A formal definition of Bottom-up Embedded Push-Down Automata and their tabulation technique

Miguel A. Alonso[1], Eric de la Clergerie[2] and Manuel Vilares[3]

[1] Departamento de Computación, Universidad de La Coruña
Campus de Elviña s/n, 15071 La Coruña, Spain
alonso@dc.fi.udc.es, vilares@dc.fi.udc.es
http://coleweb.dc.fi.udc.es/
[2] Institut National de Recherche en Informatique et en Automatique
Domaine de Voluceau, Rocquecourt, B.P. 105, 78153 Le Chesnay Cedex, France
Eric.De_La_Clergerie@inria.fr
http://atoll.inria.fr/

Abstract. The task of designing parsing algorithms for tree adjoining grammars could be simplified by providing a separation between the description of the parsing strategy and the execution of the parser. This can be accomplished through the use of Bottom-up Embedded Push-Down Automata. Towards this aim, we provide a formal and consistent definition of this class of automata and, by removing the finite-state control, we obtain an alternative definition which is adequate to define a tabulation framework for this model of automata and to show the equivalence with respect to other kinds of automata accepting tree adjoining languages.

1 Introduction

Tree Adjoining Languages (TAL) belong to the class of mildly context-sensitive languages, placed between context-free languages and context-sensitive languages. They can be described by Tree Adjoining Grammars (TAG) [12] and several grammar formalisms which have been shown to be equivalent to TAG with respect to their weak generative capacity [23]: Linear Indexed Grammars [10], Head Grammars [17], Combinatory Categorial Grammars [21], Context-Free Recursive Matrix Systems of index 2 [5], Positive Range Concatenation Grammars of arity 2 [6], Coupled Context-Free Grammars of range 2 [16], and Control Grammars of level 2 [24]. Several parsing algorithms have been proposed for all of them, but the design of correct and efficient parsing algorithms is a difficult task that could be simplified by providing a separation between the description of the parsing strategy and the execution of the parser. This can be accomplished through the use of automata: the actual parsing strategy can be described by means of the construction of a non-deterministic pushdown automaton, and tabulation can be introduced by means of some generic mechanism such as memoization. The construction of parsers in this way allows more straightforward proofs

of correctness and makes parsing strategies easier to understand and cheaper to implement [4].

Bottom-up embedded push-down automata (BEPDA) have been described in [19] as an extension of push-down automata adequate to implement parsing strategies for TAG in which adjunctions are recognized in a bottom-up way. In fact, BEPDA are the dual version of embedded push-down automata (EPDA) [22, 2], a model of automata in which adjunctions must be recognized in a top-down way. A less informal presentation of BEPDA has been shown in [18], with some inconsistencies between the set of allowed transitions and the set of configurations attainable.

Right-oriented linear indexed automata [14] and bottom-up 2–stack automata [8] can be used to implement parsing strategies similar to those of BEPDA. Both models of automata have associated a tabulation framework allowing their execution in polynomial time with respect to the size of the input string.

This article is outlined as follows. The rest of this section is devoted to introduce tree adjoining grammars. Section 2 provides a formal and consistent definition of classic BEPDA. In Sect. 3 the finite-state control is removed to obtain an alternative definition. Section 4 shows how this new definition is useful to implement parsers by defining a compilation schema from TAG into BEPDA. The relation between right-oriented linear indexed automata, bottom-up 2–stack automata and BEPDA is considered in Sect. 5. Derivations of bottom-up embedded push-down automata are studied in Sect. 6 in order to define a tabular technique allowing them to be executed efficiently. Section 7 presents final conclusions.

1.1 Tree Adjoining Grammars

Tree adjoining grammars (TAG) [12] are an extension of context-free grammars that use trees instead of productions as the primary representing structure. Formally, a TAG is a 5-tuple $\mathcal{G} = (V_N, V_T, S, \boldsymbol{I}, \boldsymbol{A})$, where V_N is a finite set of non-terminal symbols, V_T a finite set of terminal symbols, $S \in V_N$ is the axiom of the grammar, $\boldsymbol{I}$ a finite set of *initial trees* and $\boldsymbol{A}$ a finite set of *auxiliary trees*. $\boldsymbol{I} \cup \boldsymbol{A}$ is the set of *elementary trees*. Internal nodes are labeled by non-terminals and leaf nodes by terminals or ε, except for just one leaf per auxiliary tree (the *foot*) which is labeled by the same non-terminal used as the label of its root node. The path in an elementary tree from the root node to the foot node is called the *spine* of the tree.

New trees are derived by *adjoining*: let α be a tree containing a node N^α labeled by A and let β be an auxiliary tree whose root and foot nodes are also labeled by A. Then, the adjoining of β at the *adjunction node* N^α is obtained by excising the subtree of α with root N^α, attaching β to N^α and attaching the excised subtree to the foot of β.

The operation of *substitution* does not increase the generative power of the formalism but it is usually considered when we are dealing with lexicalized tree adjoining grammars. In this case, non-terminals can also label leaf nodes (called

substitution nodes) of elementary trees. An initial tree can be substituted at a substitution node if its root is labeled by the same non-terminal that labels the substitution node.

2 A formal definition of BEPDA

Bottom-up embedded push-down automata are an extension of push-down automata (PDA), so we first introduce this last model of automata, the operational device for parsing context-free grammars. A push-down automata [11] consists of a finite-state control, an input tape and a stack made up of stack symbols. Formally, a PDA can be defined as a tuple $(Q, V_T, V_S, \delta', q_0, Q_F, \$_f)$ where:

- Q is a finite set of states.
- V_T is a finite set of terminal symbols.
- V_S is a finite set of stack symbols.
- $q_0 \in Q$ is the initial state.
- $Q_F \subseteq Q$ is the set of final states.
- $\$_f \in V_S$ is the final stack symbol.
- δ' is a mapping from $Q \times V_T \cup \{\epsilon\} \times V_S$ into finite subsets of $Q \times V_S^*$.

In the case of bottom-up embedded push-down automata, the stack (that we call the main stack) is made up of non-empty stacks containing stack symbols. Formally, a BEPDA is defined by a tuple $(Q, V_T, V_S, \delta, q_0, Q_F, \$_f)$ where Q, V_T, V_S, $q_0 \in Q$, $Q_F \subseteq Q$ and $\$_f \in V_S$ are defined as before, but δ is now defined as a mapping from $Q \times V_T \cup \{\epsilon\} \times (\lceil V_S^+)^* \times V_S^* \times (\lceil V_S^+)^*$ into finite subsets of $Q \times V_S \cup \{\lceil V_S\}$, where $\lceil \notin V_S$ is a new symbol used as stack separator.

An *instantaneous configuration* is a triple (q, Υ, w), where q is the current state, $\Upsilon \in (\lceil V_S^+)^*$ represents the contents of the automaton stack and w is the part of the input string that is yet to be read. It is important to remark that every individual stack contained in the main stack must store at least one stack symbol. The main stack will be empty only in the initial configuration (q_0, ϵ, w).

Transitions in δ allow an automaton to derive a new configuration (q', Υ', w) from a configuration (q, Υ, aw), which is denoted $(q, \Upsilon, aw) \vdash (q', \Upsilon', w)$. The reflexive and transitive closure of $\vdash$ is denoted by $\overset{*}{\vdash}$. There exist two different types of transitions:

1. Transitions of the first type are of the form

$$(q', \lceil Z) \in \delta(q, a, \epsilon, \epsilon, \epsilon)$$

   and they can be applied to a configuration (q, Υ, aw) to yield a configuration $(q', \Upsilon\lceil Z, w)$.
2. Transitions of the second type are of the form

$$(q', Z) \in \delta(q,\ a,\ \lceil\alpha_k \ldots \lceil\alpha_{i+1},\ Z_m \ldots Z_1,\ \lceil\alpha_i \ldots \lceil\alpha_1)$$

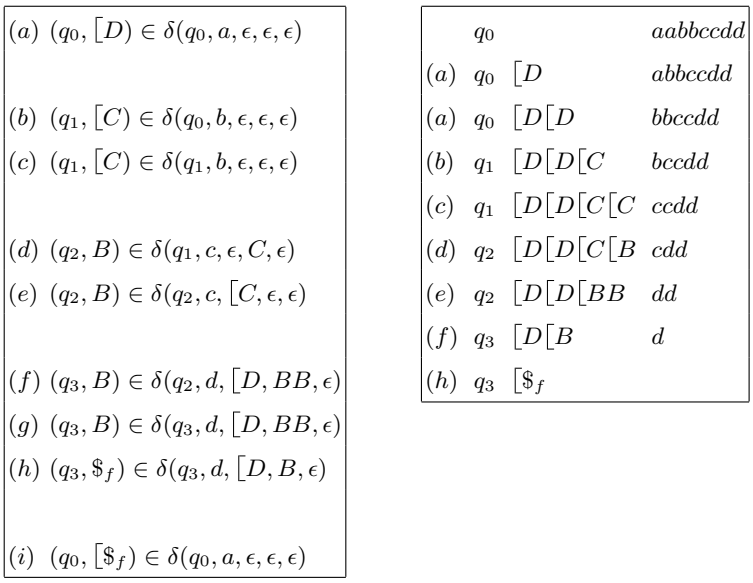

Fig. 1. BEPDA accepting $\{a^n b^n c^n d^n \mid n > 0\}$ and configurations for *aabbccdd*

where $m \geq 0$ and $k \geq i \geq 0$. They can be applied to a configuration

$$(q,\ \Upsilon \lfloor \alpha_k \ldots \lfloor \alpha_{i+1} \lfloor \alpha Z_m \ldots Z_1 \lfloor \alpha_i \ldots \lfloor \alpha_1,\ aw)$$

to yield a configuration

$$(q', \Upsilon \lfloor \alpha Z, w)$$

The *language accepted by final state* by a BEPDA is the set $w \in V_T^*$ of input strings such that $(q_0, \epsilon, w) \vdash {}^*(p, \Upsilon, \epsilon)$, where $p \in Q_F$ and $\Upsilon \in (\lfloor V_S^+)^*$.

The *language accepted by empty stack* by a BEPDA is the set $w \in V_T^*$ of input strings such that $(q_0, \epsilon, w) \vdash {}^*(q, \lfloor \$_f, \epsilon)$ for any $q \in Q$. At this point it is interesting to observe the duality with respect to embedded push-down automata: computations in EPDA start with a stack $\lfloor \$_0$ to finish with an empty stack while computations in BEPDA start with an empty stack to finish with a stack $\lfloor \$_f$.

It can be proved that for any BEPDA accepting a language L by final state there exists a BEPDA accepting the same language by empty stack and vice versa[1].

Example 1. The bottom-up embedded push-down automaton defined by the tuple $(\{q_0, q_1, q_2, q_3\}, \{a, b, c, d\}, \{B, C, D\}, \delta, q_0, \emptyset, \$_f)$, with δ containing the transitions shown in Fig. 1 (left box), accepts the language $\{a^n b^n c^n d^n \mid n \geq 0\}$ by

[1] The proof is analogous to the proof of equivalence of acceptance by final state and empty stack in the case of push-down automata [11].

empty stack. The sequence of configurations for the recognition of the input string *aabbccdd* is shown in Fig. 1 (right box), where the first column shows the transition applied, the second one the current state, the third one the contents of the stack and the fourth column shows the part of the input string to be read.

3 BEPDA without states

Finite-state control is not a fundamental component of push-down automata, as the current state in a configuration can be stored in the top element of the stack of the automaton [13]. Thus, we can obtain an alternative definition that considers a PDA as a tuple $(V_T, V_S, \Theta', \$_0, \$_f)$ where:

- V_T is a finite set of terminal symbols.
- V_S is a finite set of stack symbols.
- $\$_0 \in V_S$ is the initial stack symbol.
- $\$_f \in V_S$ is the final stack symbol.
- Θ' is a finite set of three types of transition:
  SWAP: Transitions of the form $C \overset{a}{\longmapsto} F$ that replace the top element of the stack while scanning a. The application of such a transition on a stack ξC returns the stack ξF.
  PUSH: Transitions of the form $C \overset{a}{\longmapsto} C\,F$ that push F onto C. The application of such a transition on a stack ξC returns the stack ξCF.
  POP: Transitions of the form $C\ F \overset{a}{\longmapsto} G$ that replace C and F by G. The application of such a transition on ξCF returns the stack ξG.
  where $C, F, G \in V_S$, $\xi \in V_S^*$ and $a \in V_T \cup \{\epsilon\}$.

Finite-state control can also be eliminated from bottom-up embedded push-down automata, obtaining a new definition that considers a BEPDA as a tuple $(V_T, V_S, \Theta, \$_0, \$_f)$ where V_T, V_S, $\$_0 \in V_S$ and $\$_f \in V_S$ are defined as before but Θ is now defined as a finite set of six types of transition:

SWAP: Transitions of the form $C \overset{a}{\longmapsto} F$ that replace the top element of the top stack while scanning a. The application of such a transition on a stack $\Upsilon\lceil\alpha C$ returns the stack $\Upsilon\lceil\alpha F$.

PUSH: Transitions of the form $C \overset{a}{\longmapsto} C\ F$ that push F onto C. The application of such a transition on a stack $\Upsilon\lceil\alpha C$ returns the stack $\Upsilon\lceil\alpha CF$.

POP: Transitions of the form $C\ F \overset{a}{\longmapsto} G$ that replace C and F by G. The application of such a transition on $\Upsilon\lceil\alpha CF$ returns the stack $\Upsilon\lceil\alpha G$.

WRAP: Transitions of the form $C \overset{a}{\longmapsto} C, \lceil F$ that push a new stack $\lceil F$ on the top of the main stack. The application of such a transition on a stack $\Upsilon\lceil\alpha C$ returns the stack $\Upsilon\lceil\alpha C\lceil F$.

UNWRAP-A: Transitions *unwrap-above* of the form $C, \lceil F \overset{a}{\longmapsto} G$ that delete the top stack $\lceil F$ and replace the new top element by G. The application of such a transition on a stack $\Upsilon\lceil\alpha C\lceil F$ returns the stack $\Upsilon\lceil\alpha G$.

UNWRAP-B: Transitions *unwrap-below* of the form $\lceil C, F \overset{a}{\longmapsto} G$ that delete the stack $\lceil C$ placed just below the top stack and replace the top element by G. The application of such a transition on a stack $\Upsilon\lceil C\lceil\alpha F$ returns the stack $\Upsilon\lceil\alpha G$.

(a)	$\$_0 \overset{a}{\longmapsto} \$_0, \lceil D$	
(b)	$D \overset{a}{\longmapsto} D, \lceil D$	
(c)	$D \longmapsto D, \lceil C$	
(d)	$C \overset{b}{\longmapsto} C, \lceil C$	
(e)	$C \longmapsto B$	
(f)	$B \longmapsto BE$	
(g)	$\lceil C, E \overset{c}{\longmapsto} C$	
(h)	$BC \longmapsto B$	
(i)	$\lceil D, B \overset{d}{\longmapsto} D$	
(j)	$BD \longmapsto B$	
(k)	$\$_0, \lceil D \longmapsto \$_f$	

	$\lceil \$_0$	$aabbccdd$
(a)	$\lceil \$_0 \lceil D$	$aabbccdd$
(b)	$\lceil \$_0 \lceil D \lceil D$	$abbccdd$
(c)	$\lceil \$_0 \lceil D \lceil D \lceil C$	$bbccdd$
(d)	$\lceil \$_0 \lceil D \lceil D \lceil C \lceil C$	$bccdd$
(d)	$\lceil \$_0 \lceil D \lceil D \lceil C \lceil C \lceil C$	$ccdd$
(e)	$\lceil \$_0 \lceil D \lceil D \lceil C \lceil C \lceil B$	$ccdd$
(f)	$\lceil \$_0 \lceil D \lceil D \lceil C \lceil C \lceil BE$	$ccdd$
(g)	$\lceil \$_0 \lceil D \lceil D \lceil C \lceil BC$	cdd
(e)	$\lceil \$_0 \lceil D \lceil D \lceil C \lceil BB$	cdd
(f)	$\lceil \$_0 \lceil D \lceil D \lceil C \lceil BBE$	cdd
(g)	$\lceil \$_0 \lceil D \lceil D \lceil BBC$	dd
(h)	$\lceil \$_0 \lceil D \lceil D \lceil BB$	dd
(i)	$\lceil \$_0 \lceil D \lceil BD$	d
(j)	$\lceil \$_0 \lceil D \lceil B$	d
(i)	$\lceil \$_0 \lceil D$	
(k)	$\lceil \$_f$	

Fig. 2. BEPDA without finite-state control for $\{a^n b^n c^n d^n \mid n > 0\}$

where $C, F, G \in V_S$, $\Upsilon \in (\lceil V_S^*)^*$, $\alpha \in V_S^*$ and $a \in V_T \cup \{\epsilon\}$.

An *instantaneous configuration* is a pair (Υ, w), where Υ represents the contents of the automaton stack and w is the part of the input string that is yet to be read. A configuration (Υ, aw) derives a configuration (Υ', w), denoted $(\Upsilon, aw) \vdash (\Upsilon', w)$, if and only if there exist a transition that applied to Υ gives Υ' and scans a from the input string. We use $\vdash^*$ to denote the reflexive and transitive closure of $\vdash$. An input string is accepted by an BEPDA if $(\lceil \$_0, w) \vdash^* (\lceil \$_f, \epsilon)$. The language accepted by an BEPDA is the set of $w \in V_T^*$ such that $(\lceil \$_0, w) \vdash^* (\lceil \$_f, \epsilon)$.

Example 2. The bottom-up embedded push-down automaton without states defined by the tuple $(\{a, b, c, d\}, \{B, C, D, E, F, \$_0, \$_f\}, \Theta, \$_0, \$_f)$, with Θ containing the transitions shown in Fig. 2 (left box), accepts the language $\{a^n b^n c^n d^n \mid n \geq 0\}$. The sequence of configurations to recognize the input string *aabbccdd* is also shown in Fig. 2 (right-box), where the first column shows the transition applied in each step, the second one shows the contents of the stack and the third column shows the part of the input string to be read.

It can be proved that transitions of a BEPDA with states can be emulated by transitions a BEPDA without states and vice versa.

Sketch of the proof: As a first step, we must consider a normalized version of transitions for BEPDA with states. These transitions are of the form:

$$(q', \lceil Z) \in \delta(q, a, \epsilon, \epsilon, \epsilon)$$

$$(q', Z) \in \delta(q,\ a,\ \lceil Z'_k \dots \lceil Z'_{i+1},\ Z_m \dots Z_1,\ \lceil Z'_i \dots \lceil Z'_1)$$

where $q, q' \in Q$, $a \in V_T \cup \{\epsilon\}$, $Z, Z_1, \dots, Z_m \in V_S$, $Z'_1, \dots, Z'_k \in V_S$, $0 \leq i \leq k$ and $0 \leq m \leq 2$. Then, we show these transitions can be emulated by a set of SWAP, PUSH, POP, WRAP, UNWRAP-A and UNWRAP-B transitions, and vice versa, with the help of a kind of complex transitions of the form

$$\lceil F_k \dots \lceil F_{i+1}, DC_1 \dots C_m, \lceil F_i \dots \lceil F_1 \overset{a}{\longmapsto} \ DB$$

created by the application of a sequence of simple transitions, where $a \in V_T \cup \{\epsilon\}$; $0 \leq m \leq 2$; $B, C_1, \dots, C_m, F_1, \dots, F_k, \in V_S$; and if $m = 0$ then $D \in V_S$ else $D = \epsilon$. □

4 Compiling TAG into BEPDA

Automata are interesting for parsing because they allow us to separate two different problems that arise during the definition of parsing algorithms: the description of the parsing strategy and the execution of the parser. By means of automata, a parsing strategy for a given grammar can be translated into a set of transitions defining a (possibly non deterministic) automaton and then the automaton can be executed using some standard technique.

In this section we define a generic compilation schema for tree adjoining grammars based on a call/return model [9]. We consider each elementary tree γ of a TAG as formed by a set of context-free productions $\mathcal{P}(\gamma)$: a node N^γ and its g children $N^\gamma_1 \dots N^\gamma_g$ are represented by a production $N^\gamma \rightarrow N^\gamma_1 \dots N^\gamma_g$. The elements of the productions are the nodes of the tree, except for the case of elements belonging to $V_T \cup \{\varepsilon\}$ in the right-hand side of production. Those elements may not have children and are not candidates to be adjunction nodes, so we identify such nodes labeled by a terminal with that terminal. We use $\beta \in \text{adj}(N^\gamma)$ to denote that a tree $\beta \in \boldsymbol{A}$ may be adjoined at node N^γ. If adjunction is not mandatory at N^γ, then $\mathbf{nil} \in \text{adj}(N^\gamma)$. If a tree $\alpha \in \boldsymbol{I}$ may be substituted at node N^γ, then $\alpha \in \text{subs}(N^\gamma)$. We consider the additional productions $\top^\alpha \rightarrow \mathbf{R}^\alpha$, $\top^\beta \rightarrow \mathbf{R}^\beta$ and $\mathbf{F}^\beta \rightarrow \bot$ for each initial tree α and each auxiliary tree β, where $\mathbf{R}^\alpha$ is the root node of α and $\mathbf{R}^\beta$ and $\mathbf{F}^\beta$ are the root node and foot node of β, respectively.

Fig. 3 shows the compilation rules from TAG to BEPDA, where symbols $\nabla^\gamma_{r,s}$ has been introduced to denote dotted productions

$$N^\gamma_{r,0} \rightarrow N^\gamma_{r,1} \dots N^\gamma_{r,s} \bullet N^\gamma_{r,s+1} \dots N^\gamma_{r,n_r}$$

where n_r is the length of the right hand side of the production. The meaning of each compilation rule is graphically shown in Fig. 4. This schema is parameterized by $\overrightarrow{N^\gamma}$, the information propagated top-down with respect to the node N^γ,

[INIT]	$\$_0 \longmapsto \$_0 \; [\; \overrightarrow{\top^\alpha}$	$\alpha \in \boldsymbol{I},\ S = \text{label}(\mathbf{R}^\alpha)$
[FINAL]	$\$_0 \; [\; \overleftarrow{\top^\alpha} \longmapsto \$_f$	$\alpha \in \boldsymbol{I},\ S = \text{label}(\mathbf{R}^\alpha)$
[CALL]	$\nabla^\gamma_{r,s} \longmapsto \nabla^\gamma_{r,s},\ [\; \overrightarrow{N^\gamma_{r,s+1}}$	$N^\gamma_{r,s+1} \notin \text{spine}(\gamma),\ \mathbf{nil} \in \text{adj}(N_{r,s+1})$
[SCALL]	$\nabla^\beta_{r,s} \longmapsto \nabla^\beta_{r,s} \; [\; \overrightarrow{N^\beta_{r,s+1}}$	$N^\beta_{r,s+1} \in \text{spine}(\beta),\ \mathbf{nil} \in \text{adj}(N^\beta_{r,s+1})$
[SEL]	$\overrightarrow{N^\gamma_{r,0}} \longmapsto \nabla^\gamma_{r,0}$	
[PUB]	$\nabla^\gamma_{r,n_r} \longmapsto \overleftarrow{N^\gamma_{r,0}}$	
[RET]	$\nabla^\gamma_{r,s},\ [\; \overleftarrow{N^\gamma_{r,s+1}} \longmapsto \nabla^\gamma_{r,s+1}$	$N^\gamma_{r,s+1} \notin \text{spine}(\gamma),\ \mathbf{nil} \in \text{adj}(N_{r,s+1})$
[SRET]	$[\; \nabla^\beta_{r,s},\ \overleftarrow{N^\beta_{r,s+1}} \longmapsto \nabla^\beta_{r,s+1}$	$N^\beta_{r,s+1} \in \text{spine}(\beta),\ \mathbf{nil} \in \text{adj}(N^\beta_{r,s+1})$
[SCAN]	$\overrightarrow{N^\gamma_{r,0}} \overset{a}{\longmapsto} \overleftarrow{N^\gamma_{r,0}}$	$N^\gamma_{r,0} \rightarrow a$
[AdjCALL]	$\nabla^\gamma_{r,s} \longmapsto \nabla^\gamma_{r,s},\ [\; \overrightarrow{\top^\beta}$	$\beta \in \text{adj}(N^\gamma_{r,s+1})$
[AdjRET-a]	$[\; \nabla^\gamma_{r,s},\ \overleftarrow{\top^\beta} \longmapsto \top$	$\beta \in \text{adj}(N^\gamma_{r,s+1})$
[AdjRET-b]	$\Delta^\gamma_{r,s}\ \top \longmapsto \nabla^\gamma_{r,s+1}$	
[FootCALL]	$\nabla^\beta_{f,0} \longmapsto \nabla^\beta_{f,0},\ [\; \overrightarrow{N^\gamma_{r,s+1}}$	$N^\beta_{f,0} = \mathbf{F}^\beta,\ \ \beta \in \text{adj}(N^\gamma_{r,s+1})$
[FootRET-a]	$[\; \nabla^\beta_{f,0},\ \overleftarrow{N^\gamma_{r,s+1}} \longmapsto \Delta^\gamma_{r,s}$	$N^\beta_{f,0} = \mathbf{F}^\beta,\ \ \beta \in \text{adj}(N^\gamma_{r,s+1})$
[FootRET-b]	$\Delta^\gamma_{r,s} \longmapsto \Delta^\gamma_{r,s}\ \nabla^\beta_{f,1}$	$N^\beta_{f,0} = \mathbf{F}^\beta,\ \ \beta \in \text{adj}(N^\gamma_{r,s+1})$
[SubsCALL]	$\nabla^\gamma_{r,s} \longmapsto \nabla^\gamma_{r,s},\ [\; \overrightarrow{\top^\alpha}$	$\alpha \in \text{subs}(N_{r,s+1})$
[SubsRET]	$\nabla^\gamma_{r,s},\ [\; \overleftarrow{\top^\alpha} \longmapsto \nabla^\gamma_{r,s+1}$	$\alpha \in \text{subs}(N_{r,s+1})$

Fig. 3. Generic compilation schema for TAG

and by $\overleftarrow{N^\gamma}$, the information propagated bottom-up. When the schema is used to implement a top-down traversal of elementary trees $\overrightarrow{N^\gamma} = N^\gamma$ and $\overleftarrow{N^\gamma} = \Box$, where $\Box$ is a fresh stack symbol. A bottom-up traversal requires $\overrightarrow{N^\gamma} = \Box$ and $\overleftarrow{N^\gamma} = N^\gamma$. For a mixed traversal of elementary trees, $\overrightarrow{N^\gamma} = \overline{N^\gamma}$ and $\overleftarrow{N^\gamma} = \overline{\overline{N^\gamma}}$, where $\overline{N^\gamma}$ and $\overline{\overline{N^\gamma}}$ are used to distinguish the top-down prediction from the bottom-up propagation of a node.

With respect to adjunctions, we can observe in Fig. 3 that each stack stores pending adjunctions with respect to the node placed on the top of the stack in a bottom-up treatment of adjunctions: when a foot node is reached, the adjunction node is stored on the top of the stack ([**FootCALL-a**]); the traversal of

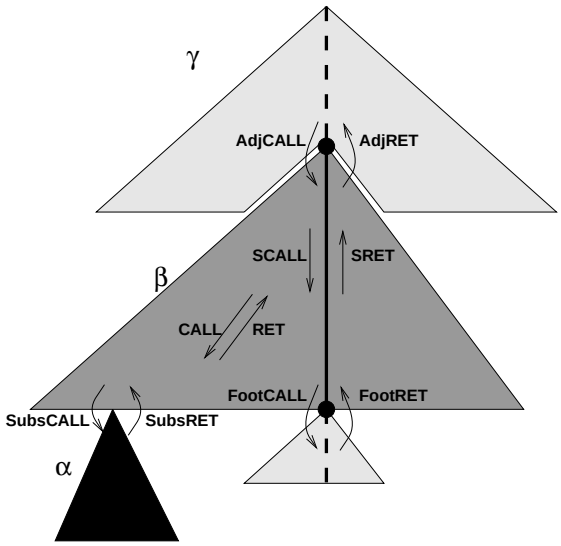

Fig. 4. Meaning of compilation rules

the elementary tree is suspended to continue with the traversal of the adjoined auxiliary tree ([**FootCALL-b**]); the adjunction stack is propagated through the spine ([**SRET**]) up to the root node ([**AdjRET-a**]); and then the stack element corresponding to the auxiliary tree is eliminated to resume the traversal of the elementary tree ([**AdjRET-b**]). To avoid confusion, we store $\Delta^{\gamma}_{r,s}$ instead of $\nabla^{\gamma}_{r,s}$ to indicate that an adjunction was started at node $N^{\gamma}_{r,s+1}$. A symbol Δ can be seen as a symbol ∇ waiting an adjunction to be completed.

Substitution is managed trough transitions generated by compilation rules [**SubsCALL**], which start the traversal of the substituted trees, and [**SubsRET**], which resume the traversal of the tree containing the substitution node once the substituted tree has been completely traversed.

5 BEPDA and other automata for TAG

5.1 Right-oriented linear indexed automata

Linear indexed automata (LIA) [4] are an extension of push-down automata in which each stack symbol has been associated to a list of indices. Right-oriented linear indexed automata (R-LIA) [14, 15] are a subclass of linear indexed automata that can be used to implement parsing strategies for TAG in which adjunctions are recognized in a bottom-up way. BEPDA and R–LIA are equivalent classes of automata. Given a BEPDA, the equivalent R-LIA is obtained by means of a simple change in the notation: the top element of a stack is considered a stack symbol, and the rest of the stack is considered the indices list

Transition	BEPDA	R–LIA
SWAP	$C \overset{a}{\longmapsto} F$	$C[\circ\circ] \overset{a}{\longmapsto} F[\circ\circ]$
PUSH	$C \overset{a}{\longmapsto} CF$	$C[\circ\circ] \overset{a}{\longmapsto} F[\circ\circ C]$
POP	$CF \overset{a}{\longmapsto} G$	$F[\circ\circ C] \overset{a}{\longmapsto} G[\circ\circ]$
UNWRAP-A	$C, \lceil F \overset{a}{\longmapsto} G$	$C[\circ\circ]\ F[\,] \overset{a}{\longmapsto} G[\circ\circ]$
UNWRAP-B	$\lceil C, F \overset{a}{\longmapsto} G$	$C[\,]\ F[\circ\circ] \overset{a}{\longmapsto} G[\circ\circ]$
WRAP	$C \overset{a}{\longmapsto} C, \lceil F$	$C[\circ\circ] \overset{a}{\longmapsto} C[\circ\circ]\ F[\,]$

Fig. 5. Equivalence between BEPDA and R-LIA

associated to it, as is shown in Fig. 5. The same procedure also serves to obtain the BEPDA equivalent to a given R–LIA.

5.2 Bottom-up 2–stack automata

Strongly-driven 2-stack automata (SD–2SA) [7] are an extension of push-down automata working on a pair of asymmetric stacks, a master stack and an auxiliary stack. These stacks are partitioned into sessions. Computations in each session are performed in one of two modes *write* and *erase*. A session starts in mode write and switches at some point to mode erase. In mode write (resp. erase), no element can be popped from (resp. pushed to) the master stack. Switching back from erase to write mode is not allowed. Bottom-up 2–stack automata (BU–2SA) [8] are a projection of SD–2SA requiring the emptiness of the auxiliary stack during computations in mode write. When a new session is created in a BU–2SA, a mark $\models$ is left on the master stack, other movements performed in write mode leaving a mark $\triangleright$. These marks are popped in erase mode.

The full set of BU–2SA transitions is shown in Fig. 6. Transitions of type **SWAP2** are equivalent to $\lceil C \overset{a}{\longmapsto} \lceil F$ in BEPDA, compound transitions obtained from the consecutive application of $C \longmapsto C, \lceil F'$ and $\lceil C, F' \overset{a}{\longmapsto} F$, where F' is a fresh stack symbol. In a similar way, transitions of type $\nearrow$**ERASE** are translated into compound transitions formed by an UNWRAP-B and a POP transition, and transitions of type $\searrow$**ERASE** are translated into the composition of UNWRAP-B and PUSH transitions. Slightly different is the case for transitions of type $\triangleright$**WRITE**, equivalent to $\lceil C \overset{a}{\longmapsto} \lceil C, \lceil F$ transitions in BEPDA, which are obtained as the consecutive application of $\lceil C \overset{a}{\longmapsto} \lceil C'$ and $C' \longmapsto C', \lceil F$, an additional transition $C', \lceil G \overset{b}{\longmapsto} K$ for each transition $C, \lceil G \overset{b}{\longmapsto} K$ in the automaton, and an additional transition $\lceil C', G \overset{b}{\longmapsto} K$ for each transition $\lceil C, G \overset{b}{\longmapsto} K$, where C' is a fresh stack symbol.

As a consequence, it is possible to build a BEPDA for any given BU–2SA. However, the reverse is not always true: PUSH and POP transitions can only

BEPDA transition		BU–2SA transition	
SWAP	$C \overset{a}{\longmapsto} F$	$(m, C, \epsilon) \overset{a}{\longmapsto} (m, F, \epsilon)$	**SWAP1**
	$[C \overset{a}{\longmapsto} [F$	$(\mathbf{w}, C, \models^m) \overset{a}{\longmapsto} (\mathbf{e}, F, \models^m)$	**SWAP2**
WRAP	$C \overset{a}{\longmapsto} C, [F$	$(m, C, \epsilon) \overset{a}{\longmapsto} (\mathbf{w}, C \models^m F, \models^m)$	$\models$**WRITE**
	$[C \overset{a}{\longmapsto} [C, [F$	$(\mathbf{w}, C, \epsilon) \overset{a}{\longmapsto} (\mathbf{w}, C \rhd F, \epsilon)$	$\rhd$**WRITE**
UNWRAP-A	$C, [F \overset{a}{\longmapsto} G$	$(\mathbf{e}, C \models^m F, \models^m) \overset{a}{\longmapsto} (m, G, \epsilon)$	$\models$**ERASE**
UNWRAP-B	$[C, F \overset{a}{\longmapsto} G$	$(\mathbf{e}, C \rhd F, \epsilon) \overset{a}{\longmapsto} (\mathbf{e}, G, \epsilon)$	$\rightarrow$**ERASE**
	$[C, XF \overset{a}{\longmapsto} G$	$(\mathbf{e}, C \rhd F, X) \overset{a}{\longmapsto} (\mathbf{e}, G, \epsilon)$	$\nearrow$**ERASE**
	$[C, F \overset{a}{\longmapsto} XG$	$(\mathbf{e}, C \rhd F, \epsilon) \overset{a}{\longmapsto} (\mathbf{e}, G, X)$	$\searrow$**ERASE**

Fig. 6. Correspondence between BEPDA and BU–2SA

be translated into BU–2SA if they are merged with an UNWRAP-B transition. So, a BEPDA implementing a shift-reduce strategy (requiring the use of PUSH and POP transitions in combination with UNWRAP-A transitions[2]) can not be translated into a BU–2SA.

6 Tabulation

The direct execution of (bottom-up embedded) push-down automata may be exponential with respect to the length of the input string and may even loop. To get polynomial complexity, we must avoid duplicating stack contents when several transitions may be applied to a given configuration. Instead of storing all the information about a configuration, we must determine the information we need to trace to retrieve that configuration. This information is stored into a table of *items*.

6.1 Tabulation of PDA

In a context-free grammar, if $B \overset{*}{\Rightarrow} \delta$ then $\alpha B \beta \overset{*}{\Rightarrow} \alpha \delta \beta$ for all $\alpha, \beta \in (V_N \cup V_T)^*$. This context-freeness property can be translated into push-down automata: given a derivation

$$(B, a_{i+1} \dots a_{j+1} \dots a_n) \overset{*}{\vdash} (C, a_{j+1} \dots a_n)$$

for all $\xi \in V_S^*$ we also have

$$(\xi B, a_{i+1} \dots a_{j+1} \dots a_n) \overset{*}{\vdash} (\xi C, a_{j+1} \dots a_n)$$

[2] A linear indexed automata implementing a LR-like strategy for linear indexed grammars using this kind of transitions is described in [1].

Thus, the only information we need to store about this last derivation are the stack elements B and C and the input positions i and j. We store this information in the form of an item $[B, i, C, j]$. New items[3] are derived from existing items by means of inference rules of the form $\frac{antecedents}{consequent}$ *conditions* similar to those used in grammatical deduction systems [20], meaning that if all antecedents are present and conditions are satisfied then the consequent item should be generated. Conditions usually refer to transitions of the automaton and to terminals from the input string.

In the case of PDA we have three inference rules, one for each type of transition:

- The inference rule for SWAP transitions

$$\frac{[B, i, C, j]}{[B, i, F, k]} \quad C \overset{a}{\longmapsto} F$$

  means that given a derivation $(\xi B, a_{i+1} \dots a_n) \overset{*}{\vdash} (\xi C, a_{j+1} \dots a_n)$, the application a a transition $C \overset{a}{\longmapsto} F$ yields a derivation $(\xi B, a_{i+1} \dots a_n) \overset{*}{\vdash} (\xi F, a_{k+1} \dots a_n)$, where $k = j + |a|$.
- In the case of a PUSH transition, the inference rule

$$\frac{[B, i, C, j]}{[F, k, F, k]} \quad C \overset{a}{\longmapsto} CF$$

  means that given an derivation $(\xi B, a_{i+1} \dots a_n) \overset{*}{\vdash} (\xi C, a_{j+1} \dots a_n)$, the application of a transition $C \overset{a}{\longmapsto} CF$ yields a derivation $(\xi F, a_{k+1} \dots a_n) \overset{0}{\vdash} (\xi F, a_{k+1} \dots a_n)$, where $k = j + |a|$.
- In the case of a POP transition, the following inference rule

$$\frac{\begin{array}{l}[F', k', F, k] \\ [B, i, C, j]\end{array}}{[B, i, G, l]} \quad \begin{array}{l} C \overset{b}{\longmapsto} CF' \\ CF \overset{a}{\longmapsto} G \end{array}$$

  is applied to indicate that given a derivation $(\xi B, a_{i+1} \dots a_n) \overset{*}{\vdash} (\xi C, a_{j+1} \dots a_n)$, a PUSH transition $C \overset{b}{\longmapsto} CF'$ that yields a derivation $(\xi C, a_{j+1} \dots a_n) \vdash (\xi CF', a_{k'+1} \dots a_n)$ and a derivation $(\xi F', a_{k'+1} \dots a_n) \overset{*}{\vdash} (\xi F, a_{k+1} \dots a_n)$, the application of a POP transition $CF \overset{a}{\longmapsto} G$ yields a derivation $(\xi CF, a_{k+1} \dots a_n) \vdash (\xi G, a_{l+1} \dots a_n)$, where $k' = j + |b|$ and $l = k + |a|$.

A PDA computation starts with the initial item $[\$_0, 0, \$_0, 0]$. An input string $a_1 \dots a_n$ has been recognized if the final item $[\$_0, 0, \$_f, n]$ is generated indicating that a derivation $(\$_0, a_1 \dots a_n) \overset{*}{\vdash} (\$_f, \epsilon)$ has been attained.

[3] In this article we are considering items based on SWAP transitions, as in [15], but items can be also defined based on PUSH transitions, as in [13].

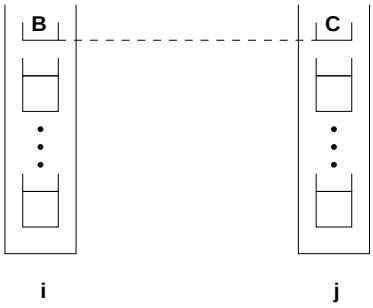

Fig. 7. Call derivations

6.2 Tabulation of BEPDA

For BEPDA, we extend the technique proposed for push-down automata in the previous subsection. We must take into account that the push-down in BEPDA is made up of stacks instead of single stack elements. So, an item $[B, i, C, j]$ can only be used to represent derivations of the form $(\lfloor B, a_{i+1} \dots a_n) \vdash^* (\lfloor C, a_{j+1} \dots a_n)$. We can generalize this result by considering items of the form $[B, i, \alpha C, j]$ in order to represent derivations $(\lfloor B, a_{i+1} \dots a_n) \vdash^* (\lfloor \alpha C, a_{j+1} \dots a_n)$. However, the size of α is not bounded and so the complexity of inference rules is not polynomial. To get polynomial complexity we must study in detail the form of derivations. We can observe that two different kinds of derivation can be attained in BEPDA:

Call derivations. Correspond to the placement of an unitary stack onto the top of the main stack, typically by means of a WRAP transition:

$$(\lfloor B, a_{i+1} \dots a_n) \vdash^* (\lfloor C, a_{j+1} \dots a_n)$$

where $B, C \in V_S$. These derivations are context-freeness in the sense that for any $\Upsilon \in (\lfloor V_S^*)^*$ we have

$$(\Upsilon \ \lfloor B, a_{i+1} \dots a_n) \vdash^* (\Upsilon \ \lfloor C, a_{j+1} \dots a_n)$$

Thus, they can be represented by call items of the form

$$[B, i, C, j, - \mid -, -, -, -]$$

A graphical representation of call derivations and items is shown in figure 7.

Return derivations. Correspond to the placement of a non-unitary stack onto the top of the main stack:

$$\begin{aligned}
&(\lfloor B, a_{i+1} \dots a_n) \\
&\vdash^* (\lfloor B \ \ \Upsilon_1 \ \lfloor D, a_{p+1} \dots a_n) \\
&\vdash^* (\lfloor B \ \ \Upsilon_1 \ \lfloor \alpha E, a_{q+1} \dots a_n) \\
&\vdash^* (\lfloor \alpha X C, a_{j+1} \dots a_n)
\end{aligned}$$

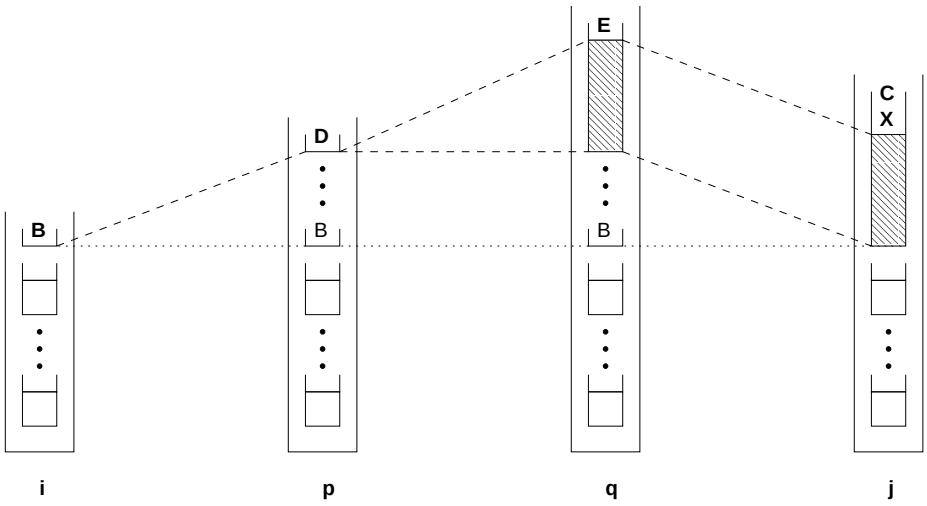

Fig. 8. Return derivations

where $B, C, D, E, X \in V_S$ and $\alpha \in V_S^*$. The two occurrences of α denote the same stack in the sense that α is neither consulted nor modified through derivation. These derivations are context-free with respect to the low part of the main stack and so, for any $\Upsilon, \Upsilon_1 \in (\lceil V_S^*)^*$ we have

$$\begin{aligned}
&(\Upsilon \ \lceil B, a_{i+1} \dots a_n) \\
&\vdash^* (\Upsilon \ \lceil B \ \ \Upsilon_1 \ \lceil D, a_{p+1} \dots a_n) \\
&\vdash^* (\Upsilon \ \lceil B \ \ \Upsilon_1 \ \lceil \alpha E, a_{q+1} \dots a_n) \\
&\vdash^* (\Upsilon \ \lceil \alpha X C, a_{j+1} \dots a_n)
\end{aligned}$$

but they are not independent with respect to the subderivation

$$(\lceil D, a_{p+1} \dots a_n) \vdash^* (\lceil \alpha E, a_{q+1} \dots a_n)$$

We can not store a entire subderivation into items if we want to get polynomial complexity. To solve this problem, we need to consider an item as formed by two different parts, a *head* storing some information about the top elements of the initial and final configurations of the derivation we are representing, and a *tail* that store information about a subderivation. The tail acts like a pointer to another configuration. Following the tail pointers we can retrieve the whole contents of a given stack. Therefore, for return derivations we are considering return items of the form

$$[B, i, C, j, X \mid D, p, E, q]$$

where (B, i, C, j, X) is the head and (D, p, E, q) is the tail. A graphical representation of call derivations and items is shown in figure 8.

Call and return items are combined by means of the following set of inference rules:

$$\frac{[B, i, C, j, - \mid -, -, -, -]}{[B, i, F, k, - \mid -, -, -, -]} \; C \overset{a}{\longmapsto} F$$

$$\frac{[B, i, C, j, X \mid D, p, E, q]}{[B, i, F, k, X \mid D, p, E, q]} \; C \overset{a}{\longmapsto} F$$

$$\frac{[B, i, C, j, - \mid -, -, -, -]}{[B, i, F, k, C \mid B, i, C, j]} \; C \overset{a}{\longmapsto} CF$$

$$\frac{[B, i, C, j, X \mid D, p, E, q]}{[B, i, F, k, C \mid B, i, C, j]} \; C \overset{a}{\longmapsto} CF$$

$$\frac{\begin{array}{c}[B, i, F, j, C \mid D, p, E, q] \\ [D, p, E, q, - \mid -, -, -, -]\end{array}}{[B, i, G, k, - \mid -, -, -, -]} \; CF \overset{a}{\longmapsto} G$$

$$\frac{\begin{array}{c}[B, i, F, j, C \mid D, p, E, q] \\ [D, p, E, q, X' \mid O, u, P, v]\end{array}}{[B, i, G, k, X' \mid O, u, P, v]} \; CF \overset{a}{\longmapsto} G$$

$$\frac{[B, i, C, j, - \mid -, -, -, -]}{[F, k, F, k, - \mid -, -, -, -]} \; C \overset{a}{\longmapsto} C, \lfloor F$$

$$\frac{[B, i, C, j, X \mid D, p, E, q]}{[F, k, F, k, - \mid -, -, -, -]} \; C \overset{a}{\longmapsto} C, \lfloor F$$

$$\frac{\begin{array}{c}[F, k, F', k', - \mid -, -, -, -] \\ [B, i, C, j, - \mid -, -, -, -]\end{array}}{[B, i, G, l, - \mid -, -, -, -]} \; \begin{array}{l} C \overset{a}{\longmapsto} C, \lfloor F \\ C, \lfloor F' \overset{b}{\longmapsto} G \end{array}$$

$$\frac{\begin{array}{c}[F, k, F', k', - \mid -, -, -, -] \\ [B, i, C, j, X \mid D, p, E, q]\end{array}}{[B, i, G, l, X \mid D, p, E, q]} \; \begin{array}{l} C \overset{a}{\longmapsto} C, \lfloor F \\ C, \lfloor F' \overset{b}{\longmapsto} G \end{array}$$

$$\frac{\begin{array}{c}[F, k, F', k', - \mid -, -, -, -] \\ [B, i, C, j, - \mid -, -, -, -]\end{array}}{[B, i, G, l, - \mid -, -, -, -]} \; \begin{array}{l} C \overset{a}{\longmapsto} C, \lfloor F \\ \lfloor C, F' \overset{b}{\longmapsto} G \end{array}$$

$$\frac{\begin{array}{c}[F, k, F', k', X \mid D, p, E, q] \\ [B, i, C, j, - \mid -, -, -, -]\end{array}}{[B, i, G, l, X \mid D, p, E, q]} \; \begin{array}{l} C \overset{a}{\longmapsto} C, \lfloor F \\ \lfloor C, F' \overset{b}{\longmapsto} G \end{array}$$

where $k = j$ if $a = \epsilon$, $k = j + 1$ if $a = a_{j+1}$, $l = k'$ if $b = \epsilon$ and $l = k' + 1$ if $b = a_{k'+1}$.

Computations start with the initial item $[\$_0, 0, \$_0, 0, - \mid -, -, -, -]$. An input string $a_1 \dots a_n$ has been recognized if the final item $[\$_0, 0, \$_f, n, - \mid -, -, -, -]$ is present. It can be proved that handling items with the inference rules is equivalent to applying the transitions on the whole stacks.

Sketch of the proof: We show that each item represents a derivation and that any derivation is represented by an item. Taking as base case the item and derivation corresponding to the initial configuration, we must apply induction on the length of the derivations. We can observe that given the set of antecedents items (each of them representing a derivation, by induction hypothesis) and the transitions specified by an inference rule, we obtain an item representing a derivation when this rule is applied. We can also observe that, given an item, it can be decomposed by some inference rule into a set of antecedent items (representing derivations by induction hypothesis) and a set of transitions, such that the application of the rule gives as a result this item representing a derivation. The proof is tedious but not difficult. The important point is to consider the exhaustive list of all kinds of derivation that can be involved in the application of each inference rule. □

The space complexity of the proposed tabulation technique with respect to the length of the input string is $\mathcal{O}(n^4)$, due to every item stores four positions of the input string. The worst case time complexity is $\mathcal{O}(n^6)$ due to the inference rule

$$\frac{[B, i, F, j, C \mid D, p, E, q] \quad [D, p, E, q, X' \mid O, u, P, v]}{[B, i, G, k, X' \mid O, u, P, v]} \; CF \overset{a}{\longmapsto} G$$

that stores 6 independent input positions i, j, p, q, u and v.

7 Conclusion

We have provided a formal definition of bottom-up embedded push-down automata. We have also shown that finite-state control can be eliminated, obtaining a new definition in which transitions are in a form useful to describe compilation schemata for TAG and suitable for tabulation. The resulting definition has been shown to be equivalent to right-oriented linear indexed automata and a superset of bottom-up 2–stack automata with respect to the parsing strategies that can be described in both models of automata.

Acknowledgments

A previous and shorter version of this article was presented at the *Second International Workshop on Parsing and Deduction (TAPD'2000)*[4] held in Vigo

[4] `http://coleweb.dc.fi.udc.es/tapd2000/`

(Spain) in September 2000 [3]. We are grateful to the participants in this workshop for their comments and suggestions. The research reported in this article has been partially supported by Plan Nacional de Investigación Científica, Desarrollo e Innovación Tecnológica (Grant TIC2000-0370-C02-01), FEDER of EU (Grant 1FD97-0047-C04-02) and Xunta de Galicia (Grant PGIDT99XI10502B).

References

1. Miguel A. Alonso, Eric de la Clergerie, and Manuel Vilares. Automata-based parsing in dynamic programming for Linear Indexed Grammars. In A. S. Narin'yani, editor, *Proc. of DIALOGUE'97 Computational Linguistics and its Applications International Workshop*, pages 22–27, Moscow, Russia, June 1997.
2. Miguel A. Alonso, Eric de la Clergerie, and Manuel Vilares. A redefinition of Embedded Push-Down Automata. In *Proc. of 5th International Workshop on Tree Adjoining Grammars and Related Formalisms (TAG+5)*, pages 19–26, Paris, France, May 2000.
3. Miguel A. Alonso, Eric de la Clergerie, and Manuel Vilares. A formal definition of Bottom-up Embedded Push-Down Automata and their tabulation technique. In David S. Warren, Manuel Vilares, Leandro Rodríguez Liñares and Miguel A. Alonso (eds.), *Proc. of Second International Workshop on Tabulation in Parsing and Deduction (TAPD 2000)*, pp. 101-112, Vigo, Spain, September 2000.
4. Miguel A. Alonso, Mark-Jan Nederhof, and Eric de la Clergerie. Tabulation of automata for tree adjoining languages. *Grammars* **3** (2000) 89–110.
5. Tilman Becker and Dominik Heckmann. Recursive matrix systems (RMS) and TAG. In *Proc. of Fourth International Workshop on Tree Adjoining Grammars and Related Frameworks (TAG+4)*, pages 9–12, Philadelphia, PA, USA, August 1998.
6. Pierre Boullier. A generalization of mildly context-sensitive formalisms. In *Proc. of Fourth International Workshop on Tree Adjoining Grammars and Related Frameworks (TAG+4)*, pages 17–20, Philadelphia, PA, USA, August 1998.
7. Eric de la Clergerie and Miguel A. Alonso. A tabular interpretation of a class of 2-Stack Automata. In *COLING-ACL'98, 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics, Proceedings of the Conference*, volume II, pages 1333–1339, Montreal, Quebec, Canada, August 1998. ACL.
8. Eric de la Clergerie, Miguel A. Alonso, and David Cabrero. A tabular interpretation of bottom-up automata for TAG. In *Proc. of Fourth International Workshop on Tree-Adjoining Grammars and Related Frameworks (TAG+4)*, pages 42–45, Philadelphia, PA, USA, August 1998.
9. Eric de la Clergerie and François Barthélemy. Information flow in tabular interpretations for generalized Push-Down Automata. *Theoretical Computer Science*, 199(1–2):167–198, 1998.
10. Gerald Gazdar. Applicability of indexed grammars to natural languages. In U. Reyle and C. Rohrer, editors, *Natural Language Parsing and Linguistic Theories*, pages 69–94. D. Reidel Publishing Company, 1987.
11. John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Series in Computer Science. Addison-Wesley Publishing Company, Reading, Massachussetts, USA, 1979.

12. Aravind K. Joshi and Yves Schabes. Tree-adjoining grammars. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages. Vol 3: Beyond Words*, chapter 2, pages 69–123. Springer-Verlag, Berlin/Heidelberg/New York, 1997.
13. Bernard Lang. Towards a uniform formal framework for parsing. In Masaru Tomita, editor, *Current Issues in Parsing Technology*, pages 153–171. Kluwer Academic Publishers, Norwell, MA, USA, 1991.
14. Mark-Jan Nederhof. Linear indexed automata and tabulation of TAG parsing. In *Proc. of First International Workshop on Tabulation in Parsing and Deduction (TAPD'98)*, pages 1–9, Paris, France, April 1998.
15. Mark-Jan Nederhof. Models of tabulation for TAG parsing. In *Proc. of the Sixth Meeting on Mathematics of Language (MOL 6)*, pages 143–158, Orlando, Florida, USA, July 1999.
16. Gisela Pitsch. $LL(k)$ parsing of coupled-context-free grammars. *Computational Intelligence* **10** (1994) 563–578.
17. C. Pollard. *Generalized Phrase Structure Grammars, Head Grammars and Natural Language*. PhD thesis, Stanford University, 1984.
18. Owen Rambow. *Formal and Computational Aspects of Natural Language Syntax*. PhD thesis, University of Pennsylvania, 1994. Available as IRCS Report 94-08 of the Institute of Research in Cognitive Science, University of Pennsylvania.
19. Yves Schabes and K. Vijay-Shanker. Deterministic left to right parsing of tree adjoining languages. In *Proc. of 28th Annual Meeting of the Association for Computational Linguistics*, pages 276–283, Pittsburgh, Pennsylvania, USA, June 1990. ACL.
20. Stuart M. Shieber, Yves Schabes, and Fernando C. N. Pereira. Principles and implementation of deductive parsing. *Journal of Logic Programming* **24** (1995) 3–36.
21. M. Steedman. Combinators and grammars. In R. Oehrle, E. Bach, and D Wheeler, editors, *Categorial Grammars and Natural Language Structures*, pages 417–442. Foris, Dordrecht, 1986.
22. K. Vijay-Shanker. *A Study of Tree Adjoining Grammars*. PhD thesis, University of Pennsylvania, January 1988. Available as Technical Report MS-CIS-88-03 LINC LAB 95 of the Department of Computer and Information Science, University of Pennsylvania.
23. K. Vijay-Shanker and David J. Weir. The equivalence of four extensions of context-free grammars. *Mathematical Systems Theory* **27** (1994) 511–545.
24. K. Vijay-Shanker, David J. Weir, and Aravind K. Joshi. On the progression from context-free to tree adjoining languages. In Alexis Manaster-Ramer, editor, *Mathematics of Language*, pages 389–401. John Benjamins Publishing Company, Amsterdam/Philadelphia, 1987.