Acta Informatica manuscript No.
(will be inserted by the editor)

Fast Deterministic Parsers for Transition Networks

Angelo Borsotti · Luca Breveglieri · Stefano Crespi Reghizzi · Angelo Morzenti

Received: date / Accepted: date

Abstract Extended BNF grammars (EBNF) allow regular expressions in the right parts of their rules. They are widely used to define languages, and can be represented by recursive Transition Networks (TN) consisting of a set of finite-state machines. We present a novel direct construction of efficient shift-reduce ELR(1) parsers for TNs. We show that such a parser works deterministically if the TN is free from the classical shift-reduce and reduce-reduce conflicts of the LR(1) parsers, and from a new conflict type called *convergence conflict*. Such a novel condition for determinism is proved correct and is more general than those proposed in the past for EBNF grammars or TNs. Such ELR(1) parsers perform fewer shift moves than the equivalent LR(1) parsers. A simple optimization of the reduction moves is described.

Keywords extended grammar · EBNF · syntax chart · ELR(1) · LR(1) · syntax analysis · shift-reduce · bottom-up parsing · parsing conflicts · parser performance

1 Introduction

The Extended Backus-Naur Form grammars (EBNF) are widely used for language specification, because the presence of regular expressions (RE) in the right-hand sides of the rules makes these grammars more readable than the pure context-free (CF) ones (here referred to as BNF). Such grammars are often represented by graphs called *Transition Networks* (TN), pioneered, e.g., by [7], or also by the dual

A preliminary account of this research is in [4].

A. Borsotti
E-mail: angelo.borsotti@mail.polimi.it

L. Breveglieri
DEIB - Politecnico di Milano, E-mail: luca.breveglieri@polimi.it

S. Crespi Reghizzi
DEIB - Politecnico di Milano and CNR-IEIIT, E-mail: stefano.crespireghizzi@polimi.it

A. Morzenti
DEIB - Politecnico di Milano, E-mail: angelo.morzenti@polimi.it

graphs called syntax diagrams (or charts) [15]. A TN is essentially a collection of finite automata (FA), typically deterministic (DFA), each one acting as recognizer of the regular expression associated to a nonterminal symbol of the grammar. In the BNF case, since the right-hand sides are trivial REs that do not contain star operators or inner alternatives, the state-transition graph of every FA in the TN is a tree graph, where the only branching node is the initial one.

This paper contributes a new method for constructing fast deterministic parsers starting from general TNs.

The Knuth's LR(1) condition [17] characterizes the grammars of the deterministic CF language family. Knuth's classical parser-generation algorithm, to be referred to as the *standard* method, maps a BNF grammar to a shift-reduce parser, i.e., a deterministic pushdown automaton (DPDA). However, if the language is specified by an EBNF grammar or by a general TN, the standard method by itself does not suffice. The EBNF rules must be first converted to BNF or, if the language is specified by a TN, their graphs must be made tree-like. Therefore, we call *indirect* such approaches to parser construction. Although the above transformations are straightforward, as they essentially involve transforming iterations into left- or right-recursive rules, there are practical drawbacks. First, the indirect LR(1) parser obtained from such a BNF grammar is not in a one-to-one relation with the official EBNF grammar (or syntax diagram) of the language. This may cause a misalignment of semantic actions or of diagnostics issued by the compiler. Second, the indirect parser pays an overhead (discussed in Sect. 4.1) corresponding to the additional parser moves caused by the grammar transformation.

To eliminate both inconveniences, we present a new direct method, called ELR(1), for constructing a shift-reduce parser from a general TN, i.e., not restricted to the tree graph case. The problem is not new, but none of the past attempts to solve it (discussed in Sect. 4.2) has been completely successful, either because these attempts impose overly restrictive hypotheses, or because they introduce various complications that have not been rigorously analyzed. For such a reason, the classical book [11] on parsing methods does not deal with the LR parsers for EBNF grammars, since "the implementation of the iterative interpretation is far from trivial". Similar is also the conclusion of the accurate and extensive survey [14], where the EBNF grammars are called extended context-free grammars (ECFG) or regular right-part grammars (RRPG):

> Any EBNF rule may be converted into a few equivalent BNF rules by replacing each regular expression (RE) with a BNF subgrammar. Yet the known methods for directly checking whether the resulting BNF grammar is deterministic, without actually building and analyzing it, are difficult or overly restricted, and none of them has reached consensus. What has been published on the LR-like parsing theory is so complex that not many feel tempted to use it ... Such a simplistic *resumé* does not do justice to the great efforts that have gone into the research and implementation of the methods described. But it is a striking phenomenon that the ideas behind the recursive descent parsing of ECFGs can be grasped and applied immediately, whereas most literature on the LR-like parsing of RRPGs is very difficult to access. Given the developments in computing power and software engineering, and the practical importance of ECFGs and RRPGs, a uniform and coherent treatment of the subject seems in order.

Such a treatment is our objective: a practical and general direct construction for such parsers. Most proposed methods for the construction of deterministic parsers when TNs have bifurcating paths and cyclic paths, as well as recursive invocations, add extra bookkeeping to the standard LR(1) method, to manage the

reduction of strings of unbounded length. Our construction does not introduce extra bookkeeping and the parser can be implemented very efficiently.

To ensure that the parser is deterministic, every parser generator needs to check that the input grammar or the TN satisfies a condition, such as the LR(1) condition for BNF grammars, which detects the violations, named *shift-reduce* (SR) and *reduce-reduce* (RR) conflicts. We formulate a new ELR(1) condition on TNs by adding a new conflict type called *convergence* conflict (CV). To prove that our ELR(1) parser is correct, we show that it is equivalent to the LR(1) parser indirectly obtained by the standard method, after a straightforward conversion of the the original TN into a collection of tree graphs, which are in one-to-one correspondence with a BNF grammar. Such a conversion creates a new nonterminal symbol for each internal node of the TN graphs.

We have compared both the descriptional and the computational complexity of direct ELR(1) parsers vs. indirect LR(1) parsers, and we have found that the direct ones are better. An implementation of our parser generator is freely available at `http://github.com/FLC-project/ELRparser`.

Sect. 2 sets the terminology and notation for (EBNF) grammars and TNs. Sect. 3 contains the ELR(1) condition, the main property and its proof, and the direct construction of shift-reduce parsers. Sect. 4 compares the descriptive and computational complexities of the new parsers and of the standard indirect ones, before and after an effective optimization for speeding-up the reduction operations. It also discusses previous related research.

2 Basic definitions

The *terminal* alphabet is denoted by Σ and the empty word by ε. The special character "$\dashv$", included in Σ, is used as a marker for the parser to identify the end of the input word.

Let V be an alphabet and let f be a mapping defined on V such that, for every $a \in V$, the image $f\,(a)$ is a non-empty language. For two mappings f and g, we denote their composition $f\,\big(\,g\,(a)\,\big)$ by $f\,g$, we denote the iteration $f \ldots f$ (n times) by f^n, and in particular the identity mapping by f^0. As usual, a substitution (e.g., in [9]) is a mapping from V into languages, which is extended to the words over V by $f\,(\varepsilon) = \varepsilon$ and $f\,(\,a_1 \ldots a_n\,) = f\,(a_1) \ldots f\,(a_n)$, with $a_i \in V$, and to the languages over V by $f\,(L) = \bigcup_{w\,\in\,L} f\,(w)$. It is said to be a *substitution* over V if $f\,(a)$ is a language over V, for all $a \in V$.

Let f be a substitution over V and let L be a language over V. The *iterated substitution* f^* of L is defined by

$$f^*\,(L) = \bigcup_{n\,=\,0}^{\infty} f^n\,(L)\;.$$

A *non-deterministic finite automaton* (NFA) M is defined by a tuple $(\,V,\,Q,\,\delta,\,q_0,\,F\,)$, where V is an alphabet and Q is a state set, with initial state $q_0 \in Q$ and final states $F \subseteq Q$. The state-transition relation of M (often represented as an edge-labeled graph) is $\delta \subseteq Q \times V \times Q$. Relation δ is extended as usual to the domain $Q \times V^* \times Q$. In the state-transition graph a path from the state q_0 to a state $q \in F$ is called *accepting*, and the concatenation of the edge labels of such a path forms

an accepted word. The *language recognized* by M, with $L(M) \subseteq V^*$, is the set of all the accepted words. The automaton is deterministic (DFA) if the relation δ is functional from $Q \times V$ to Q, i.e., $\delta\colon Q \times V \to Q$. A DFA such that no edge of δ enters the initial state is called *non-reentrant*. Clearly every DFA can be made non-reentrant by adding one extra state and at most $|V|$ edges.

2.1 Transition network

A syntax diagram is formalized by means of a set of DFAs, to be called a transition network (TN). Let V be a bipartite alphabet, i.e., $V = \Sigma \cup V_N$ and $\Sigma \cap V_N = \emptyset$, where set Σ is a terminal alphabet and set $V_N = \{ S, A, B, \ldots \}$ is a nonterminal alphabet that includes a distinguished symbol S, called axiom.

Definition 1 (transition network) A *transition network* TN is a finite collection of DFAs, called *machines*[1], $\mathcal{M} = \{ M_S, M_A, M_B, \ldots \}$. The machine denoted by M_S is distinguished as the *starting machine*. The set of machine names, $V_N = \{ S, A, B, \ldots \}$, is called *nonterminal alphabet*. Every machine $M_A \in \mathcal{M}$ is defined by a tuple $M_A = (V, Q_A, \delta_A, 0_A, F_A)$ where the *input alphabet* V is the union $V = V_N \cup \Sigma$ of the nonterminal and terminal alphabets. We assume that every machine is non-reentrant.

The state set of the TN $\mathcal{M}$ is $Q = \bigcup_{M_A \in \mathcal{M}} Q_A$. Without loss of generality, by assuming that for any two distinct machines M_A and M_B it holds $Q_A \cap Q_B = \emptyset$, we may safely define the TN transition function as $\delta = \bigcup_{M_A \in \mathcal{M}} \delta_A$ and write δ instead of δ_A at no risk of confusion.

The (regular) language over V recognized by machine M_A starting in some state q_A and ending in a final state is

$$R(M_A, q_A) = \left\{ x \in V^* \mid \delta(q_A, x) \in F_A \right\} .$$

The TN $\mathcal{M}$ defines a *substitution* $f_{\mathcal{M}}$, or only f if $\mathcal{M}$ is understood, by means of

$$\begin{cases} f_{\mathcal{M}}(a) = \{ a \} & a \in \Sigma \\ f_{\mathcal{M}}(A) = R(M_A, 0_A) & A \in V_N . \end{cases}$$

The (context-free) language L over Σ recognized by a machine M_A starting in some state q_A, is defined by means of the iterated substitution f^* as

$$L(M_A, q_A) = f^*\left(R(M_A, q_A)\right) \cap \Sigma^* .$$

We may simply write $L(q_A)$ instead of $L(M_A, q_A)$ since the state name q_A identifies the machine. Then the language defined by the TN, $L(\mathcal{M}) \subseteq \Sigma^*$, is

$$L(\mathcal{M}) = L(M_S, 0_S) ,$$

therefore it follows $L(\mathcal{M}) = f^*_{\mathcal{M}}(S) \cap \Sigma^*$. □

[1] We reserve the term "automata" to the push-down machines of the parsers.

We briefly motivate the assumptions of Def. 1 about finite-state machines. We have chosen deterministic machines because, to the best of our knowledge, the syntax diagrams used in the language reference manuals are deterministic. A different choice would moderately complicate the parser construction algorithm without much benefit. Similarly, the non-reentrance hypothesis is unnecessary, but it simplifies the (implementation of the) reduction moves of the parser as any TN path never revisits the initial state. Every machine can be so normalized by adding a new initial state and a few outgoing arcs, with a negligible overhead.

The transition function δ of a TN $\mathcal{M}$ may be partial and the machines of $\mathcal{M}$ are not necessarily minimal. For every machine $M_A \in \mathcal{M}$, we also assume the following: M_A is trim, M_A is recursively reachable from M_S and it holds $L\,(\,M_A,\,0_A\,) \neq \emptyset$.

The set of *starting characters* for a set of words $L \subseteq \Sigma^*$ is defined as

$$First(L) = \left\{\ a \in \Sigma \mid \quad \left(\,a\,\Sigma^*\ \cap\ L\,\right) \neq \emptyset\ \right\}\ .$$

Given a TN and a state $q \in Q$ thereof, we define $First(q) = First\left(\,L\,(q)\,\right)$.

2.2 BNF and EBNF grammar

A BNF (i.e., context-free) *grammar* G is a 4-tuple $(\,\Sigma,\,V_N,\,P,\,S\,)$, where the *nonterminal* alphabet is V_N, the set of *rules* is P and the *axiom* is $S \in V_N$. A *grammar symbol* is an element of the *total alphabet* $V = \Sigma\ \cup\ V_N$. In a rule $A \to \alpha$, the nonterminal $A \in V_N$ is the *left part* and the string $\alpha \in V^*$ is the *right part*. A grammar is *right-linear* if every right part α has the form $a\,B$ with $a \in \Sigma$ and $B \in V_N$, or if α is the empty word ε.

We also use *Extended* BNF grammar rules containing regular expressions. An EBNF grammar G has exactly one rule $A \to \alpha$ for each nonterminal $A \in V_N$. The right part α is a *regular expression* (RE) over the total alphabet V, with the usual operators: concatenation, union (represented by a vertical bar "$|$"), Kleene star "$*$" and cross "$+$", and parentheses. The language over V defined by the RE α is called the *regular language associated* with A and is denoted R_A, or also $R\,(\alpha)$.

We observe that a BNF grammar is also an EBNF one, provided we group all the alternative rules $A \to \alpha_1$, $A \to \alpha_2$, $\ldots$, $A \to \alpha_n$ of BNF into one EBNF rule $A \to \alpha$, with $\alpha = \alpha_1\ \mid\ \alpha_2\ \mid\ \ldots\ \mid\ \alpha_n$.

For an EBNF grammar, an *immediate derivation* is a relation $\Rightarrow$ over $V^* \times V^*$ such that $z \Rightarrow z'$ if $z = u\,A\,v$, $z' = u\,w\,v$, $A \to \alpha \in P$ and $w \in R_A$. A *derivation* is its reflexive and transitive closure, denoted $\overset{*}{\Rightarrow}$. A derivation is *rightmost* if at any step it holds $v \in \Sigma^*$. The reverse relation of an immediate derivation $w \Rightarrow z$ is named *reduction* and is denoted by $z \rightsquigarrow w$.

We assume that all grammars are *reduced*: for every nonterminal $A \in V_N$ there exist derivations $S \overset{*}{\Rightarrow} u\,A\,v$ and $A \overset{*}{\Rightarrow} w$, where $u, v \in V^*$ and $w \in \Sigma^*$. This implies that for every $A \in V_N$ the regular language R_A is not empty.

The language generated by grammar G starting from a nonterminal $A \in V_N$ is $L\,(G,\,A\,) = \left\{\ x \in \Sigma^* \,\middle|\, A \underset{G}{\overset{*}{\Rightarrow}} x\ \right\}$. The *language generated* by G is $L\,(G) = L\,(G,\,S\,)$. A language that contains the empty word ε is called *nullable*. Since grammar G is reduced, for every $A \in V_N$ the regular language R_A contains some string (as said before), from which some terminal string derives, therefore $L\,(G,\,A\,) \neq \emptyset$.

It is well-known that the EBNF grammars have the same generative capacity as the BNF ones, namely they generate the context-free languages. Two grammars are *equivalent* if they define the same language.

2.3 Equivalence of TN and grammar

Given a TN $\mathcal{M}$, we may write different EBNF or BNF grammars for the language of $\mathcal{M}$, and we need to examine two cases. First, we consider the EBNF grammars, to be called *associate*, that reflect the structure of $\mathcal{M}$ as a collection of finite-state machines, i.e., that have a bijective correspondence between their nonterminal symbols and the machines of $\mathcal{M}$.

Definition 2 (TN and EBNF) Let $\mathcal{M} = \{ M_S, M_A, M_B, \ldots \}$ be a TN over $V = \Sigma \cup V_N$. An EBNF grammar $G = \left(\Sigma, V'_N, P', S' \right)$ is said to be *associated* with $\mathcal{M}$ if $V'_N = V_N$, $S' = S$ and, for every machine $M_A \in \mathcal{M}$ and rule $A \to \alpha \in P'$, it holds $R\,(\,M_A, 0_A\,) = R_A$, i.e., the language over V recognized by machine M_A is the same as that defined by the RE α. □

It is obvious that a TN and any associated grammar define the same context-free language for each machine and corresponding nonterminal.

Lemma 1 (equivalence of TN and EBNF) *For every* TN $\mathcal{M}$ *and grammar* G *associated with* $\mathcal{M}$*, and for every machine* $M_A \in \mathcal{M}$*, it holds*

$$f^*_{\mathcal{M}}\left(R\,(\,M_A, 0_A\,)\,\right) = \left\{ \; x \in V^* \;\middle|\; A \overset{*}{\underset{G}{\Rightarrow}} x \; \right\} .$$

Therefore, it also holds $L\,(\,M_A, 0_A\,) = L\,(\,G, A\,)$ *and* $L\,(\,\mathcal{M}\,) = L\,(\,G\,)$.

In other words, a TN represents the class of all the equivalent EBNF grammars such that they use the same nonterminal symbols and the regular languages corresponding to nonterminal classes of identical name are equal. To construct an associated grammar, by means of well-known methods we compute a regular expression R_A for each machine M_A and we create the rule $A \to R_A$. We illustrate with an example.

*Example 1 (*TN *and derivation)* Fig. 1.a shows a TN $\mathcal{M} = \{ M_S \}$ with just one machine. The EBNF grammar listed in Fig. 1.b is associated with $\mathcal{M}$ since $R_S = (\,\varepsilon \mid b\,)\,\left(a\;(\,b \mid S\,c\,)\right)^* = R\,(\,M_S, 0_S\,)$. Therefore the correspondence between derivations and iterated language substitutions (Lemma 1) is one-to-one. For instance, we have

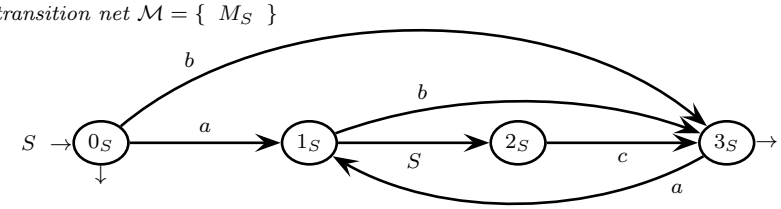
$$f_{\mathcal{M}}\,(S) \ni a\,S\,c\,a\,b \qquad \text{and} \qquad f^2_{\mathcal{M}}\,(S) \ni a\,b\,c\,a\,b$$

and since $a\,S\,c\,a\,b \in R_S$ and $b \in R_S$, there exists a derivation

$$S \Rightarrow a\,S\,c\,a\,b \Rightarrow a\,b\,c\,a\,b\,. \qquad \square$$

We observe that for many TNs it is impossible to find an associated grammar that is in BNF. This simply comes from the fact that for some machine of the TN, defining the regular language of the machine by means of a BNF grammar may require more than one nonterminal symbol [12].

(a) *transition net* $\mathcal{M} = \{\ M_S\ \}$



(b) *an associated* EBNF *grammar* $\quad G\colon\ S \to (\ \varepsilon\ \mid\ b\ )\ (\ a\ (\ b\ \mid\ S\ c\ )\ )^*$

(c) *right-linearized grammar of* $\mathcal{M}$ $\quad \hat{G}\ \begin{cases} 0_S \ \to\ a\ 1_S\ \mid\ b\ 3_S\ \mid\ \varepsilon & 2_S \ \to\ c\ 3_S \\ 1_S \ \to\ b\ 3_S\ \mid\ 0_S\ 2_S & 3_S \ \to\ a\ 1_S\ \mid\ \varepsilon \end{cases}$

Fig. 1 Part (a): transition network $\mathcal{M}$ with one machine M_S. Part (b): an associated EBNF grammar G with one rule. Part (c): corresponding right-linearized grammar $\hat{G}$.

Second, we focus on a special BNF grammar type, called right-linearized (RLZ), that precisely reproduces the structure of a TN, using as many nonterminal symbols as there are states in the TN. Such grammars will be used to compare our parsers with the standard LR(1) ones.

Since for every NFA there exists an equivalent (BNF) right-linear grammar, which encodes the transition arcs as rules, we consider the union of all such right-linear grammars for the machines of a given TN $\mathcal{M}$.

Definition 3 (corresponding RLZ grammar) Consider a TN $\mathcal{M}$. For each machine $M_A = (\ V_N\ \cup\ \Sigma,\ Q_A,\ \delta_A,\ 0_A,\ F_A\ ) \in \mathcal{M}$, the corresponding *right-linearized* (RLZ) *grammar* is the BNF grammar $\hat{G}_A = \big(\ \Sigma,\ Q_A \cup \{\ 0_B \mid \quad B \in V_N\ \}\ ,\ P_A,\ 0_A\ \big)$ with the following rules

$$\begin{aligned} (\ q_A \to a\ r_A\ ) \in P_A \quad &\Leftrightarrow \quad \Big(\ p_A \xrightarrow{a} r_A\ \Big) \in \delta_A \qquad \text{where } a \in \Sigma \\ (\ q_A \to 0_B\ r_A\ ) \in P_A \quad &\Leftrightarrow \quad \Big(\ p_A \xrightarrow{B} r_A\ \Big) \in \delta_A \qquad \text{where } B \in V_N \\ (\ q_A \to \varepsilon\ ) \in P_A \quad &\Leftrightarrow \qquad q_A \in F_A\ . \end{aligned}$$

The *right-linearized grammar* that corresponds to a TN $\mathcal{M}$ is denoted $\hat{G}_{\mathcal{M}} = (\ \Sigma,\ Q,\ P,\ 0_S\ )$, where

$$P = \bigcup_{M_A \in \mathcal{M}} P_A\ .$$

When no ambiguity arises, we drop the subscript $\mathcal{M}$ and we write $\hat{G}$ instead. □

Notice that in general grammar $\hat{G}$ is not right-linear, since the right part of a rule may be in $V_N\,V_N$. From the assumptions on TNs in Sect. 2.1, it follows that $\hat{G}$ is reduced. For illustration, see the RLZ grammar in Fig. 1.c.

Lemma 2 (equivalence of TN and RLZ grammar) *Let* $\mathcal{M}$ *be a* TN *and* $\hat{G}$ *be the corresponding* RLZ *grammar. Then it holds* $L\ (\mathcal{M}\ ) = L\ \Big(\hat{G}\Big)$.

We omit the obvious proof by induction.

3 From TN to Shift-Reduce Parser

This section starts with the reformulation for TNs of the basic notions that support the standard theory of LR(1) parsers. Subsequently we present the main result: a new algorithm that directly constructs the parser state-transition graph, to be called ELR(1) *graph*, starting from a TN. On such a graph we identify the situations that cause a parsing conflict: shift-reduce, reduce-reduce and convergence conflict. We call ELR(1) TNs the conflict-free ones. Then, since a convergence conflict corresponds to a new situation not present in standard parsers, we briefly contrast the two cases. The important result (Th. 1) proves the equivalence of the ELR(1) condition for a TN and the LR(1) condition for the corresponding right-linearized grammar. Finally, we present and exemplify the shift-reduce parsing algorithm, i.e., a deterministic push-down automaton controlled by the ELR(1) graph.

3.1 ELR(1) condition for TN

It is necessary to reformulate for a TN $\mathcal{M}$ the classical notions pertinent to the LR(1) analysis, originally stated in [17] for a BNF grammar, assuming the reader has some familiarity with them.

First, it is obvious that now a so called *dotted rule* (of a BNF grammar) should be replaced by a machine state (of a TN), e.g., in Fig. 1.a the dotted rule $S \to \bullet\, b$ is synonymous with the machine state 0_S.

Definition 4 (items and item sets) For a TN $\mathcal{M} = \{\ M_S,\ M_A,\ \dots\ \}$, an ELR(1) *item* is a pair of the form $\langle\, q,\, \pi \,\rangle \in Q \times \wp\,(\,\Sigma\,)$, such that $\pi \neq \emptyset$ and if q is the initial state 0_S of M_S then it holds $\dashv \in \pi$. Set π is called the *look-ahead (set)* of state q.

An *item set* I is a finite non-empty collection of items $I = \{\ \langle\, q_1,\, \pi_1 \,\rangle, \langle\, q_2,\, \pi_2 \,\rangle, \dots\ \}$. If two (or more) items $\langle\, q_i,\, \pi_i \,\rangle, \langle\, q_j,\, \pi_j \,\rangle \in I$ have identical states, i.e., $q_i = q_j$, they are usually coalesced into one item $\langle\, q_i,\, \pi_i \cup \pi_j \,\rangle$ with a unified look-ahead set. □

We recall that in the standard approach to LR(1) analysis an item set represents a parser state, a *p-state* for short.

The standard closure function from item sets to item sets [17] is next reformulated for the item sets of a TN instead of a BNF grammar.

Definition 5 (closure function) Let $\mathcal{M}$ be a TN and I be an item set of $\mathcal{M}$. Let 0_A be the initial state of machine $M_A \in \mathcal{M}$. The function

$$\mathit{closure}\colon \wp\left(\, Q \times \wp\,(\,\Sigma\,)\,\right) \to \wp\left(\, Q \times \wp\,(\,\Sigma\,)\,\right)$$

recursively computes the smallest set such that

$$\mathit{closure}\,(I) = I \ \cup \left\{\ \langle\, 0_A,\, \sigma \,\rangle \ \middle|\ \begin{array}{l} \langle\, q,\, \pi \,\rangle \in \mathit{closure}\,(I) \\ \textbf{and}\ \left(\, q \xrightarrow{A} r \,\right) \in \delta \\ \textbf{and}\ \sigma = \mathit{First}\left(\, L\,(r) \cdot \pi \,\right) \end{array} \right\}. \qquad \square$$

Notice that the look-ahead σ includes π if the language $L\,(r)$ is nullable.

Given a TN $\mathcal{M}$, we present the new direct ELR(1) construction of the finite state-transition function or graph ϑ of the DFA that controls the parser, the latter

being a DPDA. Such a DFA will be referred to as ELR(1) *graph*, to shorten its traditional name "recognizer of viable LR(1) prefixes". Each state is an item set and is called a *parser state* (*p-state*) to avoid confusion with the states of TN machines. The construction involves three phases:

1. from the TN $\mathcal{M}$, construct the ELR(1) graph;
2. check the forthcoming ELR(1) condition, which makes sure that the automaton is deterministic by excluding all conflicts; there are three conflict types: the standard shift-reduce (SR) and reduce-reduce (RR) conflicts, and the new *convergence* conflict (CV);
3. if the test in (2.) is passed, construct the DPDA, i.e., the parser.

Graph construction The ELR(1) graph of a TN $\mathcal{M}$ is a DFA $\mathcal{P} = (V, R, \vartheta, I_0, R)$, with alphabet $V = \Sigma \cup V_N$. The set R contains the p-states, each of which is an item set; all the p-states are final. The state-transition function[2] is $\vartheta\colon R \times V \to R$. The graph is constructed by Alg. 1, starting from $\mathcal{M}$, and outputting R and ϑ.

Algorithm 1: Construction of the graph $\mathcal{P}$ for a TN $\mathcal{M}$, i.e., the p-state set $R = \{ I_0, I_1, \ldots \}$ and the transition function $\vartheta\colon R \times V \to R$.

Input: transition network (TN) $\mathcal{M}$
Output: graph $\mathcal{P} = (R, \vartheta)$ of $\mathcal{M}$
// initial, base and closed p-states: I_0, I' and I''
// consolidated and growing p-state sets: R and R'
// consolidated transition set: ϑ
$I_0 := closure(\{ \langle 0_S, \{ \dashv \} \rangle \})$ // create initial p-state
$R' := \{ I_0 \}$ // initialize p-state set
$\vartheta := \emptyset$ // initialize transition set
repeat // build graph nodes & trans
  $R := R'$ // consolidate p-state set
  foreach $(I \in R$ **and** $X \in V)$ **do** // scan p-states & symbols
    // compute the base of the (new) successor p-state
    $I' := \{ \langle \delta(p, X), \pi \rangle \mid \langle p, \pi \rangle \in I$ **and** $\delta(p, X)$ is defined $\}$
    if $(I' \neq \emptyset)$ **then** // proceed with p-state
      $I'' := closure(I')$ // close p-state base
      $R' := R' \cup \{ I'' \}$ // add p-state to graph
      $\vartheta := \vartheta \cup \left\{ \text{arc} \left(I \xrightarrow{X} I'' \right) \right\}$ // add trans to graph
    // else p-state I does not have a successor for symbol X
until $(R' = R)$ // exit if graph unchanged

For the item sets I and I', and for the grammar symbol $X \in V$, if it holds $\vartheta(I, X) = I'$ (we equivalently write "arc $\left(I \xrightarrow{X} I' \right) \in \vartheta$") we say that the function ϑ defines a *shift* from I to I' under X. The shift is qualified as *terminal* or *nonterminal* according to the nature of symbol X. From Alg. 1 it is clear that it holds $\vartheta(I, X) = I'$ if, and only if, the TN has the arc $p_A \xrightarrow{X} q_A$ for some items $\langle p_A, \pi \rangle, \langle q_A, \pi \rangle \in I, I'$. In this case we also say that the item $\langle p_A, \pi \rangle$ in the item set I *shifts under symbol* X.

[2] This function is analogous to the "GOTO" component of the canonical parsing table described in the classical textbooks on compilers such as [1].

Alg. 1 is iterative and clearly terminates. Since all the machines of the TN $\mathcal{M}$ are non-reentrant (i.e., the initial state does not have ingoing arcs), see Sect. 2 and Def. 1, every p-state $I \in R$ is made of two disjoint subsets of items, respectively called *base* and *closure*, which are defined as follows

$$I_{|\,base} = \{\, \langle\, q,\, \pi \,\rangle \in I \mid \quad \text{state } q \text{ is not initial} \,\}$$
$$I_{|\,closure} = \{\, \langle\, q,\, \pi \,\rangle \in I \mid \quad \text{state } q \text{ is initial} \,\}\ .$$

In Fig. 2.b we represent the base and the closure one on top of the other, separated by a double line. Every p-state, but the initial one I_0, has a non-empty base, whereas the closure may be empty in some p-states.

ELR(1) *condition* Intuitively a p-state has a conflict if the parser, on finding itself in that p-state, is unable to choose a move deterministically. First, we recall the standard conflicts of the LR(1) parsers, since they concern the ELR(1) parsers too, then we formalize a new conflict type that may only occur in the latter ones.

Definition 6 (shift-reduce and reduce-reduce conflicts) A p-state I has a *Shift-Reduce* (SR) *conflict* if

$$\exists \text{ item } \langle\, q,\, \pi \,\rangle \in I \text{ such that state } q \text{ is final}$$
$$\text{and } \exists \text{ arc } \left(I \xrightarrow{a} I' \right) \in \vartheta \text{ with symbol } a \in \pi.$$

A p-state I has a *Reduce-Reduce* (RR) *conflict* if

$$\exists \text{ items } \langle\, q,\, \pi \,\rangle\,,\, \langle\, r,\, \rho \,\rangle \in I, \text{ with } q \neq r, \text{ such that}$$
$$\text{states } q \text{ and } r \text{ are final, and it holds } \pi \cap \rho \neq \emptyset. \qquad \square$$

In an ELR(1) parser, if in a p-state I two (or more) items shift under the same symbol, then determinism may be defeated. This situation is next analyzed to pinpoint the dangerous case.

Definition 7 (convergence conflict) Let I be a p-state such that for a symbol $X \in V$ the transition function $\vartheta\,(I,\, X)$ is defined. We say that

1. the transition $\vartheta\,(I,\, X)$ is *multiple (double or more)* if

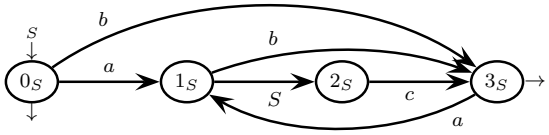
$$\exists \text{ items } \langle\, q,\, \pi \,\rangle\,,\, \langle\, r,\, \rho \,\rangle \in I, \text{ with states } q \neq r,$$
$$\text{such that } \exists \text{ arcs } \left(q \xrightarrow{X} q' \right),\, \left(r \xrightarrow{X} r' \right) \in \delta,$$

   where δ is the transition function of the TN;
2. a multiple transition is *convergent* if $q' = r'$, i.e., the destination states coincide;
3. a convergent transition is *conflicting* if $\pi \cap \rho \neq \emptyset$, i.e., the look-ahead sets overlap; if so, the p-state I has a *Convergence conflict* (CV). $\square$

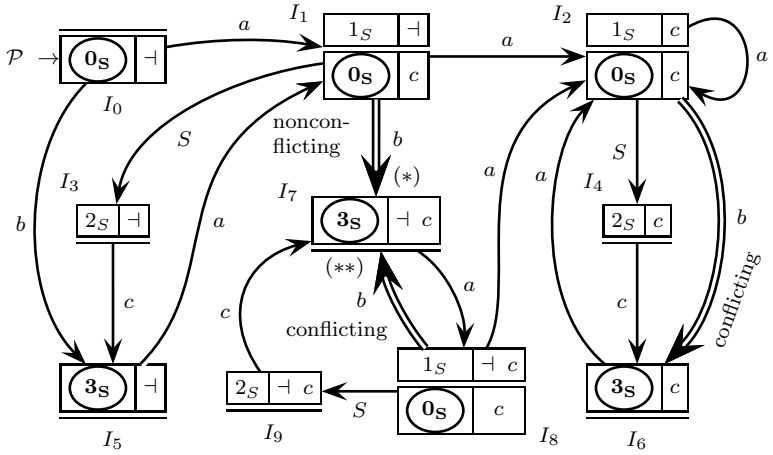
One may suspect that situations other than SR, RR and CV exist, which would introduce non-deterministic transitions in an ELR(1) parser. Yet this is not the case: the absence of the three conflict types ensures that parsing is deterministic, as we later prove.

Definition 8 (ELR(1) condition) A TN meets the ELR(1) *condition* if no p-state contains any SR, RR or CV conflict. $\square$

(a) *transition net* $\mathcal{M} = \{\ M_S\ \}$



(b) ELR(1) *graph* $\mathcal{P}$ of $\mathcal{M}$



(c) *parsing trace of word "a b a a c c" — stack shown before each reduction*

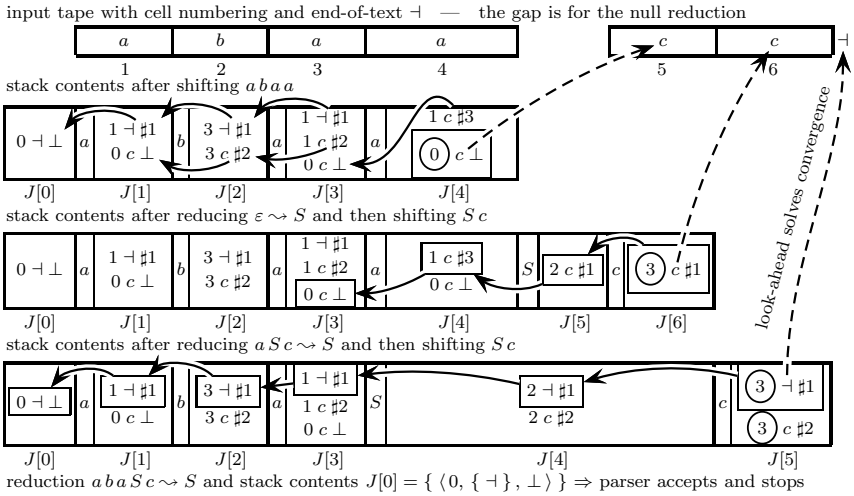


Fig. 2 Parts: (a) TN $\mathcal{M}$, (b) ELR(1) graph $\mathcal{P}$ and (c) parsing trace. In (b) the base and closure parts of each p-state are separated by a double line, convergent transitions are represented by double arrows and asterisks are for later reference (see Fig. 3).

(a) *right-linearized grammar* $\hat{G}$

$$\begin{cases} 0_S \rightarrow a\,1_S \mid b\,3_S \mid \varepsilon & 2_S \rightarrow c\,3_S \\ 1_S \rightarrow b\,3_S \mid 0_S\,2_S & 3_S \rightarrow a\,1_S \mid \varepsilon \end{cases}$$

(b) LR(1) *graph (partial) of* $\hat{G}$

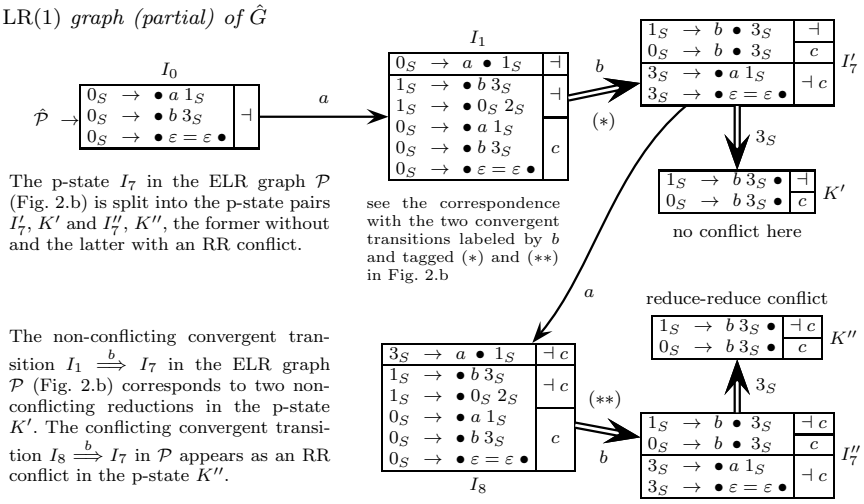


Fig. 3 Part (a): the right-linearized grammar $\hat{G}$ for the TN $\mathcal{M}$ of Fig. 2.a. Part (b): the standard LR(1) graph $\hat{\mathcal{P}}$ of $\hat{G}$, where the items are represented by dotted rules instead of machine states, to comply with tradition. Classification of p-states: K', K'' are sink reduction states and all the remaining states, apart from I_0, are intermediate.

We show the automaton graph and we discuss the ELR(1) condition for two examples.

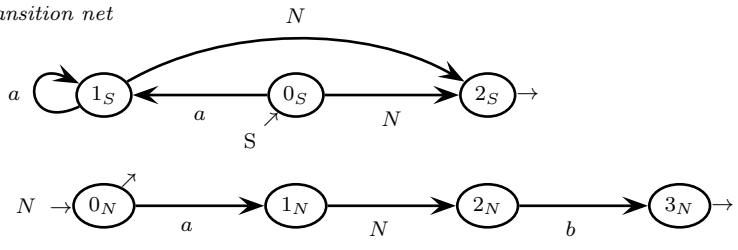
*Example 2 (*ELR(1) *condition)* For the TN in Fig. 2.a, the graph $\mathcal{P}$ is depicted in Fig. 2.b. None of the p-states has any SR or RR conflicts. The p-states I_1, I_2 and I_8 have convergent transitions. The transition $I_1 \stackrel{b}{\rightarrow} I_7$ is convergent, yet it is not conflicting since the look-ahead sets of items $\langle\, 1_S, \{\ \dashv\ \}\,\rangle$ and $\langle\, 0_S, \{\ c\ \}\,\rangle$ are disjoint. On the other hand, in the p-state I_8 (and also in the p-state I_2) the convergent transitions are conflicting because it holds $\{\ \dashv, c\ \} \cap \{\ c\ \} \neq \emptyset$ (and in I_2 the look-ahead sets of the two items are equal to $\{\ c\ \}$). It follows that this TN is not ELR(1).

The presence of a non-conflicting transition and a conflicting one, both ending in the same p-state I_7, shows that a CV conflict is a property of the source p-state, not of the destination one. □

Example 3 presents an ELR(1) TN with multiple but non-convergent transitions.

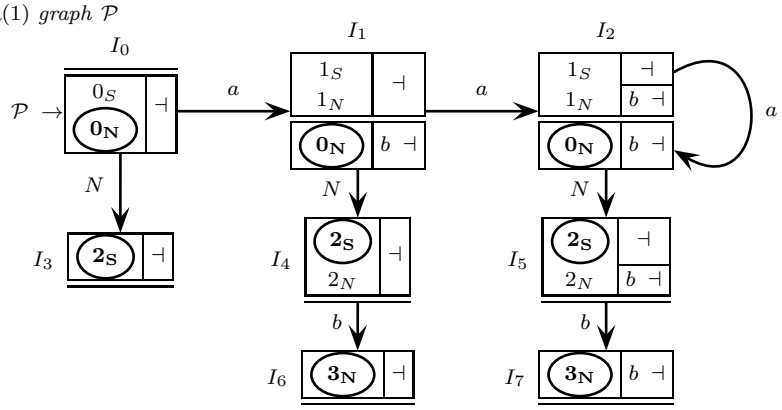
Example 3 (multiple transition) The TN shown in Fig. 4.a generates the deterministic language $\{\ a^n\, b^m \mid \quad n \geq m \geq 0\ \}$. The graph shown in Fig. 4.b has no conflict of type SR or RR. The p-state I_0 has multiple yet not convergent transitions to p-state I_1. Since none of the p-states has any conflicts, the TN meets the ELR(1) condition. □

(a) *transition net*



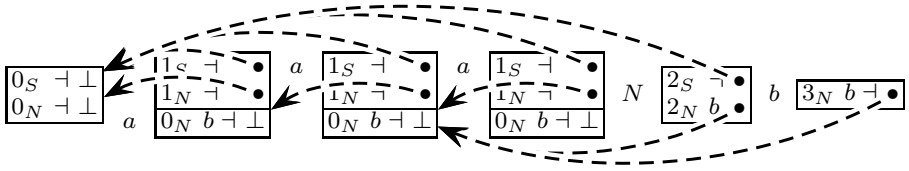
an associated EBNF grammar: $S \to a^*\ N$ $N \to a\ N\ b\ \mid\ \varepsilon$

(b) ELR(1) *graph* $\mathcal{P}$

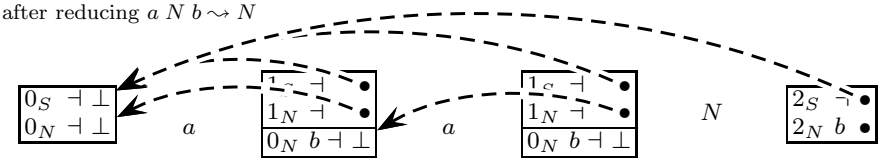


(c) *optimized parsing trace of word "a a a b"*

vector stack after shifting $a\,a\,a$, reducing $\varepsilon \rightsquigarrow N$ and shifting b



after reducing $a\ N\ b \rightsquigarrow N$



accepting configuration after reducing $a\ a\ N \rightsquigarrow S$

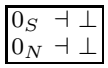


Fig. 4 Part (a): A TN and an associated EBNF grammar. Part (b): ELR(1) graph that exhibits multiple non-convergent transitions to two items in the bases of p-states I_1, I_2, I_4 and I_5. Part (c): optimized parsing of "$a\,a\,a\,b$" using Alg. 3 (parser with vector stack).

3.2 The equivalence of ELR and LR conditions

It may help to informally compare the standard LR(1) condition and Def. 8, to prepare for the proof of their equivalence in Th. 1.

ELR(1) *versus standard* LR(1) *conditions* First, we explain why a convergence conflict may not exist in a standard LR(1) parser. Then, we show that some standard RR conflicts may turn into CV conflicts in an equivalent ELR(1) parser.

Consider a BNF grammar G and the sub-grammar consisting of a nonterminal symbol A together with its alternative rules $A \to \alpha \mid \beta \mid \ldots$. Since grammar G is BNF, each sub-grammar thereof, viewed as a finite-state machine N_A, has a tree-shaped (acyclic) graph and satisfies the non-reentrance hypothesis (Def. 1), yet in general it is non-deterministic, like sub-grammar $A \to B\ c \mid B\ d$. Since a nondeterminism of this type is easily disposed of in an equivalent TN (by using as machine the DFA that recognizes $B\ (c \mid d)$), for brevity we only discuss the case that machine N_A is deterministic.

Now imagine the LR(1) graph constructed from G by the standard method. A little thought suffices to see that a p-state may not have any multiple transitions (Def. 7), hence any convergent transitions either. Therefore, the situation causing CV conflicts is impossible in a LR(1) graph.

To address the natural question of how CV conflicts originate in a TN, we consider an example where several BNF rules are equivalent to a single TN machine, and we show that a standard RR conflict in the BNF grammar turns into a CV conflict. Return in Fig. 2.b to the ELR(1) graph for the TN $\mathcal{M} = \{\ M_S\ \}$ of Fig. 2.a. Observe in Fig. 3.a the BNF grammar $\hat{G}$, which is equivalent to $\mathcal{M}$, and in Fig. 3.b the standard LR(1) graph (called GOTO graph in [1]). Notice how the CV conflict in the p-state I_8 of Fig. 2.b matches the RR conflict in the p-state K" of Fig. 3.b. However, such a correspondence between CV and RR conflicts may be not fully understood through an example, thus it will be carefully analyzed in the sequel.

The next central result supports the view that in some sense the ELR(1) condition is the most general one to ensure that a ELR(1) parser is deterministic.

Let $\hat{G}$ be the right-linearized grammar corresponding to a given TN, and let $\hat{\mathcal{P}}$ be its LR(1) graph. The alphabets of graphs $\mathcal{P}$ and $\hat{\mathcal{P}}$ slightly differ, and to simplify the notation in the coming discussion we introduce a mapping " $\hat{}$ " (not surjective) from the input alphabet of $\mathcal{P}$, i.e., from $V = \Sigma \cup V_N$, to the input alphabet of $\hat{\mathcal{P}}$, i.e., to $\Sigma \cup Q$ with $Q = \bigcup_{A \in V_N} Q_A$. The mapping " $\hat{}$ " is defined as: $\forall\, a \in \Sigma \quad \hat{a} = a$ and $\forall\, A \in V_N \quad \hat{A} = 0_A$.

For the proof of the subsequent Th. 1, we analyze the correspondence between the two graphs $\mathcal{P}$ and $\hat{\mathcal{P}}$, namely between their transition functions ϑ and $\hat{\vartheta}$, and between their p-states, to be denoted by I and $\hat{I}$. It helps to compare the graphs in Fig.s 2.b and 3.b.

We observe that since the RLZ grammar $\hat{G}$ is BNF, all the arcs of $\hat{\mathcal{P}}$ that enter the same p-state have identical labels. But since this property does not hold for $\mathcal{P}$, a p-state of $\mathcal{P}$ may be split into several p-states of $\hat{\mathcal{P}}$.

It is important to notice that any non-empty rule $X \to Y\ Z$ of grammar $\hat{G}$ has a rather restricted form: $X \in Q$, $Y \in \Sigma \cup \{\ 0_A \mid \ 0_A \in Q\ \}$ and $Z \in Q \setminus \{\ 0_A \mid \ 0_A \in Q\ \}$. It follows that the p-states $\hat{I}$ of graph $\hat{\mathcal{P}}$ can be categorized into three disjoint classes, as follows

initial p-state	*intermediate p-state* $\hat{I}$	*sink reduction p-state* $\hat{I}$
$\hat{I}_0$	such that each item in $\hat{I}_{\mid base}$ has the form $p_A \to Y \bullet q_A$	such that each item in $\hat{I}$ has the form $p_A \to Y\ q_A \bullet$

See Fig. 3.b for illustration.

For the graphs $\mathcal{P}$ and $\hat{\mathcal{P}}$ we define a correspondence relation between the sets R and $\hat{R}$ that excludes the sink reduction p-states.

Definition 9 (correspondent p-states) Consider the p-states $I \in R$ and $\hat{I} \in \hat{R}$. Let item $t = \langle\, q, \rho \,\rangle \in I$ and let set $T = \{\ \langle\, p_i \to s \bullet r, \rho_i \,\rangle\ \} \subseteq \hat{I}$. We say that

$$\begin{aligned} &\text{item } t = \langle\, q, \rho \,\rangle \textit{ corresponds} \text{ to set } T = \{\ \langle\, p_i \to s \bullet r, \rho_i \,\rangle\ \} \\ &\text{if it holds } q = r \text{ and } \rho = \bigcup_i \rho_i \end{aligned} \tag{1}$$

p-states I and $\hat{I}$ *are correspondent* if for every item $t \in I_{\mid base}$ there exists a correspondent set of items $T \subseteq \hat{I}_{\mid base}$.

The initial p-states I_0 and $\hat{I}_0$ are correspondent as their bases are empty. □

Some properties of the correspondent p-states are stated in Lemma 3.

Lemma 3 (properties of the ELR(1) and LR(1) graphs) *The following properties hold:*

1. *For every symbol $s \in V$, and for every pair of correspondent p-states I and $\hat{I}$, the p-state $\vartheta(I, s) = I'$ is defined if, and only if, the p-state $\hat{\vartheta}(\hat{I}, \hat{s}) = \hat{I}'$ is defined and is intermediate. Furthermore, the p-states I' and $\hat{I}'$ are correspondent. The mapping* (2)*, defined by* (1)

$$\left\{\ \hat{I}_0\ \right\} \cup \left\{\ \hat{I} \in \hat{R} \mid \quad \text{p-state } \hat{I} \text{ is intermediate}\ \right\} \to R \tag{2}$$

   is total and surjective.
2. *For every final but non-initial state $f_A \in Q$, it holds: item $\langle\, f_A, \lambda \,\rangle \in I$ (or more exactly $\in I_{\mid base}$) if, and only if, for some symbol $s \in V$ a correspondent p-state $\hat{I}$ contains both the correspondent set of items $\{\ \langle\, p_{A_i} \to \hat{s} \bullet f_A, \lambda_i \,\rangle\ \}$, with $\lambda = \bigcup_i \lambda_i$, and the item $\langle\, f_A \to \varepsilon \bullet, \lambda \,\rangle$.*
3. *For every final and initial state $0_A \in Q$, with $A \neq S$, it holds: item $\langle\, 0_A, \pi \,\rangle \in I$ if, and only if, a correspondent p-state $\hat{I}$ contains the item $\langle\, 0_A \to \varepsilon \bullet, \pi \,\rangle$. Furthermore and similarly, if the (axiomatic) initial state 0_S is also final, then the item $\langle\, 0_S, \pi \,\rangle$ is in the p-state I_0 if, and only if, it holds: item $\langle\, 0_S \to \varepsilon \bullet, \pi \,\rangle \in \hat{I}_0$.*
4. *For every pair of correspondent p-states I and $\hat{I}$, and for every initial state 0_A, it holds item $\langle\, 0_A, \rho \,\rangle \in I_{\mid closure}$ if, and only if, it holds: item $\langle\, 0_A \to \bullet\, \alpha, \rho \,\rangle \in \hat{I}_{\mid closure}$ for every rule $0_A \to \alpha$.*

Proof

Claim 1. It is proved by induction on the paths of the graph.

base step Let I_0 and $\hat{I}_0$ be the initial p-states of $\mathcal{P}$ and $\hat{\mathcal{P}}$. The p-state I_0 contains only items with an initial state, i.e., a state of the kind 0_A with $A \in V_N$. From the definition of the closure function (Def. 5) and from the standard construction of the GOTO graph for the usual BNF grammars (see for instance

[1]), for every item $\langle\, 0_A,\, \rho\,\rangle \in I_0$ (hence there exists a set of arcs $\left(\, 0_A \xrightarrow{s_i} p_i \,\right) \in \delta$) there is a set of items $\langle\, 0_A \to \bullet\ \hat{s_i}\ p_i,\, \rho\,\rangle \in \hat{I}_0$. In addition, if state 0_A is final then p-state $\hat{I}_0$ also includes item $\langle\, 0_A \to \bullet\ \varepsilon = \varepsilon\ \bullet,\, \rho\,\rangle$.
Notice that while in the p-state I_0 there is no more than one item with state 0_A, p-state $\hat{I}_0$ might include several items corresponding to such a state, one for each arc in the TN leaving state 0_A.
Then, $\forall\, s \in (\Sigma \cup V_N)$, p-state $I' = \vartheta\,(I_0,\, s)$ is defined if, and only if, p-state $\hat{I}' = \hat{\vartheta}\,(\hat{I}_0,\, \hat{s})$ is also defined and is intermediate. Moreover, for every item $\langle\, p,\, \rho\,\rangle \in I'_{|\,base}$ (hence there exist an item $\langle\, 0_A,\, \rho\,\rangle \in I_0$ and an arc $\left(\, 0_A \xrightarrow{s} p \,\right) \in \delta$) there is an item as $\langle\, 0_A \to \hat{s}\ \bullet\ p,\, \rho\,\rangle \in \hat{I}'_{|\,base}$, whence p-states I' and $\hat{I}'$ are correspondent.
The inductive hypothesis includes the following two items:

(i) for each item of type $\langle\, p,\, \rho\,\rangle \in I'_{|\,base}$ and for each arc of type $\left(\, p \xrightarrow{s} q \,\right) \in \delta$, there is an item of type $\langle\, p \to \bullet\ \hat{s}\ q,\, \rho\,\rangle \in \hat{I}'_{|\,closure}$; notice that in the p-state I' there is a unique item corresponding to the source state p of the TN transition, while p-state $\hat{I}'$ includes as many items as there are TN arcs, how ever labeled, departing from state p;

(ii) for each item of type $\langle\, 0_A,\, \rho\,\rangle \in I'_{|\,closure}$ (which is included therein because there are an item $\langle\, p,\, \sigma\,\rangle \in I'$ and an arc $\left(\, p \xrightarrow{A} q \,\right) \in \delta$) and for every arc $\left(\, 0_A \xrightarrow{s} q \,\right) \in \delta$, $s \in \Sigma \cup V_N$, there is an item of type $\langle\, 0_A \to \bullet\ \hat{s}\ q,\, \rho\,\rangle \in \hat{I}'_{|\,closure}$.

inductive step Let $I \in \mathcal{P}$ and $\hat{I} \in \hat{\mathcal{P}}$ be two corresponding non-initial p-states, with p-state $\hat{I}$ being necessarily intermediate.
If p-state $I' = \vartheta\,(I,\, s)$ is defined, with symbol $s \in \Sigma \cup V_N$, then $I'_{|\,base}$ includes a set of items of type $\langle\, q_i,\, \rho_i\,\rangle$, p-state I contains, either in its base or in its closure part, a set of items of the type $\left\langle\, q_{ij},\, \rho_{ij}\,\right\rangle$ such that arc $\left(\, q_{ij} \xrightarrow{s} q_i \,\right) \in \delta$, and it holds $\bigcup_j\, \rho_{ij} = \rho_i$.
From the inductive hypothesis, p-state $\hat{I}$ contains, either in its base or in its closure part, a set of items of type $\left\langle\, q_{ij} \to \bullet\ \hat{s}\ q_i,\, \rho_{ij}\,\right\rangle$, therefore, $\hat{I}'_{|\,base}$ contains a set of items of type $\left\langle\, q_{ij} \to \hat{s}\ \bullet\ q_i,\, \rho_{ij}\,\right\rangle$, with $\bigcup_j\, \rho_{ij} = \rho_i$, whence p-states I' and $\hat{I}'$ are correspondent.
Moreover, the properties described in the points (i) and (ii) of the base step hold for p-states I' and $\hat{I}'$ as well.
Conversely, if p-state $\hat{I}' = \hat{\vartheta}\,(\hat{I},\, s)$ is defined, with the TN state $s \in Q - \{\ 0_A \mid\ \ A \in V_N\ \}$ being non-initial, then p-state $\hat{I}'$ is a sink reduction and function $\vartheta\,(I,\, s)$ is undefined. Hence, Claim 1. holds true.

Claim 2. Since the mapping (2) from the initial state and the intermediate states of $\hat{\mathcal{P}}$ to R is total (Claim 1.), and since state f_A is final (Def. 3), grammar $\hat{G}$ includes the rule $f_A \to \varepsilon$.

Claim 3. If item $\langle\, 0_A,\, \pi\,\rangle$ is in the p-state I, it is in $I_{|\,closure}$. Thus $I_{|\,base}$ contains an item $\langle\, p_B,\, \lambda\,\rangle$ such that $\mathit{closure}\left(\{\,\langle\, p_B,\, \lambda\,\rangle\,\}\right) \ni \langle\, 0_A,\, \pi\,\rangle$. Therefore, some intermediate p-state contains item $\langle\, q_B \to s\ \bullet\ p_B,\, \lambda\,\rangle$ in its base and item $\langle\, 0_A \to \varepsilon\ \bullet,\, \pi\,\rangle$ in its closure. The converse argument is analogous. The proof of the second part of the claim is obvious by considering the definition of initial p-state.

Claim 4. We consider the case where p-state I is not initial and state $0_A \in I_{|\,closure}$ results from closure applied to an item of $I_{|\,base}$. The other cases, where I is the initial p-state I_0 or where state 0_A results from closure applied to an item of $I_{|\,closure}$, can be similarly dealt with. Then from the definition of graphs $\mathcal{P}$ and $\hat{\mathcal{P}}$, it follows

$$\begin{aligned}
&\text{item } \langle\, 0_A,\, \rho \,\rangle \in I_{|\,closure} \\
\Longleftrightarrow\ &\exists \text{ a machine } M_B \in \mathcal{M} \text{ that contains an arc } p_B \xrightarrow{A} q_B \\
\Longleftrightarrow\ &\text{item } \langle\, p_B,\, \pi \,\rangle \in I_{|\,base} \\
\Longleftrightarrow\ &\forall \text{ correspondent state } \hat{I}\ \exists \text{ a symbol } s \in V \text{ and a set} \\
&\{\ \langle\, r_{i_B} \to \hat{s}\ \bullet\ p_B,\, \pi_i \,\rangle\ \} \subseteq \hat{I}_{|\,base} \text{ such that } \pi = \textstyle\bigcup_i \pi_i \\
\Longleftrightarrow\ &\text{item } \langle\, p_B \to \bullet\, 0_A\ q_B,\, \pi \,\rangle \in \hat{I}_{|\,closure} \\
\Longleftrightarrow\ &\text{item } \langle\, 0_A \to \bullet\, \alpha,\, \rho \rangle \in \hat{I}_{|\,closure} \text{ for each alternative right-hand side } \alpha \\
&\text{of nonterminal } 0_A. \qquad \square
\end{aligned}$$

Theorem 1 (equivalence of ELR and LR conditions) *Let $\mathcal{M}$ be a* TN *and let $\hat{G}$ be the corresponding right-linearized grammar* (RLZ). *The* TN $\mathcal{M}$ *meets the* ELR(1) *condition if, and only if, grammar $\hat{G}$ meets the standard* LR(1) *condition.*

Proof

***Part "if" of Th.* 1** For graph $\mathcal{P}$, we examine each one of the conflict types and show that if $\mathcal{P}$ violates the ELR(1) condition, then graph $\hat{\mathcal{P}}$ violates the standard LR(1) condition.

SR conflict Consider such a conflict in p-state $I \ni \langle\, f_B,\, \{\ a,\, \dots\ \} \,\rangle$, where f_B is final non-initial and $\vartheta\,(I,\, a)$ is defined. Following Lemma 3, Claims 1. and 2., there is a correspondent p-state $\hat{I}$ such that $\hat{\vartheta}\,(\hat{I},\, a)$ is defined and it holds $\langle\, f_B \to \varepsilon\ \bullet,\, \{\ a,\, \dots\ \} \,\rangle \in \hat{I}$, hence the same conflict is in $\hat{\mathcal{P}}$. A similar reasoning, by exploiting Claims 1. and 3., applies to a conflict in p-state $I \ni \langle\, 0_B,\, \{\ a,\, \dots\ \} \,\rangle$, where 0_B is final and initial, and $\vartheta\,(I,\, a)$ is defined.
RR conflict If in p-state $I \supseteq \{\ \langle\, f_A, \{\ a,\, \dots\ \} \,\rangle, \langle\, f_B,\, \{\ a,\, \dots\ \} \,\rangle\ \}$ there is such a conflict, where f_A and f_B are final non-initial, then in one or more p-states $\hat{I} \supseteq \{\ \langle\, f_A \to \varepsilon\ \bullet, \{\ a,\, \dots\ \} \,\rangle, \langle\, f_B \to \varepsilon\ \bullet, \{\ a,\, \dots\ \} \,\rangle\ \}$, by Claim 2. there is the same conflict. A similar reasoning applies to the cases where one final state or both final states in the items are initial.
CV conflict Consider a convergent transition $I \xrightarrow{X} I'$, such that

$$I \supseteq \{\ \langle\, p_A,\, \{\ a,\, \dots\ \} \,\rangle, \langle\, q_A,\, \{\ a,\, \dots\ \} \,\rangle\ \} \qquad \delta\,(p_A,\, X) = \delta\,(q_A,\, X) = r_A$$

therefore $I'_{|\,base} \ni \langle\, r_A,\, \{\ a,\, \dots\ \} \,\rangle$. If both states p_A and q_A are non-initial, both items are in the base of I. By Claim 1., there exist correspondent intermediate p-states and a transition $\hat{I} \xrightarrow{\hat{X}} \hat{I}'$ with

$$\hat{I}' \supseteq \left\{\ \left\langle\, p_A \to \hat{X}\ \bullet\ r_A,\, \{\ a,\, \dots\ \} \,\right\rangle, \left\langle\, q_A \to \hat{X}\ \bullet\ r_A,\, \{\ a,\, \dots\ \} \,\right\rangle\ \right\}$$

therefore the sink reduction p-state $\hat{\vartheta}\left(\hat{I}',\, r_A\right)$ has an RR conflict.
Similarly, if one of the states p_A and q_A is initial, say $q_A = 0_A$, then it holds $\langle\, q_A,\, \{\ a,\, \dots\ \} \,\rangle \in I_{|\,closure}$ as the initial state of machine M_A is non-reentrant,

therefore $I_{|\,base}$ contains an item t such that $closure\left(\{\,t\,\}\right) \ni \langle\, 0_A,\, \{\ a,\, \ldots\ \}\,\rangle$. Hence there exists a p-state $\hat{I}$ correspondent of I such that

$$\left\langle\, 0_A \to \bullet\ \hat{X}\ r_A, \{\ a,\, \ldots\ \}\,\right\rangle \in \hat{I}_{|\,closure}$$

whence $\hat{\vartheta}\left(\hat{I},\, \hat{X}\right) = \hat{I}'$ and $\hat{I}' \ni \left\langle\, 0_A \to \hat{X}\ \bullet\ r_A,\, \{\ a,\, \ldots\ \}\,\right\rangle$. It follows that p-state $\hat{\vartheta}\left(\hat{I}',\, r_A\right)$ is a sink reduction and that it has an RR conflict.

***Part "only-if" of Th.* 1** Consider the two conflict types possible in the graph $\hat{\mathcal{P}}$.

SR conflict The conflict is in a p-state $\hat{I}$ where $\langle\, f_B \to \varepsilon\ \bullet,\, \{\ a,\, \ldots\ \}\,\rangle \in \hat{I}$ holds and $\hat{\vartheta}\left(\hat{I},\, a\right)$ is defined. By Lemma 3, Claims 1. and 2. (or 3.), the correspondent p-state I contains item $\langle\, f_B,\, \{\ a,\, \ldots\ \}\,\rangle$ and the function $\vartheta\ (I,\, a)$ is defined, thus resulting in the same conflict.

RR conflict Take a p-state $\hat{I}$ such that

$$\{\ \langle\, f_A \to \varepsilon\ \bullet,\, \{\ a,\, \ldots\ \}\,\rangle,\, \langle\, f_B \to \varepsilon\ \bullet,\, \{\ a,\, \ldots\ \}\,\rangle\ \} \subseteq \hat{I}_{|\,closure}\,,$$

where states f_A and f_B are final non-initial. By Claim 2., the correspondent p-state I contains items $\langle\, f_A,\, \{\ a,\, \ldots\ \}\,\rangle$ and $\langle\, f_B,\, \{\ a,\, \ldots\ \}\,\rangle$, and has the same conflict. A similar reasoning, based on Claims 2. and 3., applies if either state f_A or f_B is initial.

Finally, take an RR conflict in a sink reduction p-state $\hat{I}$ such that, for some $s \in V$, $\{\ \langle\, p_A \to \hat{s}\ r_A\ \bullet,\, \{\ a,\, \ldots\ \}\,\rangle,\, \langle\, q_A \to \hat{s}\ r_A\ \bullet,\, \{\ a,\, \ldots\ \}\,\rangle\ \} \subseteq \hat{I}$. Then there exist p-states $\hat{I}'$, $\hat{I}''$ and transitions $\hat{I}'' \xrightarrow{\hat{s}} \hat{I}' \xrightarrow{r_A} \hat{I}$ such that $\hat{I}'$ contains items $\langle\, p_A \to \hat{s}\ \bullet\ r_A,\, \{\ a,\, \ldots\ \}\,\rangle$ and $\langle\, q_A \to \hat{s}\ \bullet\ r_A,\, \{\ a,\, \ldots\ \}\,\rangle$, the correspondent p-state I' contains item $\langle\, r_A,\, \{\ a,\, \ldots\ \}\,\rangle$, and

$$\{\ \langle\, p_A \to \bullet\ \hat{s}\ r_A,\, \{\ a,\, \ldots\ \}\,\rangle,\, \langle\, q_A \to \bullet\ \hat{s}\ r_A,\, \{\ a,\, \ldots\ \}\,\rangle\ \} \subseteq \hat{I}''_{|\,closure}\,.$$

Since $\hat{I}''_{|\,closure} \neq \emptyset$, p-state $\hat{I}''$ is not a sink reduction. Let I'' be its correspondent p-state. Then it follows: if state p_A is initial, by Claim 4. there is an item $\langle\, p_A,\, \{\ a,\, \ldots\ \}\,\rangle \in I''_{|\,closure}$; moreover, if p_A is non-initial, there exists an item $\langle\, t_A \to Z\ \bullet\ p_A,\, \{\ a,\, \ldots\ \}\,\rangle \in \hat{I}''_{|\,base}$, hence $\langle\, p_A,\, \{\ a,\, \ldots\ \}\,\rangle \in I''$. A similar reasoning applies to state q_A, hence $\langle\, q_A,\, \{\ a,\, \ldots\ \}\,\rangle \in I''$, and transition $I'' \xrightarrow{\hat{s}} I'$ in $\mathcal{P}$ is multiple, convergent and conflicting. □

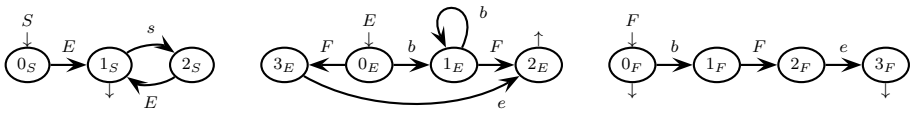
We address a possible criticism to the significance of Th. 1. Namely, starting from a TN, several equivalent BNF grammars can be obtained that differ in their implementation of the operations of the regular expressions, especially the Kleene star. Such BNF grammars may happen to be LR(1) or not, a fact that would seem to make somewhat arbitrary our reference to the right-linearized BNF grammar in Th. 1.

We defend our choice on two grounds. First, since our language specification is not by a set of REs, but by a set of DFAs, the choice of transforming a DFA into a right-linear grammar is not only natural. In fact the other natural form — left-linear — would exhibit conflicts in most cases as shown by Heilbrunner [13]. Second, the same author argues that for an EBNF grammar the choice of the

equivalent right-linearized grammar (RLZ) is preferable to any other choice, because if any other equivalent BNF grammar is LR(1), then also the right-linearized grammar is so. To illustrate this point, we rely on an example from [13].

*Example 4 (equivalent non-*LR(1) BNF *grammar)* In Fig. 5 (top) we show a TN $\mathcal{M}$ that meets the ELR(1) condition, therefore the equivalent right-linearized grammar (RLZ) is LR(1) by Th. 1. On the contrary, the equivalent BNF grammar in Fig. 5 (bottom) has an SR conflict. Notice that this BNF grammar has been obtained by quite natural transformations from the machines of $\mathcal{M}$ to BNF rules, such as the introduction of a new nonterminal B to generate the substrings "$b\,b^*$".

We leave to the reader to check that no SR conflict occurs in the right-linearized grammar corresponding to $\mathcal{M}$ (Def. 3), because RLZ grammars defer the reduction operations as late as possible. □



$$S \to E\,s\,S \mid E \qquad E \to B\,F \mid F\,e \qquad F \to b\,F\,e \mid \varepsilon \qquad B \to b\,B \mid b$$

Fig. 5 Top: A TN $\mathcal{M}$ (axiom S) that meets the ELR(1) condition (Def. 8). Bottom: an equivalent BNF grammar (but not the right-linearized one) that has LR(1) conflicts.

3.3 ELR(1) parser for TN

It is well-known that for a BNF grammar the LR(1) graph specifies the state-transition function of a deterministic pushdown automaton (DPDA), which accepts and parses the language by performing so-called shift and reduce operations. The (pushdown) stack alphabet consists of the set of p-states and of the grammar nonterminal and terminal alphabets.

Similarly, the ELR(1) parser operates by shifts and reductions, but with important differences mainly due to the fact that a TN machine may contain convergent and cyclic paths, which may cause the parser to reduce strings of unbounded length. The other major difference is that the p-states of the ELR(1) graph cannot be used as stack symbols of the DPDA, because more information is needed to track all the possible alternative paths traversed by a TN machine until a reduction occurs (which is unique by the ELR(1) condition). Here we describe the DPDA derived from the ELR(1) graph, first informally, then as an algorithm and finally by an example. Complexity analysis and comparisons with the standard LR(1) parser are in Sect. 4.1.

Recalling Def. 4 and the related ones, consider a TN $\mathcal{M}$ with machines M_S, M_A, ..., and let Q be the state set of $\mathcal{M}$. The ELR(1) graph $\mathcal{P}$ of $\mathcal{M}$ represents a DFA that has the graph p-states as its nodes and the function ϑ as its transitions. Each p-state contains a set of items of the form $\langle\, q_A,\, \pi \,\rangle \in Q \times \wp\,(\Sigma)$. We assume that

the graph $\mathcal{P}$ meets the ELR(1) condition. Clearly, for any such TN $\mathcal{M}$ there are finitely many different items, the number of p-states is bounded, and the number of items in any p-state is bounded by the constant $|\,Q\,|$.

We need to enrich the information contained in the p-states of $\mathcal{P}$ in order to determine the reduction to be performed, out of several possible paths in the graph of a TN machine. To this end, each item is extended with an integer field, called *item identifier* (*iid*). The *iid* will be used to link an item to another one at parsing time. Such extended items are called (pushdown) *stack items*, and a set of stack items is called a (pushdown) *stack p-state*.

Definition 10 (stack items and stack p-states) A *stack item* is a 3-tuple of the form ⟨ state, look-ahead, item identifier ⟩

$$\langle\, q,\, \rho,\, i \,\rangle \in Q \times \wp\,(\,\Sigma\,) \times \big(\,\mathbb{N} \cup \{\ \bot\ \}\,\big)\,,$$

where i is a value in the interval $1 \leq i \leq |\,Q\,|$ or $i = \bot$ (null value).
A *stack p-state* J is a non-empty set of stack items. We allow the presence in a stack p-state of two or more stack items that have the same (machine) state, such as $\langle\, q,\, \rho_1,\, i_1 \,\rangle$ and $\langle\, q,\, \rho_2,\, i_2 \,\rangle$, but with $i_1 \neq i_2$. Moreover, we assume that the stack items in a stack p-state are (arbitrarily) ordered.
Let $\langle\, q,\, \rho_k,\, i_k \,\rangle \in J$. A mapping μ from stack items to items is next defined.

$$\mu\big(\ \langle\, q,\, \rho_k,\, i_k \,\rangle\ \big) = \langle\, q,\, \pi \,\rangle \qquad\qquad \text{where}\ \ \pi = \bigcup_{\substack{\forall\, h \text{ such that} \\ \langle q,\, \rho_h,\, i_h \rangle\ \in\ J}} \rho_h\ .$$

The mapping μ can be naturally extended to stack item sets (stack p-states).
Let $I_0 \in R$ be the *initial* p-state of graph $\mathcal{P}$. The *initial* stack p-state J_0 is defined as follows

$$J_0 = \{\ \langle\, q,\, \pi,\, \bot \,\rangle\ \mid\ \ \text{item}\ \langle\, q,\, \pi \,\rangle \in I_0\ \}\ . \tag{3}$$

Clearly, it holds $\mu\big(\,J_0\,\big) = I_0$. □

Since for any TN the set of stack p-states is clearly finite, the parser uses it as pushdown stack alphabet. For a better readability of algorithms and examples, the item identifiers will be marked with a sign "♯". Thanks to the ordering imposed on the stack items of each stack p-state, an item identifier ♯i (with $i \neq \bot$) in the topmost stack p-state points to the i-th item of the stack p-state immediately below.

The pseudo-code of our parser is shown in Alg. 2. It is patterned as the classical shift-reduce parser [1] and we especially highlight the differences. The stack alphabet has two element types: stack p-states and symbols over the alphabet V. As in the classical parsers, stack p-states and symbols alternate in the stack. Stack p-states and graph p-states are enumerated with $J[r]$ and $I_{(s)}$ $(r, s \geq 0)$, respectively, and for convenience we stipulate there is a correspondence between them at parsing runtime: $\mu\big(\,J[r]\,\big) = I_{(r)}$, i.e., we pose $s = r$. The p-states at the stack bottom and top are denoted $J[0]$ and $J[k]$ $(k \geq 0)$, respectively.

There are three move types: terminal shift, nonterminal shift and reduction. For terminal shift, relations (4-5) compute, from the current top stack p-state $J[k]$, the new stack p-state $J[k+1]$ to push: it holds $\mu\big(J[k+1]\big) = I_{(k+1)}$ and the new stack item $\big\langle\, q'_A,\, \rho,\, \sharp i \,\big\rangle \in J[k+1]$ is linked to the previous one $\langle\, q_A,\, \rho,\, \sharp j \,\rangle$ located

Algorithm 2: Pseudo-code of the ELR(1) parser for a TN $\mathcal{M}$.

Input: the graph (R, ϑ) of a TN $\mathcal{M}$ and a word w
Output: the ELR(1) parsing of w according to $\mathcal{M}$
// current stack: $J[0]\,a_1\,J[1]\,a_2\,\ldots\,a_k\,J[k]$ $(k \geq 0)$; $J[k]$ top
// current symbol: a (may be a terminal or nonterminal)
// pointer to running stack item: p (used for reduction)
stack := J_0 // initial stack p-state, see (3)
$k := 0$
$a :=$ first char of word w (or $\dashv$ if $w = \varepsilon$)
forever do // loop for parsing the word w
  if $\left(\exists \text{ arc } \left(I_{(k)} \xrightarrow{a} I_{(k+1)}\right) \in \vartheta \text{ such that } \mu\left(J[k]\right) = I_{(k)}\right)$ **then**
    // terminal shift move: $\mu\left(J[k]\right) \xrightarrow{a} \vartheta\left(\mu\left(J[k]\right), a\right)$
    $J[k+1] :=$ $\left\{ \langle\, q'_A,\, \rho,\, \sharp i \,\rangle \;\middle|\; \begin{array}{l}\text{arc } \left(q_A \xrightarrow{a} q'_A\right) \in \delta \quad \textbf{and} \\ \langle\, q_A,\, \rho,\, \sharp j \,\rangle = i^{th} \text{ item} \in J[k]\end{array} \right\} \cup$ (4)
    $\{\, \langle\, 0_B,\, \sigma,\, \perp \,\rangle \mid \text{item } \langle\, 0_B,\, \sigma \,\rangle \in I_{(k+1)|\,closure} \,\}$ (5)
    push symbol a (surely $\neq \dashv$)
    push stack p-state $J[k+1]$
    k ++
    $a :=$ next char of word w (or $\dashv$)
  else if $\left(\begin{array}{c}\exists \text{ item } \langle\, f_A,\, \rho,\, \sharp i \,\rangle \in J[k] \text{ such} \\ \textbf{that } f_A \text{ is a final state } \textbf{and } a \in \rho\end{array}\right)$ **then**
    // reduction move: $a_{h+1} \ldots a_k \rightsquigarrow A$ or $\varepsilon \rightsquigarrow A$ (null)
    $p :=$ back pointer field $\sharp i$ in the (final) item $\langle\, f_A,\, \rho,\, \sharp i \,\rangle$
    while $(p \neq \perp)$ **do** // loop for popping the handle
      pop stack p-state $J[k]$
      pop symbol a_k $(\neq \dashv)$
      k −−
      $p :=$ back pointer field $\sharp i$ in the p^{th} item $\in J[k]$
    // no iterations ⇒ null reduction ⇒ no handle pop
    if $(A \neq S')$ **then**
      // nonterm shift move: $\mu\left(J[k]\right) \xrightarrow{A} \vartheta\left(\mu\left(J[k]\right), A\right)$
      execute a nonterminal shift move as of (4-5) with $a = A$:
      push A, **push** $J[k+1]$, k ++, yet do not read any input
    else accept and stop // final reduction to axiom S'
  else reject and stop // no move - word w invalid

at position i in $J[k]$. The same applies to nonterminal shift. For reduction, a while loop scans, by using a running item pointer p, the chain of stack items that make the reduction handle, starting from the final item $\langle\, f_A,\, \rho,\, \sharp i \,\rangle$ in $J[k]$. At every loop iteration, the stack top p-state $J[k]$ and its entrance symbol a_k are popped (for a null reduction nothing is popped). Finally, a nonterminal shift concludes the reduction move.

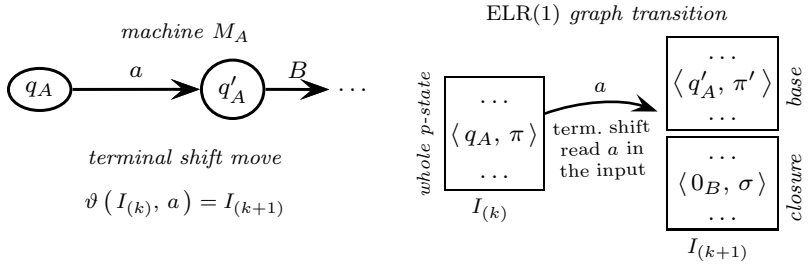
Termination occurs in acceptance or rejection. To simplify the acceptance condition, we assume that the grammar contains a unique starting rule $S' \to S$ (more precisely a TN machine), where nonterminal S' is the axiom and no other rule may contain it (as in [1]).

The ELR(1) condition makes sure that the conditions enabling a terminal shift or a reduction are mutually exclusive, so that Alg. 2 is deterministic. We do not

describe how our parser builds the rightmost derivation and the syntax tree of the input word, as it essentially works like the classical one.

Fig. 6 illustrates the scheme of a shift move. Recall that the stack p-state $J[k+1]$ computed by a shift move (terminal or non-) may contain two or more stack items that have the same TN state and different item identifiers, see relations (4-5) in Alg. 2. This happens whenever a shift move takes a convergent transition (Def. 7)[3].

The stack items are linked in a list to make a stack item chain, e.g., stack item $\langle\, q'_A,\, \rho,\, \sharp i \,\rangle$ is linked to $\langle\, q_A,\, \rho,\, \sharp j \,\rangle$ via the identifier $\sharp i$ (see also Alg. 2). Every stack item $\langle\, q_A,\, \rho,\, \sharp j \,\rangle$ is mapped by function μ onto an item $\langle\, q_A,\, \pi \,\rangle$ of graph $\mathcal{P}$. The stack item has the same machine state as the graph item, i.e., q_A, but in general it has a smaller look-ahead set, i.e., $\rho \subseteq \pi$, due to the possible presence of convergent transitions in the graph; due to the same reason in general it holds $\pi \subseteq \pi'$. The two look-ahead sets ρ and π coincide if at parsing time the DPDA does not take a convergent transition.



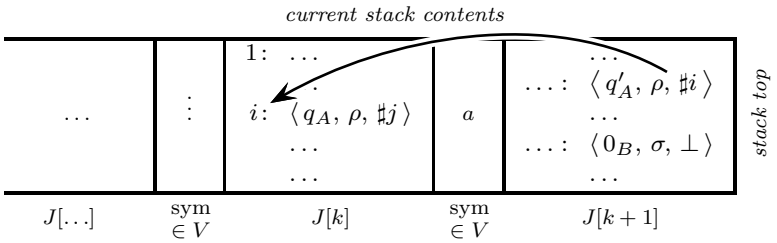


Fig. 6 Scheme of a shift move. Top: part of a machine M_A and of the ELR(1) graph. Bottom: parser stack, assuming that $\mu\,(\,J[k]\,) = I_{(k)}$ and $\mu\,(\,J[k+1]\,) = I_{(k+1)}$.

Fig. 7 shows the scheme of a (non-null) reduction move. The stack p-states of the reduction handle are linked as in Fig. 6 and are popped one by one (see Alg. 2). After so exposing on the stack top the initial stack p-state $J[h]$ (for some $h < k$) of the accepting path $0_A \xrightarrow{a_{h+1} \cdots a_k} f_A$, a nonterminal shift move on symbol A is applied exactly as in Fig. 6, but without reading an input symbol. For a null reduction, no handle is popped since the stack p-state $J[h]$ (with $h = k$) is already directly exposed on the stack top.

[3] When applying the function μ to a stack p-state, if two items have identical (machine) states, then they are coalesced as said in Def.s 7 and 10.

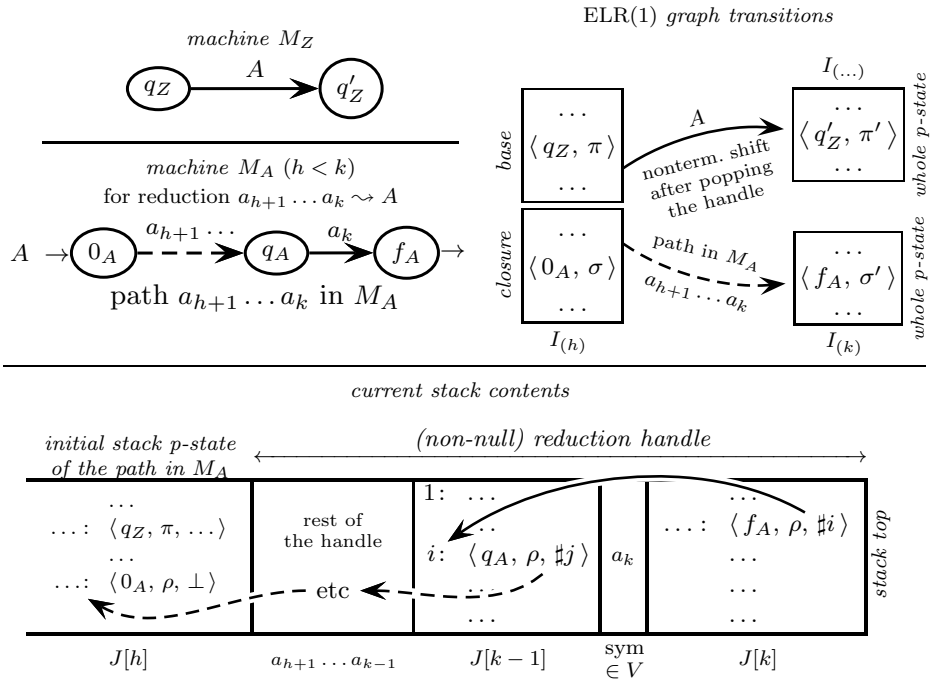


Fig. 7 Scheme of a non-null reduction move. Top: part of machines M_Z and M_A, and of the ELR(1) graph. Bottom: handle, with $\mu\big(J[k]\big) = I_{(k)}$ and $\mu\big(J[h]\big) = I_{(h)}$ $(h < k)$. As shown in Fig. 6, due to convergence in general it holds $\pi \subseteq \pi'$, $\sigma \subseteq \sigma'$ and $\rho \subseteq \pi$.

To prove the correctness of the parser algorithm it suffices to reexamine the correspondence, established in Lemma 3, between the moves of Alg. 2 and those of the LR(1) standard parser for the right-linearized grammar associated with the TN. In particular, the terminal shift moves of the two parsers are in one-to-one correspondence, while a single ELR(1) reduction move corresponds to a series of RLZ reductions. We omit the straightforward but tedious correctness proof and we show an example instead.

Example 5 (Ex. 1 *continued)* Although the graph in Fig. 2.b violates the ELR(1) condition, for brevity we use this TN to illustrate Alg. 2, taking care to choose an input that does not involve conflicts. Fig. 2.c shows the parse trace of word "$a\,b\,a\,a\,c\,c$".

The superimposed arrows are for visualization. Solid arrows represent the stack item chains. More precisely, in the first stack all the stack item chains are represented, while in the second and third stacks only the stack item chains that lead to the current reduction are shown. Dashed arrows associate the look-ahead and the current input characters at reduction.

The stack items in the reduction handle are framed. The final TN states on the stack top, which trigger a reduction move if their look-ahead matches the current input, are encircled. To avoid cluttering, we denote the TN states without

the subscript that identifies the machine, e.g., state 2_S is written 2, and we drop commas and angular/curly brackets in the stack item 3-tuples.

Each stack item in a stack p-state $J[l]$ ($l \geq 0$) is numbered starting with 1 from the top. An identifier $\sharp i$ of a stack item belonging to the stack p-state $J[l+1]$ refers back to the i-th stack item in $J[l]$.

The trace shows from top to bottom: a null reduction, a non-null reduction corresponding to an acyclic path, and a non-null one corresponding to a cyclic path, all in the graph of Fig. 2.a. The mapping μ is as follows:

- $\mu\big(J[3]\big) = I_8$: the stack p-state $J[3]$ contains the following three stack items $\langle 1_S, \{ \dashv \}, \sharp 1 \rangle$, $\langle 1_S, \{ c \}, \sharp 2 \rangle$ and $\langle 0_S, \{ c \}, \perp \rangle$, where the two ones with the TN state 1_S are derived from splitting the look-ahead set of the (graph) item $\langle 1_S, \{ \dashv, c \} \rangle$, which comes from convergence;
- $\mu\big(J[2]\big) = I_7$: the stack p-state $J[2]$ contains two stack items (with the TN state 3_S) derived from splitting one (graph) item (in I_7).

More comments can be found in Fig. 2, parts (b) and (c). □

4 Complexity comparisons and related work

4.1 Complexity comparisons

First, we analyze and compare the descriptional complexity of direct and indirect parsers. We remind that, given a transition net TN, we call *indirect* a parser obtained by first transforming the TN into an equivalent BNF grammar and then by constructing the standard LR(1) parser, and that we call *direct* the ELR(1) parser constructed by our method.

Second, we analyze the computational complexity of the indirect and direct parsers, namely their time complexity and pushdown stack size. We also present a simple optimization in the stack symbols that makes reduction moves faster.

Third, we list and comment earlier research work related to ours.

Size of standard* LR(1) *graph vs* ELR(1) *graph We argue that the direct parser has fewer p-states than the indirect LR(1) graph of an equivalent BNF grammar.

First, we consider a family of regular languages L_n and their one-rule EBNF grammars $S \to L_n$, where the *star height* of L_n is $n \geq 1$. For simplicity, we show a language with an alphabet size that grows with n, but other examples exist with a bounded alphabet size. The family is

$$L_1 = a^* \qquad L_2 = \big(a^*\, b\big)^* \qquad L_3 = \Big(\big(a^*\, b\big)^*\, c\Big)^* \qquad \ldots$$

The minimal DFA (with a non-reentrant initial state) that recognizes language L_n has $n+1$ states, and it is isomorphic to the ELR(1) graph of rule (i.e., machine) $S \to L_n$. On the other hand, to convert such a grammar to BNF one needs n nonterminal symbols (see [12] for a similar case), and the LR(1) graph of the BNF grammar has a number of p-states lower-bounded by $2n$, which almost doubles the ELR(1) case.

Second, a typical transformation of a BNF grammar into an EBNF one produces a smaller grammar, which in turn results into a TN, the ELR(1) graph of

which is smaller than the LR(1) graph of the original BNF grammar. We only discuss two such chief transformations:

- Consider the transformation of a left-recursive BNF rule, such as $A \to A\ u \mid v$, into an iterative rule $A \to v\ u^*$. Clearly, the size of the ELR(1) graph shrinks (the case of a right-recursive rule is similar).
- If a rule is substituted by another, e.g., by replacing the two BNF rules $A \to a\ B\ b$ and $B \to u \mid v$ with the rule $A \to a\ (\ u \mid v\ )\ b$, then the number of nonterminal symbols, i.e., of DFA machines of the TN, decreases, and so also does the size of the ELR(1) graph.

Clearly, the repeated application of left-recursion removal and substitution yields a TN with a smaller ELR(1) graph[4].

Computational complexity of parsers We start by comparing the space complexity of the direct and indirect parsers, namely the memory required for the parser stack. It is well known that for the indirect parser the stack length is linear in the length of the input word, and the same clearly holds for the direct case, therefore we have to compare the exact stack sizes.

The stack of the indirect parser contains only p-states (or their names) [17][5]. Instead, the stack of the direct parser presented in Alg. 2 alternates stack p-states and symbols over V, the latter being required for building the syntax tree when applying a reduction move. However, it would be simple to remove this requirement from the algorithm, by extending the alphabet of stack p-states with an extra field of domain V. When applying a shift move $\mu\ (\ J[k]\ ) \xrightarrow{a} \mu\ (\ J[k+1]\ )$, the symbol $a \in V$ is recorded in the extended stack p-state $J[k+1]$, which corresponds to p-state $I_{(k+1)}$. This change equalizes the stack lengths of the direct and indirect parsers, and though it enlarges the stack alphabet, it does not decrease the run-time efficiency.

Next we move to time complexity. To start, we compare the number of PDA moves (*push* and *pop* operations) performed by the direct and indirect parsers for the same language. Tab. 1 summarizes the number of operations that implement the shift and reduction moves of the parser for general grammars. The number of terminal shifts is identical in both the LR(1) and ELR(1) cases. Obviously, for any input word the overall number of nonterminal shifts is equal to the number of reductions in each case separately, but the two numbers may be quite different in either case.

There are many ways of obtaining a BNF grammar equivalent to a TN, and we initially consider the right-linearized grammar (RLZ) of Sect. 2. Whenever an ELR(1) parser performs a reduction with a (machine) path of length $n \geq 0$, the LR(1) parser of the corresponding RLZ grammar performs n non-null reductions with a handle of length 2 for the rules as in Def. 3, plus one null reduction. Thus

[4] It would be interesting to supplement the above analysis with experimental measurements for realistic grammars. A few results are mentioned in [3], which point to a greater compactness of the ELR(1) graphs. Just an example: for the Oracle EBNF grammar of the Java language, the TN has 532 machine states and 599 machine transitions. The ELR(1) graph has $1,937$ p-states and $25,231$ transitions, while the standard LR(1) graph has $2,946$ p-states and $25,837$ transitions.

[5] It is well known that the terminal and nonterminal symbols, typically represented in the stack when the LR(1) techniques are taught, are unnecessary.

move type	LR(1) *parser*	ELR(1) *parser*
terminal shift	1 *push* operation	1 *push* operation
nonterminal shift	1 *push* operation	1 *push* operation
reduction	for a rule $A \to \alpha$: $\mid \alpha \mid$ *pop* operations	for a path $0_A \to \ldots \to f_A$ of length $n \geq 0$, with f_A final: n *pop* operations

Table 1 Number of DPDA operations for the LR(1) and ELR(1) parsers. The terminal and nonterminal symbols in the stack are disregarded.

in total it performs $n + 1$ reductions, i.e., $2n$ pop operations. Consequently, the number of nonterminal shifts is 1 for ELR(1) and $n + 1$ for LR(1). Therefore in the ELR(1) case the number of push operations is reduced by an amount equal to the sum of the lengths of the reduction paths, though their number changes only linearly in both cases.

Optimizing* ELR(1) *reductions One might object that an RLZ grammar is the worst possible case for an LR(1) parser, since a BNF grammar with longer right-hand sides would perform fewer reductions, implying fewer nonterminal shifts. But in such a case, the reduction for a longer right-hand rule, say $A \to \alpha$, performs a unique multiple pop operation that removes in one step a number of stack items equal to $\mid \alpha \mid$. Such an implementation of reduction would make the number of pop operations (including multiple pops) of LR(1) smaller than that of ELR(1), while the number of push operations would remain greater or equal. Notice also that, if a multiple pop is allowed, the LR(1) parser is slightly more general than a DPDA because it directly operates on the stack elements below the top one.

The use of a multiple pop operation for implementing the reduction move is also feasible and rewarding for the ELR(1) parser. Alg. 3 shows such a parser variant, which is modeled according to Alg. 2 with the following differences. The parser stack is implemented as a random access array (vector), with indexeing from 0 (stack bottom) to $k \geq 0$ (top). The stack items are 3-tuples, the third field of which is a value that points back to the lowest element (i.e., the deepest in the stack) of the stack segment (the handle) to be deleted when performing a reduction move. The initial stack p-state J_0 is (re)defined (see (3)) as follows (but it still holds $\mu\left(J_0\right) = I_0$)

$$J_0 = \{ \ \langle q, \pi, 0 \rangle \mid \quad \text{item } \langle q, \pi \rangle \in I_0 \ \} \ . \tag{8}$$

From a stack item $\langle q_A, \rho, h \rangle \in J[k]$ with back pointer h $(0 \leq h \leq k)$, the terminal shift move creates a new stack p-state $J[k+1]$ that contains shifted stack items with the same pointer h (6), and initial stack items with a new pointer equal to $k + 1$ that refers to the current stack top (7). The same applies to nonterminal shift. The reduction move does not include the while loop that appears in Alg. 2 for stepwise popping the reduction handle, as now the back pointer h in a final stack item $\langle f_A, \rho, h \rangle \in J[k]$ goes directly to the handle bottom and so allows to immediately pop the whole handle. The rest of terminal and nonterminal shift, of reduction, and of acceptance and rejection, is as in Alg. 2. In Ex. 6 we show the operation of Alg. 3.

Algorithm 3: Pseudo-code of the ELR(1) parser with a vector stack.

Input: the graph (R, ϑ) of a TN $\mathcal{M}$ and a word w
Output: the ELR(1) parsing of w according to $\mathcal{M}$
// current stack: $J[0]\, a_1\, J[1]\, a_2 \ldots a_k\, J[k]$ ($k \geq 0$); $J[k]$ top
// current symbol: a (may be a terminal or nonterminal)
stack := J_0 // initial stack p-state, see (8)
$k := 0$
$a :=$ first char of word w (or $\dashv$ if $w = \varepsilon$)
forever do // loop for parsing the word w
  if $\left(\exists \text{ arc } \left(I_{(k)} \xrightarrow{a} I_{(k+1)}\right) \in \vartheta \text{ such that } \mu\left(J[k]\right) = I_{(k)}\right)$ **then**
    // terminal shift move: $\mu\left(J[k]\right) \xrightarrow{a} \vartheta\left(\mu\left(J[k]\right), a\right)$
    $J[k+1] :=$ $\left\{ \langle\, q'_A,\, \rho,\, h \,\rangle \;\middle|\; \begin{array}{l} \text{arc } \left(q_A \xrightarrow{a} q'_A\right) \in \delta \text{ and} \\ \text{item } \langle\, q_A,\, \rho,\, h \,\rangle \in J[k] \end{array} \right\} \cup$ (6)
    $\left\{ \langle\, 0_B,\, \sigma,\, k+1 \,\rangle \mid \text{item } \langle\, 0_B,\, \sigma \,\rangle \in I_{(k+1)\mid closure} \right\}$ (7)
    push symbol a (surely $\neq \dashv$)
    push stack p-state $J[k+1]$
    k ++
    $a :=$ next char of word w (or $\dashv$)
  else if $\left(\begin{array}{c} \exists \text{ item } \langle\, f_A,\, \rho,\, h \,\rangle \in J[k] \text{ such} \\ \text{that } f_A \text{ is a final state and } a \in \rho \end{array}\right)$ **then**
    // reduction move: $a_{h+1} \ldots a_k \rightsquigarrow A$ or $\varepsilon \rightsquigarrow A$ (null)
    if $(h < k)$ **then** // for popping the whole handle
      multiple pop: $J[k]$, a_k, $J[k-1]$, a_{k-1}, …, $J[h+1]$, a_{h+1}
      $k := h$
    // untaken then ⇒ null reduction ⇒ no handle pop
    if $(A \neq S')$ **then**
      // nonterm shift move: $\mu\left(J[k]\right) \xrightarrow{A} \vartheta\left(\mu\left(J[k]\right), A\right)$
      execute a nonterminal shift move as of (6-7) with $a = A$:
      push A, **push** $J[k+1]$, k ++, yet do not read any input
    else accept and stop // final reduction to axiom S'
  else reject and stop // no move - word w invalid

Example 6 (Ex. 1 continued) Reconsider the TN and ELR(1) graph of Fig. 4, parts (a) and (b) (same as Ex. 1); the part (c) of the same figure shows the parsing of the input word "$a\,a\,a\,b$" by using a vector stack. □

Alg. 3 achieves at least the same decrease in the number of *pop* operations as the LR(1) implementation mentioned above, and a further decrease because the length of the reduction paths may be unbounded. The latter case occurs frequently in practice as the TN machines often have cyclic graphs. Notice that such a stack optimization does not affect the stack size. Our implementation of ELR(1) parsers for TNs (`http://github.com/FLC-project/ELRparser`) incorporates this optimization (among others).

To sum up, we have shown that the stack sizes of the direct and indirect parsers are equal, and that the number of *push* and *pop* operations of the direct parser is reduced.

4.2 Related work

Many authors have proposed to extend the standard LR method to EBNF grammars, each proposal purporting to improve previous attempts, but no clear-cut optimal solution had surfaced so far. The following discussion particularly draws from the later papers [14,16,22], which also include relevant surveys. An old-fashioned distinction concerns the source language specification: either an EBNF grammar, i.e., a set of rules with regular expressions as right-hand sides, or a TN. Since REs and finite automata are easily interchangeable notations, the distinction has lost much relevance.

Some authors imposed restrictions on the REs, for instance by limiting the star depth to one or by forbidding common sub-expressions. Although the original motivation to simplify the parser construction has since vanished, it is fair to say that in practice the REs used in the language manuals are typically quite simple for the reason of avoiding obscurity.

Others, notably [22], specify the rule right parts by DFAs, i.e., they use a TN as we also do. Notice that the use of NFAs instead of DFAs has not been considered presumably because the lower state complexity thus achievable would make little difference in terms of readability, since the typical practical languages considered have a very small state complexity. In our case, the presence of non-deterministic machines in the TN could be easily accommodated by a little change to our parser generation framework.

For TN specifications two approaches to parser construction exist. Approach (A) eliminates branching paths and cyclic paths from the DFAs. In terms of grammars, this is tantamount to eliminating union operations and star operations from the grammar rules, thus obtaining a BNF grammar. To the latter, the standard LR(1) construction is applied to obtain what we call an indirect parser, discussed above in this section. Approach (B) directly constructs the parser. In [5] a systematic transformation from EBNF to BNF is used to obtain an ELR(1) parser that simulates the standard Knuth's one. It is generally agreed that approach (B) is superior, because transforming to BNF may add inefficiency and may obscure the semantic structure [22] of the language. Furthermore, we have seen in Sect. 4.1 that the size of the directly produced parser is smaller. The only advantage of approach (A) is to leverage on existing parser generators.

The major difficulty (solved by Alg. 2) with approach (B) is to find the left end of the reduction handle, since its length is variable and generally unbounded. A list of proposed solutions is in the cited surveys. Some algorithms use a special shift move, sometimes called *stack-shift*, to record the left end of the handle into the stack, when a new computation on a machine is started. But if the initial state is reentered, then a conflict between stack-shift and normal shift is unavoidable, and various complicated devices have been invented to resolve the conflict. Some authors add so called read-back states to control how deep the parser should dig into the stack [6,19], while others use counters for the same purpose, e.g., [24], let alone further proposed devices. Unfortunately, it was shown in [10,16] that several proposals do not precisely characterize the grammars they apply to, and in certain cases they may fall into unexpected errors. Motivated by the flaws of past attempts, paper [20] offers a characterization of the LR(k) property for TNs. Although their definition is intended to ensure that the languages "can be parsed from left to right with a look-ahead of k symbols", the authors admit that "the

subject of efficient techniques for locating the left end of a handle is beyond the scope of this paper".

In contrast with the twisted history of the ELR(1) methods, early efforts to develop deterministic *top-down* parsing algorithms for EBNF grammars have met with remarkable success and widespread application. We do not need to discuss them, but we just cite the main references and we explain why our work adds value to them. Deterministic parsers operating top-down were among the first to be constructed by compilation pioneers, and their theory for BNF grammars was shortly after developed under the acronym LL(k) by [18, 23]. A practical method to extend such parsers to EBNF grammars was popularized by Wirth [25] in his recursive-descent Pascal compiler, systematized as ELL(1) method in the book [21], and included in widely known compiler textbooks, e.g., [1]. However, in such books the top-down deterministic parsing is presented before the shift-reduce method and independently of it, presumably because it is simpler and more general than the LR(1) parser available at that time, which suffered from the restriction to BNF grammars.

On the contrary, in [8] it is shown that the classical ELL(1) parser generation method is strictly included in our ELR(1) method, although space prevents us from presenting details. It suffices to recall that for pure BNF grammars, the relationship between the grammar and language families of type LR(k) and LL(k) was carefully investigated in the past, in particular by Beatty [2]. Building on the concept of multiple convergent transitions, which we have introduced for ELR(1) analysis, it is possible to extend the Beatty's characterization to the EBNF case and to show that the ELL(1) parsing algorithm is an optimized version of the ELR(1) algorithm under the hypotheses that the TN is free from multiple transitions.

5 Conclusion

After a long history of moderately successful attempts at producing deterministic bottom-up parsers directly from syntax diagrams or transition networks, the new ELR(1) condition and the naturally corresponding algorithm presented here offer a clear and efficient solution to this practical problem. Our approach is more or equally general as any previous proposal known to us, and is simple: it just adds the treatment of convergent transitions to the Knuth's definition. To sum up, the technical difficulties were understood since long, and we have combined and improved existing ideas into a practical and provably correct solution.

The software tool that implements our parsing method is freely available at `http://github.com/FLC-project/ELRparser`.

References

1. Aho, A., Lam, M., Sethi, R., Ullman, J.: Compilers: principles, techniques and tools. Prentice-Hall, Englewoof Cliffs, NJ (2006)
2. Beatty, J.: On the relationship between the LL(1) and LR(1) grammars. JACM **29**(4), 1007–1022 (1982)
3. Borsotti, A., Breveglieri, L., Crespi Reghizzi, S., Morzenti, A.: Complexity of extended vs classic LR parsers. In: Descriptional Complexity of Formal Systems (DCFS), *LNCS*, vol. 8614, pp. 77–89. Springer (2014)
4. Breveglieri, L., Crespi Reghizzi, S., Morzenti, A.: Shift-reduce parsers for transition networks. In: Language and Automata Theory and Applications (LATA), *LNCS*, vol. 8370, pp. 222–235. Springer (2014)
5. Celentano, A.: LR parsing technique for extended context-free grammars. Comput. Lang. **6**(2), 95–107 (1981)
6. Chapman, N.: LALR(1,1) parser generation for regular right part grammars. Acta Inform. **21**, 29–45 (1984). URL `http://dx.doi.org/10.1007/BF00289138`
7. Conway, M.: Design of a separable transition-diagram compiler. Comm. ACM **6**(7), 396–408 (1963)
8. Crespi Reghizzi, S., Breveglieri, L., Morzenti, A.: Formal languages and compilation, 2nd edn. Springer, London (2013)
9. Engelfriet, J.: Iterating iterated substitution. Theor. Comput. Sci. **5**(1), 85–100 (1977). DOI 10.1016/0304-3975(77)90043-3. URL `http://dx.doi.org/10.1016/0304-3975(77)90043-3`
10. Gálvez, J.: A note on a proposed LALR parser for extended context-free grammars. Inf. Process. Lett. **50**(6), 303–305 (1994). URL `http://dx.doi.org/10.1016/0020-0190(94)00051-4`
11. Grune, D., Jacobs, C.: Parsing techniques: a practical guide, 2nd ed. Springer, London (2009)
12. Gruska, J.: On a classification of context-free languages. Kybernetika **3**(1), 22–29 (1967). URL `http://www.kybernetika.cz/content/1967/1/22`
13. Heilbrunner, S.: On the definition of ELR(k) and ELL(k) grammars. Acta Inform. **11**, 169–176 (1979)
14. Hemerik, K.: Towards a taxonomy for ECFG and RRPG parsing. In: Language and Automata Theory and Applications (LATA), *LNCS*, vol. 5457, pp. 410–421. Springer (2009). URL `http://dx.doi.org/10.1007/978-3-642-00982-2`
15. Jensen, K., Wirth, N.: Pascal User Manual and Report, Second Edition, *LNCS*, vol. 18. Springer (1975). DOI 10.1007/3-540-06950-X. URL `http://dx.doi.org/10.1007/3-540-06950-X`
16. Kannapinn, S.: Reconstructing LR theory to eliminate redundance, with an application to the construction of ELR parsers (in German). Ph.D. thesis, Tech. Univ. Berlin (2001)
17. Knuth, D.: On the translation of languages from left to right. Inform. and Contr. **8**, 607–639 (1965)
18. Knuth, D.: Top-down syntax analysis. Acta Inform. **1**, 79–110 (1971)
19. LaLonde, W.: Constructing LR parsers for regular right part grammars. Acta Inform. **11**, 177–193 (1979). URL `http://dx.doi.org/10.1007/BF00264024`
20. Lee, G., Kim, D.: Characterization of extended LR(k) grammars. Inf. Process. Lett. **64**(2), 75–82 (1997). URL `http://dx.doi.org/10.1016/S0020-0190(97)00152-X`
21. Lewi, J., De Vlaminck, K., Huens, J., Huybrechts, M.: A programming methodology in compiler construction. North-Holland, Amsterdam (1979). I and II
22. Morimoto, S., Sassa, M.: Yet another generation of LALR parsers for regular right part grammars. Acta Inform. **37**, 671–697 (2001)
23. Rosenkrantz, D., Stearns, R.: Properties of deterministic top-down grammars. Inform. and Contr. **17**(3), 226–256 (1970)
24. Sassa, M., Nakata, I.: A simple realization of LR-parsers for regular right part grammars. Inf. Process. Lett. **24**(2), 113–120 (1987). URL `http://dx.doi.org/10.1016/0020-0190(87)90104-9`
25. Wirth, N.: Algorithms + data structures = programs. Prentice-Hall, Englewood Cliffs, NJ (1975)