Linear indexed automata and tabulation of TAG parsing

Mark-Jan Nederhof
University of Groningen, Faculty of Arts, Humanities Computing
P.O. Box 716, NL-9700 AS Groningen, The Netherlands
E-mail: markjan@let.rug.nl

Abstract

We present a new kind of recognizer for tree-adjoining languages, the linear indexed automata. Such recognizers allow straightforward realization by means of logical programming languages. We show that the computations by such automata, which are in general nondeterministic, can be tabulated by means of an extension of a technique originally devised for context-free languages. The proposed application of this work is the design of efficient parsing algorithms for tree-adjoining grammars.

1 Introduction

Design of correct and efficient parsing algorithms for tree-adjoining grammars (TAGs) is a difficult task. As discussed in [9], certain problems relating to TAG parsing were solved only after many years, and in one case a parsing algorithms was shown to be incorrect beyond repair as much as seven years after its publication. Many open problems still exist.

A possible way to simplify the task of developing tabular algorithms is to apply well-known techniques from the realm of context-free parsing and logical programming, which allow tabulation to be seen separately from the parsing strategy: the actual parsing strategy can be described by means of a (nondeterministic) pushdown automaton or a set of Horn clauses, and tabulation is introduced by means of some generic mechanism such as memoization. For example, if we choose the parsing strategy to be LR parsing [13] and generate a nondeterministic LR parser in the form of a pushdown automaton, then we may construct a *tabular* LR parser by applying the generic technique from [4, 3], which allows tabulation of any pushdown automaton.

This modular way of constructing tabular algorithms has obvious advantages over direct constructions, as exemplified for tabular LR parsing by [14]. For example, it allows more straightforward proofs of correctness, is easier to understand and cheaper to implement.

The first modular approach to TAG parsing was proposed in [6]: a TAG is compiled into a logical pushdown automaton, which is interpreted by means of dynamic programming. However, it turns out that the chosen dynamic programming technique is too general for this particular task, and therefore requires fine-tuning in order to obtain an appropriate TAG parsing algorithm.

Notable are further [7, 16], which propose to separate the parsing problem into the intersection of the grammar with an input and reduction of the resulting grammar.

In this paper we propose a different approach, which has strong advantages for certain parsing strategies. In particular, in a recent publication [10] we presented a new non-deterministic LR parsing algorithm for TAGs that is expressed in terms of a particular kind of recognizer, different from existing types of recognizer for TAGs such as embedded pushdown automata [12] and 2-SA [2]. The work presented here shows that the new type of recognizer can be simulated by means of tabulation. This results in a new tabular algorithm. Furthermore, we conjecture that by our new approach many existing parsing algorithms for TAGs can be reformulated in a modular way, by distinguishing between the parsing strategy and the tabulation technique.

2 Linear indexed grammars

Linear indexed grammars (LIGs) generate the same class of languages as the tree-adjoining grammars. Other kinds of grammar generating this class of languages are the head grammars and the combinatory categorial grammars [17].

For presentational reasons we restrict ourselves to LIGs in a normal form: each right-hand side consists of either a terminal, the empty string, or one or two nonterminals, and each production manipulates at most one index. The last restriction helps us to avoid some of the technical problems that had to be overcome in [15]. The weak equivalence to standard forms of LIG will not be proven here.

Thus, a LIG G is a 5-tuple $(\Sigma, \mathcal{N}, S, \mathcal{I}, \mathcal{P})$, where Σ is a finite set of *terminals*, $\mathcal{N}$ is a finite set of *nonterminals*, $S \in \mathcal{N}$ is the *start symbol*, $\mathcal{I}$ is a finite set of *indices* and $\mathcal{P}$ is a finite set of *productions*, having one of the following forms: $A[\circ\circ\,\eta] \rightarrow B[\circ\circ\,\eta']\ C[\,]$, $A[\circ\circ\,\eta] \rightarrow B[\,]\ C[\circ\circ\,\eta']$, $A[\circ\circ\,\eta] \rightarrow B[\circ\circ\,\eta']$, $A[\,] \rightarrow a$, or $A[\,] \rightarrow \varepsilon$, where $A, B, C \in \mathcal{N}$, $\eta, \eta' \in \mathcal{I} \cup \{\varepsilon\}$, $a \in \Sigma$; and for each production, either η or η' (or both) must be the empty string (denoted by ε).

For a precise description of how LIGs generate languages, we refer to [17]. Let it suffice here to mention that $[\circ\circ\,\eta]$ stands for a list. In the case that η is of the form $p \in \mathcal{I}$ then p is the head of that list and $\circ\circ$ is the unspecified tail. In the case that $\eta = \varepsilon$ however, $\circ\circ$ is the list itself. Both occurrences of $\circ\circ$ in a production refer to the *same* list ("consistent substitution"). The empty list is written as $[\,]$. Thus, a production $A[\circ\circ\,p] \rightarrow B[\circ\circ]\ C[\,]$ can be written as the following production of a definite clause grammar [11]:

```
big_a([p | List]) --> big_b(List), big_c([]).
```

As another example, a production $A[\,] \rightarrow a$ can be written as

```
big_a([]) --> "a".
```

The start symbol is to be attached to the empty list of indices: $S[\,]$. Thus, a Prolog query for input $a_1 \cdots a_n$, would be of the form

```
?- big_s([], [a_1, ...,  a_n], []).
```

An example of a LIG is given by the following set of productions:

$$
\begin{array}{llll}
S[\circ\circ] \rightarrow A[\,]\ X[\circ\circ] & Y[\circ\circ] \rightarrow B[\,]\ Z[\circ\circ] & A[\,] \rightarrow a \\
X[\circ\circ] \rightarrow Y[\circ\circ\,p]\ D[\,] & Z[\circ\circ\,p] \rightarrow P[\circ\circ]\ C[\,] & B[\,] \rightarrow b \\
Y[\circ\circ] \rightarrow A[\,]\ X[\circ\circ] & P[\circ\circ] \rightarrow B[\,]\ Z[\circ\circ] & C[\,] \rightarrow c \\
 & P[\,] \rightarrow \varepsilon & D[\,] \rightarrow d
\end{array}
$$

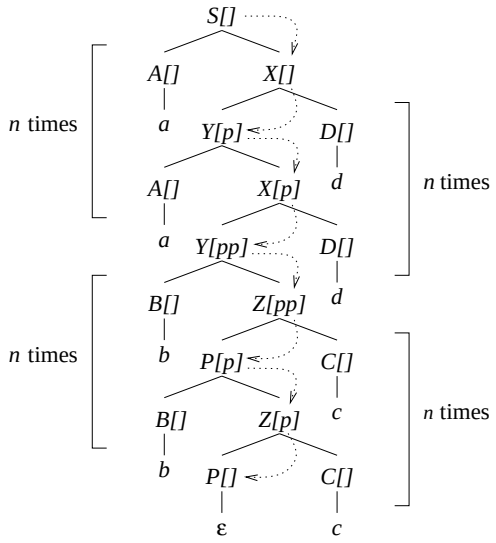

Figure 1: Parse tree for a LIG.

The language it generates is $\{a^n b^n c^n d^n \mid n > 0\}$. A parse tree for the string *aabbccdd* is given in Figure 1.

3 Linear indexed automata

We introduce a type of recognizer that is equivalent to linear indexed grammars and that is based on the well-known pushdown automata, which are equivalent to context-free grammars. The rationale is that LIGs are nothing more than context-free grammars extended with parameters in the form of lists of indices, and therefore pushdown automata extended with the same kind of parameter suffice to built recognizers for languages generated by LIGs.

Again for presentational reasons, we give a restricted type. A *linear indexed automaton* M is a 6-tuple $(\Sigma, \mathcal{Q}, I, F, \mathcal{I}, \mathcal{T})$, where Σ and $\mathcal{I}$ are as before, $\mathcal{Q}$ is a finite set of *stack symbols*, $I \in \mathcal{Q}$ is the *initial stack symbol*, $F \in \mathcal{Q}$ is the *final stack symbol* and $\mathcal{T}$ is a finite set of *transitions*, having one of the following forms:

- $X[\circ\circ] \overset{z}{\mapsto} X[\circ\circ]\ Z[]$;
- $X[\circ\circ\eta] \overset{z}{\mapsto} Z[\circ\circ\eta']$;
- $Y[]\ X[\circ\circ\eta] \overset{z}{\mapsto} Z[\circ\circ\eta']$; or
- $Y[\circ\circ\eta]\ X[] \overset{z}{\mapsto} Z[\circ\circ\eta']$,

where $X, Y, Z \in \mathcal{Q}$, $\eta, \eta' \in \mathcal{I} \cup \{\varepsilon\}$, $z \in \Sigma \cup \{\varepsilon\}$; and for each transition that is of one of the last three forms, either η or η' (or both) must be the empty string.

We define a *configuration* to be an element of $(\mathcal{Q} \times \mathcal{I}^*)^* \times \Sigma^*$. Each element from $\mathcal{Q} \times \mathcal{I}^*$ contains a stack symbol and a list of indices. A list of such elements from $\mathcal{Q} \times \mathcal{I}^*$ represents the stack of the automaton. The stack is constructed from left to right, i.e. the bottom element will be represented as the leftmost element. An element from Σ^*

Stack	Input	Next step
$I[\,]$	$aaabbbcccddd$	$I[\circ\circ] \overset{a}{\hookrightarrow} I[\circ\circ]\ X[\,]$
$I[\,]\ X[\,]$	$aabbbcccddd$	$X[\circ\circ] \overset{a}{\hookrightarrow} X[\circ\circ]\ X[\,]$
$I[\,]\ X[\,]\ X[\,]$	$abbbcccddd$	$X[\circ\circ] \overset{a}{\hookrightarrow} X[\circ\circ]\ X[\,]$
$I[\,]\ X[\,]\ X[\,]\ X[\,]$	$bbbcccddd$	$X[\circ\circ] \overset{b}{\hookrightarrow} X[\circ\circ]\ Y[\,]$
$I[\,]\ X[\,]\ X[\,]\ X[\,]\ Y[\,]$	$bbcccddd$	$Y[\circ\circ] \overset{b}{\hookrightarrow} Y[\circ\circ]\ Y[\,]$
$I[\,]\ X[\,]\ X[\,]\ X[\,]\ Y[\,]\ Y[\,]$	$bcccddd$	$Y[\circ\circ] \overset{b}{\hookrightarrow} Y[\circ\circ]\ Y[\,]$
$I[\,]\ X[\,]\ X[\,]\ X[\,]\ Y[\,]\ Y[\,]\ Y[\,]$	$cccddd$	$Y[\circ\circ] \overset{c}{\hookrightarrow} Z[\circ\circ p]$
$I[\,]\ X[\,]\ X[\,]\ X[\,]\ Y[\,]\ Y[\,]\ Z[p]$	$ccddd$	$Y[\,]\ Z[\circ\circ] \overset{c}{\hookrightarrow} Z[\circ\circ p]$
$I[\,]\ X[\,]\ X[\,]\ X[\,]\ Y[\,]\ Z[pp]$	$cddd$	$Y[\,]\ Z[\circ\circ] \overset{c}{\hookrightarrow} Z[\circ\circ p]$
$I[\,]\ X[\,]\ X[\,]\ X[\,]\ Z[ppp]$	ddd	$X[\,]\ Z[\circ\circ p] \overset{d}{\hookrightarrow} P[\circ\circ]$
$I[\,]\ X[\,]\ X[\,]\ P[pp]$	dd	$X[\,]\ P[\circ\circ p] \overset{d}{\hookrightarrow} P[\circ\circ]$
$I[\,]\ X[\,]\ P[p]$	d	$X[\,]\ P[\circ\circ p] \overset{d}{\hookrightarrow} P[\circ\circ]$
$I[\,]\ P[\,]$	ε	$I[\circ\circ]\ P[\,] \overset{\varepsilon}{\hookrightarrow} F[\circ\circ]$
$F[\,]$	ε	recognition

Figure 2: Sequence of configurations of a linear indexed automaton.

represents the remaining suffix of the input v after a certain number of symbols at the left end have been consumed.

The finite set of transitions can conceptually be seen as the infinite set that results by consistently substituting $\circ\circ$ in the right-hand and left-hand sides of transitions by arbitrary elements from $\mathcal{I}^*$. For example, a transition $X[\circ\circ\eta] \overset{z}{\hookrightarrow} Z[\circ\circ\eta']$ may be seen as an infinite set of transitions of the form $X[\eta''\eta] \overset{z}{\hookrightarrow} Z[\eta''\eta']$, where $\eta'' \in \mathcal{I}^*$.

Let the resulting infinite set of "instantiated transitions" be called $\mathcal{T}'$. We define the binary relation $\vdash$ between configurations as: $(\alpha\beta, zw) \vdash (\alpha\gamma, w)$ if and only if $\beta \overset{z}{\hookrightarrow} \gamma$ in $\mathcal{T}'$, for any $\alpha \in (\mathcal{Q} \times \mathcal{I}^*)^*$ and $w \in \Sigma^*$. The transitive and reflexive closure of $\vdash$ is denoted by $\vdash^*$.

Some input v is *recognized* if $(I[\,], v) \vdash^* (F[\,], \varepsilon)$. The language *accepted* by M is defined to be the set of all v that are recognized. A language is accepted by a linear indexed automaton if and only if it is generated by a linear indexed grammar. A proof will not be given here.

An example of a linear indexed automaton is given by the following set of transitions.

$$
\begin{array}{rcl@{\qquad}rcl}
I[\circ\circ] & \overset{a}{\mapsto} & I[\circ\circ]\ X[\,] & X[\circ\circ] & \overset{a}{\mapsto} & X[\circ\circ]\ X[\,] \\
X[\circ\circ] & \overset{b}{\mapsto} & X[\circ\circ]\ Y[\,] & Y[\circ\circ] & \overset{b}{\mapsto} & Y[\circ\circ]\ Y[\,] \\
Y[\circ\circ] & \overset{c}{\mapsto} & Z[\circ\circ p] & Y[\,]\ Z[\circ\circ] & \overset{c}{\mapsto} & Z[\circ\circ p] \\
X[\,]\ Z[\circ\circ p] & \overset{d}{\mapsto} & P[\circ\circ] & X[\,]\ P[\circ\circ p] & \overset{d}{\mapsto} & P[\circ\circ] \\
I[\circ\circ]\ P[\,] & \overset{\varepsilon}{\mapsto} & F[\circ\circ] & & &
\end{array}
$$

The automaton accepts the language $\{a^n b^n c^n d^n \mid n > 0\}$. Figure 2 shows how the input $a^3b^3c^3d^3$ is recognized.

Linear indexed automata represent a subclass of the logical pushdown automata [5]. The reason this subclass is considered in isolation is that it allows a specific form of

tabulation, as will be explained below.

4 Tabulation

Linear indexed automata manipulate stacks on two levels, since the lists of indices act as stacks embedded in the other stack formed by elements from $\mathcal{Q} \times \mathcal{I}^*$. (We will further abstain from referring to lists of indices as stacks in order to avoid the obvious confusion that would ensue from this.) The consequence is that we can apply in a 2-dimensional way the tabulation technique from [4, 3], which was originally devised for pushdown automata with only one kind of stack. The nature of the resulting tabular algorithm shows some similarities to the tabular, LR-like algorithm from [1].

The algorithm is described as follows. Given a linear indexed automaton M and an input v, we construct a table U in polynomial time. From certain entries in U, we can effectively decide whether the input is in the language. The procedure can be extended so that a representation of all parse trees, the *parse forest*, is produced as side-effect of the construction of U.

The table U contains *items* consisting of 9 fields which we pragmatically divide into two tuples of 4 and 5 fields respectively: $((X, Y, i, j), (p, Z, P, k, l))$, where $X, Y, Z, P \in \mathcal{Q}$, $p \in \mathcal{I}$ and i, j, k, l are natural numbers between 0 and n, representing positions in the input $v = a_1 \cdots a_n \in \Sigma^*$.

The meaning of the 4-tuple is unchanged with regard to the original tabulation method for pushdown automata: by reading the input from i to j we may push Y on top of X. The 5-tuple contains information with respect to the list of indices that is associated with Y: its head is p and its tail is a list that is associated with some P that can be pushed on top of Z by reading the input from k to l. See Figure 3 for a pictorial representation.

More precisely, an element $((X, Y, i, j), (p, Z, P, k, l))$ indicates the existence of a sequence of steps of the automaton of the form $(\alpha\ X[\eta],\ a_{i+1} \cdots a_n) \vdash^* (\alpha\ X[\eta]\ \beta\ Z[\eta'],\ a_{k+1} \cdots a_n) \vdash^* (\alpha\ X[\eta]\ \beta\ Z[\eta']\ P[\eta''],\ a_{l+1} \cdots a_n) \vdash^* (\alpha\ X[\eta]\ Y[\eta''p],\ a_{j+1} \cdots a_n)$, where $\alpha, \beta \in (\mathcal{Q} \times \mathcal{I}^*)^*$ and $\eta, \eta', \eta'' \in \mathcal{I}^*$, and

- nowhere between $(\alpha\ X[\eta],\ a_{i+1} \cdots a_n)$ and $(\alpha\ X[\eta]\ Y[\eta''p],\ a_{j+1} \cdots a_n)$ does the stack shrink to the height of $\alpha\ X[\eta]$;
- nowhere between $(\alpha\ X[\eta]\ \beta\ Z[\eta'],\ a_{k+1} \cdots a_n)$ and $(\alpha\ X[\eta]\ \beta\ Z[\eta']\ P[\eta''],\ a_{l+1} \cdots a_n)$ does the stack shrink to the height of $\alpha\ X[\eta]\ \beta\ Z[\eta']$; and
- the two occurrences of η'' are the same list in the sense that it is passed on unaffected through the steps from $(\alpha\ X[\eta]\ \beta\ Z[\eta']\ P[\eta''],\ a_{l+1} \cdots a_n)$ to $(\alpha\ X[\eta]\ Y[\eta''p],\ a_{j+1} \cdots a_n)$. It is allowed that elements from $\mathcal{I}$ are pushed on η'' and then popped again, but it is not allowed that elements from η'' are popped and then other elements are pushed to accidentally result in the same list.

If we want to indicate that the list of indices associated with Y is empty, we use a "dummy" 5-tuple $(\diamond, \square, \square, 0, 0)$, where $\diamond$ is a fresh symbol representing an imaginary index in an empty list of indices, and similarly $\square$ is a "dummy" stack symbol. In the initial item that is added to U, $\square$ also acts as imaginary element below the actual bottom element I.

The steps of the tabular algorithm are the following:

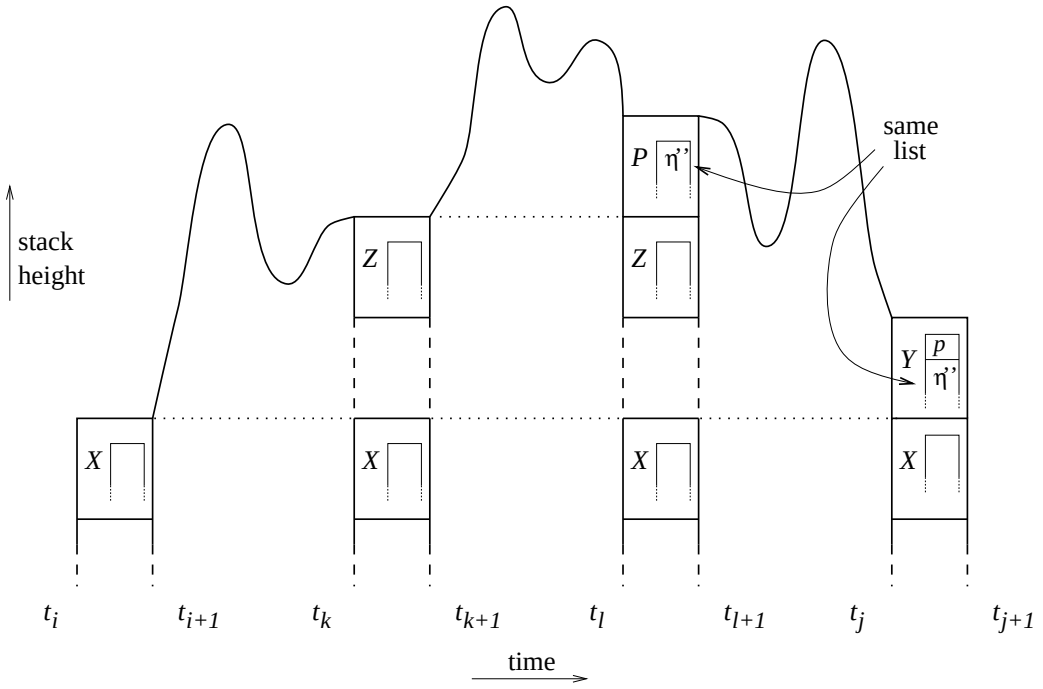

Figure 3: Meaning of an item $((X, Y, i, j), (p, Z, P, k, l))$. Time t_m is when a_m is read.

1. We initialize $U = \{((\Box, I, 0, 0), (\Diamond, \Box, \Box, 0, 0))\}$.
2. We repeat the following until no more new items can be added to U.
   (a) We choose an item $((Y, X, i, j), (p, P, R, k, l)) \in U$.
   (b) We choose some transition of one of the 4 types:
      i. $X[\circ\circ] \overset{z}{\hookrightarrow} X[\circ\circ]\ Z[\,]$;
      ii. $X[\circ\circ\,\eta] \overset{z}{\hookrightarrow} Z[\circ\circ\,\eta']$;
      iii. $Y[\,]\ X[\circ\circ\,\eta] \overset{z}{\hookrightarrow} Z[\circ\circ\,\eta']$; or
      iv. $Y[\circ\circ\,\eta'']\ X[\,] \overset{z}{\hookrightarrow} Z[\circ\circ\,\eta']$,

   and

   - in the case of type (iii) we choose an item $((Q, Y, m, i), (\Diamond, \Box, \Box, 0, 0)) \in U$; and
   - in the case of type (iv) we choose an item $((Q, Y, m, i), (p'', P'', R'', k'', l'')) \in U$,

   such that the following is satisfied:

   - If $z = \varepsilon$, then we define $j' = j$. If $z \in \Sigma$, then we require $z = a_{j+1}$ and define $j' = j + 1$.
   - (for type (ii) and (iii):) If $\eta = \varepsilon$ and $\eta' = \varepsilon$, then we define $(p', P', R', k', l') = (p, P, R, k, l)$. If $\eta = \varepsilon$ and $\eta' \in \mathcal{I}$, then we define $(p', P', R', k', l') = (\eta', Y, X, i, j)$. If $\eta \in \mathcal{I}$ (and therefore $\eta' = \varepsilon$), then we require $\eta = p$ and we choose an item $((P, R, k, l), (p', P', R', k', l')) \in U$.

Nr.	Item in U	Derived from
0	$((\Box, I, 0, 0), (\Diamond, \Box, \Box, 0, 0))$	initial item
1	$((I, X, 0, 1), (\Diamond, \Box, \Box, 0, 0))$	$I[\circ\circ] \overset{a}{\hookrightarrow} I[\circ\circ]\ X[\,]$ and 0
2	$((X, X, 1, 2), (\Diamond, \Box, \Box, 0, 0))$	$X[\circ\circ] \overset{a}{\hookrightarrow} X[\circ\circ]\ X[\,]$ and 1
3	$((X, X, 2, 3), (\Diamond, \Box, \Box, 0, 0))$	$X[\circ\circ] \overset{a}{\hookrightarrow} X[\circ\circ]\ X[\,]$ and 2
4	$((X, Y, 3, 4), (\Diamond, \Box, \Box, 0, 0))$	$X[\circ\circ] \overset{b}{\hookrightarrow} X[\circ\circ]\ Y[\,]$ and 3
5	$((Y, Y, 4, 5), (\Diamond, \Box, \Box, 0, 0))$	$Y[\circ\circ] \overset{b}{\hookrightarrow} Y[\circ\circ]\ Y[\,]$ and 4
6	$((Y, Y, 5, 6), (\Diamond, \Box, \Box, 0, 0))$	$Y[\circ\circ] \overset{b}{\hookrightarrow} Y[\circ\circ]\ Y[\,]$ and 5
7	$((Y, Z, 5, 7), (p, Y, Y, 5, 6))$	$Y[\circ\circ] \overset{c}{\hookrightarrow} Z[\circ\circ p]$ and 6
8	$((Y, Z, 4, 8), (p, Y, Z, 5, 7))$	$Y[\,]\ Z[\circ\circ] \overset{c}{\mapsto} Z[\circ\circ p]$ and $7+5$
9	$((X, Z, 3, 9), (p, Y, Z, 4, 8))$	$Y[\,]\ Z[\circ\circ] \overset{c}{\mapsto} Z[\circ\circ p]$ and $8+4$
10	$((X, P, 2, 10), (p, Y, Z, 5, 7))$	$X[\,]\ Z[\circ\circ p] \overset{d}{\hookrightarrow} P[\circ\circ]$ and $9+3+8$
11	$((X, P, 1, 11), (p, Y, Y, 5, 6))$	$X[\,]\ P[\circ\circ p] \overset{d}{\hookrightarrow} P[\circ\circ]$ and $10+2+7$
12	$((I, P, 0, 12), (\Diamond, \Box, \Box, 0, 0))$	$X[\,]\ P[\circ\circ p] \overset{d}{\hookrightarrow} P[\circ\circ]$ and $11+1+6$
13	$((\Box, F, 0, 12), (\Diamond, \Box, \Box, 0, 0))$	$I[\circ\circ]\ P[\,] \overset{\varepsilon}{\hookrightarrow} F[\circ\circ]$ and $12+0$

Figure 4: Tabular recognition of the input $a_1 a_2 \cdots a_{12} = aaabbbcccddd$.

- (for type (iv):) We require $(p, P, R, k, l) = (\Diamond, \Box, \Box, 0, 0)$. If $\eta'' = \varepsilon$ and $\eta' = \varepsilon$, then we define $(p', P', R', k', l') = (p'', P'', R'', k'', l'')$. If $\eta'' = \varepsilon$ and $\eta' \in \mathcal{I}$, then we define $(p', P', R', k', l') = (\eta', Q, Y, m, i)$. If $\eta'' \in \mathcal{I}$, then we require $\eta'' = p''$ and we choose an item $((P'', R'', k'', l''), (p', P', R', k', l')) \in U$.

(c) Depending on which of the 4 types of transition we have chosen, we add a new item to U.

- In the case of (i) we add $((X, Z, j, j'), (\Diamond, \Box, \Box, 0, 0))$.
- In the case of (ii) we add $((Y, Z, i, j'), (p', P', R', k', l'))$.
- In the case of (iii) we add $((Q, Z, m, j'), (p', P', R', k', l'))$.
- In the case of (iv) we add $((Q, Z, m, j'), (p', P', R', k', l'))$.

3. We accept the input if and only if U contains $((\Box, F, 0, n), (\Diamond, \Box, \Box, 0, 0))$.

Application of the tabular algorithm on the running example is presented in Figure 4.

The proof of correctness will not be discussed here.

In each iteration of step 2, at most seven different input positions are manipulated at the same time. This implies that the algorithm can straightforwardly be implemented to have a time-complexity of $\mathcal{O}(n^7)$. By using ideas from [9], this can easily be improved to $\mathcal{O}(n^6)$. The space-complexity is $\mathcal{O}(n^4)$, since items contain four input positions.

5 A dual type of linear indexed automata

The type of linear indexed automaton introduced in Section 3 is biased towards manipulation of lists of indices while the stack shrinks. The consequence for the kind of recognition we may express by our automata is best illustrated by means of Figure 1. The arrows leading from $S[\,]$ to $P[\,]$ indicate how lists are passed on from mother to daughter nodes. A maximal collection of nodes that conceptually pass on lists from one to the other will be called a *spine*. The spine between $S[\,]$ and $P[\,]$ in Figure 1 is the only nontrivial spine. In general, several spines can be found in a parse tree.

The task of a recognizer in this example is to verify that the numbers of a's, b's, c's and d's match. In the automaton we gave for the same language in Section 3 this was done by two cooperating mechanisms. First, the automaton stores X's and Y's on the stack for all a's and b's that it reads and later matches these to the numbers of d's and c's, respectively. This means that the stack symbols by themselves, without the lists of indices, ensure that the input string is of the form $a^n b^m c^m d^n$.

The second mechanism consists of the manipulation of indices. This mechanism is initiated only after reading the first c, as can be seen in Figure 2. The indices ensure that the number of d's equals the number of c's, and only due to the first mechanism this also ensures that the number of a's equals the number of b's.

That the second mechanism is only initiated after reading the first c is a consequence of the type of automaton defined in Section 3, which does not allow manipulation of indices while the stack grows. As a result, we cannot in general define automata that satisfy the *correct-prefix property*.

The correct-prefix property means that the algorithm does not read past the position of the first syntax error in the input. This position can be defined as the rightmost symbol of the shortest prefix of the input which cannot be extended to be a correct sentence in the language L.

In formal notation, this prefix for a given erroneous input $v \notin L$ is defined as the string wa, where $v = wax$, some x, such that $wy \in L$, for some y, but $waz \notin L$, for any z. The occurrence of a in v indicates the error position.

In our example, the automaton will detect that *aabbbcccdd* is incorrect only *after* it has read the complete input, when it finds no applicable transition for the configuration $(I[\,]\ P[p],\ \varepsilon)$. However, the error position can be defined by the fifth input position, where the third b already indicates that the numbers of a's and b's do not match. Therefore the automaton does not satisfy the correct-prefix property.

The bias towards manipulation of lists of indices while the stack shrinks is related to similar biases of other recognizers for tree-adjoining languages, such as embedded pushdown automata [12] and 2-SA [2]. Often a dual type of recognizer exists. In our case, a straightforward dual type results by simply reversing the constraints on allowable transitions. Strictly speaking, this means that transitions can be of one of the following forms:

- $X[\circ\circ]\ Z[\,] \overset{z}{\hookrightarrow} X[\circ\circ]$;
- $Z[\circ\circ\,\eta] \overset{z}{\hookrightarrow} X[\circ\circ\,\eta']$;
- $Z[\circ\circ\,\eta] \overset{z}{\hookrightarrow} Y[\,]\ X[\circ\circ\,\eta']$; or
- $Z[\circ\circ\,\eta] \overset{z}{\hookrightarrow} Y[\circ\circ\,\eta']\ X[\,]$,

An example of such an automaton is obtained by reversing the transitions of the automaton from the running example and making F the initial stack symbol and I the final stack symbol. This automaton accepts the language $\{d^n c^n b^n a^n \mid n > 0\}$ and satisfies the correct-prefix property.

We conjecture that a dual form of the tabulation algorithm from Section 4 exists that preserves the correct-prefix property for linear indexed automata of the dual type. Due to the unusual form of the two types of transition that increase the height of the stack, the items may need to be of a slightly different form. See a similar problem for context-free parsing in [8].

It is a subject of future research to establish how tabular algorithms such as that described in [9] can be reformulated in a modular way by this approach.

6 Acknowledgments

This research was carried out within the framework of the Priority Programme Language and Speech Technology (TST). The TST-Programme is sponsored by NWO (Dutch Organization for Scientific Research).

I would like to thank Tilman Becker and Eric de la Clergerie for interesting discussions.

References

[1] M.A. Alonso Pardo, E. de la Clergerie, and M. Vilares Ferro. Automata-based parsing in dynamic programming for Linear Indexed Grammars. In A.S. Narin'yani, editor, *Computational Linguistics and its Applications, proceedings*, pages 22–27, Moscow, Russia, June 1997.

[2] T. Becker. A new automaton model for TAGs: 2-SA. *Computational Intelligence*, 10(4):422–430, 1994.

[3] S. Billot and B. Lang. The structure of shared forests in ambiguous parsing. In *27th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference*, pages 143–151, Vancouver, British Columbia, Canada, June 1989.

[4] B. Lang. Deterministic techniques for efficient non-deterministic parsers. In *Automata, Languages and Programming, 2nd Colloquium*, Lecture Notes in Computer Science, volume 14, pages 255–269, Saarbrücken, 1974. Springer-Verlag.

[5] B. Lang. Complete evaluation of Horn clauses: An automata theoretic approach. Rapport de Recherche 913, Institut National de Recherche en Informatique et en Automatique, Rocquencourt, France, November 1988.

[6] B. Lang. The systematic construction of Earley parsers: Application to the production of $\mathcal{O}(n^6)$ Earley parsers for tree adjoining grammars. Unpublished paper, December 1988.

[7] B. Lang. Recognition can be harder than parsing. *Computational Intelligence*, 10(4):486–494, 1994.

[8] M.J. Nederhof. Reversible pushdown automata and bidirectional parsing. In J. Dassow, G. Rozenberg, and A. Salomaa, editors, *Developments in Language Theory II*, pages 472–481. World Scientific, Singapore, 1996.

[9] M.J. Nederhof. Solving the correct-prefix property for TAGs. In T. Becker and H.-U. Krieger, editors, *Proceedings of the Fifth Meeting on Mathematics of Language*, pages 124–130. Deutsches Forschungszentrum für Künstliche Intelligenz GmbH, August 1997. Document D-97-02, Accepted for publication in CL journal.

[10] M.J. Nederhof. An alternative LR algorithm for TAGs. Submitted for publication, 1998.

[11] F.C.N. Pereira and D.H.D. Warren. Definite clause grammars for language analysis—a survey of the formalism and a comparison with the augmented transition networks. *Artificial Intelligence*, 13:231–278, 1980.

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

[13] S. Sippu and E. Soisalon-Soininen. *Parsing Theory, Vol. II: LR(k) and LL(k) Parsing*, EATCS Monographs on Theoretical Computer Science, volume 20. Springer-Verlag, 1990.

[14] M. Tomita. *Efficient Parsing for Natural Language.* Kluwer Academic Publishers, 1986.

[15] K. Vijay-Shanker and D.J. Weir. Parsing some constrained grammar formalisms. *Computational Linguistics*, 19(4):591–636, 1993.

[16] K. Vijay-Shanker and D.J. Weir. The use of shared forests in tree adjoining grammar parsing. In *Sixth Conference of the European Chapter of the Association for Computational Linguistics, Proceedings of the Conference*, pages 384–393, Utrecht, The Netherlands, April 1993.

[17] K. Vijay-Shanker and D.J. Weir. The equivalence of four extensions of context-free grammars. *Mathematical Systems Theory*, 27:511–546, 1994.