An alternative LR algorithm for TAGs

Mark-Jan Nederhof
DFKI
Stuhlsatzenhausweg 3
D-66123 Saarbrücken, Germany
E-mail: nederhof@dfki.de

Abstract

We present a new LR algorithm for tree-adjoining grammars. It is an alternative to an existing algorithm that is shown to be incorrect. Furthermore, the new algorithm is much simpler, being very close to traditional LR parsing for context-free grammars. The construction of derived trees and the computation of features also become straightforward.

1 Introduction

The efficiency of LR(k) parsing techniques (Sippu and Soisalon-Soininen, 1990) appears to be very attractive from the perspective of natural language processing. This has stimulated the computational linguistics community to develop extensions of these techniques to general context-free grammar parsing. The best-known example is *generalized* LR parsing (Tomita, 1986).

A first attempt to adapt LR parsing to tree-adjoining grammars (TAGs) was made by Schabes and Vijay-Shanker (1990). The description was very complicated however, and not surprisingly, no implementation of the algorithm seems to have been made up to now. Apart from presentational difficulties, the algorithm as it was published is also incorrect. Brief indications of the nature of the incorrectness have been given before by Kinyon (1997). There seems to be no straightforward way to correct the algorithm.

We therefore developed an alternative to the algorithm from Schabes and Vijay-Shanker (1990). This alternative is novel in presentational aspects, and is fundamentally different in that it incorporates reductions of subtrees.

The new algorithm has the benefit that many theoretically and practically useful properties carry over from the context-free case. For example, by making a straightforward translation from TAGs to linear indexed grammars, one may identify computations of the parser with rightmost derivations in reverse. Also the extensions needed for construction of parse trees (or "derived trees" as they are often called for TAGs) and the computation of features are almost identical to the corresponding extensions for context-free LR parsing.

Section 2 discusses our notation. The algorithm for constructing the LR table is given in Section 3, and the automaton that operates on these tables is given in Section 4. Section 5 first explains why the algorithm from Schabes and Vijay-Shanker (1990) is incorrect, and then provides an example of how our new algorithm works. Some extensions are discussed in Section 6, and the implementation in Section 7.

2 Notation

For a good introduction to TAGs, the reader is referred to Joshi (1987). In this section we merely summarize our notation.

A tree-adjoining grammar is a 4-tuple (Σ, NT, I, A), where Σ is a finite set of *terminals*, I is a finite set of *initial trees* and A is a finite set of *auxiliary trees*. We refer to the trees in $I \cup A$ as *elementary trees*. The set NT, a finite set of *nonterminals*, does not play any role in this paper.

Each auxiliary tree has a distinguished leaf, call the *foot*. We refer to the foot of an auxiliary tree t as F_t. We refer to the root of an elementary tree t as R_t. The set of all nodes of an elementary tree t is denoted by $\mathcal{N}(t)$, and we define the set of all nodes in the grammar by $\mathcal{N} = \bigcup_{t \in I \cup A} \mathcal{N}(t)$.

For each non-leaf node N we define $children(N)$ as the list of children nodes. For other nodes, the function *children* is undefined. The dominance relation $\lhd^*$ is the reflexive and

transitive closure of the parent relation $\lhd$ defined by $N \lhd M$ if and only if $children(N) = \alpha M \beta$, for some $\alpha, \beta \in \mathcal{N}^*$.

Each leaf N in an elementary tree, except when it is a foot, is labelled by either a terminal from Σ or the empty string ε. We identify such a node N labelled by a terminal with that terminal. Thus, we consider Σ to be a subset of $\mathcal{N}$.[1]

For now, we will disallow labels to be ε, since this causes a slight technical problem. We will return to this issue in Section 6.

For each node N that is not a leaf or that is a foot, we define $Adjunct(N)$ as the set of auxiliary trees that can be adjoined at N. This set may contain the element *nil* to indicate that adjunction at that node is not obligatory.

An example of a TAG is given in Figure 1. There are two initial trees, α_1 and α_2, and one auxiliary tree β. For each node N, $Adjunct(N)$ has been indicated to the right of that node, unless $Adjunct(N) = \{nil\}$, in which case that information is omitted from the picture.

3 Construction of the LR table

For technical reasons, we assume an additional node for each elementary tree t, which we denote by $\top$. This node has only one child, viz. the actual root node R_t. We also assume an additional node for each auxiliary tree t, which we denote by $\bot$. This is the unique child of the actual foot node F_t. The domain of the function *children* is extended to include foot nodes, by defining $children(F_t) = \bot$, for each $t \in A$.

For the algorithm, two kinds of tree need to be distinguished: elementary trees and subtrees of elementary trees. A subtree can be identified by a pair (t, N), where t is an elementary tree and N is a node in that tree; the pair indicates the subtree of t rooted at N. The set of all trees needed by our algorithm is given by:

$$T = I \cup A \cup \{(t, N) \mid t \in I \cup A, N \in \mathcal{N}(t)\}$$

From here on, we will use the symbol t exclusively to range over $I \cup A$, and τ to range over T in general.

[1] With this convention, we can no longer distinguish between different leaves in the grammar with the same terminal label. This merging of leaves with identical labels is not an inherent part of our algorithm, but it simplifies the notation considerably.

For each $\tau \in T$, we may consider a part of the tree consisting of a node N in τ and the list of its children nodes γ. Analogously to the notation for context-free parsing, we separate the list of children nodes into two lists, separated by a dot, and write $N \rightarrow \alpha \bullet \beta$, where $\alpha\beta = \gamma$, to indicate that the children nodes in α have already been matched against a part of the input string, and those in β have as yet not been processed.

The set of such objects for an elementary tree t is given by:

$$P_t = \{(\top \rightarrow \alpha \bullet \beta) \mid \alpha\beta = R_t\} \cup \{(N \rightarrow \alpha \bullet \beta) \mid N \in \mathcal{N}(t), children(N) = \alpha\beta\}$$

For subtrees (t, M) we define:

$$P_{(t,M)} = \{(N \rightarrow \alpha \bullet \beta) \mid M \lhd^* N, children(N) = \alpha\beta\}$$

Such objects are attached to the trees $\tau \in T$ to which they pertain, to form the set of *items*:

$$Items = \{[\tau, N \rightarrow \alpha \bullet \beta] \mid \tau \in T, (N \rightarrow \alpha \bullet \beta) \in P_\tau\}$$

A *completed* item is an item that indicates a completely recognized elementary tree or subtree. Formally, items are completed if they are of the form $[t, \top \rightarrow R_t \bullet]$ or of the form $[(t, N), N \rightarrow \alpha \bullet]$.

The main concept needed for the construction of the LR table is that of *LR states*. These are particular elements from 2^{Items} to be defined shortly.

First, we introduce the function *closure* from 2^{Items} to 2^{Items} and the functions *goto* and $goto_\bot$ from $2^{Items} \times \mathcal{N}$ to 2^{Items}. For any $q \subseteq Items$, $closure(q)$ is the smallest set such that:

1. $q \subseteq closure(q)$;
2. $[\tau, N \rightarrow \alpha \bullet M\beta] \in closure(q)$, $nil \in Adjunct(M)$ and $children(M) = \gamma$ implies $[\tau, M \rightarrow \bullet \gamma] \in closure(q)$;
3. $[\tau, N \rightarrow \alpha \bullet M\beta] \in closure(q)$ and $t \in Adjunct(M)$ implies $[t, \top \rightarrow \bullet R_t] \in closure(q)$;
4. $[\tau, F_t \rightarrow \bullet \bot] \in closure(q)$, $t \in Adjunct(N)$, $N \in \mathcal{N}(t')$ and $children(N) = \gamma$ implies $[(t', N), N \rightarrow \bullet \gamma] \in closure(q)$; and
5. $[\tau, M \rightarrow \gamma \bullet] \in closure(q)$ and $[\tau, N \rightarrow \alpha M \bullet \beta] \in Items$ implies $[\tau, N \rightarrow \alpha M \bullet \beta] \in closure(q)$.

The clauses 1 thru 4 are reminiscent of the clo-

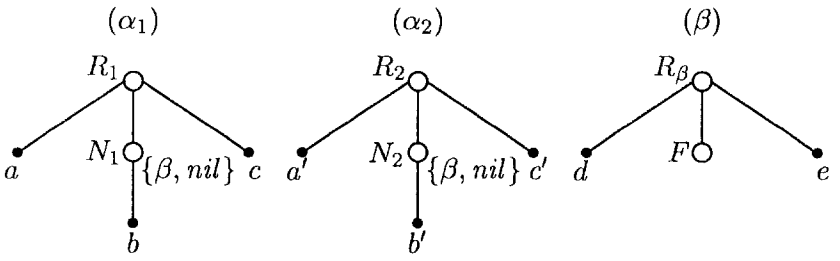

Figure 1: A tree-adjoining grammar.

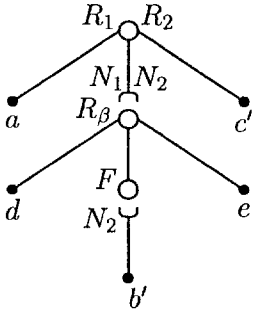

Figure 2: An incorrect "parse tree" (Section 5).

sure function for traditional LR parsing. Note that in clause 4 we set out to recognize a subtree (t', N) of elementary tree t'. Clause 5 is unconventional: we traverse the tree τ upwards when the dot indicates that all children nodes of M have been recognized.

Next we define the function *goto*, for any $q \subseteq \mathit{Items}$, and any $M \in \Sigma$ or $M \in \mathcal{N}$ such that $\mathit{Adjunct}(M)$ includes at least one auxiliary tree.

$$\mathit{goto}(q, M) = \{[\tau, N \to \alpha M \bullet \beta] \mid [\tau, N \to \alpha \bullet M\beta] \in \mathit{closure}(q)\}$$

The function $\mathit{goto}_\perp$ is similar in that it shifts the dot over a node, in this case the imaginary node $\perp$ which is the unique child of an actual foot node F_t. However, it only does this if t is a tree which can be adjoined at the node that is given as the second argument.

$$\mathit{goto}_\perp(q, M) = \{[\tau, F_t \to \perp \bullet] \mid [\tau, F_t \to \bullet \perp] \in \mathit{closure}(q) \wedge t \in \mathit{Adjunct}(M)\}$$

The *initial* LR state is the set

$$q_{in} = \{[t, \top \to \bullet R_t] \mid t \in I\}$$

We construct the set Q of all LR states as the smallest collection of sets satisfying the conditions:

1. $q_{in} \in Q$;
2. $q \in Q$, $M \in \mathcal{N}$ and $q' = \mathit{goto}(q, M) \neq \emptyset$ imply $q' \in Q$; and
3. $q \in Q$, $M \in \mathcal{N}$ and $q' = \mathit{goto}_\perp(q, M) \neq \emptyset$ imply $q' \in Q$.

An LR state is *final* if its closure includes a completed item corresponding to an initial tree:

$$Q_{fin} = \{q \in Q \mid \mathit{closure}(q) \cap \{[t, \top \to R_t \bullet] \mid t \in I\} \neq \emptyset\}$$

Final LR states indicate recognition of the input. Other completed items give rise to a *reduction*, a type of stack manipulation by the LR automaton to be defined in the next section. As defined below, reductions are uniquely identified by either auxiliary trees t or by nodes N obtained from the corresponding completed items.

$$\mathit{reductions}(q) = \{t \in A \mid [t, \top \to R_t \bullet] \in \mathit{closure}(q)\} \cup \{N \in \mathcal{N} \mid [(t, N), N \to \alpha \bullet] \in \mathit{closure}(q)\}$$

For each node N in a tree, we consider the set $CS(N)$ of strings that represent horizontal cross-sections through the subtree rooted at N. If we do not want to include the cross-section through N itself, we write $CS(N)^+$. A cross-section can also be seen as the yield of the subtree after removal of a certain number of its subtrees.

For convenience, each node of an auxiliary tree (or subtree thereof) that dominates a foot node is paired with a stack of nodes. The intuition behind such a stack of nodes $[N_1, \ldots, N_m]$ is that it indicates a path, the so called *spine*, through the derived tree in the direction of the foot nodes, where each N_i, with $1 \leq i \leq m$, is a node at which adjunction has taken place. Such stacks correspond to the stacks of linear indexed grammars.

The set of all stacks of nodes is denoted by $\mathcal{N}^*$. The empty stack is denoted by $[]$, and stacks consisting of head H and tail T are denoted by $[H|T]$. We define:

$$\mathcal{M} = \mathcal{N} \cup (\mathcal{N} \times \mathcal{N}^*)$$

and we simultaneously define the functions CS and CS^+ from $\mathcal{N}$ to $2^{\mathcal{M}^*}$ as the least functions

satisfying:

- $CS(N)^+ \subseteq CS(N)$, for each N;
- $(N, L) \in CS(N)$, for each N such that $N \lhd^* \bot$, and each $L \in \mathcal{N}^*$;
- $N \in CS(N)$, for each N such that $\neg(N \lhd^* \bot)$; and
- for each N, $children(N) = M_1 \cdots M_m$ and $x_1 \in CS(M_1), \ldots, x_m \in CS(M_m)$ implies $x_1 \cdots x_m \in CS^+(N)$.

4 The recognizer

Relying on the functions defined in the previous section, we now explore the steps of the LR automaton, which as usual reads input from left to right and manipulates a stack.

We can divide the stack elements into two classes. One class contains the LR states from Q, the other contains elements of $\mathcal{M}$. A stack consists of an alternation of elements from these two classes. More precisely, each stack is an element from the following set of strings, given by a regular expression:

$$\mathcal{S} = q_{in}(\mathcal{M}Q)^*$$

Note that the bottom element of the stack is always q_{in}. We will use the symbol Δ to range over stacks and substrings of stacks, and the symbol X to range over elements from $\mathcal{M}$.

A configuration (Δ, w) of the automaton consists of a stack $\Delta \in \mathcal{S}$ and a remaining input w. The steps of the automaton are given by the binary relation $\vdash$ on pairs of configurations. There are three kinds of step:

shift $(\Delta q, aw) \vdash (\Delta q a q', w)$, provided $q' = goto(q, a) \neq \emptyset$.

reduce subtree $(\Delta q_0 X_1 q_1 X_2 q_2 \cdots X_m q_m, w) \vdash (\Delta q_0 (\bot, [N|L]) q', w)$, provided $N \in reductions(q_m)$, $X_1 \cdots X_m \in CS^+(N)$ and $q' = goto_\bot(q_0, N) \neq \emptyset$, where L is determined by the following. If for some j $(1 \leq j \leq m)$ X_j is of the form (M, L) then this provides the value of L, otherwise we set $L = []$.[2]

reduce aux tree $(\Delta q_0 X_1 q_1 X_2 q_2 \cdots X_m q_m, w) \vdash (\Delta q_0 X q', w)$, provided $t \in reductions(q_m)$, $X_1 \cdots X_m \in CS(R_t)$ and $q' = goto(q_0, N) \neq \emptyset$, where we obtain node N from the (unique) X_j $(1 \leq j \leq m)$ which is of the form $(M, [N|L])$, and set $X = N$ if $L = []$ and $X = (N, L)$ otherwise.[3]

The shift step is identical to that for context-free LR parsing. There are two reduce steps that must be distinguished. The first takes place when a subtree of an elementary tree t has been recognized. We then remove the stack symbols corresponding to a cross-section through that subtree, together with the associated LR states. We replace these by 2 other symbols, the first of which corresponds to the foot of an auxiliary tree, and the second is the associated LR state. In the case that some node M of the cross-section dominates the foot of t, then we must copy the associated list L to the first of the new stack elements, after pushing N onto that list to reflect that the spine has grown one segment upwards.

The second type of reduction deals with recognition of an auxiliary tree. Here, the head of the list $[N|L]$, which indicates the node at which the auxiliary tree t has been adjoined according to previous bottom-up calculations, must match a node that occurs directly above the root node of the auxiliary tree; this is checked by the test $q' = goto(q_0, N) \neq \emptyset$.

Input v is recognized if $(q_{in}, v) \vdash^* (q_{in}\Delta q, \varepsilon)$ for some Δ and $q \in Q_{fin}$. Then Δ will be of the form $X_1 q_1 X_2 q_2 \cdots q_{m-1} X_m$, where $X_1 \cdots X_m \in CS(R_t)$, for some $t \in I$.

Up to now, it has been tacitly assumed that the recognizer has some mechanism to its disposal to find the strings $X_1 \cdots X_m \in CS(R_t)$ and $X_1 \cdots X_m \in CS^+(N)$ in the stack. We will now explain how this is done.

For each N, we construct a deterministic finite automaton that recognizes the strings from $CS^+(N)$ from right to left. There is only one final state, which has no outgoing transitions. This is related to the fact that $CS^+(N)$ is suffix-closed. A consequence is that, given any stack that may occur and any N, there is at most one string $X_1 \cdots X_m \in CS^+(N)$ that can be found from the top of the stack downwards, and this string is found in linear time. For each $t \in I \cup A$ we also construct a deterministic finite automaton for $CS(R_t)$. The procedure for $t \in I$ is given in Figure 3, and an example of its application is given in Figure 4. The procedure for $t \in A$ is

[2] Exactly in the case that N dominates a footnote will (exactly) one of the X_j be of the form (M, L), some M.

[3] Exactly in the case that N dominates a footnote will $L \neq []$.

let $K = \emptyset$, $\mathcal{T} = \emptyset$;
let $s = fresh\_state$, $f = fresh\_state$;
$make\_fa(f, R_t, s)$.

procedure $make\_fa(q_1, M, q_0)$:
let $\mathcal{T} = \mathcal{T} \cup \{(q_0, M, q_1)\}$;
if $children(M)$ is defined
  then $make\_fa\_list(q_1, children(M), q_0)$
endproc.

procedure $make\_fa\_list(q_1, M\alpha, q_0)$:
if $\alpha = \varepsilon$
  then $make\_fa(q_1, M, q_0)$
  else let $q = fresh\_state$;
    $make\_fa\_list(q, \alpha, q_0)$; $make\_fa(q_1, M, q)$
endproc.

procedure $fresh\_state()$:
create some fresh object q;
let $K = K \cup \{q\}$; **return** q
endproc.

Figure 3: Producing a finite automaton $(K, N, \mathcal{T}, s, \{f\})$ that recognizes $CS(R_t)$, given some $t \in I$. K is the set of states, N acts as alphabet here, $\mathcal{T}$ is the set of transitions, s is the initial state and f is the (only) final state.

similar except that it also has to introduce transitions labelled with pairs (N, L), where N dominates a foot and L is a stack in $\mathcal{N}^*$; it is obvious that we should not actually construct different transitions for different $L \in \mathcal{N}^*$, but rather one single transition $(N, \_)$, with the placeholder "_" representing all possible $L \in \mathcal{N}^*$.

The procedure for $CS^+(N)$ can easily be expressed in terms of those for $CS(R_t)$.

5 Extended example

For the TAG presented in Figure 1, the algorithm from Schabes and Vijay-Shanker (1990) does not work correctly. The language described by the grammar contains exactly the strings abc, $a'b'c'$, $adbec$, and $a'db'ec'$. The algorithm from Schabes and Vijay-Shanker (1990) however also accepts $adb'ec'$ and $a'dbec$. In the former string, it acts as if it were recognizing the (ill-formed) tree in Figure 2: it correctly matches the part to the "south" of the adjunction to the part to the "north-east". Then, after reading c', the information that would indicate

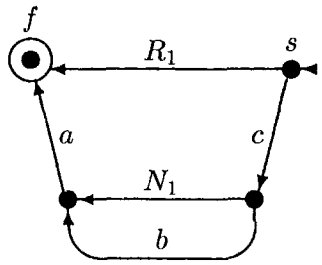

Figure 4: Example of the construction for $CS(R_1)$, where R_1 is the root node of α_1 (Figure 1).

whether a or a' was read is retrieved from the stack, but this information is merely popped without investigation. Thereby, the algorithm fails to perform the necessary matching of the elementary tree with regard to the part to the "north-west" of the adjunction.

Our new algorithm recognizes exactly the strings in the language. For the running example, the set of LR states and some operations on them are shown in Figure 5. Arrows labelled with nodes N represent the *goto* function and those labelled with $\perp(N)$ represent the $goto_\perp$ function. The initial state is 0. The thin lines separate the items resulting from the *goto* and $goto_\perp$ functions from those induced by the *closure* function. (This corresponds with the distinction between *kernel* and *nonkernel* items as known from context-free LR parsing.)

That correct input is recognized is illustrated by the following:

Stack	Input	Step
0	*adbec*	*shift a*
0 *a* 1	*dbec*	*shift d*
0 *a* 1 *d* 5	*bec*	*shift b*
0 *a* 1 *d* 5 *b* 7	*ec*	*reduce* N_1
0 *a* 1 *d* 5 $(\perp, [N_1])$ 9	*ec*	*shift e*
0 *a* 1 *d* 5 $(\perp, [N_1])$ 9 *e* 10	*c*	*reduce* β
0 *a* 1 N_1 3	*c*	*shift c*
0 *a* 1 N_1 3 *c* 6		*accept*

Note that as soon as all the terminals in the auxiliary tree have been read, the "south" section of the initial tree is matched to the "north-west" section through the goto function. Through subsequent shifts this is then matched to the "north-east" section.

This is in contrast to the situation when incorrect input, such as $adb'ec'$, is provided to the

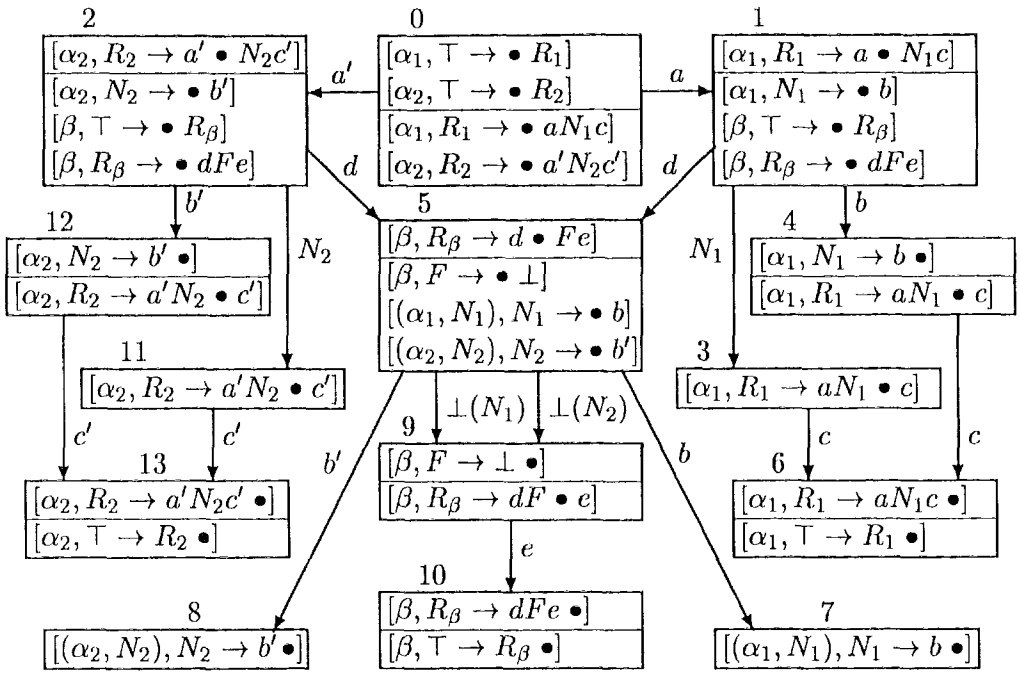

Figure 5: The set of LR states.

automaton:

Stack	Input	Step
0	$adb'ec'$	*shift* a
0 a 1	$db'ec'$	*shift* d
0 a 1 d 5	$b'ec'$	*shift* b'
0 a 1 d 5 b' 8	ec'	*reduce* N_2
0 a 1 d 5 $(\bot, [N_2])$ 9	ec'	*shift* e
0 a 1 d 5 $(\bot, [N_2])$ 9 e 10	c'	

Here, the computation is stuck. In particular, a reduction with auxiliary tree β fails due to the fact that $goto(1, N_2) = \emptyset$.

6 Extensions

The recognizer can be turned into a parser by attaching information to the stack elements from $\mathcal{M}$. At reductions, such information is gathered and combined, and the resulting data is attached to the new element from $\mathcal{M}$ that is pushed onto the stack. This can be used for computation of derived trees or derivation trees, and for computation of features. Since this technique is almost identical to that for the context-free case, it suffices to refer to existing literature, e.g. Aho et al. (1986, Section 5.3).

We have treated a classical type of TAG, which has adjunction as the only operation for composing trees. Many modern types of TAG also allow tree substitution next to adjunction. Our algorithm can be straightforwardly extended to handle tree substitution. The main changes that are required lie in the closure function, which needs an extra case (much like the corresponding operation in context-free LR parsing), in adding a third type of goto function, and in adding a fourth step, consisting of reduction of initial trees, which is almost identical to the reduction of auxiliary trees. The main difference is that all X_j are elements from $\mathcal{N}$; the X that is pushed can be a substitution node or a nonterminal (see also Section 7).

Up to now we have assumed that the grammar does not assign the empty string as label to any of the leaves of the elementary trees. The problem introduced by allowing the empty string is that it does not leave any trace on the stack, and therefore $CS(R_t)$ and $CS^+(N)$ are no longer suffix-closed. We have solved this by extending items with a third component E, which is a set of nodes labelled with ε that have been traversed by the closure function. Upon encountering a completed item $[\tau, N \to \alpha \bullet, E]$, a reduction is performed according to the sets $CS(R_t, E)$ or $CS^+(N, E)$, which are subsets of $CS(R_t)$ and $CS^+(N)$, respectively, containing only those cross-sections in which the nodes la-

belled with ε are exactly those in E. An automaton for such a set is deterministic and has one final state, without outgoing transitions.

7 Implementation

We have implemented the parser generator, with the extensions from the previous section. We have assumed that each set $Adjunct(N)$, if it is not $\{nil\}$, depends only on the nonterminal label of N. This allows more compact storage of the entries $goto_\perp(q, M)$: for a fixed state q and nonterminal B, several such entries where M has B as label can be collapsed into a single entry $goto'_\perp(q, B)$. The goto function for tree substitution is represented similarly.

We have constructed the LR table for the English grammar developed by the XTAG project at the University of Pennsylvania. This grammar contains 286 initial trees and 316 auxiliary trees, which together have 5950 nodes. There are 9 nonterminals that allow adjunction, and 10 that allow substitution. There are 21 symbols that function as terminals.

Our findings are that for a grammar of this size, the size of the LR table is prohibitively large. The table represented as a collection of unit clauses in Prolog takes over 46 MB for storage. The majority of this is needed to represent the three goto functions, which together require over 2.5 million entries, almost 99% of which is consumed by $goto$, and the remainder by $goto_\perp$ and the goto function for tree substitution. The reduction functions require almost 80 thousand entries. There are 5610 LR states. The size of the automata for recognizing the sets $CS(R_t, E)$ and $CS^+(N, E)$ is negligible: together they contain just over 15 thousand transitions.

The time requirements for generation of the table were acceptable: approximately 25 minutes were needed on a standard main frame with moderate load.

Another obstacle to practical use is the equivalent of hidden left recursion known from traditional LR parsing (Nederhof and Sarbo, 1996), which we have shown to be present in the grammar for English. This phenomenon precludes realization of nondeterminism by means of backtracking. Tabular realization was investigated by Nederhof (1998) and will be the subject of further research.

Acknowledgments

Anoop Sarkar provided generous help with making the XTAG available for testing purposes.

Parts of this research were carried out within the framework of the Priority Programme Language and Speech Technology (TST), while the author was employed at the University of Groningen. The TST-Programme is sponsored by NWO (Dutch Organization for Scientific Research). This work was further funded by the German Federal Ministry of Education, Science, Research and Technology (BMBF) in the framework of the VERBMOBIL Project under Grant 01 IV 701 V0.

References

A.V. Aho, R. Sethi, and J.D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley.

A.K. Joshi. 1987. An introduction to tree adjoining grammars. In A. Manaster-Ramer, editor, *Mathematics of Language*, pages 87–114. John Benjamins Publishing Company.

A. Kinyon. 1997. Un algorithme d'analyse LR(0) pour les grammaires d'arbres adjoints lexicalisées. In D. Genthial, editor, *Quatrième conférence annuelle sur Le Traitement Automatique du Langage Naturel, Actes*, pages 93–102, Grenoble, June.

M.-J. Nederhof and J.J. Sarbo. 1996. Increasing the applicability of LR parsing. In H. Bunt and M. Tomita, editors, *Recent Advances in Parsing Technology*, chapter 3, pages 35–57. Kluwer Academic Publishers.

M.-J. Nederhof. 1998. Linear indexed automata and tabulation of TAG parsing. In *Actes des premières journées sur la Tabulation en Analyse Syntaxique et Déduction (Tabulation in Parsing and Deduction)*, pages 1–9, Paris, France, April.

Y. Schabes and K. Vijay-Shanker. 1990. Deterministic left to right parsing of tree adjoining languages. In *28th Annual Meeting of the ACL*, pages 276–283.

S. Sippu and E. Soisalon-Soininen. 1990. *Parsing Theory, Vol. II: LR(k) and LL(k) Parsing*, volume 20 of *EATCS Monographs on Theoretical Computer Science.* Springer-Verlag.

M. Tomita. 1986. *Efficient Parsing for Natural Language.* Kluwer Academic Publishers.