AN EFFICIENT LR PARSER GENERATOR FOR TREE ADJOINING GRAMMARS

Carlos A. Prolo*
Dept. of Computer and Information Science
University of Pennsylvania
Philadelphia, PA, 19104, U.S.A.
prolo@linc.cis.upenn.edu

Abstract

The first published LR algorithm for Tree Adjoining Grammars (TAGs [Joshi and Schabes, 1996]) was due to Schabes and Vijay-Shanker [1990]. Nederhof [1998] showed that it was incorrect (after [Kinyon, 1997]), and proposed a new one. Experimenting with his new algorithm over the XTAG English Grammar[XTAG Research Group, 1998] he concluded that LR parsing was inadequate for use with reasonably sized grammars because the size of the generated table was unmanageable. Also the degree of conflicts is too high.

In this paper we discuss issues involved with LR parsing for TAGs and propose a new version of the algorithm that, by maintaining the degree of prediction while deferring the "subtree reduction", dramatically reduces both the average number of conflicts per state and the size of the parser.

1 Introduction

For Context Free Grammars, LR parsing [Knuth, 1965, Aho et al., 1986] can be viewed as follows. If at a certain state q_0 of the LR automaton, during the parsing of a sentence, we expect to see the expansion of a certain non-terminal A, and there is a production $A \rightarrow X_1\ X_2\ X_3\ ...\ X_n$ in the grammar, then the automaton must have a path labeled $X_1\ X_2\ X_3\ ...\ X_n$ starting at q_0. This is usually represented by saying that each state in the path contains a "dotted item" for the production, starting with $A \rightarrow \bullet\ X_1\ X_2\ X_3\ ...\ X_n$ at q_0, with the dot moving one symbol ahead at each state in the path. We will refer to the last state of such paths as *final.* The dot being in front of a symbol X_i represents the fact that we expect to see the expansion of X_i in the string. If X_i is a non-terminal, then, before crossing to the next state, we first have to check that some expansion of X_i is actually next in the input.

The situation is depicted in Figure 1, where paths are represented as winding lines and single arcs as straight lines. At a certain state q_1, where some possible yield of the prefix α_1 of a production $A \rightarrow \alpha_1 B \alpha_2$ has just been scanned, the corresponding dotted item is at a non-terminal B. This state turns out to be itself the beginning of other paths, like β in the picture, that lead to the recognition of some match for B through some of its rules. The machine, guided by the input string, could traverse this sub-path until getting to q_4. At this final state, some sort of memory is needed to get back to the previous path for A, to then cross from q_1 to q_2 (i.e. B has just been seen). In LR parsing this

*I am grateful to Aravind Joshi, Joseph Rosenzweig, Anoop Sarkar, Mahesh Viswanathan, Fei Xia, and the very kind IWPT 2000 reviewers, for their contribution to the development of this paper.

is realized by a stack that stores the sequence of states traversed during recognition. At each final state, the recognized production is unwound from the stack, in an operation called a reduction, leaving exposed the state q_1 and making a transition over the non-terminal B to q_2 (a *goto* transition).

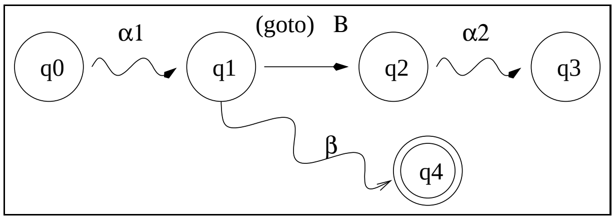
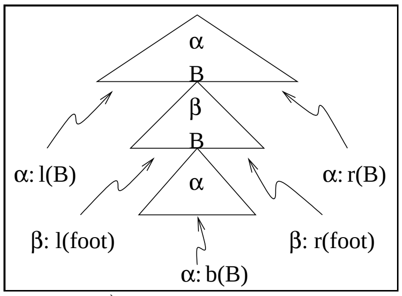


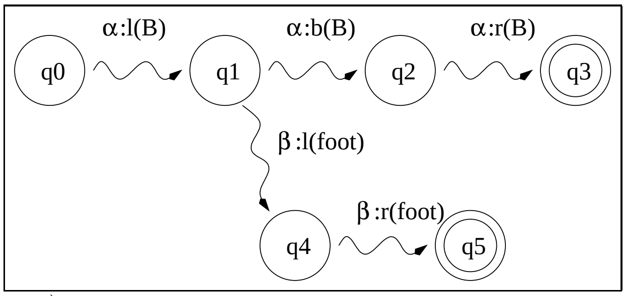
Figure 1: LR parsing for Context Free Grammars

A reduction means a commitment to a certain substructure in the attempt to find a parse for the sentence. Whenever the parser reaches a state that has a reduction (a final state), and there are also additional *reduce* operations or a *shift*, it has decide whether to do that reduction or try another alternative. That is certainly a limitation of the method, but at least, the decision has to be taken only after witnessing that the input has a complete yield for the production. That is exactly what will be hard to guarantee in the TAG case.

The problem with LR parsing for TAGs is adjunction, as illustrated in Figure 2. Figure 2.a sketches the adjunction of a tree β at a node labeled B of a tree α. $\alpha : l(B)$ is the part of α that appears before (left of) node B. Similarly, $\alpha : b(B)$ and $\alpha : r(B)$ are the parts below and to the right of B respectively. $\beta : l(foot)$ and $\beta : r(foot)$ are the two halves of β split by its spine.



a) The adjunction



b) The paths of computation for the adjunction

Figure 2: The problem of adjunction

There are two natural paths the LR parsing nature would induce the machine to have (Figure 2.b). One traverses α: state q_1 is the natural candidate to predict what comes below B (even after having recognized the left side of the adjoined tree). And State q_2 in the same way is the natural state to start the prediction of what comes after B. As for the other path, state q_1 predicts the adjunction of β, more specifically its left side. And the right side (i.e., past the foot node) is naturally predicted at q_4. Now let's look at the dynamic path of computation, corresponding to the projection of each segment in the sentence. This is:

$$q_0 \Rightarrow \alpha : l(B) \Rightarrow q_1 \Rightarrow \beta : l(foot) \Rightarrow q_4 \Rightarrow (q_1) \Rightarrow \alpha : b(B) \Rightarrow q_2 \Rightarrow (q_4) \Rightarrow \beta : r(foot) \Rightarrow$$
$$\Rightarrow q_5 \Rightarrow (q_2) \Rightarrow \alpha : r(B) \Rightarrow q_3$$

Although we marked only q_3 and q_5 as final, as we would like it to be, since they mark the end of the recognition of each tree, there are two other points of disruption at states q_4 and q_2. In a normal LR model of computation these points would correspond to reduction-like operations with *goto*'s to jump to the resuming state. Two problems then immediately arise: one that we have called "early reduction"; the other is the unsuitability of the normal stack model.

Figure 3 shows an intuitive but rather naive first attempt to solve the problem. At the State q_4 a reduction would be triggered sending the machine via a "*goto-below*" to state q_6. But notice that this operation would pop out all the states after q_1, including q_4. Later, at q_2, when a new reduction would be required, we do not know anymore how to get back to the state that contains the "*goto-foot*", the popped out q_4, unless we change the underlying storing model to something different from a stack. We are not finished. Even if we could get to q_8, we have to keep q_2 alive somehow, because when reaching q_5 we need it to access the "*goto-right*" to state q_7. It is important to note that we do not know in advance (i.e., at compilation time) which states those are. They are dependent on the actual computation. Although we could look for another storing model, the graver problem is the early reduction: the need to commit to tree β, before seeing its right side in the input.

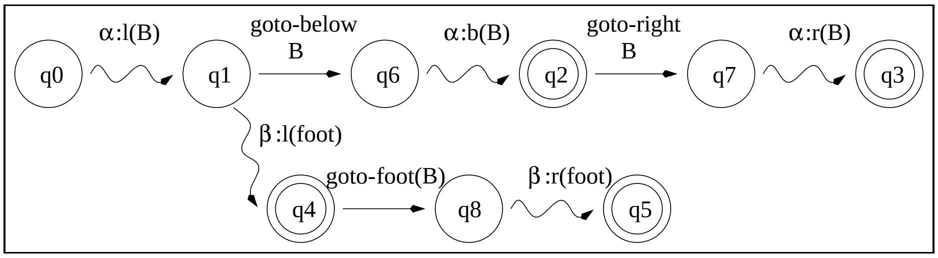


Figure 3: A naive solution for adjunction

Our second model, used by Nederhof[1998], shown in Figure 4, is clearly an enhancement of the naive one, as it increases the prediction capacity. The prediction of $\alpha : b(B)$ is made at q_4, instead of q_1. Hence we lose the need for the first reduction mentioned above. The reduction in q_2 does not cause any trouble since the state from where we want to take the *goto*, q_4, will be in the stack at that moment and is easy to recover. At q_5 the reduction takes its *goto-right* to q_7 from q_1, which is recoverable from the stack, instead of q_2 as in the previous approach, which has been popped out. Alas, things are not that simple.

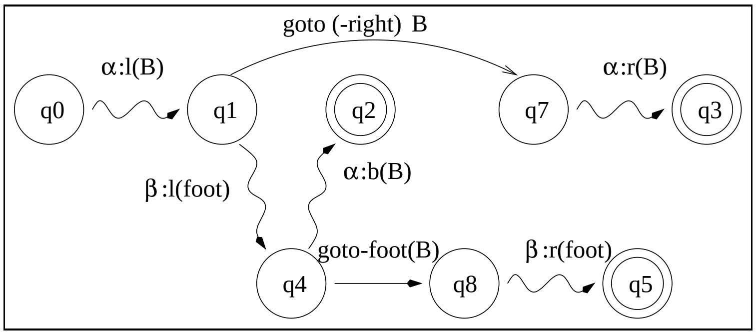


Figure 4: Nederhof's solution for adjunction

The problem of "early-reduction" is still significant, although not as much as in the first model. When the parser is in a state where (as one of the alternatives) it has just finished recognizing the subtree below a node where something has adjoined, as in state q_2 in Figure 4, somehow it has to get back to state q_4, and with a *goto* (*goto-foot*) to resume at state q_8 the recognition of the adjoined tree. In Nederhof's approach this is done by performing a reduction, that is, choosing one of the items in q_2 that signifies that a subtree below an adjunction node has been recognized (e.g., the subtree corresponding to $\alpha : b(B)$), and promoting a reduction of that subtree. This actually means committing to tree α too early, before seeing whether the rest of the input matches $\alpha : r(B)$. In the example in Figure 5, immediately after seeing a prefix "N", as in (1), that could be anchoring any of the innumerably many α_i trees generating continuations such as in (2), the parser would have to commit in advance to one of them, due to the need to turn back to the right side of the relative clause tree β.[1] The most visible consequence of this problem in the table is the huge rate of conflicts involving this kind of reduction that we drastically reduce with our approach as shown in Section 4.

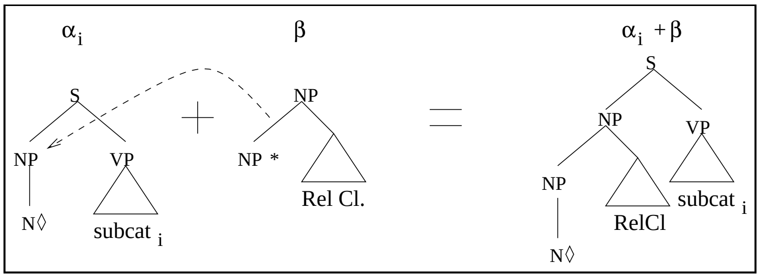


Figure 5: Example of early reduction

(1) (the) cat/N ...
(2) ... [that John saw] died.
... [that John saw] ate an apple.
... [that John saw] asked for MEOW MIX.

The second problem is an immediate consequence of the first. *goto*'s are traditionally made on a symbol basis. In Nederhof's approach, however, because we have already committed to a (certain node of a) tree at state q_2, a strategy of having one *goto* per node is adopted. Besides increasing the number of states, it simultaneously causes an explosion in the number of *goto* transitions per state, making the size of the table unmanageable. What we had in mind when looking for a new algorithm was precisely to solve those two problems.

There is still a third problem, related to the lack of the valid prefix property. Predicting the path $\alpha : b(B)$ below a node where adjunction is supposed to take place after the left part of the adjoined tree is hard. Actually, we can show it to be impossible with Nederhof's approach as well as our own (although it would not be a problem for the "naive" solution).[2] An attempt to do that in the algorithm for table generation would lead to non-termination for some grammars. Hence, at state q_4 all items corresponding to adjunction nodes labeled B in some tree are inserted, with the dot past the node, even if many of those nodes were not possible at that state, for a particular input prefix. Although in

[1]The example given is for explanatory purposes, and one could claim that the subjects of the α_i trees should be substitution nodes, for instance, which is true. The problem of early reduction described, however, largely appears in more subtle contexts.

[2]Recall the natural candidate for such prediction would be q_1.

our approach the effects of misprediction and overgeneration are less harmful, it decisively affects the design of the algorithm as we see below.

Figure 6 sketches our proposed solution. The *goto* now depends on both q_1 and q_2. At state q_2, the items that correspond to the end of the recognition of bottom subtrees labeled B, like $\alpha{:}b(B)$, are grouped into subsets of the same size, size being the number of leaves of the subtree. Hence *goto-adj*(q_1, q_2, B, l) has the set of possibilities of continuation like $\alpha{:}r(B)$ that were predicted at q_1 and confirmed at q_2 that have l leaves under B. The dependency on q_1 has the sole purpose of fixing the overprediction we mentioned above, originating at q_4, and propagated to q_2. When q_2 is reached, an action *bpack(B,l)* extracts from the stack the l nodes under B ($\alpha{:}b(B)$, uncovering q_4 and the *goto-foot*), and puts them back in the stack as a single embedded stack element. The material is unpacked during β reduction before moving to q_7.[3] The details of the algorithm are in the next section.

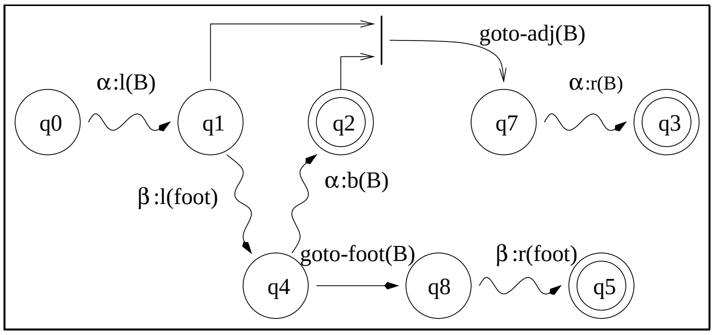


Figure 6: Proposed solution for adjunction

2 The Algorithm

2.1 The Table Generation for the NA/OA Case

We describe, in this subsection, an algorithm for a TAG grammar in which all nodes are marked either NA(for Null Adjunction) or OA(Obligatory Adjunction). Later we deal with the general case. We do not consider selective adjunction (SA). To refer to the symbol (label) of a node n, we use $symb(n)$. ϵ is used to label anchors to represent the absence of a terminal symbol (the "empty" label).

A *dotted node* is a pair (n, pos) where n is a tree node and pos, as in [Schabes, 1990], can be la (at the left and above the node), lb (left and below), rb (right and below), or ra (right and above). We also represent the dotted nodes pictorially as ${}^\bullet n$, ${}_\bullet n$, $n_\bullet$, and $n^\bullet$. A *dotted item* (*item*, for short) is a pair (t, dn), where t is an elementary tree, and dn a dotted node whose node component belongs to t.

We define a congruence relation $\cong$ as the least symmetric and transitive relation such that, for any pair of items $i_1 = (t, dn_1)$ and $i_2 = (t, dn_2)$ of the same tree t, $i_1 \cong i_2$ if any of the following conditions apply:

[1.] $dn_1 = {}^\bullet n$, $dn_2 = {}_\bullet n$, and n is marked for null adjunction (NA).
[2.] $dn_1 = {}_\bullet n$, $dn_2 = n_\bullet$, and n is an anchor labeled ϵ.
[3.] $dn_1 = {}_\bullet n$, $dn_2 = {}^\bullet m$, and m is the leftmost child of n.
[4.] $dn_1 = n_\bullet$, $dn_2 = n^\bullet$, and n is marked for null adjunction (NA).

[3] This is quite similar to how stacks of stacks are used in the theories of automata for TAGs. Also the model resembles the one used in [Schabes and Vijay-Shanker, 1990] in their attempt.

[5.] $dn_1 = n_\bullet$, $dn_2 = m^\bullet$, and m is the rightmost child of n,
[6.] $dn_1 = n^\bullet$, $dn_2 = {}^\bullet m$, and m is the right sibling of n.

Congruent items are indistinguishable for the purpose of our algorithm and its underlying theory. Hence instead of dealing with separate items, we use equivalence classes under $\cong$. If i is a term, $[i]$ is the congruence class to which i belongs. Each congruence class has one and only one *active* item, where an active item has one of the following forms:

- $(t, {}^\bullet n)$, where n is marked OA: this item triggers adjunction on n.
- $(t, {}_\bullet n)$, where n can be: a substitution node (triggers substitution), a non-ϵ anchor (triggers a *shift* action), or a foot node (triggers the return to the tree where adjunction occurred).
- $(t, n_\bullet)$, where n is marked OA: this item triggers a *bpack* action, that we will define later.
- $(t, n^\bullet)$, where n is the root of t: triggers a *reduce* (alpha or beta) operation.

Given a set S of dotted items (under $\cong$) of a TAG grammar, we define $closure(S)$ as the minimal set that satisfies the following conditions:

1. if $[i] \in S$, then $[i] \in closure(S)$.
2. if $[(t, {}_\bullet n)] \in closure(S)$, and n is a substitution node, then, for every initial tree t_1 with root node r labeled $symb(n)$, $[(t_1, {}^\bullet r)] \in closure(S)$.
3. if $[(t, {}^\bullet n)] \in closure(S)$ and n is a node marked OA, then, for every auxiliary tree t_1 with root node r labeled $symb(n)$, $[(t_1, {}^\bullet r)] \in closure(S)$.
4. if $[(t, {}_\bullet n)] \in closure(S)$, where n is a foot node, then, for every OA node m of the grammar such that $symb(m) = symb(n)$, $[(t_1, {}_\bullet m)] \in closure(S)$, where t_1 is the elementary tree that contains m. We point out that it is at this closure operation that the loss of the valid prefix property occurs.

To define a parsing table for a grammar G with goal symbol S, we first extend G by adding one new tree called *start*, with two nodes: the root, labeled with a fresh symbol ST, marked NA; and ST's single child, a substitution node labeled S. Then, let I be the set of all equivalence classes of items of G under $\cong$. Let N be the set of symbols, $T \subseteq N$ be the set of symbols that appear in some anchor, and $\mathbb{N}$ the set of non-negative integers. We define the "parsing table" as a set $Q \subseteq 2^I$ of states with initial state $q_0 = closure(\{[(start, {}^\bullet ST)]\}) \in Q$, together with the functions $GOTO_{subst} : Q \times N \to Q$, $GOTO_{foot} : Q \times N \to Q$, $GOTO_{adj} : Q \times Q \times N \times \mathbb{N} \to Q$, and $ACTIONS : Q \times T \to 2^A$, where $A = \{\text{shift } q \mid q \in Q\} \cup \{\alpha\text{-reduce } t | t \text{ is an alpha tree }\} \cup \{\beta\text{-reduce } t | t \text{ is a beta tree }\} \cup \{\text{bpack } (A, l) | A \in N, l \in \mathbb{N}\} \cup \{\text{accept}\}$. Q, $GOTO_{subst}$, $GOTO_{foot}$, $GOTO_{adj}$, and $ACTIONS$ are the minimal set and functions that satisfies the following inductive definition.

1. $q_0 \in Q$.
2. If $q \in Q$ and $p = \{[(t, n_\bullet)] \mid [(t, {}_\bullet n)] \in q$ and n is an anchor $\}$, then $p' = closure(p) \in Q$, and (shift p') $\in ACTIONS(q, symb(n))$.
3. If $q \in Q$ and $[(t, n^\bullet)] \in Q$, where n is the root of t, then, for every $a \in T$: if t is an initial tree, then (α-reduce t) $\in ACTIONS(q, a)$, else (β-reduce t) $\in ACTIONS(q, a)$.
4. If $q \in Q$ and $[(t, n_\bullet)] \in Q$, where n is marked OA, then, for every $a \in T$, (bpack($symb(n), l$)) $\in ACTIONS(q, a)$, where l is the number of non-ϵ leaves under n.

5. If $q \in Q$ and $[(start, ST^{\bullet})] \in Q$, then accept $\in ACTIONS(q, \$)$.

6. If $q \in Q$ and $p = \{[(t, n_{\bullet})] \mid [(t, {}_{\bullet}n)] \in q$ and n is a substitution node, then $p' = closure(p) \in Q$, and $GOTO_{subst}(q, symb(n)) = p'$.

7. If $q \in Q$ and $p = \{[(t, n_{\bullet})] \mid [(t, {}_{\bullet}n)] \in q$ and n is a foot node, then $p' = closure(p) \in Q$, and $GOTO_{foot}(q, symb(n)) = p'$.

8. If $q_1, q_2 \in Q$, and for some $k \in \mathbb{N}$, $p = \{[(t, n^{\bullet})]) \mid [(t, {}^{\bullet}n)] \in q_1, [(t, n_{\bullet})] \in q_2$, n is marked OA, and the number of leaves under n in t equals $k\}$, then $p' = closure(p) \in Q$, and $GOTO_{adj}(q_1, q_2, symb(n), k) = p'$.

The state corresponding to the empty set is called *error*. The domain of k, in practice, is bounded by the grammar: the maximum number of leaves of any node. As usual, if an entry contains more than one action, we say that there is a *conflict* in the entry. As expected, two *shift* actions are never in conflict. Although *reduce* and *bpack* actions do not actually depend on a terminal symbol for the current algorithm, in practice the table could be so constrained, either by having the algorithm extended to an LR(1) version or by using empirical evidence to reduce/resolve conflicts. In our inductive definition, each state was associated with the closure of a (usually much smaller) set of items. This set prior to the closure is generally known as its *kernel*. It is a property of our framework that an alternative definition whose states are associated with kernels would produce an automaton isomorphic to the one we defined above.

2.2 The Driver

The algorithm for the driver presented below uses a stack. Let *st*, st_1 be stacks and *el* be a stack element. **PUSH** (*el*,*st*), **TOP** (*st*) and **POP** (*st*) have the usual meanings. **POP** (*st*, *k*) pops out the top k elements from *st*. **EXTRACT** (*st*, *k*) pops out the top k elements returning another stack with the k elements in the same order they were in *st*. Its counterpart, **INSERT** (*st1*, *st*), pushes onto *st* all elements in *st1*, again preserving the order. null is the empty stack. size(*st*) is the number of elements in the stack *st*. The stack is a sort of higher order structure, in the sense that an element of the stack may contain an embedded stack. More precisely, an element of the stack is a pair (X, q), where q is a state of the parsing table, and X is either a grammar symbol or another stack.

The algorithm for the driver also uses *input*, the sequence of symbols to be recognized. Two operations are defined over it: **look**, which returns the leftmost symbol of *input* or \$ if *input* is the null sequence; and **advance**, which removes the leftmost symbol from *input*.

Let $ACTIONS$, $GOTO_{subst}$, $GOTO_{foot}$, and $GOTO_{adj}$ be the four tables/functions for a grammar G. Let q_0 be the initial state of the corresponding machine. Let *input* and the stacks *stack*, *emb-stack* be as defined above. The algorithm for the driver is then as follows.

```
stack = null
PUSH (<null,q0 >, stack)
forever
    let <−, state> = TOP (stack)
    let lookahead = look (input)
    if ACTIONS (state, lookahead) = error then return failure
    else  let action be in ACTIONS (state, lookahead)      (a non-deterministic choice)
          case action of:
              shift p:
```

```
    PUSH (<lookahead,p >, stack)
    advance (input)
α-reduce t:
    let k be the number of non-empty leaves of t
    let A be the symbol at the root of t
    POP (stack, k)
    let <−, state1> = TOP (stack)
    PUSH (<A,GOTO_subst(state1,A)>, stack)
β-reduce t:
    let kl and kr be the number of non-empty leaves of t
            respectively to the left and to the right of the foot of t
    POP (stack, kr)
    let <aux-stack,-> = TOP (stack)
    POP (stack)
    let k = size (aux-stack)
    let <-,state2> = TOP (if k > 0 then aux-stack else stack)
    POP (stack, kl)
    let <-, state1> = TOP (stack)
    INSERT (aux-stack, stack)
    let A be the symbol at the root of t
    let state = GOTO_adj (state1,state2,A,k)
    if state = error then return failure
            (a consequence of the lack of the prefix property)
    let <el,- > = TOP (stack)
    POP (stack)
    PUSH (<el, state>, stack)
bpack (A,k):
    emb-stack = EXTRACT (stack, k)
    let <-, state1>= TOP (stack)
    PUSH (<emb-stack, GOTO_foot (state1,A)>, stack)
accept:
    return success
```

2.3 The General Case

The obvious way to handle the general case is to transform the arbitrary input grammar into an equivalent one with all nodes marked NA/OA prior to the application of the algorithm of Subsection 2.1. For every TAG grammar G, there is a grammar G' equivalent to G with respect to the possible derivations, whose nodes are all OA/NA marked. For instance, given the grammar G, in Figure 7,[4] we can construct G' in Figure 8 by replacing each tree t of G by 2^n new trees, where n is the number of unmarked nodes of t. Each new tree corresponds to one of the possible assignments of marks NA/OA to each of the unmarked nodes. In fact, in a TAG derivation, a tree instance together with the Gorn addresses at which other trees have adjoined defines exactly one of these G' component trees.

The approach we implemented, however, takes advantage of the original compact representation, avoiding exploding the number of trees, and hence items and item sets. An OA/NA tree can be represented in items as the original unmarked one plus a list of the OA nodes, much like the Gorn addresses in derivations. It turns out that certain distinctions are irrelevant for the algorithm. For

[4]Substitution nodes are always assumed to be NA,even if not explicitly marked as such in the trees. Nodes are represented by their label, e.g. NP, plus an optional subscript for identification purposes, as in NP, NP_s, and NP_o. Subscripts are ignored by the algorithm.

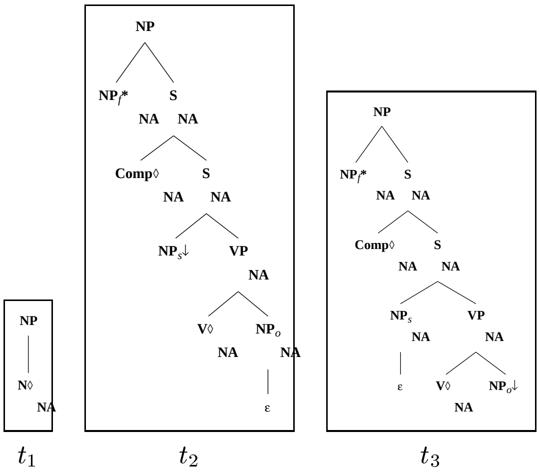


Figure 7: TAG grammar G for relative clauses

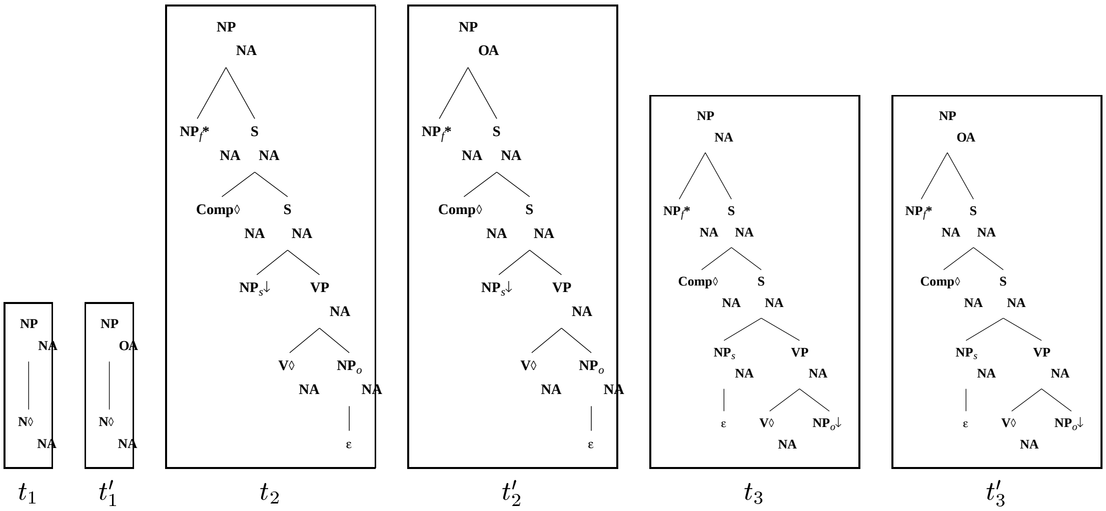


Figure 8: TAG grammar G', equivalent to G, with only OA/NA nodes

instance, it is not relevant to identify in the item whether tree nodes to the right of the dotted item allow for adjunction or not. In fact markings at nodes that come "after" the dotted node according to the usual dot traversal order defined in [Schabes, 1990] are not distinguishable. Our algorithm only keeps track of the innermost OA node that dominates the dotted node in an item, the only strong requirement, so that the *bpack* operation and the $GOTO_{adj}$ function can be obtained. We omit details of changes to be made in the algorithm of Subsection 2.1. Mostly they concern closure rule 4, and rules 4 and 8 in the definition of the automaton.

One last word: the approach we took is not totally equivalent to the bare OA/NA decomposition version. Keeping track of the history of adjunction for nodes that appear before the dotted node would generate tables with slightly more fine grain distinctions. Whether this could have some practical importance is still to be investigated.

3 An Example

The set of states and the $GOTO_{adj}$ function produced by the OA/NA algorithm for grammar G' of Figure 8 are shown below. Non-*shift* actions were defined together with the states. Functions $GOTO_{foot}$, $GOTO_{subst}$, and *shift* actions are better viewed as a graph, in Figure 9. Figure 10 shows the parsing sequence for the string "girls/N that/Comp John/N likes/V".

S_0 : { [($start$, ${}^\bullet$NP)], [(t_1, ${}_\bullet$N)], [(t_1', ${}^\bullet$NP)], [(t_2, ${}_\bullet$NP$_f$)], [(t_2', ${}^\bullet$NP)], [(t_3, ${}_\bullet$NP$_f$)], [(t_3', ${}^\bullet$NP)], [(t_1', ${}_\bullet$N)], [(t_2', ${}_\bullet$NP$_f$)], [(t_3', ${}_\bullet$NP$_f$)] }

S_1 : { [($start$, ST${}^\bullet$)] } $\quad$ $ACTIONS$(S_1,\$) = { accept }

S_2 : { [(t_1, NP${}^\bullet$)], [(t_1', NP${}_\bullet$)] } $\quad$ $ACTIONS$(S_2,-) = { α-reduce t_1, bpack(NP,1) }

S_3 : { [(t_2, ${}_\bullet$Comp)], [(t_3, ${}_\bullet$Comp)], [(t_2', ${}_\bullet$Comp)], [(t_3', ${}_\bullet$Comp)] }

S_4 : { [(t_2, ${}_\bullet$NP$_s$)], [(t_3, ${}_\bullet$V)], [(t_2', ${}_\bullet$NP$_s$)],[(t_3', ${}_\bullet$V)], [(t_1, ${}_\bullet$N)], [(t_1', ${}^\bullet$NP)], [(t_2, ${}_\bullet$NP$_f$)], [(t_2', ${}^\bullet$NP)], [(t_3, ${}_\bullet$NP$_f$)], [(t_3', ${}^\bullet$NP)], [(t_1', ${}_\bullet$N)], [(t_2', ${}_\bullet$NP$_f$)], [(t_3', ${}_\bullet$NP$_f$)] }

S_5 : { [(t_2, ${}_\bullet$V)], [(t_2', ${}_\bullet$V)] }

S_6 : { [(t_3, ${}_\bullet$NP$_o$)], [(t_3', ${}_\bullet$NP$_o$)], [(t_1, ${}_\bullet$N)], [(t_1', ${}^\bullet$NP)], [(t_2, ${}_\bullet$NP$_f$)], [(t_2', ${}^\bullet$NP)], [(t_3, ${}_\bullet$NP$_f$)], [(t_3', ${}^\bullet$NP)], [(t_1', ${}_\bullet$N)], [(t_2', ${}_\bullet$NP$_f$)], [(t_3', ${}_\bullet$NP$_f$)] }

S_7 : { [(t_2, NP${}^\bullet$)], [(t_2', NP${}_\bullet$)] } $\quad$ $ACTIONS$(S_7,-) = { β-reduce t_2, bpack(NP,4) }

S_8 : { [(t_3, NP${}^\bullet$)], [(t_3', NP${}_\bullet$)] } $\quad$ $ACTIONS$(S_8,-) = { β-reduce t_3, bpack(NP,4) }

S_9 = $GOTO_{adj}$(S_0,S_2,NP,1) = $GOTO_{adj}$(S_4,S_2,NP,1): { [(t_1', NP${}^\bullet$)] }
$\quad$ $ACTIONS$(S_9,-) = { β-reduce t_1' }

S_{10}= $GOTO_{adj}$(S_0,S_7,NP,4) = $GOTO_{adj}$(S_4,S_7,NP,4): { [(t_2', NP${}^\bullet$)] }
$\quad$ $ACTIONS$(S_{10},-) = { β-reduce t_2' }

S_{11}= $GOTO_{adj}$(S_0,S_8,NP,4) = $GOTO_{adj}$(S_4,S_8,NP,4): { [(t_3', NP${}^\bullet$)] }
$\quad$ $ACTIONS$(S_{11},-) = { β-reduce t_3' }

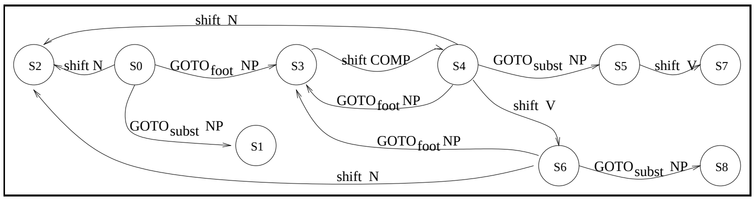


Figure 9: *shift* actions and functions $GOTO_{subst}$, $GOTO_{foot}$ for G'

4 Evaluation of the Algorithm

We show in Table 11 the results of the application of our algorithm as well as Nederhof's[5] to a recent version of the XTAG grammar with 1009 trees and 11490 nodes. *conflicts* is the average number of actions per pair (state,terminal). *reductions* and *bpacks* are respectively the average number of *reduce* and *bottom pack* actions in a state.[6] *transitions* is the total number of transition entries, including *shift*'s and *goto*'s. The number of conflicts is drastically reduced with our algorithm. In particular the nasty early reduction(*bpack*) conflicts are decreased, improving the general quality of the conflicts

[5]We reimplemented Nederhof's algorithm

[6]For Nederhof's approach *reduce* stands for "reduce tree" and *bpack* stands for "reduce subtree".

stack	input	sel. action
$[(\text{-},S_0)]$	girls/N ...	shift S_2
$[(\text{-},S_0)$ (girls/N, S_2)]	that/Comp ...	bpack (NP,1)
$[(\text{-},S_0)$ ([(girls/N, S_2)], S_3)]	that/Comp ...	shift S_4
$[(\text{-},S_0)$ ([(girls/N, S_2)], S_3) (that/Comp, S_4)]	John/N ...	shift S_2
$[(\text{-},S_0)$ ([(girls/N, S_2)], S_3) (that/Comp, S_4) (John/N, S_2)]	likes/V \$	α-reduce t_1
$[(\text{-},S_0)$ ([(girls/N, S_2)], S_3) (that/Comp, S_4) (NP, S_5)]	likes/V \$	shift S_7
$[(\text{-},S_0)$ ([(girls/N, S_2)], S_3) (that/Comp, S_4) (NP, S_5) (likes/V, S_7)]	\$	β-reduce t_2
$[(\text{-},S_0)$ (girls/N, S_9)]	\$	α-reduce t'_1
$[(\text{-},S_0)$ (NP, S_1)]	\$	accept

Figure 10: Parsing of "*girls/N that/Comp John/N likes/V*"

(shifting part of it to full tree reductions). The size of the table, roughly evaluated by the sum of the number of transitions and the number of action entries, is reduced by a factor of about 75 from unmanageable 75M to 1M. We also reported in the second half of the table the application of both algorithms to a grammar with 3161 trees and 22641 nodes, extracted from the Penn Treebank [Marcus et al., 1994] with the help of Fei Xia's accurate extractor [Xia, 1999].[7]

ALGORITHM	grammar	states	transitions	conflicts	reductions	bpacks
Nederhof's	XTAG	17490	40 M	114	1.93	120
Ours	XTAG	5861	202 K	7.6	3.2	5.1
Nederhof's	Treebank	22101	60 M	639	3.5	742
Ours	Treebank	13058	844 K	15.6	5.15	12.8

Figure 11: Summary of the parsing tables generated by the algorithms

5 Conclusion and Future Research

We aim at building a parser that, parsing sentences from left to right using LR techniques, is able to take sound decisions towards reaching their correct analysis. The algorithm we presented here is suited for a non-lexicalized TAG, and in fact the results we presented here are all for the non-lexicalized component (set of template trees) of LTAG grammars where the terminals are parts of speech. We were able to produce a parse table of small size, with reasonably low degree of conflicts, considering the degree of ambiguity inherent to the sets of trees.

The misprediction of bottom trees, due to lack of the valid prefix property, remains a concern. It is not clear whether a rescue of that property would have some significant impact in reducing conflicts in our current algorithm. Of course, as we said before, the property does not hold for the general case. But we are studying the subclass of TAG grammars for which our algorithm would never mispredict to see whether they are adequate for modeling Natural Language. That would mean, among other things,

[7] We set the parameters of the extractor to build a grammar with the same set of symbols as in the Penn Treebank, which is much higher than in the XTAG grammar, especially the terminals. In the version that converts the alphabet to the one of the XTAG project the numbers are much smaller for both algorithms, that is, the automaton makes much less distinction among states.

that function $GOTO_{adj}$ would depend only on one state (q_2 in Figure 6), a significant improvement in size.

We are currently working on an extension that takes into account the lexical items of the lexicalized LTAG grammars while keeping the size of the parsing table manageable. The effort we made in reducing the size of the generated table is essential due to the expected scaling up with the inclusion of lexicalization (which has already been confirmed in preliminary experiments).

The algorithm could be extended to have reductions dependent on lookaheads (e.g., an LALR(1) version), what is reasonably easy to do, that would not increase the number of states or transitions and would likely reduce substantially the degree of conflicts. The reason we did not do it is because we intend to use empirical evidence from annotated corpora to rank the conflict alternatives, for any given pair (state, lookahead), in the lexicalized version. Of course this subsumes the effect of the LR(1) version.

References

[Aho et al., 1986] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. 1986. *Compilers: principles, techniques, and tools.* Addison-Wesley, Reading, MA, USA.

[Joshi and Schabes, 1996] Aravind Joshi and Yves Schabes. 1996. Tree-adjoining grammars. In *Handbook of Formal Languages and Automata.* Springer-Verlag, Berlin.

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

[Knuth, 1965] Donald E. Knuth. 1965. On the translation of languages from left to right. *Information and Control*, 8(6):607–639.

[Marcus et al., 1994] Mitchell Marcus, Grace Kim, Mary Ann Marcinkiewicz, Robert MacIntyre, Ann Bies, Mark Ferguson, Karen Katz, and Britta Schasberger. 1994. The penn treebank: Annotating predicate argument structure. In *Proceedings of the 1994 Human Language Technology Workshop.*

[Nederhof, 1998] Mark-Jan Nederhof. 1998. An alternative LR algorithm for TAGs. In *Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics and 16th International Conference on Computational Linguistics*, Montreal, Canada.

[Schabes and Vijay-Shanker, 1990] Y. Schabes and K. Vijay-Shanker. 1990. Deterministic left to right parsing of tree adjoining languages. In *Proceedings of 28th Annual Meeting of the Association for Computational Linguistics*, pages 276–283, Pittsburgh, Pennsylvania, USA.

[Schabes, 1990] Yves Schabes. 1990. *Mathematical and Computational Aspects of Lexicalized Grammars.* Ph.D. thesis, Department of Computer and Information Science, University of Pennsylvania.

[Xia, 1999] Fei Xia. 1999. Extracting tree adjoining grammars from bracketed corpora. In *Proceedings of the 5th Natural Language Processing Pacific Rim Symposium(NLPRS-99)*, Beijing, China.

[XTAG Research Group, 1998] The XTAG Research Group. 1998. A Lexicalized Tree Adjoining Grammar for English. Technical Report IRCS 98-18, University of Pennsylvania.