Incremental Parser Generation for Tree Adjoining Grammars*

Anoop Sarkar
University of Pennsylvania
Department of Computer and Information Science
200 S. 33rd St., Philadelphia PA 19104-6389, USA
anoop@linc.cis.upenn.edu

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 LR Parser Generation

Tree Adjoining Grammars (TAGs) are tree rewriting systems which combine trees with the single operation of *adjoining*. (Schabes and Vijay-Shanker, 1990) describes the construction of an LR parsing algorithm for TAGs[1]. 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 explained by an automaton which is weakly equivalent to TAGs called Bottom-Up Embedded Pushdown Automata (BEPDA) (Schabes and Vijay-Shanker, 1990)[3]. Storage in a BEPDA is a sequence of stacks, where new stacks can be introduced above and below the top stack in the automaton. Recognition of adjunction is equivalent to the **unwrap** move shown in Fig. 1.

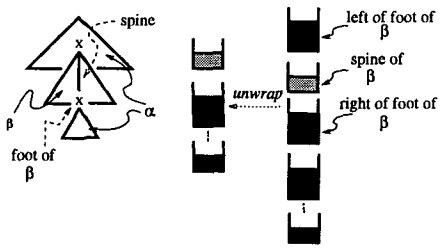

Figure 1: Recognition of adjunction in a BEPDA.

The LR parser (of (Schabes and Vijay-Shanker, 1990)) uses a parsing table and a sequence of stacks (Fig. 1) to parse the input. The parsing table encodes the actions taken by the parser as follows (using two *GOTO* functions):

- **Shift** to a new state, pushed onto a new stack which appears on top of the current sequence of stacks. The current input token is removed.
- **Resume Right** when the parser has reached right and below a node (in a dotted tree, explained below) on which an auxiliary tree has been adjoined. The $GOTO_{foot}$ function encodes the proper state such that the string to the right of the footnode can be recognized.
- **Reduce Root**, the parser executes an unwrap move to recognize adjunction (Fig. 1). The proper state for the parser after adjunction is given by the $GOTO_{right}$ function.
- **Accept** and **Error** functions as in conventional LR parsing.

There are four positions for a dot associated with a symbol in a dotted tree: left above, left below, right below and right above. A dotted tree has one such dotted symbol. The tree traversal in Fig. 2 scans the frontier of the tree from left to right while trying to recognize possible adjunctions between the

*This work is partially supported by NSF grant NSF-STC SBR 8920230 ARPA grant N00014-94 and ARO grant DAAH04-94-G0426. Thanks to Breck Baldwin, Dania Egedi, Jason Eisner, B. Srinivas and the three anonymous reviewers for their valuable comments.

[1]Familiarity with TAGs and their parsing techniques is assumed throughout the paper, see (Schabes and Joshi, 1991) for an introduction. We assume that our definition of TAG does not have the *substitution* operation. See (Aho et al., 1986) for details on LR parsing.

[2]The algorithm described here can be extended to use 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.

above and below positions of the dot. Adjunction on a node is recorded by marking it with an asterisk[4].

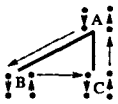

Figure 2: Left to right dotted tree traversal.

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 Fig. 3. All the states in the parse table must be closed under these operations[5].

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 three transitions: $s_i\{^\bullet a\} \xrightarrow{a} s_j\{a^\bullet\}$, a is a terminal symbol; $s_i\{A_\bullet\} \xrightarrow{\beta_{right}} s_j\{A^\bullet\}$, β can adjoin at node A; $s_i\{_\bullet A\} \xrightarrow{\beta_{foot}} s_j\{A_\bullet\}$, A is a footnode. Entries in the parse table are determined 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 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.

2 Lazy Parser Generation

The algorithm described so far assumes that the parse table is precompiled before the parser is used. Lazy parser generation generates only those parts of the parser that become necessary during actual parsing. 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 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 so. For example, a TAG and corresponding FSA is shown in Fig. 4 (na rules out adjunction at a node)[6]. Computation of closure and transitions in the state occurs while parsing as in Fig. 5 which is the result of the LR parser expanding the FSA in Fig. 4 while parsing the string aec.

The modified parse function checks the type of the state and may expand the kernel states while parsing a sentence. Memory use in the lazy technique is greater as the FSA is needed during parsing *and* parser generation.

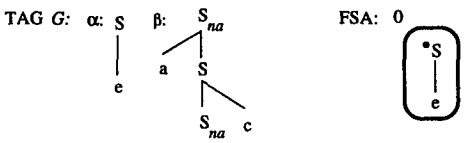

Figure 4: TAG G where $L(G) = \{a^n e c^n\}$ and corresponding FSA after lazy parse table generation.

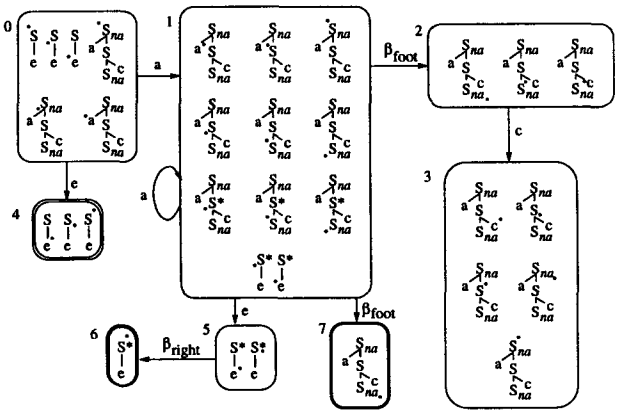

Figure 5: The FSA after parsing the string aec.

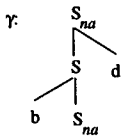

Figure 6: New tree added to G with $L(G) = \{a^n b^m e c^n d^m\}$

3 Incremental Parser Generation

An incremental parser generator responds to grammar updates by throwing away only that information from the FSA of the old grammar that is inconsistent in the updated grammar. 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 between states are computed) that are unaffected by the change in the grammar. Consider

[4]For example, $B*$. This differs from the usual notation for marking a footnode with an asterisk.

[5]Fig. 5 is a partial FSA for the grammar in Fig. 4.

[6]Unexpanded kernel states are marked with a boldfaced outline, acceptance states with double-lines.

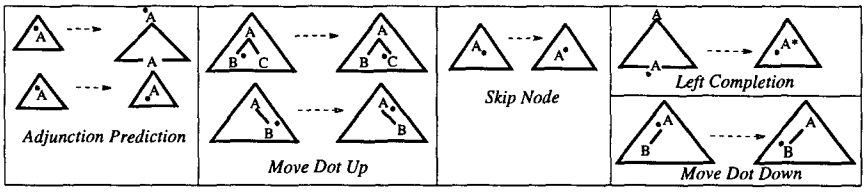

Figure 3: Closure Operations.

the addition of a tree to the grammar (deletion will be similar).

- 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.
- 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 Fig. 6 causes the FSA to fragment into the disconnected graph in Fig. 7. It is crucial to keep the disconnected states around; consider the re-expansion of a single state in Fig. 8. All states compatible with the modified grammar are eventually reused.

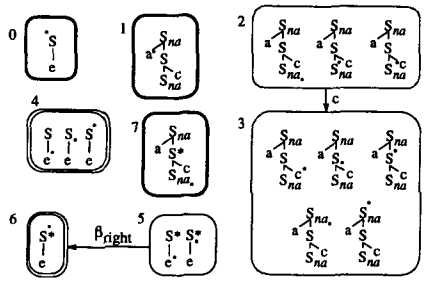

Figure 7: The parse table after the addition of γ.

The approach presented above causes certain states to become unreachable from the start state[7]. 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.

4 Conclusion

What we have described above is work in progress in implementing an LR-type parser for a wide-coverage lexicalized grammar of English using TAGs (XTAG Group, 1995). Incremental parser generation allows the addition and deletion of elementary trees from a

[7]Quantitative results on the performance of the algorithm presented are forthcoming.

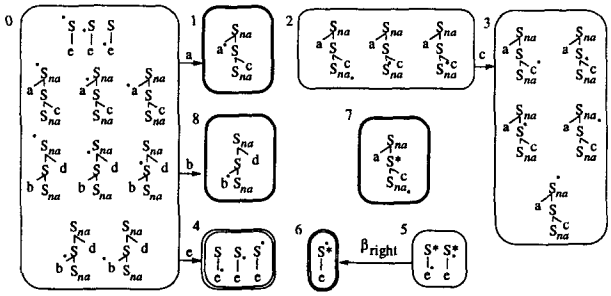

Figure 8: The parse table after expansion of state 0 with the modified grammar.

TAG without recompilation of the parse table for the updated grammar. This allows precompilation of top-down dependencies such as the prediction of adjunction while having the flexibility 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.

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.