Incremental Parser Generation for Tree Adjoining Grammars

Anoop Sarkar
<anoop@linc.cis.upenn.edu>

Dept. of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104

ACL 96, Student Session

The Promise of LR Parsing

- LR-type parsers are generally considered faster than Earley-type or CYK-type parsers.
- This is because they precompile information about the grammar into a parse table used while parsing the input.

- In practice, LR-type parsing faces problems:
  - Bloated size of the parse table for large grammars (e.g. wide coverage grammars).
  - Many parts of the grammar account for infrequent data, but are explored for each parse.
  - Modifications to the grammar involve recompiling the entire parse table.
  - If the grammar formalism is lexicalized, LR-type parsing does not exploit lexicalization, unlike Earley-type parsing.

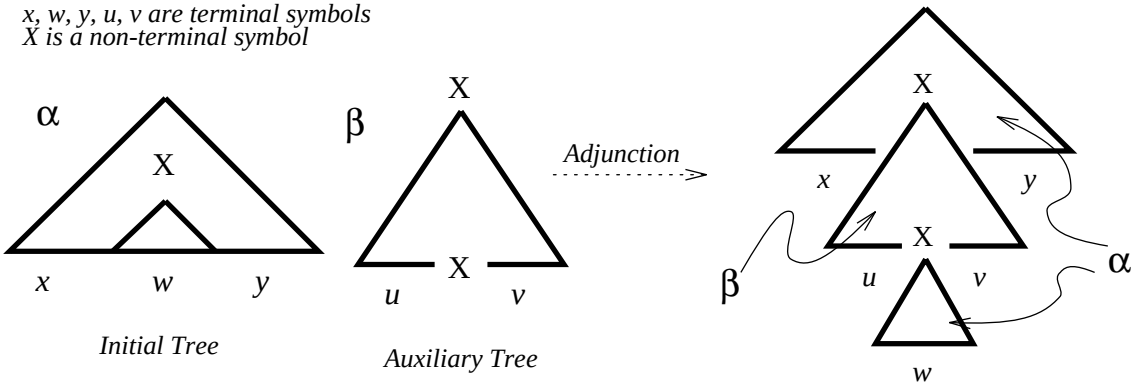
Overview

- These problems are exemplified in the LR-type parsing of lexicalized Tree Adjoining Grammars (TAGs).
- This paper offers a solution to these problems for the LR-type parsing of TAGs.

- The algorithm described here describes a lazy and incremental parse table generator in a LR-type parser for TAGs.
- It extends the work done on incremental modification of LR(0) parser generators for CFGs (Heering, et al. 1990).

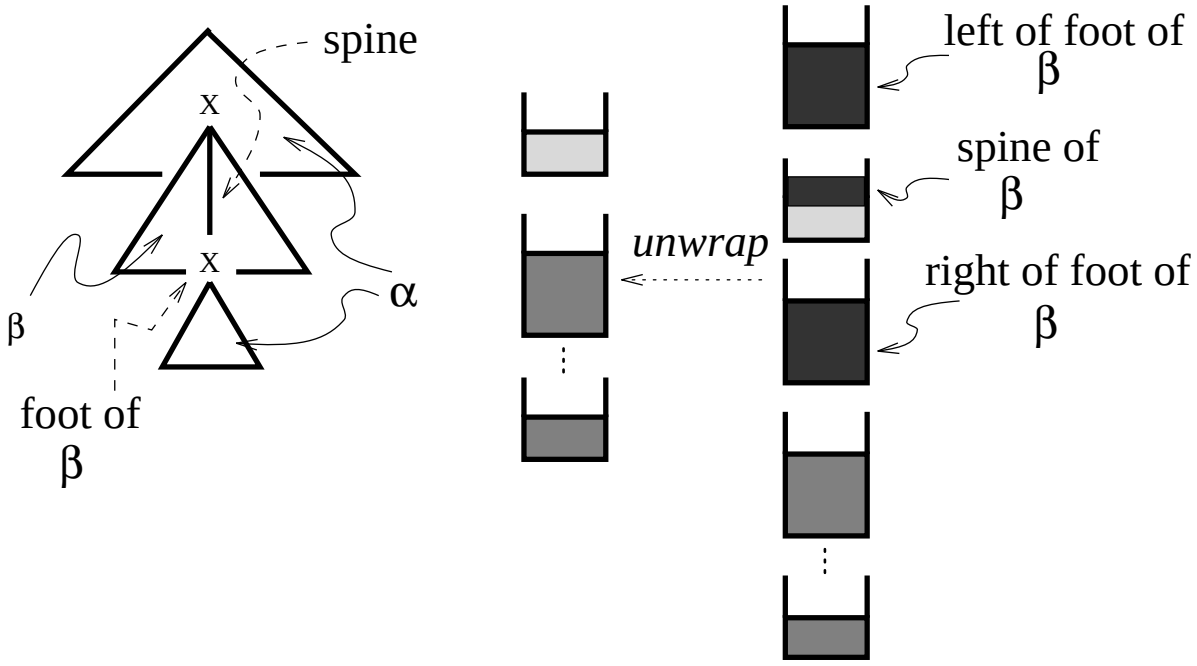
LR Parsing of TAGs

- LR parsing of TAGs (Schabes and Vijayshanker, 1990) is an extension of the conventional parsing algorithm for CFGs.
- However, the notion of shift/reduce cannot be applied to a TAG since TAGs compose via the *adjunction* operation:



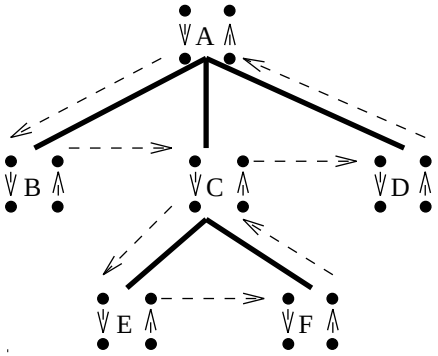
LR Parsing of TAGs

- ▷ While LR parsing of CFGs uses a parse table and a single stack, LR parsing of TAGs requires a parse table and a sequence of stacks (below).
- ▷ Instead of the conventional reduce move, the LR parser for TAGs makes the *unwrap* move on the sequence of stacks.



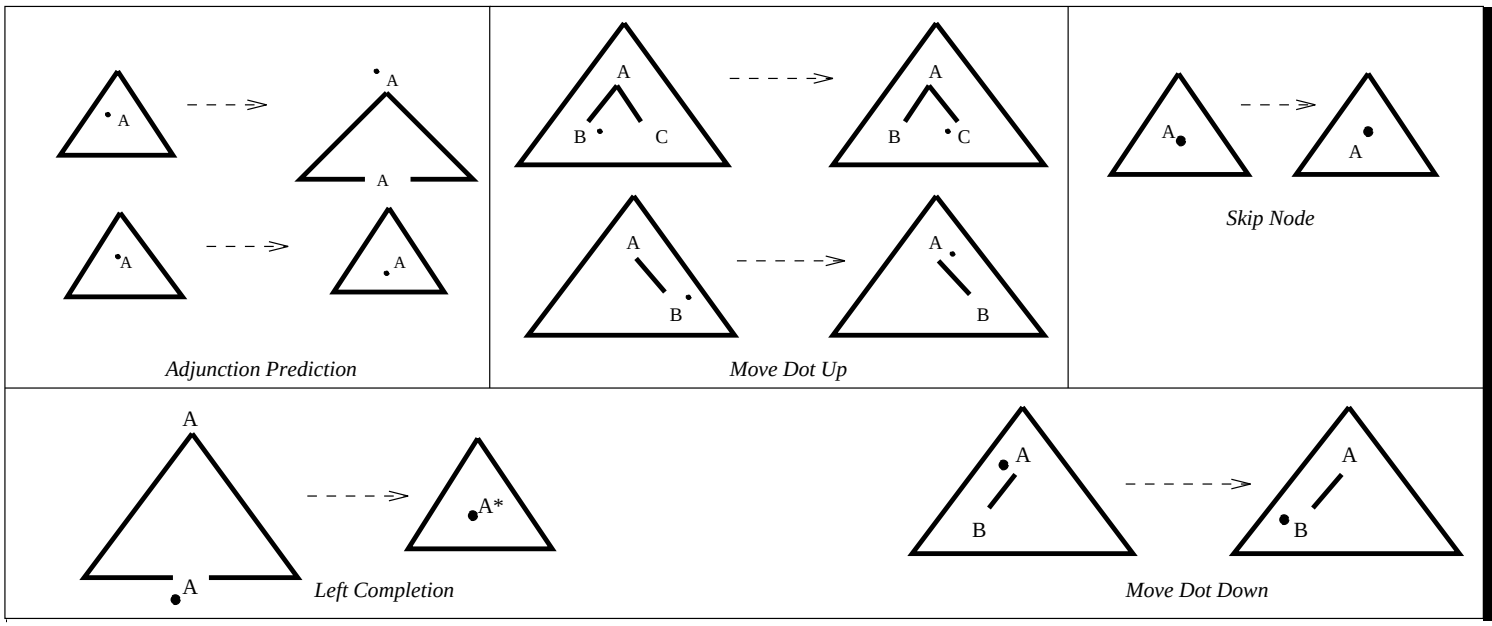
Dotted Tree Traversal

- The notion of dotted rules for CFGs is extended to trees.
- Four positions are available to a dot at each node: left above, left below, right below and right above.
- Each dotted tree has one such dot.
- The dotted tree traversal (below) scans for adjunctions between the above and below positions of each dot.
- Adjunction performed at a node is indicated with a star, e.g B*



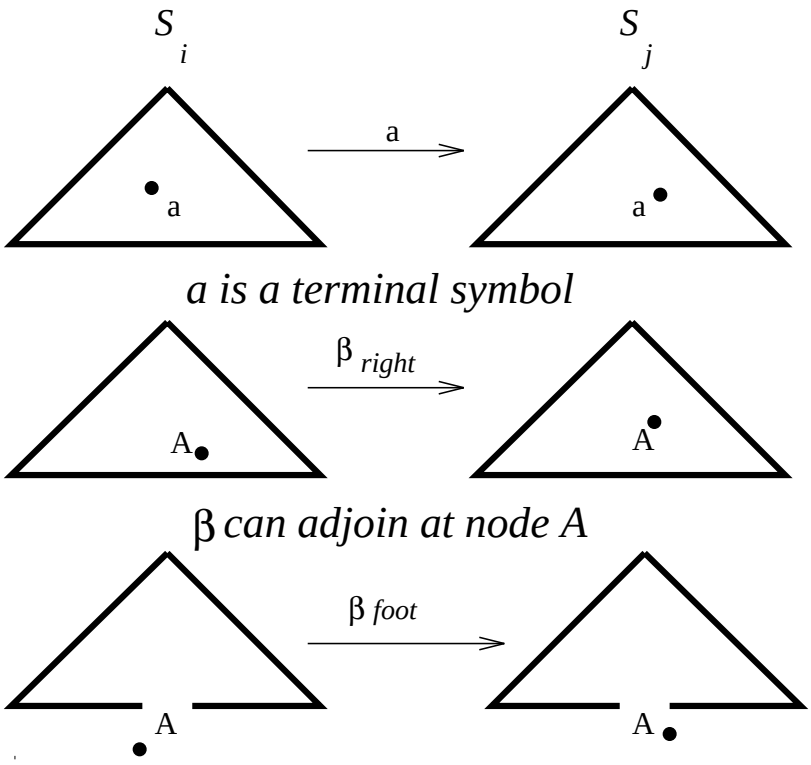
Construction of the Parse Table

- ▷ The parse table is built as a finite state automaton (FSA).
- ▷ The FSA is built by putting in the start state all initial trees with the dot left and above the root.
- ▷ The state is then closed under the following closure operations:



Transitions in the Parse Table

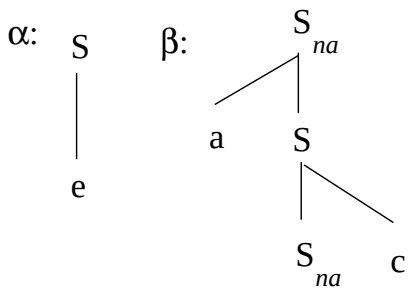
▷ New states in the parse table are built and the following transitions are added to the table:



Lazy Parser Generation

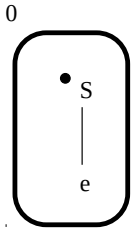
- In conventional LR parsing, the parse table is precompiled before the parser is used.
- The lazy technique spreads the generation of the parse table over the parsing of several sentences.

- For example, if we have a TAG G where $L(G) = \{a^n e c^n\}$:



Lazy Parser Generation

- The FSA after the table generation phase:

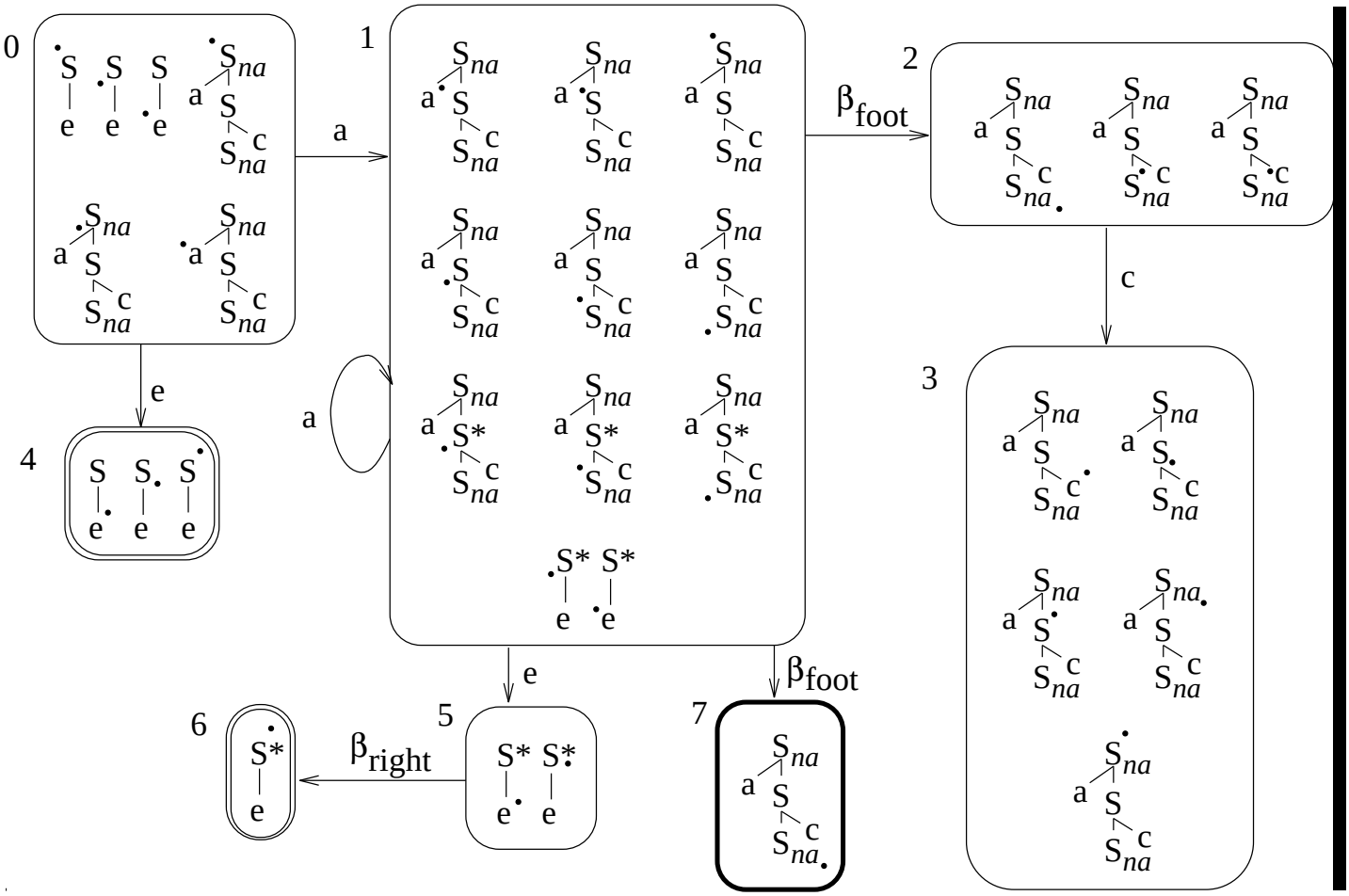


- The boldfaced outline indicates that the state is *unexpanded* or not closed.

- The FSA is needed while parsing as well, unlike conventional LR parsing.
- Computations of closure and transitions occur while parsing.

Lazy Parser Generation

▷ The FSA after parsing the string *aec*

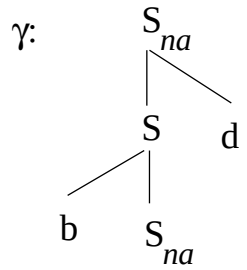


▷ Double lines indicate that the state is an acceptance state.

Incremental Parser Generation

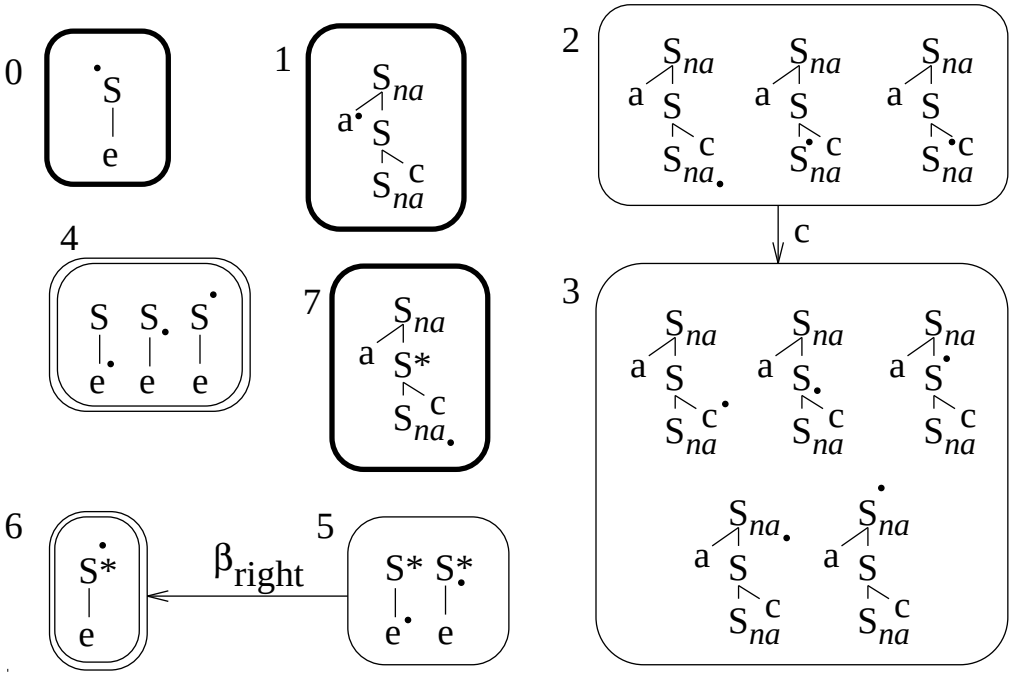
- ▷ Modifications to the grammar in conventional LR parsing results in recompiling the entire parse table.
- ▷ Lazy parser generators also throw away all of the old parse table, generating the new parse table by need.
- ▷ Incremental behaviour is obtained by selecting states affected by the change in the grammar and removing items added by closure operations (further detail in the paper).
- ▷ The lazy parser will now expand the states using the new grammar.

- ▷ Consider addition of a new tree γ added to G with $L(G) = \{a^n b^m e c^n d^m\}$:



Incremental Parser Generation

▷ The parse table after the addition of γ.

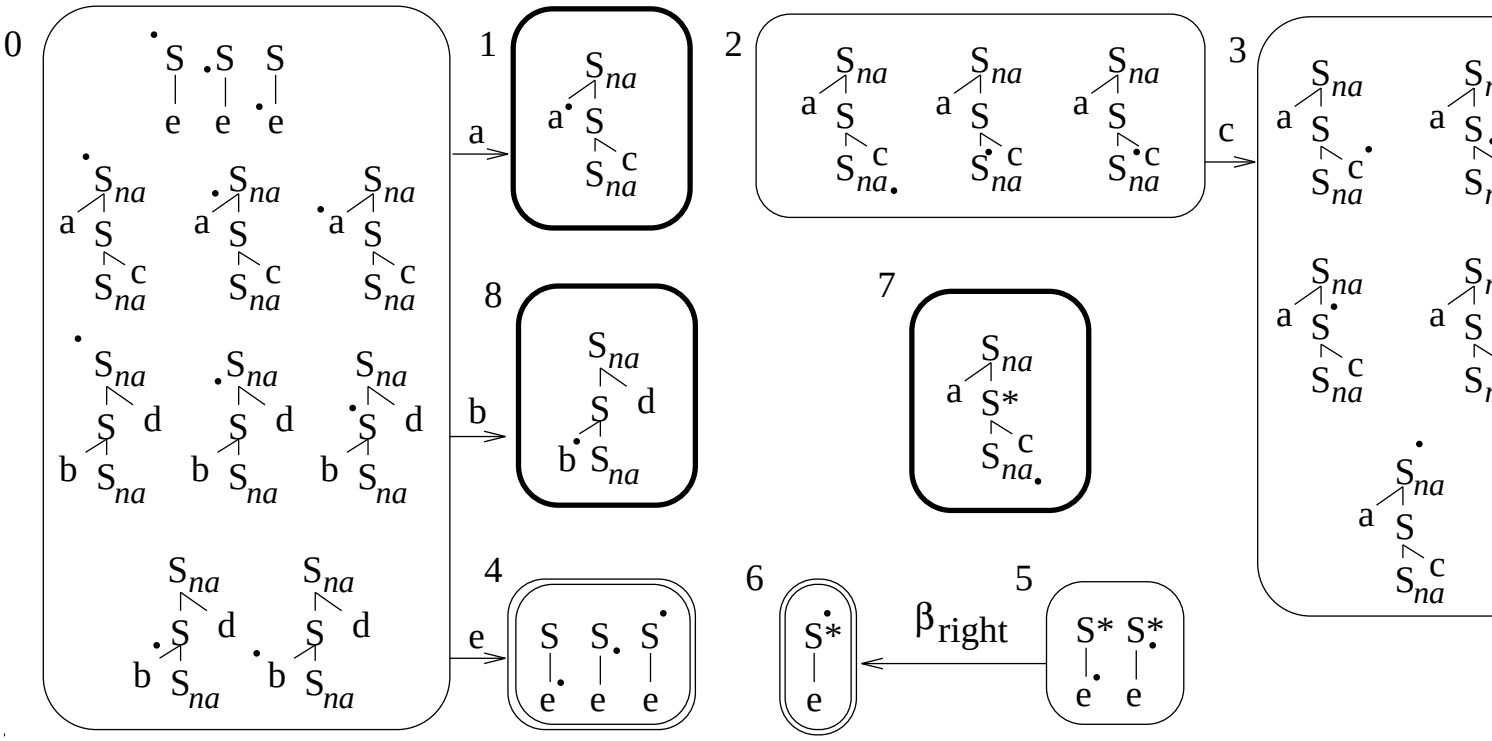


▷ Since γ was an initial tree it affects the start state (state 0) removing all applications of the closure operations.

▷ The FSA fragments into a disconnected graph.

Incremental Parser Generation

- ▷ The disconnected states are kept around by the parser.
- ▷ This is crucial, as can be seen by the re-expansion of a single state (state 0 with the modified grammar):



- ▷ All states compatible with the new grammar are eventually reused.

Conclusion

- The algorithm for incremental parse table generation given here extends a similar result for CFGs.

- The parse table generator was built on a lazy parser generator which generates parts of the table 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 of the updated grammar.

- This approach presented causes certain states to become unreachable from the start state over time. A *garbage collection* scheme is used here.