DETERMINISTIC LEFT TO RIGHT PARSING OF TREE ADJOINING LANGUAGES*

Yves Schabes
Dept. of Computer & Information Science
University of Pennsylvania
Philadelphia, PA 19104-6389, USA
schabes@linc.cis.upenn.edu

K. Vijay-Shanker
Dept. of Computer & Information Science
University of Delaware
Newark, DE 19716, USA
vijay@udel.edu

Abstract

We define a set of deterministic bottom-up left to right parsers which analyze a subset of Tree Adjoining Languages. The LR parsing strategy for Context Free Grammars is extended to Tree Adjoining Grammars (TAGs). We use a machine, called Bottom-up Embedded Push Down Automaton (BEPDA), that recognizes in a bottom-up fashion the set of Tree Adjoining Languages (and exactly this set). Each parser consists of a finite state control that drives the moves of a Bottom-up Embedded Pushdown Automaton. The parsers handle deterministically some context-sensitive Tree Adjoining Languages.

In this paper, we informally describe the BEPDA then given a parsing table, we explain the LR parsing algorithm. We then show how to construct an LR(0) parsing table (no lookahead). An example of a context-sensitive language recognized deterministically is given. Then, we explain informally the construction of SLR(1) parsing tables for BEPDA. We conclude with a discussion of our parsing method and current work.

1 Introduction

LR(k) parsers for Context Free Grammars (Knuth, 1965) consist of a finite state control (constructed given a CFG) that drives deterministically with k lookahead symbols a push down stack, while scanning the input from left to right. It has been shown that they recognize exactly the set of languages recognized by deterministic push down automata. LR(k) parsers for CFGs have been proven useful for compilers as well as recently for natural language processing. For natural language processing, although LR(k) parsers are not powerful enough, conflicts between multiple choices are solved by pseudo-parallelism (Lang, 1974, Tomita, 1987). This gives rise to a class of powerful yet efficient parsers for natural languages. It is in this context that we study deterministic (LR(k)-style) parsing of TAGs.

The set of Tree Adjoining Languages is a strict superset of the set of Context Free Languages (CFLs). For example, the cross serial dependency construction in Dutch can be generated by a TAG. [1] Walters (1970), Révész (1971), Turnbull and Lee (1979) investigated deterministic parsing of the class of context-sensitive languages. However they used Turing machines which recognize languages much more powerful than Tree Adjoining Languages. So far no deterministic bottom-up parser has been proposed for any member of the class of the so-called "mildly context sensitive" formalisms (Joshi, 1985) in which Tree Adjoining Grammars fall.[2] Since the set of Tree Adjoining Languages (TALs) is a strict superset of the set of Context Free Languages, in order to define LR-type parsers for TAGs, we need to use a more powerful configuration then a finite state automaton driving a push down stack. We investigate the design of deterministic left to right bottom up parsers for TAGs in which a finite state control drives the moves of a Bottom-up Embedded Push Down Stack. The class of corresponding non-deterministic automata recognizes exactly the set of TALs.

We focus our attention on showing how a bottom-up embedded pushdown automaton is deterministically driven given a parsing table. To illustrate the building of a parsing table, we consider the simplest case, i.e. building of LR(0) items and the corresponding LR(0)

*The first author is partially supported by Darpa grant N0014-85-K0018, ARO grant DAAL03-89-C-0031PRI NSF grant-IRI84-10413 A02. We are extremely grateful to Bernard Lang and David Weir for their valuable suggestions.

[1] The parsers that we develop in this paper can parse these constructions deterministically (see Figure 5).

[2] Tree Adjoining Grammars, Modified Head Grammars, Linear Indexed Grammars and Categorial Grammars (all of which generate the same subclass of context-sensitive languages) fall in the class of the so-called "mildly context sensitive" formalisms. The Embedded Push Down Automaton recognizes exactly this set of languages (Vijay-Shanker 1987).

parsing table for a given TAG. An example for a TAG generating a context-sensitive language is given in Figure 5. Finally, we consider the construction of SLR(1) parsing tables.

We assume that the reader is familiar with TAGs. We refer the reader to Joshi (1987) for an introduction to TAGs. We will assume that the trees can be combined by adjunction only.

2 Automata Models of Tags

Before we discuss the Bottom-up Embedded Pushdown Automaton (BEPDA) which we use in our parser, we will introduce the Embedded Pushdown Automaton (EPDA). An EPDA is similar to a pushdown automaton (PDA) except that the storage of an EPDA is a sequence of pushdown stores. A move of an EPDA (see Figure 1) allows for the introduction of bounded pushdowns above and below the current top pushdown. Informally, this move can be thought of as corresponding to the adjoining operation move in TAGs with the pushdowns introduced above and below the current pushdown reflecting the tree structure to the left and right of the foot node of an auxiliary being adjoined. The spine (path from root to foot node) is left on the previous stack.

The generalization of a PDA to an EPDA whose storage is a sequence of pushdowns captures the generalization of the nature of the derived trees of a CFG to the nature of derived trees of a TAG. From Thatcher (1971), we can observe that the path set of a CFG (i.e. the set of all paths from root to leaves in trees derived by a CFG) is a regular set. On the other hand, the path set of a TAG is a CFL. This follows from the nature of the adjoining operation of TAGs, which suggests stacking along the path from root to a leaf. For example, as we traverse down a path in a tree γ (in Figure 1), if adjunction, say by β, occurs then the spine of β has to be traversed before we can resume the path in γ.

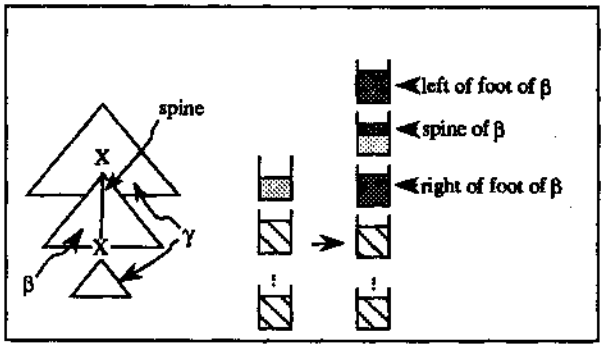


Figure 1: Embedded Pushdown Automaton

3 Bottom-up Embedded Pushdown Automaton[3]

For any TAG G, an EPDA can be designed such that its moves correspond to a top-down parse of a string generated by G (EPDA characterizes exactly the set of Tree Adjoining Languages, Vijay- Shanker, 1987). If we wish to design a bottom-up parser, say by adopting a shift reduce parsing strategy, we have to consider the nature of a reduce move of such a parser (i.e. using EPDA storage). This reduce move, for example applied after completely considering an auxiliary tree, must be allowed to 'remove' some bounded pushdowns above and below some (not necessarily bounded) pushdown. Thus (see Figure 2), the reduce move is like the dual of the wrapping move performed by an EPDA.

Therefore, we introduce Bottom-up Embedded Pushdown Automaton (BEPDA), whose moves are dual of an EPDA. The two moves of a BEPDA are the unwrap move depicted in Figure 2 – which is an inverse of the wrap move of an EPDA – and the introduction of new pushdowns on top of the previous pushdown (push move). In an EPDA, when the top pushdown is emptied, the next pushdown automatically becomes the new top pushdown. The inverse of this step is to allow for the introduction of new pushdowns above the previous top pushdown. These are the two moves allowed in a BEPDA, the various steps in our parsers are sequences of one or more such moves.

Due to space constraints, we do not show the equivalence between BEPDA and EPDA apart from noting that the moves of the two machines are dual of each other.

4 LR Parsing Algorithm

An LR parser consists of an input, an output, a sequence of stacks, a driver program, and a parsing table that has three parts (ACTION, GOTO_{right} and GOTO_{foot}). The parsing program is the same for all LR parsers, only the parsing tables change from one grammar to another. The parsing program reads characters from the input one character at a time. The program uses the sequence of stacks to store states.

The parsing table consists of three parts, a parsing action function ACTION and two goto functions GOTO_{right} and GOTO_{foot}. The program driving the LR parser first determines the state i currently on top of the top stack and the current input token a_r. Then it consults the ACTION table entry for state i and token

[3] The need to use bottom-up version of an EPDA in LR style parsing of TAGs was suggested to us by Bernard Lang and David Weir. Also their suggestions played an instrumental role in the definition of BEPDA, for example restriction on the moves allowed.

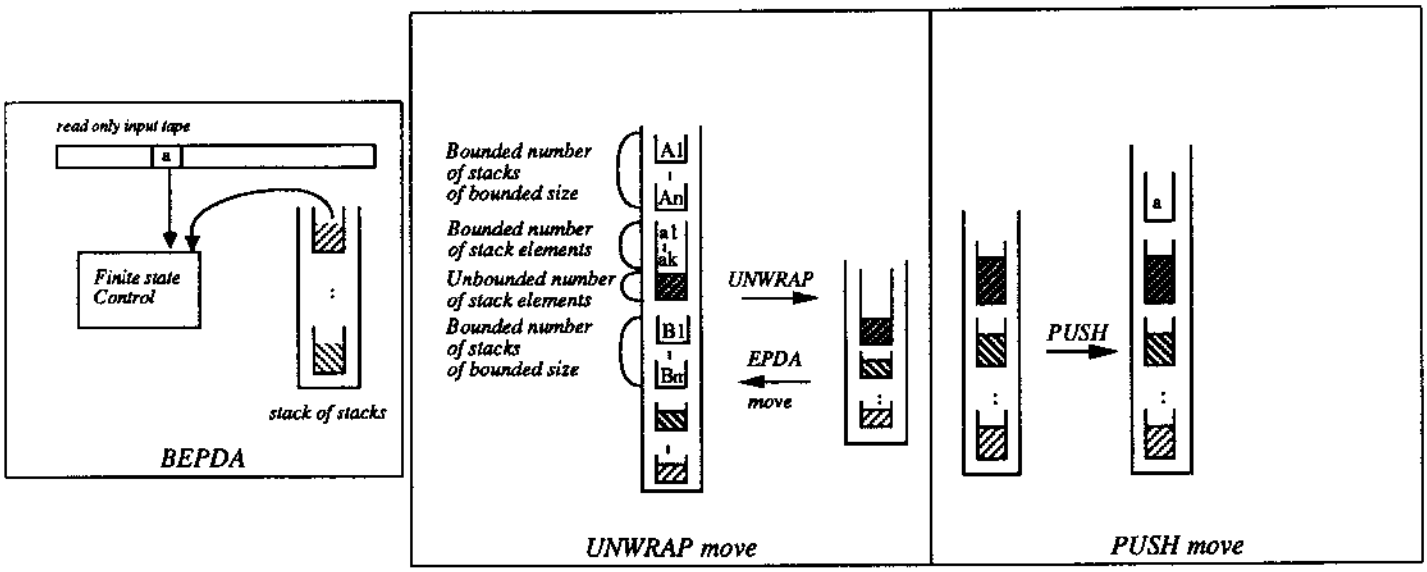


Figure 2: Bottom-up Embedded Pushdown Automaton

a_r. The entry in the action table can have one of the following five values:

- **Shift** j (sj), where j is a state;
- **Resume Right** of δ at address dot (rsδ@dot)), where δ is an elementary tree and dot is the address of a node in δ;
- **Reduce Root** of the auxiliary tree β in which the last adjunction on the spine was performed at address $star$ (rdβ@$star$);
- **Accept** (acc);
- **Error**, no action applies, the parsers rejects the input string (errors are associated with empty table entries).

The function GOTO_{right} and GOTO_{foot} take a state i and an auxiliary tree β and produce a state j.

An example of a parsing table for a grammar generating $L = \{a^n b^n e c^n d^n | n \geq 0\}$ is given in Figure 5.

We denote an *instantaneous description* of the BEPDA by a pair whose first component is the sequence of pushdowns and whose second component is the unexpanded input:

$$(\|t_m \cdots t_1\| \cdots \|s_1 \cdots s_w,\ a_r a_{r+1} \cdots a_n\$)$$

In the above sequence of pushdowns, the stacks are piled up from left to right. $\|$ stands for the bottom of a stack. s_w is the top element of the top stack, s_1 is the bottom element of the top stack, t_1 is the top element of the bottom stack and t_m is the bottom element of the bottom stack.

The initial configuration of the parser is set to:

$$(\|0,\ a_1 \cdots a_n\$)$$

where 0 is the start state and $a_1 \cdots a_n\$$ is the input string to be read with an end marker (\$).

Suppose the parser reaches the configuration:

$$(\|t_m \cdots t_1\| \cdots \|i_w \cdots i_1 i,\ a_r a_{r+1} \cdots a_n\$)$$

The next move of the parser is determined by reading a_r, the current input token and the state i on top of the sequence of stacks, and then consulting the parsing table entry for ACTION$[i, a_r]$. The parser keeps applying the move associated with ACTION$[i, a_r]$ until acceptance or error occurs. The following moves are possible:

(i) ACTION$[i, a_r]$ = **shift state** j (sj). The parser executes a push move, entering the configuration:

$$(\|t_m \cdots t_1\| \cdots \|i_w \cdots i_1 i\|j,\ a_{r+1} \cdots a_n\$)$$

(ii) ACTION$[i, a_r]$ = **resume right** of δ at address dot (rsδ@dot). The parser is coming to the right and below of the node at address dot in δ, say η, on which an auxiliary tree has been adjoined. The information identifying the auxiliary tree is in the sequence of stacks and must be recovered. There are two cases:

Case 1: η does not subsume a foot node. Let k be the number of terminal symbols subsumed by η. Before applying this move, the current configuration looks like:

$$(\|\cdots\|i_k\|\cdots\|i_1\|i,\ a_r \cdots a_n\$)$$

The k top first stacks are merged into one stack and the stack $\|$m is pushed on top of it, where $m = \text{GOTO}_{foot}[i_k, \beta]$ for some auxiliary tree β that can be adjoined in δ at η, and the parser enters the configuration:

$$(\|\cdots\|i_k\|i_{k-1} \cdots i_1 i\|m,\ a_r \cdots a_n\$)$$

Case 2: η subsumes the foot node of δ. Let k (resp. k') be the number of terminal symbols to the right (resp. to the left) of the foot node subsumed by η. Before applying this move, the configuration looks like:

$(\|\cdots\|n_{k'+1}\|\cdots\|n_1\|s_1\cdots s_l\|i_k\cdots\|i_1\|i,\ a_r\cdots a_n\$)$
The k' stacks below the $k+2^{th}$ stack from the top as well as the $k+1$ top stacks are rewritten onto the $k+2^{th}$ stack and the stack $\|m$ is pushed on top of it, where $m = \text{GOTO}_{foot}[n_{k'+1}, \beta]$ for some auxiliary tree β that can be adjoined in δ at η, and the parser enters the configuration:
$(\|\cdots\|n_{k'+1}\|[s_1\cdots s_l n_{k'}\cdots n_1 i_k\cdots i_1 i\|m,\ a_r\cdots a_n\$)$

(iii) ACTION$[i, a_r]$ = **reduce root** of an auxiliary tree β in which the last adjunction on the spine was performed at address *star* (rdβ@*star*). The parser has finished the recognition of the auxiliary tree β. It must remove all information about β and continue the recognition of the tree in which β was adjoined. The parser executes an unwrap move. Let k (resp. k') be the number of terminal symbols to the left (resp. to the right) of the foot node of β. Let ζ be the node at address *star* in β ($\zeta = nil$ if *star* is not set). Let p be the number of terminal symbols to the left of the foot node subsumed by ζ ($p = 0$ if $\zeta = nil$). $p + k' + 1$ symbols from the top of the sequence of stacks popped. Then $k - p$ single element stacks below the new top stack are unwrapped. Let j be the new top element of the top stack. Let $m = \text{GOTO}_{right}[j, \beta]$. j is popped and the single element stack $\|m$ is pushed on top of the top stack.

By keeping track of the auxiliary trees being reduced, it is possible to output a parse instead of acceptance or an error.

The parser recognizes the derived tree inside out: it extracts recursively the innermost auxiliary tree that has no adjunction performed in it.

5 LR(0) Parsing Tables

This section explain how to construct an LR(0) parsing table given a TAG. The construction is an extension of the one used for CFGs. Similarly to Schabes and Joshi (1988), we extend the notion of dotted rules to trees. We define the closure operations that correspond to adjunction. Then we explain how transitions between states are defined. We give in Figure 5 an example of a finite state automaton used to build the parsing table for a TAG (see Figure 5) generating a context-sensitive language.

We first explain preliminary concepts (originally defined to construct an Earley-type parser for TAGs) that will be used by the algorithm. Dotted rules are extended to trees. Then we recall a tree traversal that the algorithm will mimic in order to scan the input from left to right.

A **dotted symbol** is defined as a symbol associated with a dot *above* or *below* and either *to the left* or *to the right* of it. The four positions of the dot are annotated by la, lb, ra, rb (resp. left above, left below, right above, right below): ${}^{la}_{lb}A^{ra}_{rb}$. In practice, only two dot positions can be used (to the left and to the right of a node). However, for sake of simplicity, we will use four different dot positions. A **dotted tree** is defined as a tree with exactly one dotted symbol. Furthermore, some nodes in the dotted tree can be marked with a star. A star on a node expresses the fact that an adjunction has been performed on the corresponding node. A dotted tree is referred as *[α, dot, pos, stars]*, where α is a tree, *dot* is the address of the dot, *pos* is the position of the dot (la, lb, ra or rb) and *stars* is a list of nodes in α annotated by a star.

Given a dotted tree with the dot above and to the left of the root, we define a tree traversal of a dotted tree (as shown in the Figure 3) that will enable us to scan the frontier of an elementary tree from left to right while trying to recognize possible adjunctions between the above and below positions of the dot of interior nodes.

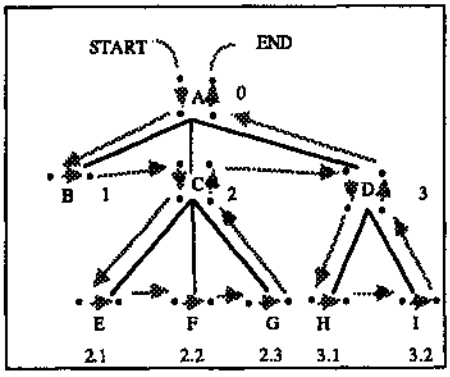


Figure 3: Left to Right Tree Traversal

A **state** in the finite state automaton is defined to be a set of dotted trees closed under the following operations: Adjunction Prediction, Left Completion, Move Dot Down, Move Dot Up and Skip Node (See Figure 4).[4]

Adjunction Prediction predicts all possible auxiliary trees that can be adjoining at a given node. **Left Completion** occurs when an auxiliary tree is recognized up to its foot node. All trees in which that tree can be adjoined are pulled back with the node on which adjunction has been performed added to the list of stars. **Move Dot Down** moves the dot down the links. **Move Dot Up** moves the dot up the links. **Skip Node** moves the dot up on the right hand side of a node on which no adjunction has been performed.

All the states in the finite state automaton (FSA) must be closed under the closure operations. The FSA is

[4]These operations correspond to processors in the Earley-type parser for TAGs.

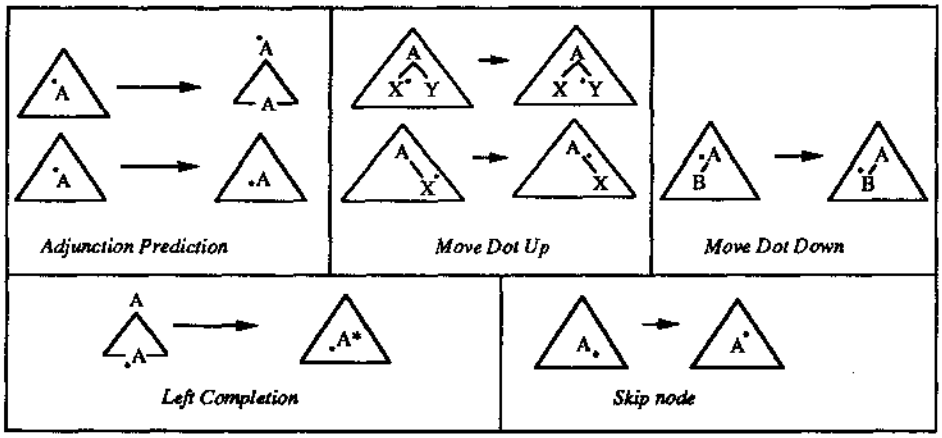


Figure 4: Closure Operations

build as follows. In states set 0, we put all initial trees with a dot to the left and above the root. The state is then closed. Then recursively we build new states with the following transitions (we refer to Figure 5 for an example of such a construction).

- A transition on a (where a is a terminal symbol) from S_i to S_j occurs if and only if in S_i there is a dotted tree *[δ, dot, la, stars]* in which the dot is to the left and above a terminal symbol a; S_j consists of the closure of the set of dotted trees of the form *[δ, dot, ra, stars]*.
- A transition on β_{right} from S_i to S_j occurs iff in S_i there is a dotted tree *[δ, dot, rb, stars]* such that the dot is to the right and below a node on which β can be adjoined; S_j consists of the closure of the set of dotted trees of the form *[δ, dot, ra, stars']*. If the dotted node of *[δ, dot, rb, stars]* is not on the spine[5] of δ, $star'$ consists of all the nodes in $star$ that strictly dominate the dotted node. When the dotted node is on the spine, $stars'$ consists of all the nodes in $star$ that strictly dominate the dotted node, if there are some, otherwise $stars' = \{dot\}$.
- A **Skip foot** of *[β, dot, lb, stars]* transition from S_i to S_j occurs iff in S_i there is a dotted tree *[β, dot, lb, stars]* such that the dot is to the left and below the foot node of the auxiliary tree β; S_j consists of the closure of the set of dotted trees of the form *[β, dot, rb, stars]*.

The parsing table is constructed from the FSA built as above. In the following, we write $trans(i, x)$ for set of states in the FSA reached from state i on the transition labeled by x.

The actions for $ACTION(i, a)$ are:

- **Shift** j $(sc(j))$. It applies iff $j \in trans(i, a)$.
- **Resume Right** of *[δ, dot, rb, stars]* (rsδ@*dot*). It applies iff in state i there is a dotted tree *[δ, dot, rb, stars]*, where $dot \in stars$.
- **Reduce Root** of β (rdβ@*star*). It applies iff in state i there is a dotted tree *[β, 0, ra, {star}]*, where β is an auxiliary tree.[6]
- **Accept** occurs iff a is the end marker ($a = \$$) and there is a dotted tree *[α, 0, ra, {star}]*, where α is an initial tree and the dot is to the right and above the root node.
- **Error**, if none of the above applies.

The GOTO table encodes the transitions in the FSA on non-terminal symbols. It is indexed by a state and by β_{right} or β_{foot}, for all auxiliary trees β: $j \in GOTO(i, label)$ iff there is a transition from i to j on the given label ($label \in \{\beta_{right}, \beta_{foot} | \beta$ is an auxiliary tree$\}$.

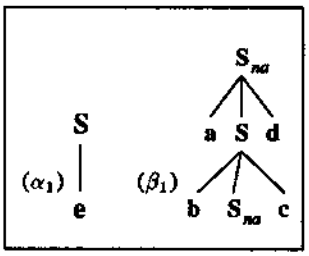
If more than one action is possible in an entry of the action table, the grammar is not LR(0): there is a conflict of action, the grammar cannot be parsed deterministically without lookahead.

An example of a finite state automaton used for the construction of the LR(0) table for a TAG (trees α_1, β_1 in Figure 5) generating[7] $L = \{a^n b^n e c^n d^n | n \geq 0\}$, its corresponding parsing table is given and an example of sequences of moves are given in Figure 5.

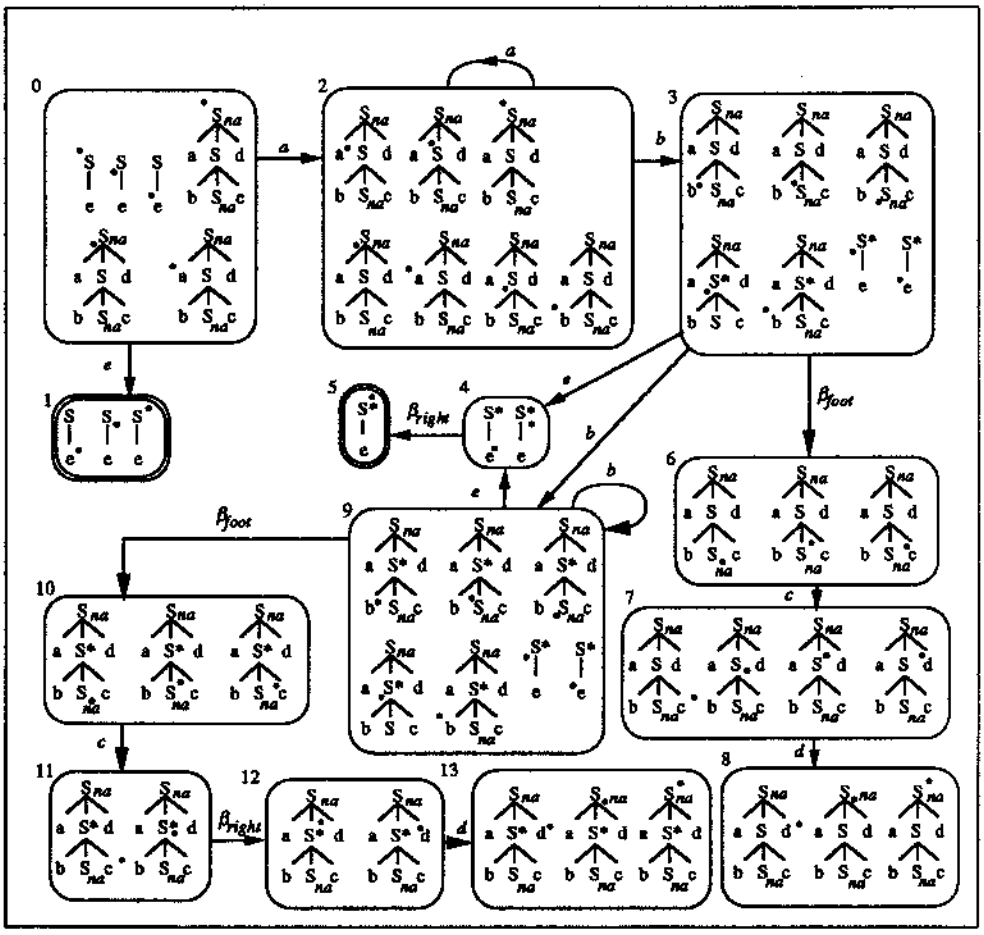
[5] Nodes on the path from root node to foot node.

[6] 0 is the address of the root node.

[7] In the given TAG (trees α_1 and β_1), if we omit a and c, we obtain a TAG that is similar to the one for the Dutch cross-serial construction. This grammar can still be handled by an LR(0) parser.
In the trees α and β, na stand for null adjunction constraint (i.e. no auxiliary tree can be adjoined on a node with null adjunction constraint).



TAG for $L = \{a^n b^n e c^n d^n\}$



Finite State Automaton for a BEPDA Recognizing $L = \{a^n b^n e c^n d^n\}$

	PARSING ACTION						*GOTO* foot	*GOTO* right
	a	b	c	d	e	$	β	β
0	s2				s1			
1						acc		
2	s2	s3						
3		s9			s4		6	
4	rsα@0	rsα@0	rsα@0	rsα@0	rsα@0	rsα@0		5
5						acc		
6			s7					
7				s8				
8	rdβ@–	rdβ@–	rdβ@–	rdβ@–	rdβ@–	rdβ@–		
9		s9			s4		10	
10			s11					
11	rsβ@2	rsβ@2	rsβ@2	rsβ@2	rsβ@2	rsβ@2		12
12				s13				
13	rdβ@2	rdβ@2	rdβ@2	rdβ@2	rdβ@2	rdβ@2		

Example of LR(0) Parsing Table

Parser configuration	Next move
(‖0, *aabbeccdd*$)	s2
(‖0‖2, *abbeccdd*$)	s2
(‖0‖2‖2, *bbeccdd*$)	s3
(‖0‖2‖2‖3, *beccdd*$)	s9
(‖0‖2‖2‖3‖9, *eccdd*$)	s4
(‖0‖2‖2‖3‖9‖4, *ccdd*$)	rsα@0
(‖0‖2‖2‖3‖9‖4‖10, *ccdd*$)	s11
(‖0‖2‖2‖3‖9‖4‖10‖11, *cdd*$)	rsβ@2
(‖0‖2‖2‖3‖4 9 10 11‖6, *cdd*$)	s7
(‖0‖2‖2‖3‖4 9 10 11‖6‖7, *dd*$)	s8
(‖0‖2‖2‖3‖4 9 10 11‖6‖7‖8, *d*$)	rdβ@–
(‖0‖2‖4 9 10‖12, *d*$)	s13
(‖0‖2‖4 9 10‖12‖13, $)	rdβ@2
(‖0‖5, $)	acc

Example of sequences of moves

sj ≡ Shift *j*; rsδ@*dot* ≡ Resume Right of δ at *dot*; rdβ@*star* ≡ Reduce Root of β with star at address *star*; $ ≡ end of input.

Figure 5: Example of the construction of an LR(0) parser for a TAG recognizing $L = \{a^n b^n e c^n d^n\}$

6 SLR(1) Parsing Tables

The tables that we have constructed are LR(0) tables. The Resume Right and Reduce Root moves are performed regardless of the next input token. The accuracy of the parsing table can be improved by computing lookaheads. FIRST and FOLLOW can be extended to dotted trees.[8] FIRST of a dotted tree corresponds to the set of left most symbols appearing below the subtree dominated by the dotted node. FOLLOW of a dotted tree defines the set of tokens that can appear in a derivation immediately following the dotted node. Once FIRST and FOLLOW computed, the LR(0) parsing table can be improved to an SLR(1) table: Resume Right and Reduce Root are applicable only on the input tokens in the follow set of the dotted tree.

For example, the SLR(1) table for the TAG built with trees α_1 and β_1 is given in Figure 6.

	PARSING ACTION						*GOTO*	
							foot	right
	a	b	c	d	e	$	β	β
0	s2				s1			
1						acc		
2	s2	s3						
3		s9			s4		6	
4			rsα@0					5
5						acc		
6			s7					
7				s8				
8				rdβ@–		rdβ@–		
9		s9			s4		10	
10			s11					
11			rsβ@2					12
12				s13				
13				rdβ@2		rdβ@2		

Figure 6: Example of SLR(1) Parsing Table

By associating dotted trees with lookaheads, one can also compute LR(k) items in the finite state automaton in order to build LR(k) parsing tables.

7 Current Research

The deterministic parsers we have developed do not satisfy an important property satisfied by LR parsers for CFG. This property is often described as the viable prefix property which states that as long as the portion of the input considered so far leads to some stack configuration (i.e. does not lead to error), it is always possible to find a suffix to obtain a string in the language.

Our parsers do not satisfy this property because the left completion move is not a 'reduce' move. This move applies when we have reached a bottom-left end (to the left of the foot node) of an auxiliary tree, say β. If we had considered this move to be a reduce move, then by popping appropriate amount of elements off the storage would allow us to figure out which tree (into which β was adjoined), say α, to proceed with. Rather than using this information (that is available in the storage of the BEPDA), by putting left completion in the closure operations, we apply a move that is akin to the predict move of Earley parser. That is we continue by considering every possible nodes β could have been adjoined at, which could include nodes in trees that were not used so far. However, we do not accept incorrect strings, we only lose the prefix property (for an example see Figure 7). As a consequence, errors are always detected but not as soon as possible.

Parser configuration	Next move
($\|\|0$, *aabeccdd*\$)	s2
($\|\|0\|\|2$, *abeccdd*\$)	s2
($\|\|0\|\|2\|\|2$, *beccdd*\$)	s3
($\|\|0\|\|2\|\|2\|\|3$, *eccdd*\$)	s4
($\|\|0\|\|2\|\|2\|\|3\|\|4$, *ccdd*\$)	rsα@0
($\|\|0\|\|2\|\|2\|\|3\|\|4\|\|6$, *ccdd*\$)	s7
($\|\|0\|\|2\|\|2\|\|3\|\|4\|\|6\|\|7$, *cdd*\$)	error

Figure 7: Example of error detecting

The reason why we did not consider the left completion move to be a reduce move is related to the restrictions on moves of BEPDA which is weakly equivalent to TAGs (perhaps also due to the fact that left to right parsing may not be most natural for parsing TAGs which produce trees with context-free path sets). In CFGs, where there is only horizontal stacking, a single reduction step is used to account for the application of rule in left to right parsing. On the other hand, with TAGs, if a tree is used successfully, it appears that a prediction move and more than one reduction move are necessary for auxiliary tree. In left to right parsing, a prediction is made to start an auxiliary tree β at top left end; a reduction is appropriate to recover the node β was adjoined at the left completion stage; a reduction is needed again at resume right state to resume the right end of β; finally a reduction is needed at the right completion stage. In our algorithm, reductions are used at right resume stage and reduce right state. Even if a reduction step is applied at left completion stage, an encoding of the fact that left part of β (as well as the left part of trees adjoined on the spine of β) has been completed has to be restored in the storage (note in a reduction move of any shift reduce parser for CFGs, any information about the rule used is discarded once reduction step applied). So far we have not been able to apply a reduction step at the left completion stage, reinsert the left part of β and yet maintain

[8] Due to the lack of space, we do not define FIRST and FOLLOW. However, we explain the basic principles used for the computation of FIRST and FOLLOW.

the correct sequence in the storage so that the right part of β can be recovered at the resume right stage. We are considering alternative strategies for shift reduce parsing with BEPDA as well as considering whether there are other automata models equivalent to TAGs better suited for deterministic left to right parsing of tree-adjoining languages.

Conclusion

We have introduced a bottom-up machine (Bottom-up Embedded Push Down Automaton) that enabled us to define LR-like parsers for TAGs. The machine recognizes in a bottom-up fashion exactly the set of Tree Adjoining Languages.

We described the LR parsing algorithm and a method for computing LR(0) parsing tables. We also mentioned the possibility of building SLR(k) parsing tables by defining the notions of FIRST and FOLLOW sets for TAGs.

As shown for the example, no lookaheads are necessary to parse deterministically the language $L = \{a^n b^n e c^n d^n | n \geq 0\}$. If instead of using e, we had the empty string ϵ in the initial tree, LR(0)-like parser will not be enough. On the other hand SLR(1)-like parser will suffice.

We have noted that our parsers do not satisfy the valid prefix property. As a consequence, errors are always detected but not as soon as possible.

Similar to the work of Lang (1974) and Tomita (1987) extending LR parsers for arbitrary CFGs, the LR parsers for TAGs can be extended to solve by pseudo-parallelism the conflicts of moves.

References

Joshi, Aravind K., 1985. How Much Context-Sensitivity is Necessary for Characterizing Structural Descriptions—Tree Adjoining Grammars. In Dowty, D., Karttunen, L., and Zwicky, A. (editors), *Natural Language Processing—Theoretical, Computational and Psychological Perspectives*. Cambridge University Press, New York. Originally presented in a Workshop on Natural Language Parsing at Ohio State University, Columbus, Ohio, May 1983.

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

Knuth, D. E., 1965. On the translation of languages from left to right. *Inf. Control* 8:607–639.

Lang, Bernard, 1974. Deterministic Techniques for Efficient Non-Deterministic Parsers. In Loeckx, Jacques (editor), *Automata, Languages and Programming, 2nd Colloquium, University of Saarbrücken*. Lecture Notes in Computer Science, Springer Verlag.

Révész, G., 1971. Unilateral context sensitive grammars and left to right parsing. *J. Comput. System Sci.* 5:337–352.

Schabes, Yves and Joshi, Aravind K., June 1988. An Earley-Type Parsing Algorithm for Tree Adjoining Grammars. In 26^{th} *Meeting of the Association for Computational Linguistics (ACL'88)*. Buffalo.

Thatcher, J. W., 1971. Characterizing Derivations Trees of Context Free Grammars through a Generalization of Finite Automata Theory. *J. Comput. Syst. Sci.* 5:365–396.

Tomita, Masaru, 1987. An Efficient Augmented-Context-Free Parsing Algorithm. *Computational Linguistics* 13:31–46.

Turnbull, C. J. M. and Lee, E. S., 1979. Generalized Deterministic Left to Right Parsing. *Acta Informatica* 12:187–207.

Vijay-Shanker, K., 1987. *A Study of Tree Adjoining Grammars*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania.

Walters, D.A., 1970. Deterministic Context-Sensitive Languages. *Inf. Control* 17:14–40.