all the valid trees for the automaton, but gets intersected with the annotation of any transition followed; also if two paths through the automaton meet (i.e., an item is about to be added for a second time), their annotations get unioned.

This approach supports arbitrary merging of states, including merging all the final states into one. The parser maintains a dynamic record of which trees are valid for states (in particular final states) in the parse table. This means that we can minimise our automata as well as determining them, and so can be pre-compiled as the automata themselves are built up. For each action transition (the action, plus the start and finish states) we record the tree address that the transition reaches (we call this the action-site, or just a-site for short). During parse recovery, when the parse table indicates an action that licensed an item, we look up the relevant transition to discover where in the tree (or trees, if we are traversing several simultaneously) the present item must be, so that we can correctly construct a derivation tree.

5 Technical details

5.1 Constructing the automata

We identify each node in an elementary tree \( \gamma \) with an elementary address \( \gamma/i \). The root of \( \gamma \) has the address \( \gamma/e \) where \( e \) is the empty string. Given a node \( \gamma/i \), its \( n \) children are addressed from left to right with the addresses \( \gamma/i_1, \ldots, \gamma/i_n \), respectively. For convenience, let anchor (\( \gamma \)) and foot (\( \gamma \)) denote the elementary address of the node that is the anchor and footnode (if it has one) of \( \gamma \), respectively; and label (\( \gamma/i \)) and parent (\( \gamma/i \)) denote the label of \( \gamma/i \) and the address of the parent of \( \gamma/i \), respectively.

In this paper we make the following assumptions about elementary trees. Each tree has a single anchor node and therefore a single spine. In the algorithms below we assume that nodes not on the spine have no children. In practice, not all elementary LTAG trees meet these conditions, and we discuss how the approach described here might be extended to the more gen-

---

5The path from the root to the anchor node.
eral case in Section 6.

Let \( \gamma / i \) be an elementary address of a node on the spine of \( \gamma \) with \( n \) children \( \gamma / i 1, \ldots , \gamma / i k, \ldots , \gamma / i n \) for \( n \geq 1 \), where \( k \) is such that \( \gamma / i k \) dominates anchor \((\gamma)\).

\[
\text{next}(\gamma / ij) = \begin{cases} 
\gamma / ik + 1 & \text{if } j = 1 \& n > k \\
\gamma / ij - 1 & \text{if } 2 \leq j \leq k \\
\gamma / ij + 1 & \text{if } k < j \leq n \\
\gamma / i & \text{otherwise}
\end{cases}
\]

next defines a function that traverses a spine, starting at the anchor. Traversal of an elementary tree during recognition yields a sequence of parser actions, which we annotate as follows: the two actions \( \Delta \) and \( \Delta^\circ \) indicate a substitution of a tree rooted with \( A \) to the left or right, respectively; \( \Delta_+ \) and \( \Delta_- \) indicate the presence of the foot node, a node labelled \( A \), to the left or right, respectively; Finally \( \Delta \) indicates an adjunct±on of a tree with root and foot labelled \( A \). These actions constitute the input language of the automaton that traverses the tree. This automaton is defined as follows (note that we use \( \epsilon \)-transitions between nodes to ease the construction – we assume these are removed using a standard algorithm).

Let \( \gamma \) be an elementary tree with terminal and nonterminal alphabets \( V_T \) and \( V_N \), respectively. Each state of the following automaton specifies the elementary address \( \gamma / i \) being visited. When the node is first visited we use the state \( \bot[\gamma / i] \); when ready to move on we use the state \( T[\gamma / i] \).

Define as follows the finite state automaton \( M = (Q, \Sigma, \bot[anchor (\gamma)], \delta, F) \). \( Q \) is the set of states, \( \Sigma \) is the input alphabet, \( q_0 \) is the initial state, \( \delta \) is the transition relation, and \( F \) is the set of final states.

\[
Q = \{ T[\gamma / i], \bot[\gamma / i] \mid \gamma / i \text{ is an address in } \gamma \};
\]

\( \Sigma = \{ \Delta, \Delta_+, \Delta^\circ, \Delta^\circ_+, \Delta^- \mid A \in V_N \}; \)

\( F = \{ T[\gamma / e] \}; \) and 

\( \delta \) includes the following transitions:

\[
(\bot[\text{foot (} \gamma \text{)}], \Delta, T[\text{foot (} \gamma \text{)}]) \text{ if foot (} \gamma \text{) is to the right of anchor (} \gamma \text{)}
\]

\[
(\bot[\text{foot (} \gamma \text{)}], \Delta^\circ, T[\text{foot (} \gamma \text{)}]) \text{ if foot (} \gamma \text{) is to the left of anchor (} \gamma \text{)}
\]

\[
(\bot[\gamma / i], \epsilon, \bot[\text{next (} \gamma / i \text{)}]) \mid \gamma / i \text{ is an address in } \gamma \\text{ i.e } \epsilon
\]

\[
(\bot[\gamma / i], \Delta, T[\gamma / i]) \mid \gamma / i \text{ substitution node, label } \gamma / i = A,
\]

\( \gamma / i \text{ to right of anchor (} \gamma \text{)} \}

\[
(\bot[\gamma / i], \Delta^\circ, T[\gamma / i]) \mid \gamma / i \text{ substitution node, label } \gamma / i = A,
\]

\( \gamma / i \text{ to left of anchor (} \gamma \text{)} \}

\[
(\bot[\gamma / i], \epsilon, T[\gamma / i]) \mid \gamma / i \text{ substitution node, label } \gamma / i = A
\]

\[
(\bot[\gamma / i], \epsilon, T[\gamma / i]) \mid \gamma / i \text{ substitution node, label } \gamma / i = A
\]

In order to recover derivation trees, we also define the partial function \( a-site(q,a,q') \) for \( (q,a,q') \in \delta \) which provides information about the site within the elementary tree of actions occurring in the automaton.

\[
a-site(q,a,q') = \begin{cases} 
\gamma / i & \text{if } a \neq \epsilon \& q' = T[\gamma / i] \\
\text{undefined} & \text{otherwise}
\end{cases}
\]

### 5.2 Combining Automata

Suppose we have a set of trees \( \Gamma = \{ \gamma_1, \ldots , \gamma_n \} \). Let \( M_{\gamma_1}, \ldots , M_{\gamma_n} \) be the \( \epsilon \)-free automata that are built from members of the set \( \Gamma \) using the above construction, where for \( 1 \leq k \leq n \), \( M_k = (Q_k, \Sigma_k, q_k, \delta_k, F_k) \).

Construction of a single automaton for \( \Gamma \) is a two step process. First we build an automaton that accepts all elementary computations for trees in \( \Gamma \); then we apply the standard automaton determinization and minimization algorithms to produce an equivalent, compact automaton. The first step is achieved simply by introducing a new initial state with \( \epsilon \)-transitions to each of the \( q_k \):

Let \( M = (Q, \Sigma, q_0, \delta, F) \) where 

\( Q = \{ q_0 \} \cup \bigcup_{1 \leq k \leq n} Q_k; \)

\( \Sigma = \bigcup_{1 \leq k \leq n} \Sigma_k \)

\( F = \bigcup_{1 \leq k \leq n} F_k \)

\( \delta = \bigcup_{1 \leq k \leq n} (q_0, \epsilon, q_k) \cup \bigcup_{1 \leq k \leq n} \delta_k. \)

We determinize and then minimize \( M \) using the standard set-of-states constructions to produce \( M_\Gamma = (Q', \Sigma, Q_0, \delta', F') \). Whenever two states are merged in either the determining or minimizing algorithms the resulting state is named by the union of the states from which it is formed.

For each transition \( (Q_1, a, Q_2) \in \delta' \) we define the function \( a-sites(Q_1, a, Q_2) \) to be a set of elementary nodes as follows:

\[
a-sites(Q_1, a, Q_2) = \bigcup_{q_1 \in Q_1, q_2 \in Q_2} a-site(q_1, a, q_2)
\]

Given a transition in \( M_\Gamma \), this function returns all the nodes in all merged trees which that tran-
Finally, we define:
\[
cross(Q_1, a, Q_2) = \{ \gamma \mid \gamma/i \in a\text{-sites}(Q_1, a, Q_2) \}
\]
This gives that subset of those trees whose elementary computations take the \( M_T \) through state \( Q_1 \) to \( Q_2 \). These are the transition annotations referred to above, used to constrain the parser's set of valid trees.

5.3 The Recognition Phase

This section illustrates a simple bottom-up parsing algorithm that makes use of minimized automata produced from sets of trees that anchor the same input symbol.

The input to the parser takes the form of a sequence of minimized automata, one for each of the symbols in the input. Let the input string be \( w = a_1 \ldots a_n \) and the associated automata be \( M_1, \ldots, M_n \) where \( M_k = (Q_k, \Sigma_k, q_k, \delta_k, F_k) \) for \( 1 \leq k \leq n \). Let \( \text{treesof}(M_k) = \Gamma_k \) where \( \Gamma_k \) is a set of the names of those elementary trees that were used to construct the automata \( M_k \).

During the recognition phase of the algorithm, a set \( I \) of items are created. An item has the form \( \langle T, q, [l, r, l', r'] \rangle \) where \( T \) is a set of elementary tree names, \( q \) is a automata state and \( l, r, l', r' \in \{ 0, \ldots, n, - \} \) such that either \( l \leq l' \leq r \leq l' \) or \( l \leq r \) and \( l' = r' = - \). The indices \( l, l', r, r' \) are positions between input symbols (position 0 is before the first input symbols and position \( n \) is after the final input symbol) and we use \( w_{p,p'} \) to denote that substring of the input \( w \) between positions \( p \) and \( p' \). \( I \) can be viewed as a four dimensional array, each entry of which contains a set of pairs comprising of a set of nonterminals and an automata state.

Roughly speaking, an item \( \langle T, q, [l, r, l', r'] \rangle \) is included in \( I \) when for every \( \gamma \in T \), anchored by some \( a_k \) (where \( l \leq k \leq r \) and if \( l' \neq - \) then \( k \leq l' \) or \( r' \leq k \)); \( q \) is a state in \( Q_k \), such that some elementary subcomputation reaching \( q \) from the initial state, \( q_k \), of \( M_k \) is an initial substring of the elementary computation for \( \gamma \) that reaches the elementary address \( \gamma/i \), the subtree rooted at \( \gamma/i \) spans \( w_{l,r} \), and if \( \gamma/i \) dominates a foot node then that foot node spans \( w_{l',r'} \), otherwise \( l' = r' = - \).

The input is accepted if an item \( \langle T, q_f, [0, n, -, -] \rangle \) is added to \( I \) where \( T \) contains some initial tree rooted in the start symbol \( S \) and \( q_f \in F_k \) for some \( k \).

When adding items to \( I \) we use the procedure\( \text{add}(T, q, [l, r, l', r']) \) which is defined such that if there is already an entry \( \langle T', q, [l, r, l', r'] \rangle \in I \) for some \( T' \) then replace this with the entry \( \langle T \cup T', q, [l, r, l', r'] \rangle \); otherwise add the new entry \( \langle T, q, [l, r, l', r'] \rangle \) to \( I \).

\( I \) is initialized as follows. For each \( k \in \{ 1, \ldots, n \} \) call \( \text{add}(T, q_k, [k-1, k, -, -]) \) where \( T = \text{treesof}(M_k) \) and \( q_k \) is the initial state of the automata \( M_k \).

We now present the rules with which the complete set \( I \) is built. These rules correspond closely to the familiar steps in existing bottom-up LTAG parser, in particular, the way that we use the four indices is exactly the same as in other approaches (Vijay-Shanker and Joshi, 1985). As a result a standard control strategy can be used to control the order in which these rules are applied to existing entries of \( I \).

1. If \( \langle T, q, [l, r, l', r'] \rangle, \langle T', q_f, [r, r'', -, -] \rangle \in I \), \( q_f \in F_k \) for some \( k \), \((q, A, q') \in \delta_k \) for some \( k' \), label \((\gamma/e) = A \) from some \( \gamma' \in T' \) & \( T'' = T \cap \text{cross}(q, A, q') \) then call \( \text{add}(T'', q', [l, r', l', r']) \).
2. If \( \langle T, q, [l, r, l', r'] \rangle, \langle T', q_f, [l'', l, -, -] \rangle \in I \), \( q_f \in F_k \) for some \( k \), \((q, A, q') \in \delta_k \) for some \( k' \), label \((\gamma/e) = A \) from some \( \gamma' \in T' \) & \( T'' = T \cap \text{cross}(q, A, q') \) then call \( \text{add}(T'', q', [l'', r', l', r']) \).
3. If \( \langle T, q, [l, r, -, -] \rangle \in I \), \((q, A, q') \in \delta_k \) for some \( k \) & \( T' = T \cap \text{cross}(q, A, q') \) then for each \( r' \) such that \( r < r' < n \) call \( \text{add}(T', q', [l, r', r', r']) \).
4. If \( \langle T, q, [l, r, -, -] \rangle \in I \), \((q, A, q') \in \delta_k \) for some \( k \) & \( T' = T \cap \text{cross}(q, A, q') \) then for each \( l' \) such that \( 0 < l' < l \) call \( \text{add}(T', q', [l', r, l', r']) \).
5. If \( \langle T, q, [l, r, l', r'] \rangle, \langle T', q_f, [l'', r'', l, r] \rangle \in I \), \( q_f \in F_k \) for some \( k \), \((q, A, q') \in \delta_k \) for some \( k' \), label \((\gamma/e) = A \) from some \( \gamma' \in T' \) & \( T'' = T \cap \text{cross}(q, A, q') \) then call \( \text{add}(T'', q', [l'', r'', l', r']) \).

\( ^6 \)This replacement is treated as a new entry in the table. If the old entry has already licenced other entries, this may result in some duplicate processing. This could be eliminated by a more sophisticated treatment of tree sets.

376
The running time of this algorithm is \( O(n^6) \) since the last rule must be embedded within six loops each of which varies with \( n \). Note that although the third and fourth rules both take \( O(n) \) steps, they need only be embedded within the \( l \) and \( r \) loops.

5.4 Recovering Parse Trees

Once the set of items \( I \) has been completed, the final task of the parser is to recover derivation tree\(^7\). This involves retracing the steps of the recognition process in reverse. At each point, we look for a rule that would have caused the inclusion of item in \( I \). Each of these rules involves some transition \((q, a, q') \in \delta_k\) for some \( k \) where \( a \) is one of the parser actions, and from this transition we consult the set of elementary addresses in \( \text{a-sites}(q, a, q) \) to establish how to build the derivation tree. We eventually reach items added during the initialization phase and the process ends. Given the way our parser has been designed, some search will be needed to find the items we need. As usual, the need for such search can be reduced through the inclusion of pointers in items, though this is at the cost of increasing parsing time. There are various points in the following description where nondeterminism exists. By exploring all possible paths, it would be straightforward to produce an AND/OR derivation tree that encodes all derivation trees for the input string.

We use the procedure \( \text{der}((T, q, [l, r, l', r']), r) \) which completes the partial derivation tree \( r \) by backing up through the moves of the automata in which \( q \) is a state.

A derivation tree for the input is returned by the call \( \text{der}((T, q_f, [0, n, -, -]), \tau) \) where \( (T, q_f, [0, n, -, -]) \in I \) such that \( T \) contains some initial tree \( \gamma \) rooted with the start non-terminal \( S \) and \( q_f \) is the final state of some automata \( M_k, 1 \leq k \leq n \). \( \tau \) is a derivation tree containing just one node labelled with name \( \gamma \).

In general, on a call to \( \text{der}((T, q, [l, r, l', r']), \tau) \) we examine \( I \) to find a rule that has caused this item to be included in \( I \). There are six rules to consider, corresponding to the five recogniser rules, plus lexical introduction, as follows:

1. If \((T', q', [l, r''', l'', r'''], \gamma) \in I, q_f \in F_k \) for some \( k \), \((q', \Delta, q) \in \delta_{k'} \) for some \( k' \), \( \gamma \) is the label of the root of \( \tau \), \( \gamma \in T' \), label \( (\gamma'/\epsilon) = A \) from some \( \gamma' \in T'' \) & \( \gamma' \in \text{a-sites}(q', \Delta, q) \), then let \( \tau' \) be the derivation tree containing a single node labelled \( \gamma' \), and let \( \tau'' \) be the result of attaching \( \text{der}((T'', q_f, [r'', r, -,-]), \gamma) \) under the root of \( \tau \) with an edge labelled the tree address \( i \). We then complete the derivation tree by calling \( \text{der}((T'', q', [l, r'', l', r'''], \gamma)), \tau'') \).

2. If \((T', q', [r''', r, l'', r'''], \gamma) \in I, q_f \in F_k \) for some \( k \), \((q', \Delta, q) \in \delta_{k'} \) for some \( k' \), \( \gamma \) is the label of the root of \( \tau \), \( \gamma \in T' \), label \( (\gamma'/\epsilon) = A \) from some \( \gamma' \in T'' \) & \( \gamma' \in \text{a-sites}(q', \Delta, q) \), then let \( \tau' \) be the derivation tree containing a single node labelled \( \gamma' \), and let \( \tau'' \) be the result of attaching \( \text{der}((T'', q_f, [l, r'', -,-]), \gamma) \) under the root of \( \tau \) with an edge labelled the tree address \( i \). We then complete the derivation tree by calling \( \text{der}((T'', q', [r'', r, l'', r'''], \gamma)), \tau'') \).

3. If \( r = r' \), \((T', q', [l, l', --]), \gamma) \in I \) and \((q', \Delta, q) \in \delta_{k'} \) for some \( k \), \( \gamma \) is the label of the root of \( \tau \), \( \gamma \in T' \) and label \( (\gamma'/\epsilon) = A \) from some \( \gamma' \in T'' \) and \( \gamma' \in \text{a-sites}(q', \Delta, q) \), then make the call \( \text{der}((T', q', [l, l', r, r]), \gamma) \).

4. If \( l = l' \), \((T', q', [r', r, l', r']), \gamma) \in I \) and \((q', \Delta, q) \in \delta_{k'} \) for some \( k \), \( \gamma \) is the label of the root of \( \tau \), \( \gamma \in T' \) and label \( (\gamma'/\epsilon) = A \) from some \( \gamma' \in T'' \) and \( \gamma' \in \text{a-sites}(q', \Delta, q) \), then make the call \( \text{der}((T', q', [r', r, l', r']), \gamma) \).

5. If \((T', q', [r'', r, l'', r'']), \gamma) \in I, q_f \in F_k \) for some \( k \), \((q', \Delta, q) \in \delta_{k'} \) for some \( k' \), \( \gamma \) is the label of the root of \( \tau \), \( \gamma \in T' \), label \( (\gamma'/\epsilon) = A \) from some \( \gamma' \in T'' \) and \( \gamma' \in \text{a-sites}(q', \Delta, q) \), then let \( \tau' \) be the derivation tree containing a single node labelled \( \gamma' \), and let \( \tau'' \) be the result of attaching \( \text{der}((T'', q_f, [l, r'', l'', r'']), \gamma) \) under the root of \( \tau \) with an edge labelled the tree address \( i \). We then complete the derivation tree by calling \( \text{der}((T'', q', [l, r'', l', r'']), \gamma)), \tau'') \).

6. If \( l + 1 = r \), \( r' = r'' = l' = q \) is the initial state of \( M_r \), \( \gamma \) is the label of the root of \( \tau \), \( \gamma \in T \), then return the final derivation tree \( \tau \).

6 Discussion

The approach described here offers empirical rather than formal improvements in performance. In the worst case, none of the trees
in the grammar share any structure so no optimisation is possible. However, in the typical case, there is scope for substantial structure sharing among closely related trees. Carroll et al. (1998) report preliminary results using this technique on a wide-coverage DTG (a variant of LTAG) grammar. Table 1 gives statistics for three common verbs in the grammar: the total number of trees, the size of the merged automaton (before any optimisation has occurred) and the size of the minimised automaton. The final column gives the average of the number of trees that share each state in the automaton. These figures show substantial optimisation is possible, both in the space requirements of the grammar and in the sharing of processing state between trees during parsing.

As mentioned earlier, the algorithms we have presented assume that elementary trees have one anchor and one spine. Some trees, however, have secondary anchors (for example, a subcategorised preposition). One possible way of including such cases would be to construct automata from secondary anchors up the secondary spine to the main spine. The automata for both the primary and secondary anchors associated with a lexical item could then be merged, minimized and used for parsing as above.

Using automata for parsing has a long history dating back to transition networks (Woods, 1970). More recent uses include Alshawi (1996) and Eisner (1997). These approaches differ from the present paper in their use of automata as part of the grammar formalism itself. Here, automata are used purely as a stepping-stone to parser optimisation: we make no linguistic claims about them. Indeed one view of this work is that it frees the linguistic descriptions from overt computational considerations. This work has perhaps more in common with the technology of LR parsing as a parser optimisation technique, and it would be interesting to compare our approach with a direct application of LR ideas to LTAGs.

**References**

H. Alshawi. 1996. Head automata and bilingual tilings: Translation with minimal representations. In *ACL96*, pages 167–176.

J. Carroll, N. Nicolov, O. Shaumyan, M. Smets, and D. Weir. 1998. Grammar compaction and computation sharing in automaton-based parsing. In *Proceedings of the First Workshop on Tabulation in Parsing and Deduction*, pages 16–25.

J. Chen and K. Vijay-Shanker. 1997. Towards a reduced-commitment D-theory style TAG parser. In *IWPT97*, pages 18–29.

J. Eisner. 1997. Bilexical grammars and a cubic-time probabilistic parser. In *IWPT97*, pages 54–65.

R. Evans and D. Weir. 1997. Automaton-based parsing for lexicalized grammars. In *IWPT97*, pages 66–76.

D. A. Huffman. 1954. The synthesis of sequential switching circuits. *J. Franklin Institute*.

A. K. Joshi and Y. Schabes. 1991. Tree-adjointing grammars and lexicalized grammars. In Maurice Nivat and Andreas Podelski, editors, *Definability and Recognizability of Sets of Trees*. Elsevier.

E. F. Moore, 1956. *Automata Studies*, chapter Gedanken experiments on sequential machines, pages 129–153. Princeton University Press, N.J.

Y. Schabes and A. K. Joshi. 1988. An Earley-type parsing algorithm for tree adjoining grammars. In *ACL88*.

K. Vijay-Shanker and A. K. Joshi. 1985. Some computational properties of tree adjoining grammars. In *ACL85*, pages 82–93.

K. Vijay-Shanker and D. Weir. 1993. Parsing some constrained grammar formalisms. *Computational Linguistics*, 19(4):591–636.

W. A. Woods. 1970. Transition network grammars for natural language analysis. *Commun. ACM*, 13:591–606.