Incremental Parser Generation for Tree Adjoining Grammars*

Anoop Sarkar
University of Pennsylvania
Department of Computer and Information Science
Philadelphia PA 19104
anoop@linc.cis.upenn.edu

2 February 1996

Abstract

This paper describes the incremental generation of parse tables for the LR-type parsing of Tree Adjoining Languages (TALs). The algorithm presented handles modifications to the input grammar by updating the parser generated so far. In this paper, a lazy generation of LR-type parsers for TALs is defined in which parse tables are created by need while parsing. We then describe an incremental parser generator for TALs which responds to modification of the input grammar by updating parse tables built so far.

1 Introduction

Tree Adjoining Grammars (TAGs) are tree rewriting systems which combine trees with the single operation of *adjunction* (see Figure 1). The construction of deterministic bottom-up left to right parsing of Tree Adjoining Languages (TALs)[1] (Schabes and Vijay-Shanker, 1990) is an extension of the LR parsing strategy for context free languages (Aho et al., 1986). Parser generation

*Thanks to Dania Egedi, Aravind Joshi, B. Srinivas and the student session reviewers.

[1]Familiarity with Tree Adjoining Grammars (TAGs) and their parsing techniques is assumed throughout the paper. For an introduction to TAGs, see (Joshi, 1987). We shall assume that our definition of TAG does not have the *substitution* operation. Refer to (Schabes, 1991) for a background on the parsing of TAGs.

involves precompiling as much top-down information as possible into a parse table which is used by the LR parsing algorithm. This paper gives an algorithm for the incremental generation of parse tables for the LR-type parsing of TAGs.

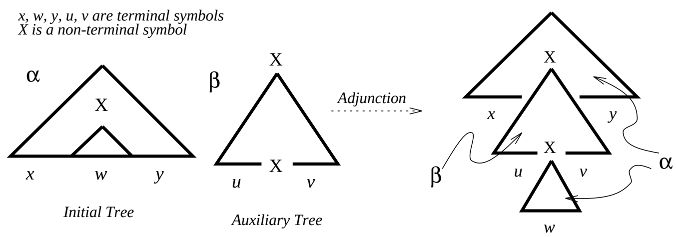

Figure 1: The Adjunction Operation

Parser generation provides a fast solution to the parsing of input sentences as certain information about the grammar is precompiled and available while parsing. However, if the grammar used to generate the parser is either dynamic or needs frequent modification then the time needed to parse the input is determined by both the parser and the parser generator.

The main application area for TAGs has been the description of natural languages. In such an area grammars are very rarely static, and modifications to the original grammar are commonplace. In such an interactive environment, conventional LR-type parsing suffers from the following disadvantages:

- Some parts of the grammar might never be used in the parsing of sentences actually given to the parser. The time taken by the parser generator over such parts is wasted.
- Usually, only a small part of the grammar is modified. So a parser generator should also correspondingly make a small change to the parser rather than generate a new one from scratch.

The algorithm described here allows the incremental incorporation of modifications to the grammar in a LR-type parser for TALs. This paper extends the work done on the incremental modification of LR(0) parser generators for CFGs in (Heering et al., 1990; Heering et al., 1989). We define a lazy and incremental parser generator having the following characteristics:

- The parse tables are generated in a lazy fashion from the grammar, i.e. generation occurs while parsing the input. Information previously precompiled is now generated depending on the input.
- The parser generator is incremental. Changes in the grammar trigger a corresponding change in the already generated parser. Parts of the parser not affected by the modifications in the grammar are reused.
- Once the needed parts of the parser have been generated, the parsing process is as efficient as a conventionally generated one.

Incremental generation of parsers gives us the following benefits:

- The LR-type parsing of lexicalized TAGs (Schabes, 1991). With the use of the lazy and incremental parser generation, lexicalized descriptions of TAGs can be parsed using LR-type parsing techniques. Parse tables can be generated without exhaustively considering all lexical items that anchor each tree.
- Modular composition of parsers, where various modules of TAG descriptions are integrated with recompilation of only the necessary parts of the parse table of the combined parser.

2 LR Parser Generation

(Schabes and Vijay-Shanker, 1990) describe the construction of an LR parsing algorithm for TAGs. Parser generation here is taken to be the construction of LR(0) tables (i.e. without any lookahead) for a particular TAG[2]. The moves made by the parser can be most succinctly explained by looking at an automaton which is weakly equivalent to TAGs called Bottom-Up Embedded Pushdown Automata (BEPDA) (Schabes and Vijay-Shanker, 1990)[3]. The storage of a BEPDA is a sequence of stacks (or pushdown stores) where stacks can be introduced above and below the top stack in the automaton. Recognition of adjunction can be informally seen to be equivalent to the **unwrap** move shown in Figure 2.

[2]The algorithm described here can be extended to a parser with SLR(1) tables (Schabes and Vijay-Shanker, 1990).

[3]Note that the LR(0) tables considered here are deterministic and hence correspond to a subset of the TALs. Techniques developed in (Tomita, 1986) can be used to resolve nondeterminism in the parser.

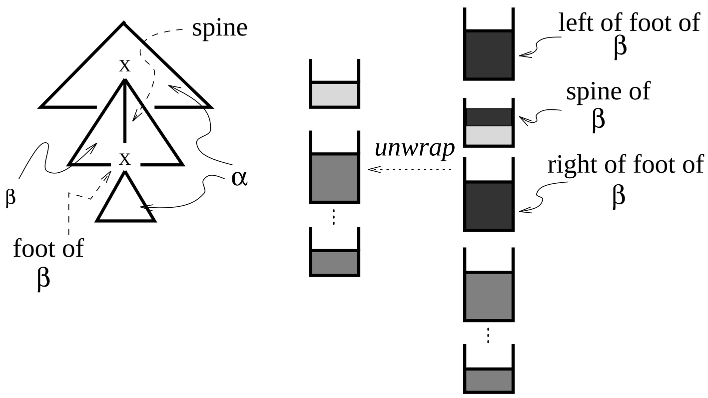

Figure 2: Recognition of adjunction in a BEPDA.

The LR parser uses a parsing table and a sequence of stacks (see Figure 2) to parse the input. The parsing table encodes the actions taken by the parser as follows (with the help of two *GOTO* functions):

- **Shift** to a new state which is pushed onto a new stack which appears on top of the current sequence of stacks.
- **Resume Right** where the parser has reached right and below a node on which an auxiliary tree has been adjoined. Figure 3 gives the two cases where the string beneath the foot node of an auxiliary tree has been recognized (in some other tree) and where the $GOTO_{foot}$ function encodes the proper state such that the right part of an auxiliary tree can be recognized.
- **Reduce Root** which causes the parser to execute an unwrap move to recognize adjunction (see Figure 2). The proper state for the parser after adjunction is given by the $GOTO_{right}$ function.
- **Accept** and **Error** functions as in conventional LR parsing.

Figure 4 shows how the concept of dotted rules for CFGs is extended to trees. There are four positions for a dot associated with a symbol: left above, left below, right below and right above. A dotted tree has one such dotted symbol. The tree traversal in Figure 4 scans the frontier of the tree from left to right while trying to recognize possible adjunctions between the

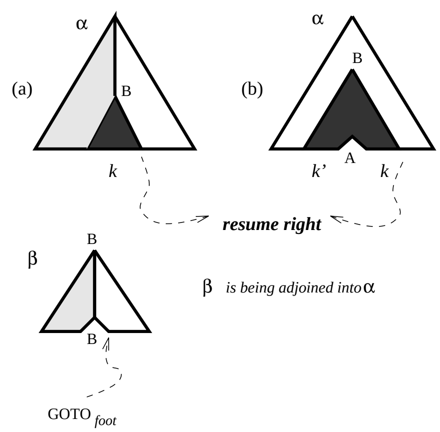

Figure 3: The **resume right** action in the parser.

above and below positions of the dot. If an adjunction has been performed on a node then it is marked with a star (e.g. $B*$).

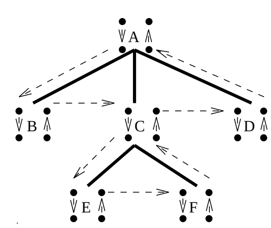

Figure 4: Left to right dotted tree traversal.

Construction of a LR(0) parsing table is an extension of the technique used for CFGs. The parse table is built as a finite state automaton (FSA) with each state defined to be a set of dotted trees. The closure operations on states in the parse table are defined in Figure 5. All the states in the parse table must be closed under these operations. Figure 9 is a partial FSA constructed for the grammar in Figure 7.

The FSA is built as follows: in state 0 put all the initial trees with the dot left and above the root. The state is then closed. New states are built by the transitions defined in Figure 6. Entries in the parse table are determined

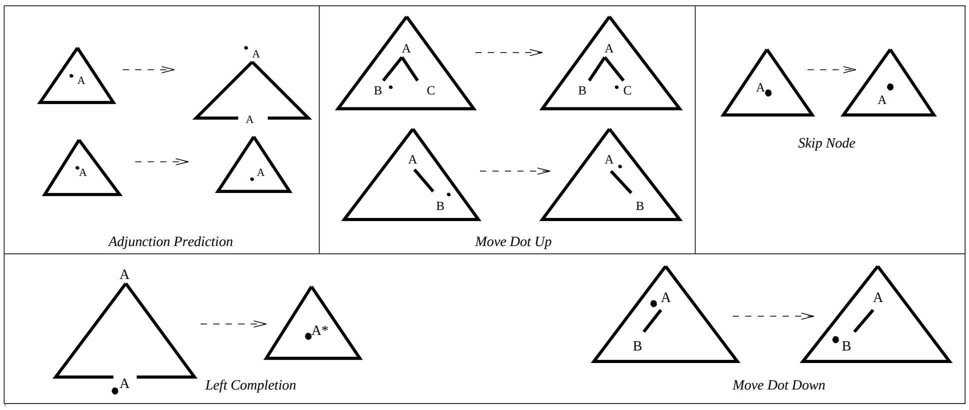

Figure 5: Closure Operations.

as follows:

- a **shift** for each transition in the FSA.
- **resume right** iff there is a node $B*$ with the dot right and below it.
- **reduce root** iff there is a rootnode in an auxiliary tree with the dot right and above it.
- **accept** and **error** with the usual interpretation.

The items created in each state before closure applies, i.e. the right hand sides in Figure 6 are called the **kernels** of each state in the FSA. The initial trees with the dot left and above the root form the kernel for state 0. A state which has not been closed is said to be in kernel form.

3 Lazy Parser Generation

The algorithm described so far assumes that the parse table is precompiled before the parser is used. Lazy parser generation spreads the generation of the parse table over the parsing of several sentences to obtain a faster response time in the parser generation stage. It generates only those parts of the parser that are needed to parse the sentences given to it. Lazy parser

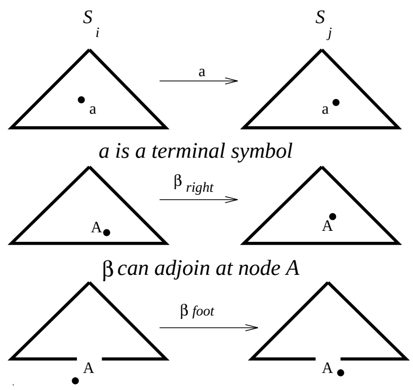

Figure 6: Transitions in the finite state automaton.

generation is useful in cases where typical input sentences are parsed with a small part of the total grammar.

We define lazy parser generation mainly as a step towards incremental parser generation. The approach is an extension of the algorithm for CFGs given in (Heering et al., 1990; Heering et al., 1989). To modify the LR parsing strategy given earlier we move the closure and computation of transitions (Figure 5 and Figure 6) from the table generation stage to the LR parser. The lazy technique expands a kernel state only when the parser, looking at the current input, indicates that the state needs expansion. For example, the TAG in Figure 7 (*na* rules out adjunction) produces the FSA in Figure 8[4]. Computation of closure and transitions in the state occurs while parsing as seen in Figure 9 which is the result of the LR parser expanding the FSA in Figure 8 while parsing the string *aec*.

The only extra statement in the modified parse function is a check on the type of the state and possible expansion of kernel states takes place while parsing a sentence. Memory use in the lazy technique is greater as the FSA is needed during parsing as well.

[4]As a convention in our FSAs we mark unexpanded kernel states with a boldfaced outline and a double-lined outline as the acceptance states.

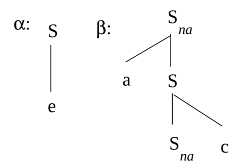

Figure 7: TAG G where $L(G) = \{a^n e c^n\}$

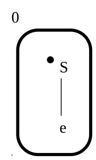

Figure 8: The FSA after parse table generation.

4 Incremental Parser Generation

The lazy parser generator described reacts to modifications to the grammar by throwing away all parts of the parser that it has generated and creates a FSA containing only the start state. In this section we describe an incremental parser generator which retains as much of the original FSA as it can. It throws away only that information from the FSA of the old grammar which is incorrect with respect to the updated grammar.

The incremental behaviour is obtained by selecting the states in the parse table affected by the change in the grammar and returning them to their kernel form (i.e. remove items added by the closure operations). The parse table FSA will now become a disconnected graph. The lazy parser will expand the states using the new grammar. All states in the disconnected graph are kept as the lazy parser will reconnect with those states (when the transitions in Figure 6 are computed) that are unaffected by the change in the grammar. Consider the addition of a tree to the grammar[5].

- for an initial tree α return state 0 to kernel form adding α with the dot left and above the root node. Also return all states where a possible *Left Completion* on α can occur to their kernel form.

[5]Deletion of a tree will be similar.

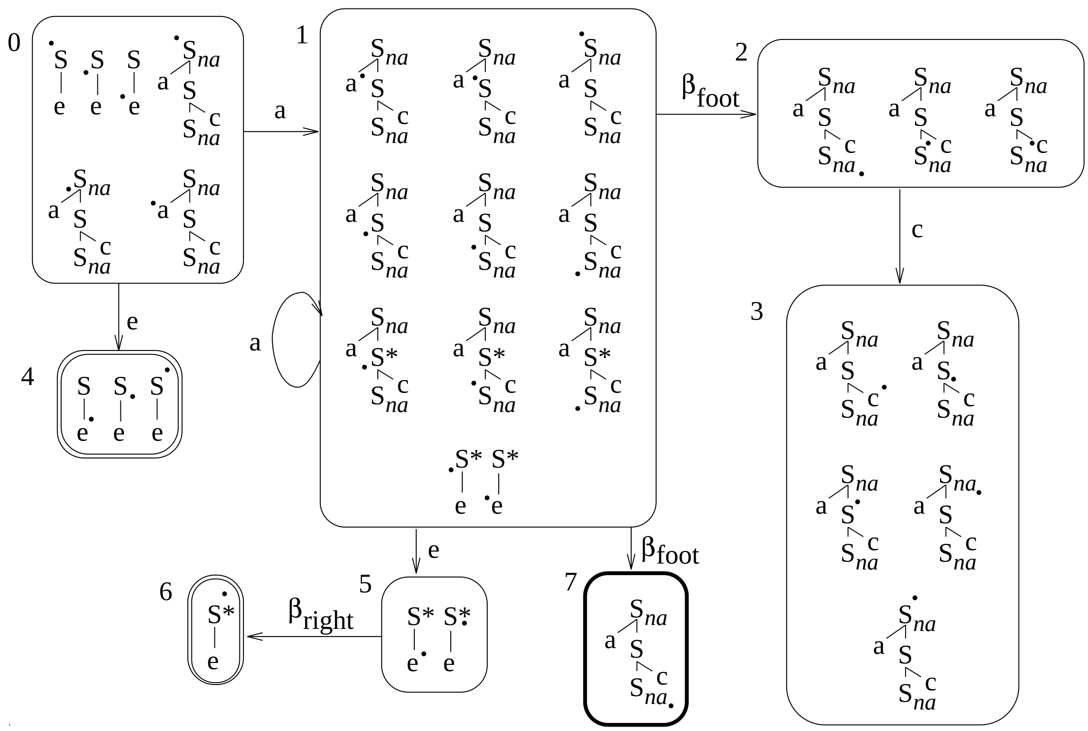

Figure 9: The FSA after parsing the string *aec*.

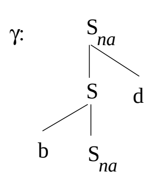

Figure 10: New tree added to G with $L(G) = \{a^n b^m e c^n d^m\}$

- for an auxiliary tree β return all states where a possible *Adjunction Prediction* on β can occur and all states with a β_{right} transition to their kernel form.

For example, the addition of the tree in Figure 10 causes the FSA to fragment into the disconnected graph in Figure 11. It is crucial that the disconnected states are kept around as can be seen from the re-expansion of a single state in Figure 12. All states compatible with the modified grammar are eventually reused.

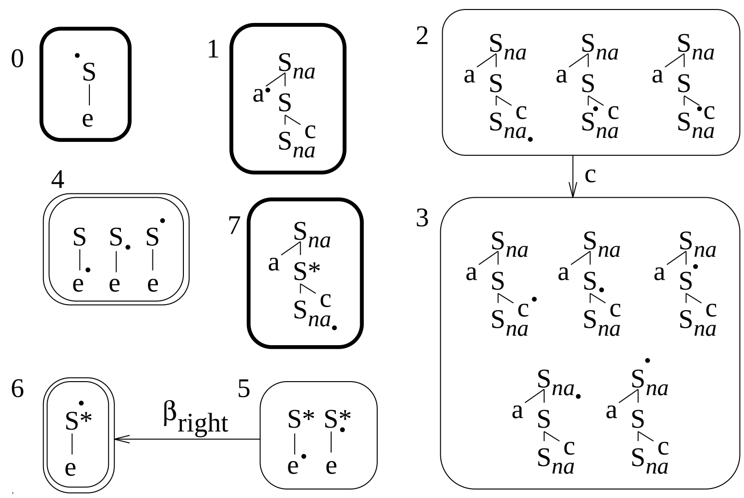

Figure 11: The parse table after the addition of γ.

The approach presented above causes certain states to become unreachable from the start state. Frequent modifications of a grammar can cause many unreachable states. A *garbage collection* scheme defined in (Heering et al., 1990) can be used here which avoids overregeneration by retaining unreachable states.

5 Conclusion

What we have described above is work in progress in implementing a LR-type parser for a wide-coverage lexicalized grammar of English in the TAG framework (XTAG Group, 1995). The algorithm for incremental parse ta-

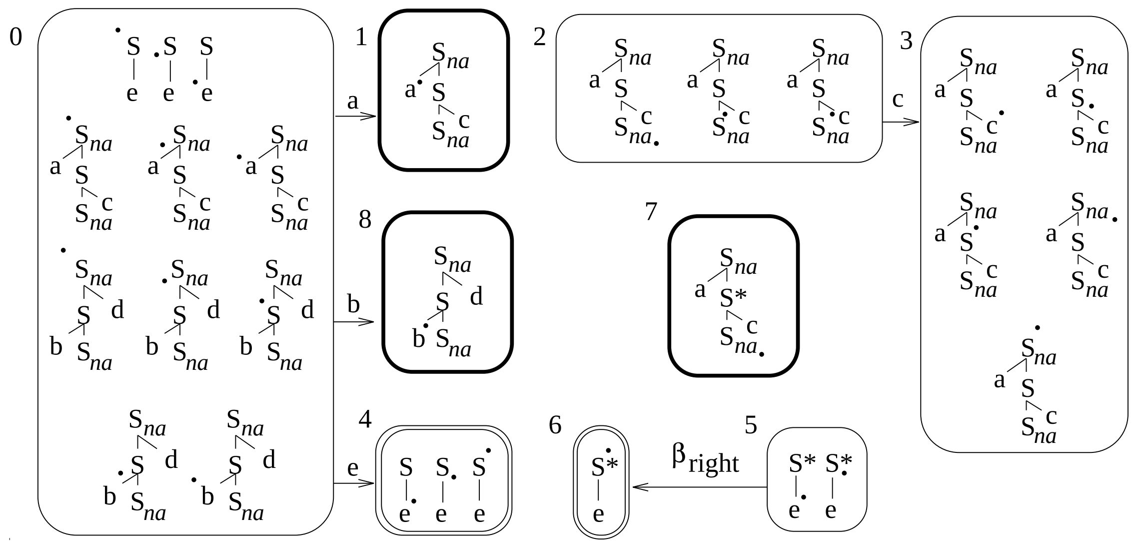

Figure 12: The parse table after expansion of state 0 with the modified grammar.

ble generation for TAGs given here extends a similar result for CFGs. The parse table generator was built on a lazy parser generator which generates the parser only when the input string uses parts of the parse table not previously generated. The technique for incremental parser generation allows the addition and deletion of elementary trees from a TAG without recompilation of the parse table for the updated grammar. This allows us to combine the speed-up obtained by precompiling top-down dependencies such as the prediction of adjunction with the flexibility in lexical description usually given by Earley-style parsers.

References

Aho, Alfred V., Ravi Sethi and Jeffrey D. Ullman, *Compilers: Principles, Techniques and Tools*, Addison Wesley, Reading, MA, 1986.

Heering, Jan, Paul Klint and Jan Rekers, Incremental Generation of Parsers, In *IEEE Transactions on Software Engineering*, vol. 16, no. 12, pp. 1344-1350, 1990.

Heering, Jan, Paul Klint and Jan Rekers, Incremental Generation of Parsers, In *ACM SIGPLAN Notices (SIGPLAN '89 Conference on Programming Language Design and Implementation)*, vol. 24, no. 7, pp. 179–191, 1989.

Joshi, Aravind K., An Introduction to Tree Adjoining Grammars, In Manaster-Ramer, Alexis (ed.) *Mathematics of Language*, John Benjamins, Amsterdam, 1987.

Schabes, Yves and K. Vijay-Shanker, Deterministic Left to Right Parsing of Tree Adjoining Languages, In *28th Meeting of the Association for Computational Linguistics (ACL '90)*, Pittsburgh, PA, 1990.

Schabes, Yves and Aravind K. Joshi, Parsing with Lexicalized Tree Adjoining Grammars, In Tomita, Masaru (ed.) *Current Issues in Parsing Technologies*, Kluwer Academic, Dordrecht, The Netherlands, 1991.

Tomita, Masaru, *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*, Kluwer Academic, Dordrecht, The Netherlands, 1986.

XTAG Research Group, *A Lexicalized Tree Adjoining Grammar for English*, IRCS Technical Report 95-03, University of Pennsylvania, Philadelphia, PA. 1995.