The Systematic Construction of Earley Parsers: Application to the Production of $O(\mathrm{n}^6)$ Earley Parsers for Tree Adjoining Grammars

Extended Abstract

Bernard Lang
INRIA
B.P. 105, 78153 Le Chesnay, France
e-mail: lang@inria.inria.fr

Abstract

Using general results on dynamic programming techniques for the evaluation of definite clause programs, we systematically derive an $O(\mathtt{n}^6)$ Earley algorithm for Tree Adjoining Grammars. Though the algorithm produced is original and interesting in its own right, *the main contribution of this paper is the collection of independent techniques used to produce it.* In particular we show how general results on dynamic programming execution of compiled DC programs can be used to organize the parsing in a strictly left-to-right discipline, even in the presence of discontinuous structures with interleaved constituents. *The same techniques can be used to produce parsers with different recognition strategies, e.g. bottom-up, top-down, or predictive bottom-up as Earley's.* They may also be applied to unification based grammatical formalisms, and to the construction of robust parsers (e.g. island parsing).

Key Words: Parsing, Definite Clauses, Logic Programming, Push-Down Automata, Non-Determinism, Dynamic Programming, Earley Parsing, Tree Adjoining Grammars.

Contents

1 Introduction

The work presented here is part of a larger effort to develop a systematic technology for building Earley-like evaluators. A first step in this direction was a generalization of Earley's algorithm to a general Earley like construction for *Push-Down Automata (PDA)* presented in [Lan-74] that subsumes all published left-to-right variants of Earley's algorithm. The interest of this result was that the PDA could be constructed using any of a large number of techniques developed for the compiler technology [AhoU-72] and also in other research areas. The choice of the technique corresponds to what has been called since the choice of a parsing schema [Kay-80]. This idea was successfully used later by Tomita who applied it to LR(1) based parsers [Tom-87] and also sketched its application to specific PDA based parsers used in the linguistics community [Tom-88].

More recently, we have extended this approach to general *Definite Clause (DC)* programs, by introducing a new operational device, the *Logical PDA*. A DC program or grammar can be compiled into a LPDA according to some computation schema (e.g. predictive bottom-up corresponds to Earley deduction), and then the LPDA may be interpreted according to a general dynamic programming construction à la Earley. The data and control structures used for such interpretations are particularly simple, and lend themselves rather well to easy complexity analysis for finite interpretations [Lan-88b], which is often the case in parsing techniques.

Since many systematic syntactic formalisms used in computational linguistics can be expressed with DC programs (possibly on non-Herbrand interpretations such as feature structures [Kay-84, KapB-82, Shi-84, AïtN-86]), we believe that the Earley construction for LPDAs could be the basis for the development of Earley parsers for these formalisms. In this paper, we attempt to support this view by showing how an efficient Earley parser can be developed for the *Tree Adjoining Grammar (TAG)* formalism. The choice of TAGs was motivated by Schabes and Joshi's report of an $O(\mathtt{n}^9)$ algorithm [SchJ-88] and their statement that finding such an algorithm "was a difficult task". When the work presented here was conducted, the author knew the existence of that algorithm and of the $O(\mathtt{n}^6)$ CYK algorithm [Hay-62, Kas-65, You-66, AhoHU-68] developed earlier by Vijay-Shanker and Joshi [VijJ-85]. However, material circumstances prevented access to the actual descriptions of these algorithms. These details are mentionned to stress that no foreknowledge of the algorithmic aspects of the result helped us in this work, but only the systematic use of general techniques, which are the real topic of this paper. These techniques are rather tedious to present (they should be mechanized), but

are straightforward to apply once the proper theoretical framework has been set.

The strategy followed in the construction of our TAG Earley parser is the following: translation of the TAG into a DC program, construction of a corresponding predictive bottom-up LPDA, modification of the general dynamic programming evaluation of this LPDA to obey the left-to-right parsing discipline despite the presence of interleaved constituents (this modification is based on *general properties* of LPDAs).

Though similar constructions could perhaps be developed for Earley deduction, we believe a fully formal analysis would be more complex (because of a lesser decomposition into independent steps) and also less general (our techniques would equally apply to, for example, pure top-down or pure bottom-up schemas, or to non-left-to-right parsers). *It is shown in [Lan-88c] that the push-down structure plays an essential role in these algorithms.*

2 Tree Adjoining Grammars and DC Programs

A *Tree Adjoining Grammar (TAG)* is a 5-tuple (V_N, V_T, S, I, A) where V_N and V_T are sets of respectively *non-terminal* and *terminal* symbols, I and A are sets of respectively *initial* and *auxiliary* elementary trees. Except as specified below, the leaf (resp. non-leaf) nodes of elementary trees are labelled with terminal symbols or the empty string (resp. with non-terminal symbols). The root of an initial tree is labelled with the distinguished non-terminal S. Each auxiliary tree has one distinguished leaf node called its *foot* which is labelled with the same non-terminal as the root of the tree. The path from root to foot is called the *spine* of the auxiliary tree.

Given a tree $\mathcal{T}$ containing a non-leaf node X labelled with a non-terminal N, the *adjonction* of an auxiliary tree $\mathcal{A}$ with its root also labelled by N is the following operation: *excise* the subtree $\mathcal{T}'$ of $\mathcal{T}$ occurring in X, replace it with a copy of the auxiliary tree $\mathcal{A}$, and graft the excised subtree $\mathcal{T}'$ to the foot of this copy of $\mathcal{A}$. We shall call (the frontier of) the excised subtree $\mathcal{T}'$ the *footer* of the adjoined auxiliary tree $\mathcal{A}$.

A tree is generated by the TAG iff it can be obtained by a succession of adjonctions of auxiliary trees to some initial tree. The language generated by the TAG is the set of terminal strings that are the frontiers of trees that the TAG can generate.

In order to parse a sentence w.r.t. a TAG, we shall first use a construction similar to the transformation of CF grammars into DC programs presented by Pereira and Warren in [PerW-80].

The basic idea in the CF case is to consider every syntactic category as a predicate over substrings of the analysed sentences. A substring is represented by two indexes corresponding to its starting and ending positions. These two indexes are the arguments of the predicates. This gives for example a translation of the rule " `sentence` $\rightarrow$ `np vp` " into the clause "*sentence*(x, y) `:-` $np(x, z)$ $vp(z, y)$". This clause means that the substring from x to y is a *sentence* if the substring from x to z is a np and the substring from z to y is a vp.

In the CF case, there is one predicate in the DC program for every non-terminal of the CF grammar. In the case of TAGs, we define a predicate for each node in the initial and auxiliary trees. There is also a predicate for each non-terminal symbol of the translated TAG.

We use the (possibly subscripted) letters X, N and T to represent respectively node predicates, non-terminal predicates and terminal predicates. These letters are also used to denote the tree nodes, non-terminal symbols and terminal symbols these predicates correspond to.

Input index variables are noted with the letters x, y, s and e. More specifically the letters s and e are used, in spine-node predicates and non-terminal predicates, to denote the **s**tarting and **e**nding points of the footer of the currently recognized auxiliary tree.

In the expansion rules below, the n_{X} children of a node X are denoted by X_i for i in $[1..n_{\mathrm{X}}]$. For a spine node X, we denote by d_{X} the index of its unique son $\mathrm{X}_{d_{\mathrm{X}}}$ that is also on the spine.

The rules constructed mimic node by node the production of a tree by TAG derivation:

Initialization , for the root X of every initial tree:

The final value returned for the variable y must be the length `n` of the input sentence.

`sentence`(y) `:-` $\mathrm{X}(0, \mathrm{y})$

Normal expansion of a non-spine node X into its constituents:

$\mathrm{X}(\mathrm{x}_0, \mathrm{x}_{n_{\mathrm{X}}})$ `:-` $\mathrm{X}_1(\mathrm{x}_0, \mathrm{x}_1)$ $\ldots$ $\mathrm{X}_{n_{\mathrm{X}}}(\mathrm{x}_{n_{\mathrm{X}}-1}, \mathrm{x}_{n_{\mathrm{X}}})$

Terminal recognition , for each leaf-node X with terminal label T:

$\mathrm{X}(\mathrm{x}, \mathrm{y})$ `:-` `scan`$(\mathrm{x}, \mathrm{y}, \mathrm{T})$

Normal Adjunction Decision for each node X with non-terminal label N:

$\mathrm{X}(\mathrm{x}, \mathrm{y})$ `:-` $\mathrm{N}(\mathrm{x}, \mathrm{y}, \mathrm{s}, \mathrm{e})$ $\mathrm{X}(\mathrm{s}, \mathrm{e})$

Adjunction , for every N-labelled root node X of an auxiliary tree:

$\mathrm{N}(\mathrm{x}, \mathrm{y}, \mathrm{s}, \mathrm{e})$ `:-` $\mathrm{X}(\mathrm{x}, \mathrm{y}, \mathrm{s}, \mathrm{e})$

Spine expansion (the extra variables s and e are the footer's end points):

$\mathrm{X}(\mathrm{x}_0, \mathrm{x}_{n_\mathrm{X}}, \mathrm{s}, \mathrm{e})$:- $\mathrm{X}_1(\mathrm{x}_0, \mathrm{x}_1)$ ... $\mathrm{X}_{d_\mathrm{X}}(\mathrm{x}_{d_\mathrm{X}-1}, \mathrm{x}_{d_\mathrm{X}}, \mathrm{s}, \mathrm{e})$ ... $\mathrm{X}_{n_\mathrm{X}}(\mathrm{x}_{n_\mathrm{X}-1}, \mathrm{x}_{n_\mathrm{X}})$

Spine Adjunction Decision (s' and e' correspond to the footer of the dominating tree):

$\mathrm{X}(\mathrm{x}, \mathrm{y}, \mathrm{s}', \mathrm{e}')$:- $\mathrm{N}(\mathrm{x}, \mathrm{y}, \mathrm{s}, \mathrm{e})\ \mathrm{X}(\mathrm{s}, \mathrm{e}, \mathrm{s}', \mathrm{e}')$

Foot Skipping for each foot node X, the end-points of the foot must be those of the footer:

$\mathrm{X}(\mathrm{s}, \mathrm{e}, \mathrm{s}, \mathrm{e})$:-

Note that the footer is actually recognized in (normal or spine) adjunction decision clauses.

Finally, for every word with position i in the input sentence, and belonging to the lexical (i.e. terminal) category T, we add the unit clause:

$\texttt{scan}(i-1, i, \mathrm{T})$:- .

3 Logical push-down automata & Dynamic programming

The standard computational device for CF parsing is the push-down automaton. Most existing parsers are built on that model. This is clearly true for deterministic parsing techniques [AhoU-72], but also for backtracking techniques [GriP-65], and for dynamic programming à la Earley as shown by the author in [Lan-74, Lan-88a].

The *Logical Push-Down Automaton (LPDA)* is the natural extension of that operational engine to DC programs and grammars. As in the CF case, it allows independent choice of the parsing schema (e.g. top-down, bottom-up, predictive, ... [GriP-65, Kay-80]) which is embodied in the construction of the LPDA, and of the evaluation strategy (e.g. backtrack, breadth-first or dynamic programming). Furthermore, the LPDA formalism leads to very simple structures in the case of dynamic programming interpretations [Lan-88b, Lan-88c], which are easier to analyse, prove, and optimize than the corresponding direct constructions on DC programs [PerW-83, Por-86, TamS-86, Vie-88], while remaining independant of the computation (parsing) schema unlike the direct constructions.

A LPDA is essentially a PDA that stores logical atoms (i.e. predicates applied to arguments) and substitutions on its stack, instead of simple symbols. The symbols of the standard CF PDA stack are (approximately, for this abstract) predicates with no arguments. A technical point is that we consider PDAs without "finite state" control: this is possible without loss

of generality by having pop transitions that replace the top two atoms by only one (this is standard in LR(k) PDA parsers[AhoU-72]).

Formally a LPDA $\mathcal{A}$ is a 6-tuple: $\mathcal{A} = (\mathbf{X}, \mathbf{F}, \mathbf{\Delta}, \overset{\circ}{\$}, \$_f, \mathbf{\Theta})$

where $\mathbf{X}$ is a set of variables, $\mathbf{F}$ is a set of functions and constants symbols, $\mathbf{\Delta}$ is a set of stack predicate symbols, $\overset{\circ}{\$}$ and $\$_f$ are respectively the initial and final stack predicates, and $\mathbf{\Theta}$ is a finite set of *transitions* having one of the following three forms:

horizontal transitions:	$B \mapsto C$	– – replace B by C on top of stack
push transitions:	$B \mapsto CB$	– – push C on top of former stack top B
pop transitions:	$BD \mapsto C$	– – replace BD by C on top of stack

where B, C and D are $\mathbf{\Delta}$-atoms, i.e. atoms built with $\mathbf{\Delta}$, $\mathbf{F}$ and $\mathbf{X}$.

Intuitively (and approximately) a pop transition $BD \mapsto C$ is applicable to a stack configuration with atoms A and A′ on top, iff there is a substitution s such that $Bs = As$ and $Ds = A's$. Then A and A′ are removed from the stack and replaced by Cs, i.e. the atom C to which s has been applied. Things are similar for other kinds of transitions. Of course a LPDA is usually non-deterministic w.r.t. the choice of the applicable transition.

In the case of dynamic programming interpretations, all possible computation paths are explored, with as much sub-computation sharing as possible. The algorithm proceeds by building a collection of *items* (analogous to those of Earley's algorithm) which are pairs of atoms. An item <A A′> represents a stack fragment of two consecutive atoms [Lan-74, Lan-88a]. If another item <A′ A″> was also created, this means that the sequence of atoms AA′A″ is to be found in some possible stack configuration, and so on (*up to the use of substitutions, not discussed here*). The computation is initialized with an initial item $\overset{\circ}{U} = < \overset{\circ}{\$} \dashv >$. New items are produced by applying the LPDA transitions to existing items, until no new application is possible (an application may often produce an already existing item). The computation terminates under similar conditions as specialized algorithms [PerW-83, TamS-86, Vie-88]. If successful, the computation produces one or several *final items* of the form $<\$_f \overset{\circ}{\$} >$, where the arguments of $\$_f$ are an answer substitution of the initial DC program. In a parsing context, one is usually interested in obtaining parse-trees rather than "answer substitutions". A parse tree is here a proof tree corresponding to the original DC program. Such proof trees may be obtained by the same techniques that are used in the case of CF parsing [Lan-74, BilL-88, Bil-88], and that actually interpret the items and their relations as a shared parse forest structure.

Substitutions are applied to items as follows (we give as example the most complex case): a pop transition $BD \mapsto C$ is applicable to a pair of items <A A′> and <E E′>, iff there is

a unifier s of <A A′> and <B D>, and a unifier s' of A′s and E. This produces the item <Css' E′s'>.

Given a DC program, *many different computation schemata may be used to build a corresponding LPDA* [Lan-88c]. Since we are interested in building Earley parsers, i.e. bottom-up parsers with a predictive top-down component, we use a corresponding LPDA construction, which gives computations that are similar to the Earley deduction algorithm [PerW-83].

The $\mathbf{n}+1$ clauses of the DC program are defined as γ_k : $\quad \mathrm{A}_{k,0}$:- $\mathrm{A}_{k,1}, \ldots, \mathrm{A}_{k,n_k}$ with γ_0 defining the goal. For each clause γ_k we note t_k the vector of variables occurring in the clause. We also define n_k+1 new predicates $\bigtriangledown_{k,i}$ for i in $[1..n_k]$, taking t_k as arguments, and corresponding to positions between the literals of the clause body.

For each predicate P of the DC program, we define 2 stack predicates P′ and P″. The predicate P′ is used to state a P-query, i.e. require recognition of a P-constituent, and P″ is used to state that it has been answered in a way described by its arguments.

For each atom $\mathrm{A}_{k,i}$ occurring in a clause γ_k, the notations $\mathrm{A}'_{k,i}$ and $\mathrm{A}''_{k,i}$ denote the same atom where the predicate symbol, say P, has been replaced respectively by P′ and P″.

We define in the LPDA the transitions:

1. $\overset{\circ}{\$} \mapsto \mathrm{A}'_{0,0}\ \overset{\circ}{\$}$
2. $\mathrm{A}'_{k,0} \mapsto \bigtriangledown_{k,0}(\mathrm{t}_k)\,\mathrm{A}'_{k,0}$ — for every clause γ_k
3. $\bigtriangledown_{k,i}(\mathrm{t}_k) \mapsto \mathrm{A}'_{k,i+1}\,\bigtriangledown_{k,i}(\mathrm{t}_k)$ — for every clause γ_k and for every position i in its body: $0 \leq i < n_k$
4. $\bigtriangledown_{k,n_k}(\mathrm{t}_k)\,\mathrm{A}'_{k,0} \mapsto \mathrm{A}''_{k,0}$ — for every clause γ_k
5. $\mathrm{A}''_{k,i+1}\,\bigtriangledown_{k,i}(\mathrm{t}_k) \mapsto \bigtriangledown_{k,i+1}(\mathrm{t}_k)$ — for every clause γ_k and for every position i in its body: $0 \leq i < n_k$

The final predicate of the LPDA is the stack predicate $\mathrm{A}''_{0,0}$.

The following is an informal explaination of the transitions defined above:

1. State the initial query $\mathrm{A}_{0,0}$ in the stack, represented by $\mathrm{A}'_{0,0}$.
2. Choose clause γ_k to prove the goal sitting on top of the stack: the head $\mathrm{A}_{k,0}$ of the clause γ_k must subsume it.

3. Then subquery successively for each atom in the body of γ_k. The position predicate $\bigtriangledown_{k,i}$ indicates that the subqueries for the first i atoms have been consistently proved.

4. When a common answer substitution θ has been found for all atoms in the body of γ_k, then the head instance $\mathrm{A}_{k,0}\theta$ is proved. It is represented here by $\mathrm{A}''_{k,0}$.

5. Having proved an instance of the i-th atom in the body of γ_k consistently with the proofs of previous ones, move to the next body atom to be subqueried by transition 3, unless there is none left in which case an answer to a previous subquery is produced by transition 4.

4 Building the Earley parser

The DC program built from the TAG as described in section 2 can then be translated into a LPDA by means of the construction given at the end of section 3, *which produces a predictive bottom-up parser*. The resulting LPDA is described in appendix A.

To comply fully with the constraints stated in [SchJ-88], we must adapt the dynamic programming interpretation of the LPDA so as to obey the *remaining* characteristics of Earley's algorithm:

1. left to right scanning of the input sentence,

2. construction for each input position i (between words) of a set of items $\mathcal{S}_i$,

3. initialization of each set $\mathcal{S}_i$ by scanning transitions applied to items in the previous set $\mathcal{S}_{i-1}$,

4. completion of the item set $\mathcal{S}_i$ before any transition is applied to items already put in the following item set $\mathcal{S}_{i+1}$ by scanning transitions. Hence, one should not add items to item set $\mathcal{S}_i$ after a later item set has been worked upon.

The items built by the dynamic programming interpretation of the LPDA are pairs of atoms built with the stack predicates of the LPDA by application of its transitions. These atoms have some arguments bound to input position indexes, while other may be left as free-variables (most of the latter arguments are actually useless in our case, but were kept to simplify the description of the LPDA construction). The atoms with a X or N predicate correspond to syntactical constituents, to be recognized when the predicate is primed once (e.g. X'), or already recognized when it is primed twice (e.g. X''). *The first two arguments*

of these predicates correspond to the end points indexes of the corresponding part of the parsed sentence. Hence a X$''$ atom must have its first two arguments bound to actual indexes, while in left-to-right parsing a X$'$ atom must have its first argument bound (to know where to start recognition) but not its second argument.

The other arguments also correspond to indexes in the parsed sentence, respecting always the following order:

- for indexed variables $\mathrm{x}_i : i \leq j$ implies $\mathrm{x}_i \leq \mathrm{x}_j$
- for others variables: $\mathrm{x} \leq \mathrm{s} \leq \mathrm{s}' \leq \mathrm{e}' \leq \mathrm{e} \leq \mathrm{y}$

To obey the above constraints, we organize the items into sets indexed by input position. We initialize the first item set $\mathcal{S}_0$ with the initial item $\overset{\circ}{U} = <\overset{\circ}{\$} \dashv >$, and we complete $\mathcal{S}_0$ by applying to its items all possible transitions, except the scanning one $\mathtt{scan}'(i-1, i, \mathrm{T}) \mapsto \mathtt{scan}''(i-1, i, \mathrm{T})$ (and 3 others discussed below). Thus $\mathcal{S}_0$ contains all items that may be computed without scanning any input. Then the scanning transition is applied to all possible items in $\mathcal{S}_0$ and the items produced are included in the next item set $\mathcal{S}_1$ which is thus initialized. The set $\mathcal{S}_1$ is completed as $\mathcal{S}_0$ was, then $\mathcal{S}_2$ is initialized with the scanning transition, and so on.

The parse terminates successfully, on a sentence of length $\mathtt{n}$, if the item set $\mathcal{S}_\mathtt{n}$ contains final items like $<\mathrm{X}''(0, \mathtt{n})\ \overset{\circ}{\$} >$, where X is the root of some initial tree of the TAG.

If the LPDA had been obtained by compiling a DC program produced from a CF grammar as in [PerW-80], then everything would work fine, essentially mimicking the original algorithm of Earley (the $\bigtriangledown$ predicates are the equivalent of Earley's dotted rules [Ear-70]).

Unfortunately, there are problems with 3 types of non-scanning transitions:

4.1 Skipping a constituent of another structure

The first case concerns the foot skipping transitions $\mathrm{X}'(\mathrm{s}, \mathrm{e}, \mathrm{s}, \mathrm{e}) \mapsto \mathrm{X}''(\mathrm{s}, \mathrm{e}, \mathrm{s}, \mathrm{e})$. The above rule about the binding of the first two arguments of the X$'$ and X$''$ predicates cannot be obeyed since these transitions bring no new binding information. This is due to the fact that the size of the footer cannot be known yet, since in a left-to-right parsing discipline, this transition is reached before scanning the footer, and furthermore the footer is to be scanned by another remotely connected part of the LPDA. However, the value of the second argument e is needed to be able to pursue the *left-to-right* parsing of the footed auxiliary tree with the proper input (i.e. item set) index. The only solution is to guess the value(s) of this argument. This is formally permitted by the following property of LPDAs:

Theorem 1: *The results computed by a LPDA are not changed if a transition is replaced by a complete set of instances of itself.*

A set S of instances of a transition τ is complete w.r.t. τ when any ground (i.e. fully instantiated) instance of τ is an instance of a transition in S.

Thus we solve our problem by replacing foot skipping transitions by instances of these transitions $\mathrm{X}'(\mathrm{s}, i, \mathrm{s}, i) \mapsto \mathrm{X}''(\mathrm{s}, i, \mathrm{s}, i)$ for any value of the index i, and placing the items they produce in the corresponding item sets $\mathcal{S}_i$.

Then there still is the problem that such a skipping (pseudo-scanning) transition creates items in $\mathcal{S}_i$ when applied to an item $\mathcal{S}_\mathrm{s}$ with $\mathrm{s} + 1 < i$. Thus we lose the incrementality of item sets creation. This difficulty is solved by using instead only two transitions that produce incrementally the same computational result (this is proved by a trivial induction). Hence we replace each foot skipping transition for a foot node X by the following two transitions (note that we start whith $i = \mathrm{s}$ rather than $i = 0$ because the above foot skipping transitions are useless for $i < \mathrm{s}$):

1. $\mathrm{X}'(\mathrm{s}, \mathrm{e}, \mathrm{s}, \mathrm{e}) \mapsto \mathrm{X}''(\mathrm{s}, \mathrm{s}, \mathrm{s}, \mathrm{s})$
2. $\mathrm{X}''(\mathrm{s}, \mathrm{e}, \mathrm{s}, \mathrm{e}) \mapsto \mathrm{X}''(\mathrm{s}, \mathrm{e}+1, \mathrm{s}, \mathrm{e}+1)$

The second one of these transition is to be considered as scanning, i.e. the items it produces must be used to initialize the next set of items.

4.2 Recognizing a constituent interleaved in another structure

The second and third case concern the 4^{th} transitions for normal and spine adjunction decision: $\nabla'_\mathrm{X}(\mathrm{x}, \mathrm{y}, \mathrm{s}, \mathrm{e}) \mapsto \mathrm{X}'(\mathrm{s}, \mathrm{e})\ \nabla'_\mathrm{X}(\mathrm{x}, \mathrm{y}, \mathrm{s}, \mathrm{e})$ and $\nabla'_\mathrm{X}(\mathrm{x}, \mathrm{y}, \mathrm{s}, \mathrm{e}, \mathrm{s}', \mathrm{e}') \mapsto \mathrm{X}'(\mathrm{s}, \mathrm{e}, \mathrm{s}', \mathrm{e}')\ \nabla'_\mathrm{X}(\mathrm{x}, \mathrm{y}, \mathrm{s}, \mathrm{e}, \mathrm{s}', \mathrm{e}')$. Here the situation is reversed. We take as example the case of the 4^{th} transition of normal adjunction decision.

This transition pushes the atom $\mathrm{X}'(\mathrm{s}, \mathrm{e})$ on top of the stack, thus requesting the recognition of the footer of an adjunction. However, at this point the corresponding adjoined tree has already been recognized, and thus the scanning position is already beyond the part of the sentence corresponding to the footer.

The solution is to have guessed earlier, at the proper time, that recognition of this footer must begin, and to have parsed it in step with the left-to-right scanning of the sentence. However, all we can do is to attempt to begin this recognition at each position index in the parsed sentence.

Again, this is formally permitted by the above theorem 1, combined with another one. First we replace the transition by a set of its instance where the beginning s of the footer has been instanciated to all possible values of the input index. Then we apply a second theorem that states:

Theorem 2: *A push transition* $\mathrm{B} \mapsto \mathrm{CB}$ *may be eliminated without changing the results of dynamic programming computations of a LPDA if this computation is initialized with an additional item* $<\mathrm{C\,B}>$.

The application to our set of transition instances amounts to an initialization of the computation with items of the form: $<\mathrm{X}'(i,\mathrm{e})\ \nabla'_{\mathrm{X}}(\mathrm{x},\mathrm{y},i,\mathrm{e})>$ for each non-leaf node X *which can be the locus of an adjunction*, and for each possible input index i. In practice, such an item can only be used to recognize a footer starting at the $(i+1)^{\mathrm{st}}$ lexical element. Hence, the items can be created progressively along with item sets: each item set $\mathcal{S}_i$ is additionally initialized with an item $<\mathrm{X}'(i,\mathrm{e})\ \nabla'_{\mathrm{X}}(\mathrm{x},\mathrm{y},i,\mathrm{e})>$ for each non-leaf node X. The same is done for spine nodes, according to the structure of the corresponding push transition.

After these transformations, all the items may be produced from left to right in successive item sets. *Some optimizations are possible to reduce the number of items*, e.g. by coordinating to some extent foot skipping and foot recognition.

4.3 Correctness and Complexity

Most of the constructions used in this paper are based on general results that need not be proved again. They can all be mechanized (we have a running implementation for some [VilZ-88]). The only proofs needed specifically here are for the correctness of the TAG to DC program construction (section 2, tedious but not hard) and for the actual left-to-right behavior of the final algorithm (a fairly simple induction), but not for its correctness or termination.

As shown by Lang in [Lan-88b], an upperbound for the time complexity of dynamic programming interpretation of a LPDA is given by $O(\mathtt{n}^{\theta})$ where $\mathtt{n}$ is the length of the input sentence, and θ is the maximum number of times that any given transition can be applied.

This number θ is precisely the number of independently bound variables that may be involved in the same transition. For transitions applied to only one item (i.e. push and horizontal transitions), this is precisely the number of independently bound variables in that item.

For transitions applied to two items (i.e. pop transitions), this number is the number of independently bound variables occuring in both items. Note that the fact that a pop transition

is applicable introduces new dependencies, since the bindings of the second atom of the first item must be compatible with the bindings of the first atom of the second item.

Space complexity may be evaluated on similar grounds. It is $O(\mathtt{n}^{\sigma})$ where σ is the maximum number of independently bound variables that can occur in one item.

Examination of the LPDA transition shows that both time and space complexities are $O(\mathtt{n}^{6})$ Of course, as is usual for Earley algorithm, the performance may be much better in specific cases.

5 Conclusion

The objective of this paper was **not** to produce an Earley parser for TAGs, but **rather** to show how very general techniques can be used to produce such a parser, and others. It may be that more ingenuity would have produced a parser with lower complexity bounds in the specific case of TAGs, but this does not limit the value of the techniques presented.

Naturally, these techniques are not limited to left-to-right parsing or to simple grammatical formalisms. The definite clause formalism can accomodate extra arguments representing more contextual or semantical information. A variety of parsing schema may be used to compile DC programs into LPDAs. The dynamic programming interpretation of LPDA is itself adaptable to many variations (extending Sheil's results [She-76] and including in particular the robust processing of noisy input (e.g. parsing in word lattices [Nak-87], island parsing [WarHSC-88], or parsing incomplete sentences [Lan-88a]).

The benefit of using LPDAs comes from the fact that it is a very explicit and well defined operational formalism, which can accomodate any parsing schema, and can also produce parse forests in a well formalized way [BilL-88].

References

[AhoHU-68] Aho, A.V.; Hopcroft, J.E.; and Ullman, J.D. 1968 Time and Tape Complexity of Pushdown Automaton Languages. *Information and Control* 13: 186-206.

[AhoU-72] Aho, A.V.; and Ullman, J.D. 1972 *The Theory of Parsing, Translation and Compiling*. Prentice-Hall, Englewood Cliffs, New Jersey.

[AïtN-86] Aït-Kaci, H.; and Nasr, R. 1986 Login: A logic Programming Language with Built-in Inheritance. *Journal of Logic Programming* 3: 185-215.

[Bil-88] Billot, S. 1988 *Analyseurs Syntaxiques et Non-Déterminisme*. Thèse de Doctorat, Université d'Orléans la Source, Orléans (France).

[BilL-88] Billot, S.; and Lang, B. 1988 *The structure of Shared Forests in Ambiguous Parsing*. Submitted to this conference.

[Ear-70] Earley, J. 1970 An Efficient Context-Free Parsing Algorithm. *Communications ACM* 13(2): 94-102.

[GriP-65] Griffiths, I.; and Petrick, S. 1965 On the Relative Efficiencies of Context-Free Grammar Recognizers. *Communications ACM* 8(5): 289-300.

[Hay-62] Hays, D.G. 1962 Automatic Language-Data Processing. In *Computer Applications in the Behavioral Sciences*, (H. Borko ed.), Prentice-Hall, pp. 394-423.

[KapB-82] Kaplan, R.M.; and Bresnan J. 1982 Lexical-Functional Grammar: A Formal System for Grammatical Representation. In *The Mental Representation of Grammatical Relations*, ch. 4, pp. 173-281, J. Bresnan (ed.), The MIT Press.

[Kas-65] Kasami, J. 1965 *An Efficient Recognition and Syntax Analysis Algorithm for Context-Free Languages*. Report of Univ. of Hawaii, also AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford (Massachusetts), also 1966, University of Illinois Coordinated Science Lab. Report, No. R-257.

[Kay-80] Kay, M. 1980 Algorithm Schemata and Data Structures in Syntactic Processing. *Proceedings of the Nobel Symposium on Text Processing*, Gothenburg.
Also: Tech. Report CSL-80-12, Xerox Palo-Alto Research Center.

[Kay-84] Kay, M. 1984 Unification in Grammar. *Proc. of the First Internat. Workshop on Natural Language Understanding and Logic Programming*: 233-240, Rennes

(France), V. Dahl and P. Saint-Dizier (eds.), Elsevier Science Pub. (North-Holland), 1985.

[Lan-74] Lang, B. 1974 Deterministic Techniques for Efficient Non-deterministic Parsers. *Proc. of the* 2^{nd} *Colloquium on Automata, Languages and Programming*, J. Loeckx (ed.), Saarbrücken, Springer Lecture Notes in Computer Science 14: 255-269.
Also: Rapport de Recherche 72, IRIA-Laboria, Rocquencourt (France).

[Lan-88a] Lang, B. 1988 Parsing Incomplete Sentences. *Proc. of the* 12^{th} *Internat. Conf. on Computational Linguistics (COLING'88)* Vol. 1 :365-371, D. Vargha (ed.), Budapest (Hungary).

[Lan-88b] Lang, B. 1988 Datalog Automata. *Proc. of the* 3^{rd} *Internat. Conf. on Data and Knowledge Bases*, C. Beeri, J.W. Schmidt, U. Dayal (eds.), Morgan Kaufmann Pub., pp. 389-404, Jerusalem (Israel).

[Lan-88c] Lang, B. 1988 *Complete Evaluation of Horn Clauses: an Automata Theoretic Approach*. INRIA Research Report 913.

[Nak-87] Nakagawa, S. 1987 Spoken Sentence Recognition by Time-Synchronous Parsing Algorithm of Context-Free Grammar. *Proc. ICASSP 87*, Dallas (Texas), Vol. 2 : 829-832.

[PerW-80] Pereira, F.C.N.; and Warren, D.H.D. 1980 Definite Clause Grammars for Language Analysis — A Survey of the Formalism and a Comparison with Augmented Transition Networks. *Artificial Intelligence* 13: 231-278.

[PerW-83] Pereira, F.C.N.; and Warren, D.H.D. 1983 Parsing as Deduction. *Proceedings of the* 21^st *Annual Meeting of the Association for Computational Linguistics*: 137-144, Cambridge (Massachusetts).

[Por-86] Porter, H.H. 3^{rd} 1986 *Earley Deduction*. Tech. Report CS/E-86-002, Oregon Graduate Center, Beaverton (Oregon).

[SchJ-88] Schabes, Y.; and Joshi, A.K. 1988 Yves Aravind An Earley-type Parsing Algorithm for Tree Adjoining Grammars. *Proc. of the* 26^{th} *Annual Meeting of the Association for Computational Linguistics*: 258-269, Buffalo (New York).

Also: Technical Report MS-CIS-88-36, LINC LAB 113, Dept. of Computer and Information Science, University of Pennsylvania, 1988.

[She-76] Sheil, B.A. 1976 Observations on Context Free Parsing. in *Statistical Methods in Linguistics*: 71-109, Stockholm (Sweden), Proc. of Internat. Conf. on Computational Linguistics (COLING-76), Ottawa (Canada).
Also: Technical Report TR 12-76, Center for Research in Computing Technology, Aiken Computation Laboratory, Harvard Univ., Cambridge (Massachusetts).

[Shi-84] Shieber, S.M. 1984 The Design of a Computer Language for Linguistic Information. *Proc. of the* 10^{th} *Internat. Conf. on Computational Linguistics — COLING'84*: 362-366, Stanford (California).

[TamS-86] Tamaki, H.; and Sato, T. 1986 OLD Resolution with Tabulation. *Proc. of* $3^{r}d$ *Internat. Conf. on Logic Programming*, London (UK), Springer LNCS 225: 84-98.

[Tom-87] Tomita, M. 1987 An Efficient Augmented-Context-Free Parsing Algorithm. *Computational Linguistics* 13(1-2): 31-46.

[Tom-88] Tomita, M. 1988 Graph-structured Stack and Natural Language Parsing *Proc. of the* 26^{th} *Annual Meeting of the Association for Computational Linguistics* : 249-257, Buffalo (New York).

[Vie-88] Vieille, L. 1988 *Recursive Query Processing: The power of Logic.* Tech. Report TR-KB-17, European ComputerIndustry Research Center (ECRC), Munich (West Germany). To appear in Theoretical Computer Science.

[VijJ-85] Vijay-Shankar, K.; and Joshi, A.K. 1985 Some Computational Properties of Tree Adjoining Grammars. *Proceedings of the* $23^{r}d$ *Annual Meeting of the Association for Computational Linguistics*: 145-152, Chicago (Illinois).

[VilZ-88] Villemonte de la Clergerie, E.; and Zanchetta, A. 1988 *Evaluateur de Clauses de Horn.* Rapport de Stage d'Option, Ecole Polytechnique, Palaiseau (France).

[WarHSC-88] Ward, W.H.; Hauptmann, A.G.; Stern, R.M.; and Chanak, T. 1988 Parsing Spoken Phrases Despite Missing Words. In *Proceedings of the 1988 International*

Conference on Acoustics, Speech, and Signal Processing (ICASSP 88), Vol. 1: 275-278.

[You-66] Younger, D.H. 1967 Recognition and Parsing of Context-Free Languages in Time n^3. *Information and Control*, 10(2): 189-208.

A Translating a TAG into a simple LPDA

We assume that the tag has been translated into a DC program as described in section 2. We now translate this DC program into a LPDA following the construction given in section 3. We do not always follow exactly the standard construction, but make a few innocuous simplifications for better readability.

In particular, though the standard construction assumes that all clause variables are to be carried in the $\bigtriangledown_{\mathrm{X}}$ literals, this is not really needed systematically. The variables not occurring in the clause head need not be carried before their first appearance in a body literal, nor after they last appearance in such a literal. Thus, *for example*, in expansion transitions below, a variable x_i is used only in clause literals $\mathrm{X}_i(\mathrm{x}_{i-1}, \mathrm{x}_i)$ and $\mathrm{X}_{i+1}(\mathrm{x}_i, \mathrm{x}_{i+1})$. Hence in the LPDA transitions, this variable appears only in the corresponding literals, and is also carried in the intervening position literal $\bigtriangledown_{\mathrm{X},i}(\mathrm{x}_i, \ldots)$. Another simplification concerns clauses with a unique literal in their body, for which the position predicates $\bigtriangledown_{\mathrm{X}}$ are not used at all.

On the other hand we do not make some obvious simplifications, so as to keep the uniformity of the presentation of the construction, and of the later discussion. For example, this may result in duplicating uselessly a variable in some litterals.

Note that we follow the convention of the general Earley construction of section 3 concerning prime and double-prime on node and non-terminal predicates (X and N) when they are used as *query* (i.e. top-down) as in X' and N', or as *answer* (i.e. bottom-up) as in X'' and N''.

The primes and double-primes on $\bigtriangledown_{\mathrm{X}}$ predicates in adjunction-decision transitions do not play this role and are just meant as distinctive marks. We do not use indexes in this latter case to avoid ambiguity with position predicates in expansion transitions, since both adjunction-decision or expansion can be applied to the same node predicates X when node X is labelled with a non-terminal.

Initialization for the root X of every initial tree of the TAG:

$\overset{\circ}{\$} \mapsto \mathrm{X}'(0, \mathrm{y})\ \overset{\circ}{\$}$

and X'' is a final predicate. The parse is successful iff a final item $<\mathrm{X}''(0, n)\ \overset{\circ}{\$}>$ is produced.

Normal expansion for every non-leaf node X, $\forall i,\ 1 \leq i \leq n_{\mathrm{X}}$

1. $\mathrm{X}'(\mathrm{x}_0, \mathrm{x}_{n_{\mathrm{X}}}) \mapsto \bigtriangledown_{\mathrm{X},0}(\mathrm{x}_0, \mathrm{x}_0, \mathrm{x}_{n_{\mathrm{X}}})\ \mathrm{X}'(\mathrm{x}_0, \mathrm{x}_{n_{\mathrm{X}}})$
2. $\bigtriangledown_{\mathrm{X},i-1}(\mathrm{x}_{i-1}, \mathrm{x}_0, \mathrm{x}_{n_{\mathrm{X}}}) \mapsto \mathrm{X}'_i(\mathrm{x}_{i-1}, \mathrm{x}_i)\ \bigtriangledown_{\mathrm{X},i-1}(\mathrm{x}_{i-1}, \mathrm{x}_0, \mathrm{x}_{n_{\mathrm{X}}})$

3. $\mathrm{X}''_i(\mathrm{x}_{i-1}, \mathrm{x}_i)\ \nabla_{\mathrm{X},i-1}(\mathrm{x}_{i-1}, \mathrm{x}_0, \mathrm{x}_{n_\mathrm{X}}) \mapsto \nabla_{\mathrm{X},i}(\mathrm{x}_i, \mathrm{x}_0, \mathrm{x}_{n_\mathrm{X}})$
4. $\nabla_{\mathrm{X},n_\mathrm{X}}(\mathrm{x}_{n_\mathrm{X}}, \mathrm{x}_0, \mathrm{x}_{n_\mathrm{X}})\ \mathrm{X}'(\mathrm{x}_0, \mathrm{x}_{n_\mathrm{X}}) \mapsto \mathrm{X}''(\mathrm{x}_0, \mathrm{x}_{n_\mathrm{X}})$

Terminal recognition for every leaf node X with a terminal label T

1. $\mathrm{X}'(\mathrm{x}, \mathrm{y}) \mapsto \mathtt{scan}'(\mathrm{x}, \mathrm{y}, \mathrm{T})\ \mathrm{X}'(\mathrm{x}, \mathrm{y})$
2. $\mathtt{scan}''(\mathrm{x}, \mathrm{y}, \mathrm{T})\ \mathrm{X}'(\mathrm{x}, \mathrm{y}) \mapsto \mathrm{X}''(\mathrm{x}, \mathrm{y})$

Normal Adjunction Decision for every node X with non-terminal label N

1. $\mathrm{X}'(\mathrm{x}, \mathrm{y}) \mapsto \nabla_\mathrm{X}(\mathrm{x}, \mathrm{y})\ \mathrm{X}'(\mathrm{x}, \mathrm{y})$
2. $\nabla_\mathrm{X}(\mathrm{x}, \mathrm{y}) \mapsto \mathrm{N}'(\mathrm{x}, \mathrm{y}, \mathrm{s}, \mathrm{e})\ \nabla_\mathrm{X}(\mathrm{x}, \mathrm{y})$
3. $\mathrm{N}''(\mathrm{x}, \mathrm{y}, \mathrm{s}, \mathrm{e})\ \nabla_\mathrm{X}(\mathrm{x}, \mathrm{y}) \mapsto \nabla'_\mathrm{X}(\mathrm{x}, \mathrm{y}, \mathrm{s}, \mathrm{e})$
4. $\nabla'_\mathrm{X}(\mathrm{x}, \mathrm{y}, \mathrm{s}, \mathrm{e}) \mapsto \mathrm{X}'(\mathrm{s}, \mathrm{e})\ \nabla'_\mathrm{X}(\mathrm{x}, \mathrm{y}, \mathrm{s}, \mathrm{e})$
5. $\mathrm{X}''(\mathrm{s}, \mathrm{e})\ \nabla'_\mathrm{X}(\mathrm{x}, \mathrm{y}, \mathrm{s}, \mathrm{e}) \mapsto \nabla''_\mathrm{X}(\mathrm{x}, \mathrm{y})$
6. $\nabla''_\mathrm{X}(\mathrm{x}, \mathrm{y})\ \mathrm{X}'(\mathrm{x}, \mathrm{y}) \mapsto \mathrm{X}''(\mathrm{x}, \mathrm{y})$

Adjunction for every N labelled root node X of an auxiliary tree:

1. $\mathrm{N}'(\mathrm{x}, \mathrm{y}, \mathrm{s}, \mathrm{e}) \mapsto \mathrm{X}'(\mathrm{x}, \mathrm{y}, \mathrm{s}, \mathrm{e})$
2. $\mathrm{X}''(\mathrm{x}, \mathrm{y}, \mathrm{s}, \mathrm{e}) \mapsto \mathrm{N}''(\mathrm{x}, \mathrm{y}, \mathrm{s}, \mathrm{e})$

Spine expansion for every spine node X, $\quad \forall i,\ 1 \leq i \leq n_\mathrm{X}$ and $i \neq d_\mathrm{X}$

1. $\mathrm{X}'(\mathrm{x}_0, \mathrm{x}_{n_\mathrm{X}}, \mathrm{s}, \mathrm{e}) \mapsto \nabla_{\mathrm{X},0}(\mathrm{x}_0, \mathrm{x}_0, \mathrm{x}_{n_\mathrm{X}}, \mathrm{s}, \mathrm{e})\ \mathrm{X}'(\mathrm{x}_0, \mathrm{x}_{n_\mathrm{X}}, \mathrm{s}, \mathrm{e})$
2. $\nabla_{\mathrm{X},i-1}(\mathrm{x}_{i-1}, \mathrm{x}_0, \mathrm{x}_{n_\mathrm{X}}, \mathrm{s}, \mathrm{e}) \mapsto \mathrm{X}'_i(\mathrm{x}_{i-1}, \mathrm{x}_i)\ \nabla_{\mathrm{X},i-1}(\mathrm{x}_{i-1}, \mathrm{x}_0, \mathrm{x}_{n_\mathrm{X}}, \mathrm{s}, \mathrm{e})$
3. $\mathrm{X}''_i(\mathrm{x}_{i-1}, \mathrm{x}_i)\ \nabla_{\mathrm{X},i-1}(\mathrm{x}_{i-1}, \mathrm{x}_0, \mathrm{x}_{n_\mathrm{X}}, \mathrm{s}, \mathrm{e}) \mapsto \nabla_{\mathrm{X},i}(\mathrm{x}_i, \mathrm{x}_0, \mathrm{x}_{n_\mathrm{X}}, \mathrm{s}, \mathrm{e})$
4. $\nabla_{\mathrm{X},d_\mathrm{X}-1}(\mathrm{x}_{d_\mathrm{X}-1}, \mathrm{x}_0, \mathrm{x}_{n_\mathrm{X}}, \mathrm{s}, \mathrm{e}) \mapsto \mathrm{X}'_{d_\mathrm{X}}(\mathrm{x}_{d_\mathrm{X}-1}, \mathrm{x}_{d_\mathrm{X}}, \mathrm{s}, \mathrm{e})\ \nabla_{\mathrm{X},d_\mathrm{X}-1}(\mathrm{x}_{d_\mathrm{X}-1}, \mathrm{x}_0, \mathrm{x}_{n_\mathrm{X}}, \mathrm{s}, \mathrm{e})$
5. $\mathrm{X}''_{d_\mathrm{X}}(\mathrm{x}_{d_\mathrm{X}-1}, \mathrm{x}_{d_\mathrm{X}}, \mathrm{s}, \mathrm{e})\ \nabla_{\mathrm{X},d_\mathrm{X}-1}(\mathrm{x}_{d_\mathrm{X}-1}, \mathrm{x}_0, \mathrm{x}_{n_\mathrm{X}}, \mathrm{s}, \mathrm{e}) \mapsto \nabla_{\mathrm{X},d_\mathrm{X}}(\mathrm{x}_{d_\mathrm{X}}, \mathrm{x}_0, \mathrm{x}_{n_\mathrm{X}}, \mathrm{s}, \mathrm{e})$
6. $\nabla_{\mathrm{X},n_\mathrm{X}}(\mathrm{x}_{n_\mathrm{X}}, \mathrm{x}_0, \mathrm{x}_{n_\mathrm{X}}, \mathrm{s}, \mathrm{e})\ \mathrm{X}'(\mathrm{x}_0, \mathrm{x}_{n_\mathrm{X}}, \mathrm{s}, \mathrm{e}) \mapsto \mathrm{X}''(\mathrm{x}_0, \mathrm{x}_{n_\mathrm{X}}, \mathrm{s}, \mathrm{e})$

Spine Adjunction Decision for every spine node X with ono-terminal label N:

1. $\mathrm{X}'(\mathrm{x}, \mathrm{y}, \mathrm{s}', \mathrm{e}') \mapsto \nabla_\mathrm{X}(\mathrm{x}, \mathrm{y}, \mathrm{s}', \mathrm{e}')\ \mathrm{X}'(\mathrm{x}, \mathrm{y}, \mathrm{s}', \mathrm{e}')$
2. $\nabla_\mathrm{X}(\mathrm{x}, \mathrm{y}, \mathrm{s}', \mathrm{e}') \mapsto \mathrm{N}'(\mathrm{x}, \mathrm{y}, \mathrm{s}, \mathrm{e})\ \nabla_\mathrm{X}(\mathrm{x}, \mathrm{y}, \mathrm{s}', \mathrm{e}')$
3. $\mathrm{N}''(\mathrm{x}, \mathrm{y}, \mathrm{s}, \mathrm{e})\ \nabla_\mathrm{X}(\mathrm{x}, \mathrm{y}, \mathrm{s}', \mathrm{e}') \mapsto \nabla'_\mathrm{X}(\mathrm{x}, \mathrm{y}, \mathrm{s}, \mathrm{e}, \mathrm{s}', \mathrm{e}')$

4. $\nabla'_{\mathrm{X}}(\mathrm{x},\mathrm{y},\mathrm{s},\mathrm{e},\mathrm{s}',\mathrm{e}') \mapsto \mathrm{X}'(\mathrm{s},\mathrm{e},\mathrm{s}',\mathrm{e}')\ \nabla'_{\mathrm{X}}(\mathrm{x},\mathrm{y},\mathrm{s},\mathrm{e},\mathrm{s}',\mathrm{e}')$

5. $\mathrm{X}''(\mathrm{s},\mathrm{e},\mathrm{s}',\mathrm{e}')\ \nabla'_{\mathrm{X}}(\mathrm{x},\mathrm{y},\mathrm{s},\mathrm{e},\mathrm{s}',\mathrm{e}') \mapsto \nabla''_{\mathrm{X}}(\mathrm{x},\mathrm{y},\mathrm{s}',\mathrm{e}')$

6. $\nabla''_{\mathrm{X}}(\mathrm{x},\mathrm{y},\mathrm{s}',\mathrm{e}')\ \mathrm{X}'(\mathrm{x},\mathrm{y},\mathrm{s}',\mathrm{e}') \mapsto \mathrm{X}''(\mathrm{x},\mathrm{y},\mathrm{s}',\mathrm{e}')$

Foot Skipping for every foot node X

1. $\mathrm{X}'(\mathrm{s},\mathrm{e},\mathrm{s},\mathrm{e}) \mapsto \mathrm{X}''(\mathrm{s},\mathrm{e},\mathrm{s},\mathrm{e})$

In addition, for every word with position i in the input sentence, and belonging to the lexical (i.e. terminal) category T, we simulate the scanning of this input word with the transition

$\mathtt{scan}'(i-1,i,\mathrm{T}) \mapsto \mathtt{scan}''(i-1,i,\mathrm{T})$

It is possible to extend the LPDA formalism to account more cleanly for input scanning [Lan-88b]. However we chose not to do so here to avoid further complexity in our formalism. Hence the necessity to simulate scanning with these dummy transitions, that play no essential role in the constructions described in this paper.