Fast GLR Parsers for Extended BNF Grammars and Transition Networks

Angelo Borsotti[a], Luca Breveglieri[a], Stefano Crespi Reghizzi[a,b], Angelo Morzenti[a]

[a] *Politecnico di Milano, Piazza Leonardo Da Vinci* 32, 20133 *Milano, Italy*
[b] *CNR-IIET, Piazza Leonardo Da Vinci* 32, 20133 *Milano, Italy*

Abstract

The Tomita's Generalized LR (1) parsing algorithm (GLR), later improved in many ways, runs in a linear time on LR (1) grammars, and degrades to a polynomial-time bound if the grammar is not deterministic. We address a useful feature not present in the current GLR (1) methods: the ability to accept grammars of the Extended BNF type (EBNF), the rules of which contain regular expressions. An EBNF grammar is conveniently represented by a collection of finite automata called a Transition Net (TN). We define, analyze and evaluate a new GLR (1) algorithm, called GELR, that combines the recent LR (1) parsing algorithm for TNs with the classical GLR data structures: the Graph-Structured Stack representing multiple stacks, and the Shared Packed Parse Forest for multiple syntax trees. The GELR algorithm is proved correct and an efficient implementation incorporating the state-of-the-art Right-Nulled parsing optimization is available. Experimental measures of the GELR parser size, speed and memory footprint are reported for current programming and web languages, and are compared with those of other parsing algorithms. The findings prove that directly parsing EBNF grammars does not penalize speed. Performance comparisons for different computer languages should also be of interest.

Keywords: GLR parsing, regular right-hand side grammar, parser performance comparison, graph-structured stack, GSS, shared packed parse forest, SPPF, infinite ambiguity.

1. Introduction

Given a grammar, a parser (syntax analyzer) is an algorithm that checks if a source text is syntactically correct and, in the case it is so, returns a syntax tree for the text, or more than one tree in the case of ambiguity. A parser generator takes a grammar and produces a parser or syntax analyzer.

Email addresses: angelo.borsotti@mail.polimi.it (Angelo Borsotti), luca.breveglieri@polimi.it (Luca Breveglieri), stefano.crespireghizzi@polimi.it (Stefano Crespi Reghizzi), angelo.morzenti@polimi.it (Angelo Morzenti)

The classical bottom-up LR (1) algorithm [12] applies to any deterministic Context-Free (CF) language, i.e., one recognized by a deterministic pushdown automaton. Such an algorithm is commonly used for parsing simple computer languages and it ensures a linear-time parsing complexity. Yet, it is not always suitable for more complex computer languages, and it is totally inadequate for the applications of natural language processing. In fact, the grammar restrictions imposed by the LR (1) condition are often violated by the CF grammars listed in the language reference manuals.

Sometimes the LR (k) algorithm, with $k \geq 1$, may overcome an LR (1) parsing conflict present in a grammar, but this is hopeless for those grammars that are ambiguous. Of course, when the shape of the language is not under the control of a compiler designer, we cannot assume the existence of an unambigous grammar, and even less the existence of a deterministic one; we have instead to take whatever grammar is listed in the language reference manual.

To address the parsing problem for such conflicting and ambiguous cases, various approaches have been followed in the past, which use a more general parsing algorithm. Clearly, to be practical a parser must not require the CF grammar to be constrained into a normal form, such as the Chomsky one needed by the CYK algorithm, e.g., in [7]. We mention the Earley parser that accepts any CF grammar, has a cubic-time parsing complexity and exists in different optimized versions such as [1, 2]. More relevant to our work, is the Tomita Generalized LR (1) parsing algorithm (GLR) [17], later improved in many ways. It provides a linear-time parsing complexity on LR (1) grammars, and it degrades to a polynomial-time complexity if non-LR (1) rules are present in the grammar. The GLR parsers are used to parse unrestricted CF languages in different applications, such as natural language processing and computer language compilation. For completeness, we mention that other general parsers use the top-down LL (k) algorithm as their base, instead of the bottom-up one.

Our contribution belongs to the line of GLR parsers and addresses a practical need disregarded by existing parsers: the ability to process a CF grammar that includes regular expressions in the rules (or productions). For many real computer languages, syntax is not defined by a "pure" CF grammar, also known as BNF, but is defined by a so-called *Extended* BNF (EBNF) grammar (also known as ECF), where the right-hand side of a rule is a regular language defined by a *regular expression* (RE). Such an RE may include the operations of union, concatenation and Kleene star, and may define languages of finite or infinite cardinality. In contrast, the right-hand side of a BNF rule is just a single string, and each nonterminal symbol is defined by a finite union of alternatives, i.e., a finite set of strings. To illustrate the improvement in expressiveness, a CF grammar needs recursive rules to generate a list of terms, while just one EBNF rule containing the iterative operator "$*$" (Kleene star) is sufficient.

Moreover, in the reference manuals an EBNF rule is often visualized by means of a graph, named syntax chart, that synthetically represents the finite-state automaton (FA) or *machine* that recognizes the regular language of the right-hand side of the rule. Thus, an EBNF grammar comprises a collection of FA graphs, one for each nonterminal symbol, which altogether we call a

transition net (TN).

The current GLR and LR parser generators lack support for EBNF grammars (or TNs), and this is a limitation that causes some annoyance and serious inconvenience for large languages such as COBOL. The critical issues raised by a conversion from EBNF to BNF, whether by hand or by a separate tool, are discussed at length in [10, 11]. First, the conversion from iterative to recursive rules, although straightforward, is not unique, especially since one may choose left or right recursion. Thus, the obtained BNF grammar is somewhat arbitrary and differs from the official EBNF one. Consequently, the structure assigned to a source text by the parser is not the same as the official one. Consider in particular the case of a list construct: through iterative rules all the list elements are generated at once, whereas through recursive rules the elements are layered one per level. Moreover, according to [10], removing the EBNF syntactic sugar may increase the size of the parsing automaton by nearly two orders of magnitude (imagine a DFA of type LR (1)). Second, the use of such an internal BNF grammar by the parser and by the subsequent compiler stages needs a costly tracking of the grammar changes from EBNF to BNF throughout the compiler.

Therefore, the main question we examine is whether a parser directly constructed from the official EBNF grammar or TN might be more efficient than a standard CF parser produced after converting the grammar into BNF. An analogy may help us introduce our approach. A comparable situation is found in the top-down parsing, where the popular deterministic recursive-descent ELL (1) parsers for EBNF grammars are generally preferred to the LL (1) parsers for CF grammars. Such ELL (1) parsers have been generalized to nondeterministic grammars, thus obtaining the so-called GLL parsers for EBNF grammars [16].

On the other hand, the development of deterministic bottom-up parsers for EBNF grammars or TNs has lagged behind the top-down case for many years. A long series of research attempts targeted at solving such a problem is critically surveyed in [8]. It includes the pioneering work [14] that, interestingly for us, starts from an EBNF grammar where each rule right part is a DFA, in other words a TN. However the latter work, as well as many others that followed, were not entirely successful [8]. Recently, a rigorous method for generating efficient deterministic parsers for TNs, and a set of related tools, have been made available under the name of ELR (1) parsing [3, 5]. Such ELR (1) parser and parser generator are here combined with the state-of-the art GLR (1) methods, in order to obtain a general Tomita-like parser for TNs.

We refer to [15] for a survey of the milestones in the development of the GLR (1) method and for the presentation of the related state-of-the-art parser, called RNGLR. Here, it suffices to say that GLR operates as a deterministic shift-reduce LR (1) parser [12],[1] until an LR (1) conflict occurs. We recall that an LR (1) parser data-structure is a pushdown stack, where the parser states (p-

[1] A simple variant of Knuth's LR (1) technique, named LALR (1) [6], is often used for those very large grammars that have an LR (1) parser size exceeding the available memory. In the GLR parsers, the LALR variant can be used without any change.

states) are pushed by shift moves and popped by reduction moves. In principle, to cope with the grammars that have LR (1) conflicts, the previous deterministic LR (1) pushdown automaton has to be turned into a nondeterministic one. By ruling out the inefficient simulation of a nondeterministic computation by backtracking, the GLR approach uses, instead of a pushdown stack, a data-structure, the so-called Graph-Structured Stack (GSS), that subsumes a multiplicity of pushdown stacks. Then, when an LR (1) conflict occurs, GLR keeps track of all the possible moves on its GSS. The GSS evolves during parsing and, at the end, it represents all the possible successful sequences of parsing moves. To compactly store the syntax tree(s) of the source text, a GLR parser uses a compact data-structure called Shared Packed Parse Forest (SPPF), where the subtrees that repeatedly occur are represented only once.

As said, the main goal of our research consisted of developing a new GLR parser generator that inputs EBNF grammars or transition nets, and outputs a parser that directly manages the syntax structures generated by EBNF grammars. The performances of our parser should then be compared against those of an existing state-of-the-art GLR parser.

Our approach augments the LR (1) parser generation algorithm for TNs, called ELR (1) parser [3], to cope with a grammar that has parsing conflicts, which cause a determinism loss possibly resulting also in ambiguity. Accordingly, we have named GELR (1) the new *Generalized* ELR parser for general unrestricted EBNF grammars. Technical problems had to be faced to make the usual data-structures (GSS and SPPF) of GLR parsers, suitable to EBNF grammars.

Furthermore, the issue of grammar ambiguity had to be carefully re-examined, because EBNF grammars may contain ambiguity caused by regular expressions, in addition to the syntactic ambiguity of CF grammars. In particular, we had to find a solution for the problem of the ambiguity of infinite degree, when such an ambiguity type is caused by an RE that applies a Kleene star to a language that includes the empty string.

This paper contributes the new parser generator and parser GELR (1) for general EBNF grammars or transition nets, and an open-source implementation thereof.[2] To ensure the needed trust and accuracy level, the paper includes the theoretical definitions, the validation of the main algorithms by formal proofs, and a testing based on parsing real computer languages. We also report time and space measurements, and an experimental performance comparison of GELR (1) with respect to a state-of-the-art GLR (1) algorithm.

Paper organization. Section 2 lists the essential definitions of EBNF grammar and transition net, and defines the structural equivalence and interchangeability of the two models. Section 3 introduces our parser generation method for TN as an evolution of the LR (1) and ELR (1) methods, combined with the GSS and SPPF data-structures of GLR (1) parsers. Section 4 proves that

[2] An interactive web tool is written in HTML-JS, and an efficient (command-line) one in Java; both are available on the GitHub repository at `https://github.com/FLC-project/GELR`.

the GSS graph resulting from parsing an input string correctly represents the derivations of the string. Section 5 describes the optimized parser generator and parser GELR, reports experimental measurements on real computer language grammars, and compares with the performance of existing parsers. Section 6 concludes and mentions future research directions.

2. Preliminaries

The following topics are needed as background: Finite Automata (FA); Regular Expressions (RE); CF grammars, their extension (named EBNF) with regular expressions (RE) in the right-hand sides of the rules, and their representation by syntax charts, which we prefer to call Transition Nets or TN; LR (1) parsing also for EBNF grammars; and Generalized LR (1) parsing or GLR (1). For brevity, we often omit the parameter (1).

For the basic concepts of FA, RE, CF grammars and LR (1) parsing, we just have to list our notation. For GLR (1) we follow the state-of-the art presentation in [15]. For the EBNF grammars and TNs, and for the ELR (1) parsing, we refer to [3] and we recall the essential elements here and in the next section.

The *terminal* alphabet is denoted by Σ and the empty string by ε. The special character "$\dashv$", included in Σ, is used exclusively as an end-marker (or terminator) of the input string. The elements (characters or letters) of a string x of length $|\,x\,| \geq 1$ are denoted by $x_0\, x_1\, \ldots\, x_{|\,x\,|-1}$. Numbering from 0 instead of 1 is more convenient for the strings that end with "$\dashv$".

A *deterministic finite automaton* (DFA) M is a tuple $(\,V,\, Q,\, \delta,\, q_0,\, F\,)$, where V is the input alphabet, Q is the state set, $q_0 \in Q$ is the initial state and $F \subseteq Q$ is the final state set. The state-transition function or graph is denoted by $\delta\colon Q \times V \to Q$, and it is extended as usual to $Q \times V^* \to Q$. In the graph of δ, a path from the state q_0 to a state $q \in F$ is *accepting*, and its label is a string over V accepted by M. The recognized language $L\,(M) \subseteq V^*$ is the set of all the accepted strings. We assume that all DFAs are trim: every state can be reached from the initial state q_0 and reaches a final state in F.

Definition 1 (extended CF grammar – EBNF). An EBNF *grammar* G is a 4-tuple $(\,\Sigma,\, V_N,\, P,\, S\,)$, where V_N is the *nonterminal alphabet* disjoint from the terminal alphabet Σ, P is the set of *rules* and $S \in V_N$ is the *axiom*. A *grammar symbol* is an element of the *total alphabet* $V = \Sigma \cup V_N$. For each nonterminal $A \in V_N$, there is exactly one rule $(\,A \to \alpha\,) \in P$: nonterminal A is the *left symbol* and α, the *right-hand side*, is a regular expression (RE) over V. Such REs may use concatenation, union (denoted as "$|$"), the Kleene star "$*$" and cross "$+$", the empty string symbol "ε" and the parentheses. The language over V defined by the RE α is called the *regular language associated* with A and is denoted by R_A, or also by $R\,(\alpha)$.

A (non-extended) CF grammar, also known as BNF, is an EBNF grammar such that for any rule $(\,A \to \alpha\,) \in P$ the right-hand side has the form: $\alpha = \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$, with $\alpha_i \in V^*$ for $1 \leq i \leq n$. □

We later assume that every grammar contains by default the *special rule* $S' \to S \dashv$, where nonterminal S' acts as axiom and does not occur in the other rules. Under such an assumption an input string is never empty and always ends by the end-marker "$\dashv$".

Definition 2 (derivation, language and syntax tree). For a given 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 $w \in R\,(\alpha)$, i.e., it is in the regular language R_A associated to nonterminal A. The *derivation*, denoted $\overset{*}{\Rightarrow}$, is the reflexive and transitive closure of the *immediate* derivation relation $\Rightarrow$. A *derivation* in n steps, with $n \geq 1$, is a sequence $z_0 \Rightarrow z_1 \ \ldots \ \Rightarrow z_n$. A derivation is *leftmost* if, $\forall\ i \in [\,0 \ldots n-1\,]$ such that $z_i = u\,A\,v$ and $z_{i+1} = u\,w\,v$, it holds $u \in \Sigma^*$. The reverse relation of an immediate derivation $w \Rightarrow z$ is named *reduction* and is denoted by $z \rightsquigarrow w$. The language generated by a grammar G starting from nonterminal $A \in V_N$ is:

$$L\,(\,G,\,A\,) = \left\{\, x \in \Sigma^* \mid \quad A \underset{G}{\overset{*}{\Longrightarrow}} x \,\right\}$$

The *language generated* by G is $L\,(G) = L\,(\,G,\,S\,)$, i.e., from the axiom S.

The derivation of a string $x \in L\,(G,\,S)$ is also represented by a *syntax tree* of x. In a syntax tree, a node having or not having child nodes is respectively called *internal* node or *leaf* node. Each internal node of a syntax tree is labeled by a nonterminal symbol (the root node by S), and each leaf node by a terminal symbol or "ε". Let t be the root node of a syntax tree, then the *frontier* of such a node, denoted by *frontier* (t), is the string x.

For every nonterminal A, assume that a derivation step $A \Rightarrow u$ rewrites A and that symbol A labels an internal node t: if $|\,u\,| > 0$ then node t has a number $|\,u\,|$ of child nodes labeled, from left to right, by u_0, u_1, $\ldots$, $u_{|\,u\,|-1}$; if $|\,u\,| = 0$ then node t has one child node labeled "ε".

Given a syntax tree of a string $x = x_0 \ \ldots \ x_{n-1}$ of length $n > 0$, the *indexed syntax tree* (IST) is obtained by decorating each tree node with two integer indices h and k, with $0 \leq h \leq k \leq n-1$, that identify a substring of x. More precisely, let t be a syntax tree node and let X be its label: then in the IST the label of node t is $X_{h,\,k}$ if, and only if:

- $h < k$ and *frontier* $(t) = x_h \ \ldots \ x_{k-1}$, where $\forall\ h \leq j \leq k-1 \colon x_j \in \Sigma$
- or $h = k$ and *frontier* $(t) = \varepsilon$ □

Some examples of derivation, syntax tree and indexed tree are shown in Ex. 1.

Remark 1. From Def. 2, it follows that if $R_A \subseteq V^*$ is an infinite language and $\beta \in R_A$, the immediate derivation $A \Rightarrow \beta$ may rewrite nonterminal A into a string of unbounded length, and in the syntax tree the node A has an unbounded arity (number of child nodes). Therefore, the shape of the EBNF syntax trees is *shallower* than that of the BNF trees. □

We assume that all the grammars are *reduced*: for every nonterminal $A \in V_N$ there exist derivations $S \stackrel{*}{\Rightarrow} u\,A\,v$ and $A \stackrel{*}{\Rightarrow} w$, where $u, v \in V^*$ and $w \in \Sigma^*$. Therefore, for every $A \in V_N$ the language R_A is not empty, and it follows that also $L\,(G, A\,) \neq \emptyset$. A language that contains the empty string ε is called *nullable*. Two grammars are *equivalent* if they define the same language. It is well known that EBNF and BNF grammars have the same generative capacity.

Transition net. As mentioned, an EBNF grammar is often graphically presented in a user friendly way by a finite collection of deterministic finite automata (DFA), named a syntax chart or a transition net (TN), the difference between the two models being in the graphical style and in the fact that the states have a name (in a TN) or are unnamed (in a syntax chart). For the coming constructions and proofs, TNs are more manageable than REs and syntax charts, and we move to their definition (in principle, it would be possible to define TNs even by using nondeterministic finite automata, but we do not pursue this direction). Given an RE, classical algorithms exist to construct a language-equivalent finite automaton, e.g., those by McNaughton-Yamada, Glushkov, Berry-Sethi, and Thompson. Anyway, we do not have to describe them, since any algorithm that produces a DFA would do in our case. By applying any RE-to-DFA conversion algorithm, we obtain a correspondence between each EBNF rule $A \to \alpha$ and a DFA that recognizes the (regular) language $R_A \subseteq V^*$.

Let $G = (\,\Sigma, V_N, P, S\,)$ be an EBNF grammar, as in Def. 1. The next formalization is taken from [3].

Definition 3 (transition net – TN). A *transition net* (TN) $\mathcal{M}$ is a finite collection of DFAs called *machines*, i.e., $\mathcal{M} = \{\, M_S, M_A, M_B, \ldots \}$, where M_S is the *starting (or axiomatic) machine*. Each machine is a 5-tuple $M_A = (\,V, Q_A, \delta_A, 0_A, F_A\,)$, where $V = \Sigma \cup V_N$ and $\Sigma \cap V_N = \emptyset$.

The *state set of* $\mathcal{M}$ is the union $Q = \bigcup_{M_A \in \mathcal{M}} Q_A$. Without loss of generality, we assume $Q_A \cap Q_B = \emptyset$ for $M_A \neq M_B$, and we define the *transition function* (or graph) of the TN as $\delta = \bigcup_{M_A \in \mathcal{M}} \delta_A$. To simplify the notation, we write δ instead of δ_A at no risk of confusion.

The *regular language*, over terminals and nonterminals, recognized by a machine M_A, starting from a state $q_A \in Q_A$, is:

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

We say that a TN $\mathcal{M}$ is *structurally equivalent* to a grammar $G = (\Sigma, V_N, P, S)$ if there is a one-to-one correspondence between the nonterminals in the set V_N and the TN machines, and if for each rule $(\,A \to \alpha\,) \in P$ the identity $R\,(M_A, 0_A) = R_A$ holds.

The (terminal) CF *language recognized by the* TN starting from a machine M_A, is defined as $L\,(M_A, 0_A) = L\,(G, A)$, and the *language defined by the* TN is $L\,(\mathcal{M}) = L\,(M_S, 0_S) = L\,(G)$.

The next property of some machine states will be important for optimizing the parser. A state q_A is *right-nullable* if the language $R\,(M_A, q_A) \subseteq V^*$ contains a non-empty string $u \subseteq V_N^+$ that is nullable, i.e., such that $u \stackrel{*}{\underset{G}{\Longrightarrow}} \varepsilon$. □

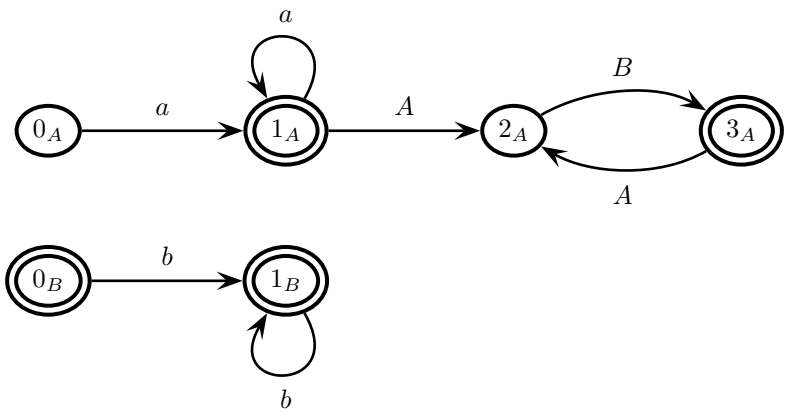
Notice that if state M_A is final, it can be right-nullable or not.

Since we use a TN as a faithful representation of a grammar, we only consider TNs that are structurally equivalent to EBNF grammars that, by previous assumptions, are reduced. Therefore, the following properties hold for any machine M_A: (*i*) M_A is trim, (*ii*) M_A is recursively reachable from M_S, and (*iii*) $L\ (\ M_A,\ 0_A\ ) \neq \emptyset$.

Derivations and trees of a TN. The language defined by a TN is equal to the one defined by a structurally equivalent grammar (Def. 3), thanks to the one-to-one correspondence between grammar rules and machines, and more precisely between grammar derivations and computations, i.e., paths in the machine graphs. It is obvious that the above correspondence extends to the syntax trees, thus allowing us to speak indifferently of trees for grammars or for TNs.

The next example illustrates the relation between a grammar and a structurally equivalent net, and shows derivations, computations and syntax trees.

(a) *transition network* $\mathcal{M} = \{\ M_A,\ M_B\ \}$, where A is the axiom



(b) *structurally equivalent* EBNF *grammar* $G\colon A \to a^+\ (\ A\ B\ )^* \qquad B \to b^*$

Figure 1: Part (a) – transition network $\mathcal{M}$. Part (b) – an EBNF grammar G structurally equivalent to $\mathcal{M}$. State 2_A is right-nullable.

Example 1 (transition net and EBNF grammar). The grammar G with rules $A \to a^+\ (\ A\ B\ )^*$ and $B \to b^*$ is structurally equivalent to the TN $\mathcal{M} = \{\ M_A,\ M_B\ \}$ shown in Fig. 1.a, since:

$$R\ (\ M_A,\ 0_A\ ) = a^+\ \ (\ A\ B\ )^* \qquad \text{and} \qquad R\ (\ M_B,\ 0_B\ ) = b^*$$

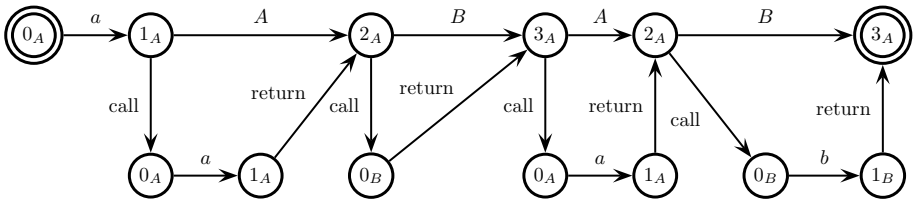
In general, we prefer to represent a grammar rule by means of a machine that is minimal with respect to the number of states. On the other hand, machine M_B is not minimal since its two states are equivalent and could be merged. The reason for not minimizing M_B is that we prefer to work with machines where

the initial state is not reentered by a path; such a constraint has a negligible effect on the machine size, yet it enables some simplification in the parser.

From structural equivalence it follows that for any string in $L(G)$ there is a one-to-one correspondence between (say) leftmost derivations and TN computations. To illustrate, the next derivation:

$$A \Rightarrow a\,A\,B\,A\,B \Rightarrow a\,a\,B\,A\,B \Rightarrow a\,a\,\varepsilon\,A\,B \Rightarrow a\,a\,a\,B \Rightarrow a\,a\,a\,b$$

corresponds to the following computation paths:



Thus, the string $a\,a\,a\,b$ is in $L(\mathcal{M})$ and has the syntax tree numbered 5 in Fig. 2. In the same figure, four additional syntax trees of the same string are shown, as the string has ambiguity degree 5. The indexed tree that represents the derivation / computation of tree n. 5 is shown in Fig. 2, bottom. Notice that the layers of the computation paths shown above correspond to those of the syntax tree. □

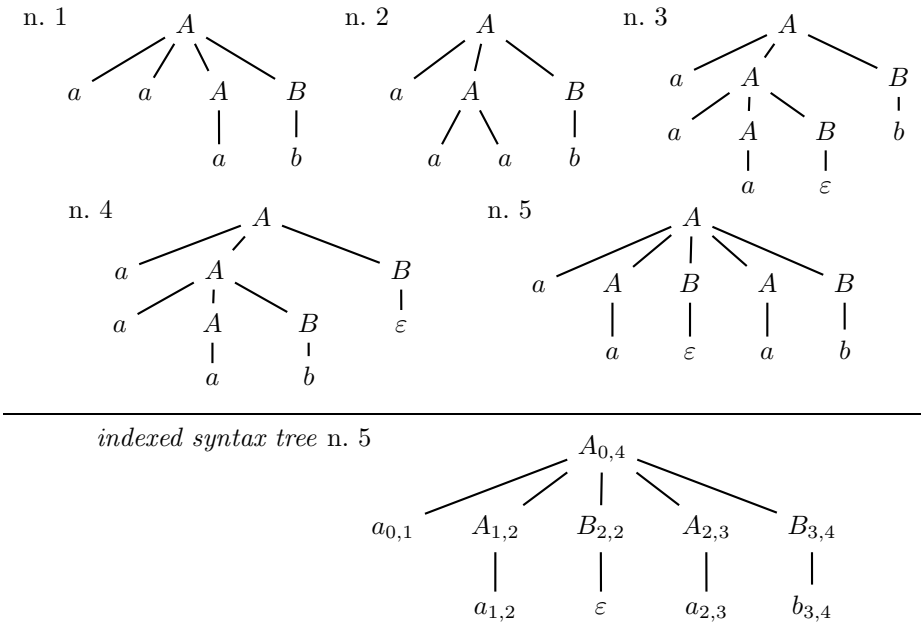


Figure 2: The five syntax trees of string $a\,a\,a\,b$ of Ex. 1; below, the indexed syntax tree n. 5.

In the next sections, we often use TNs instead of EBNF grammars as the initial representation from which the parser is constructed. But this makes little difference since the conversion from a TN to a structurally equivalent EBNF grammar is straightforward. It suffices to compute by standard methods an RE R_A for each machine M_A, to create the rule $A \to R_A$, and to choose as axiom the nonterminal corresponding to the starting machine.

3. Generalized ELR parsing

This central section is divided in five parts. It starts with our notation for (deterministic) LR (1) and ELR (1) parsers, which differs from Knuth's mainly because each so called *item* is keyed to a TN machine state rather than to a position within a grammar rule (production). The parsing tables that control a parser, encode a DFA that lacks a standard name in the literature (sometimes it is called LR automaton), to be here called *pilot*. The pilot states are called *parser states* (*p-states*), to avoid confusion with the states of the TN machines.

In the second part, we move to the nondeterministic case and we define the new parser states needed by the pilot (called Generalized ELR (1) or GELR (1)) when a grammar or a TN has parsing conflicts. We present the GELR (1) pilot construction algorithm, and a suitable representation of the syntax trees where the nodes are keyed to p-states.

The third part starts with an axiomatic definition of the classical *Graph Structured Stack* (GSS) data-structure used by GLR (1) parsers to represent multiple pushdown stacks extended to GLR (1). Then it shows, by means of an example, how the parser constructs the GSS. The fourth part recalls the classical *Shared Packed Parse Forest* (SPPF) used by the GLR (1) parsers to compactly represent the syntax trees of ambiguous sentences. The last part of the section examines the case of the sentences that are infinitely ambiguous.

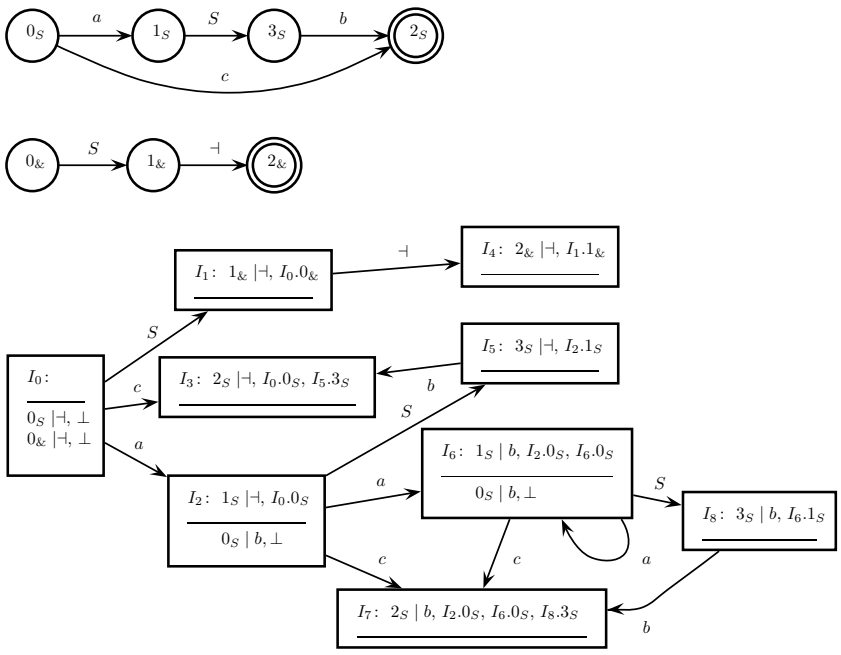
3.1. Notations for items and parser states of LR (1) *and* ELR (1) *grammars*

It would be long and unnecessary to give a full account of the construction of an LR (1) parser, which is found in any compiler textbook. Given a TN structurally equivalent to an (E)BNF grammar, we just need to present our notation for encoding the items that occur in the p-states of the pilot.

Example 2 (pilot notation). The grammar $S \to a\ S\ b \mid c$, augmented with the (axiomatic) rule $S' \to S\ \dashv$, is represented by the TN in Fig. 3 (top). Two versions of the LR pilot are depicted in the figure: in the middle, the pilot produced by our tool,[3] and at the bottom a part of the classical pilot. □

[3] All the TN and pilot graphs reproduced in this paper were produced by our Javascript parser generator tool. We trust the reader to adjust for minor differences with respect to the formal notation we use.

Grammar, transition net and pilot: $S \to a\ S\ b \mid c$



Classical LR (1) *pilot (only a fragment)*:

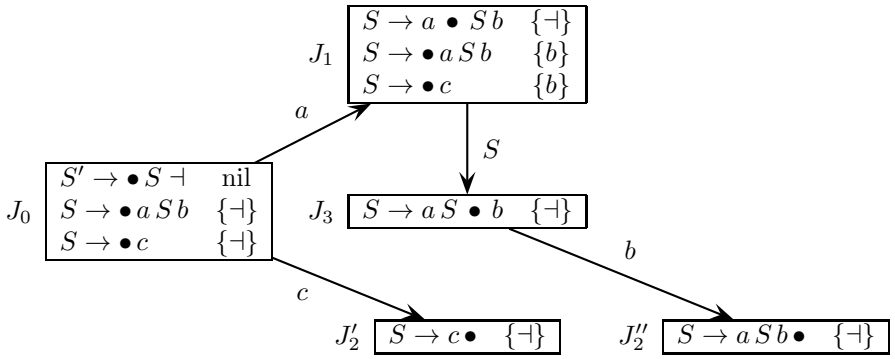


Figure 3: Grammar, TN and our ELR (1) pilot $\mathcal{P}$; parts of the classical LR (1) pilot.

We compare the two pilots. It suffices to examine the differences between the corresponding p-states I_2 and J_1 that are reached from the initial p-states by shifting a terminal a.

I_2 The p-state I_2 contains two items, written on separate lines. The first item is the state 1_S of machine M_S, which is entered from the initial state

0_S of the same machine after shifting a terminal a; the look-ahead set of the item is $\{ \dashv \}$. The second item is the state 0_S with look-ahead $\{ b \}$.

J_1 In the p-state J_1 there are three items. The first one is the dotted rule $S \to a \bullet S\ b$ with look-ahead $\{ \dashv \}$; clearly, such a dotted rule carries the same information as the state 1_S. The second and third items are the dotted rules $S \to \bullet\ a\ S\ b$ and $S \to \bullet\ c$, both with look-ahead $\{ b \}$. The two together have the same meaning as state 0_S, namely that the right-hand side $a\,S\,b$ or the right-hand side c are next expected, and that the look-ahead expected for eventually performing a reduction is $\{ b \}$.

Thus, the evident difference between the two pilots is the use of machine states instead of dotted rules. But we focus on a second difference, more relevant for us. We notice that the ELR pilot has fewer p-states than the LR one. For instance, p-states J_2' and J_2'' correspond to p-state I_3, the reason being that in the machine M_S the arcs labeled b and c enter the same final state 2_S, whereas in the grammar there are two separate right-hand sides: $a\,S\,b \mid c$. When the LR parser enters either J_2' or J_2'', it respectively performs the reductions $c \rightsquigarrow S$ or $a\,S\,b \rightsquigarrow S$. On the other hand, the ELR parser, on entering p-state I_3, needs to know which reduction to perform, since both are possible. To this end, item 2_S includes the names of the predecessor items, either $I_0.0_S$ or $I_5.3_S$. Such an information is just a convenience for ELR parsing, but it becomes more important when nondeterministic parsing comes into play.

3.2. Construction of the GELR(1) *parser pilot*

Given a TN, which represents a structurally equivalent EBNF grammar, we construct the *pilot* that controls the parser. The construction differs from the one [3] for the ELR case, described in Sect. 3.1, in that the pilot is enriched with the information needed to parse the TNs when conflicts (shift-reduce or other) occur. First, we define the new structure of the p-state items, which we call generalized.

Definition 4 (generalized ELR(1) item). Item, notation and operations:

- For a TN $\mathcal{M} = \{ M_S,\ M_A,\ \dots \}$, a GELR(1) *item* is a 3-tuple:

machine state $q_A \in Q_A$	*look-ahead set* $\pi \subseteq \Sigma$ (including $\dashv$)	*pointer set* λ: a possibly empty finite set of pairs of type $\langle$ *pilot-state*, *machine-state* $\rangle$, where each such pair points to an item in a predecessor p-state

(1)

  Since the state and look-ahead components have exactly the same meaning as in the deterministic case [3], we omit their definitions.

- Given any two *items* $\gamma_1 = \langle\, q, \pi_1, \lambda_1 \,\rangle$ and $\gamma_2 = \langle\, q, \pi_2, \lambda_2 \,\rangle$ that have an identical state q and differ in their look-ahead and/or pointer fields, we define

a *coalesce* operation that returns the item $\gamma = \langle\, q,\, \pi_1 \cup \pi_2,\, \lambda_1 \cup \lambda_2 \,\rangle$. Given a set $I = \{\, \langle\, q_1,\, \pi_1,\, \lambda_1 \,\rangle,\, \langle\, q_2,\, \pi_2,\, \lambda_2 \,\rangle,\, \ldots \,\} \neq \emptyset$, let I' be the item set obtained by applying all the possible coalesce operations to the items of I. Then in I' the state component becomes a key that identifies uniquely the items.

- For any item set I, we define the *projection*, denoted by $I_{q,\,\pi}$, on the *state* and *look-ahead* components:

$$I_{q,\,\pi} = \{\, \langle\, q,\, \pi \,\rangle \mid \quad \exists\, \lambda \quad \langle\, q,\, \pi,\, \lambda \,\rangle \in I \,\}$$

- We say that two (coalesced) item sets I and J are *pilot-equivalent*, written $p_eq\,(I,\, J)$, iff their projections are equal:

$$p_eq\,(I,\, J) \quad \Longleftrightarrow \quad I_{q,\,\pi} = J_{q,\,\pi} \qquad \square$$

As in the LR and ELR approaches, a GLR parser state (p-state) is represented by an item set. To compute the pilot p-state transition graph, the classical closure function [12] on item sets is next reformulated for the GELR (1) case.

Definition 5 (closure function). Let I be an item set of a TN $\mathcal{M}$ and let 0_A be the initial state of machine M_A. The closure is the least fixed point of the following function:

$$\mathit{closure}\,(I) = \mathit{coalesce}\left(I \cup \left\{ \langle\, 0_A,\, \sigma,\, \emptyset \,\rangle \;\middle|\; \begin{array}{l} \exists\, \langle\, q,\, \pi,\, \lambda \,\rangle \in \mathit{closure}\,(I) \\ \text{such that } r \in \delta\,(q,\, A) \\ \text{and } \sigma = \mathit{First}\,\big(L\,(r) \cdot \pi\big) \end{array} \right\} \right)$$

where the function *First* returns the set of the initial characters of any string in the language $L\,(r) \cdot \pi$ (notice that $\varepsilon \notin L\,(r) \cdot \pi$). $\square$

Notice that, in the new item $\langle\, 0_A,\, \sigma,\, \emptyset \,\rangle$ generated by a closure, the pointer component is empty.

Definition 6 (pilot graph). The pilot, denoted by $\mathcal{P}$, is a DFA defined by the following entities:

- the input alphabet $V = \Sigma \cup V_N$ (alphabet Σ also includes the end-of-text $\dashv$)
- the set $R = \{\, I_0,\, I_1,\, \ldots \,\}$ of *p-states*, where each p-state I_i contains a set of *items*, see Def. 4, and is defined as said below
- the *initial p-state* I_0, defined as $I_0 = \mathit{closure}\,(\,\{\, \langle\, 0_S,\, \{\, \dashv \,\},\, \emptyset \,\rangle \,\}\,)$
- the *pilot state-transition* (partial) *function* $\vartheta\colon R \times (\,\Sigma \cup V\,) \to R$, which is defined below in Eq. (2)

We define another partial function $\mathit{shift}\colon \mathit{set\ of\ items} \times (\Sigma \cup V) \to \mathit{set\ of\ items}$, as follows. Let $\gamma = \langle\, p_A,\, \rho,\, \lambda \,\rangle \in I$ and let $X \in V$, then:

$$\mathit{shift}\,\big(\,\langle\, p_A,\, \rho,\, \lambda \,\rangle,\, X\big) = \begin{cases} \langle\, q_A,\, \rho,\, \langle\, I,\, p_A \,\rangle \,\rangle & \text{if } M_A \text{ has an arc } p_A \xrightarrow{X} q_A \\ \text{undefined} & \text{otherwise} \end{cases}$$

For any p-state I and any grammar symbol X, the function ϑ is defined as:

$$\vartheta\,(I,\,X) = \mathit{coalesce}\left(\bigcup_{\forall\ \text{item}\ \gamma_i\ \in\ I} \mathit{shift}\,(\gamma_i,\,X)\right) \tag{2}$$

Function ϑ is computed as the (coalesced) union of the shifts of all the items in the p-state I (clearly, if such a union is empty the next p-state is undefined).

In Eq. (2), notice that all the duplicated items are merged, hence the value $\vartheta\,(I,\,X)$ contains h items, with $0 \leq h \leq |\,I\,|$. A state $J \in R$ is said to be a *reduction p-state* for a nonterminal $A \in V_N$ if, and only if, it includes an item that has a final state of machine M_A.[4] □

Algorithm for pilot construction. Given a TN, Alg. 1 computes the pilot $\mathcal{P}$ of the TN. Although the algorithm is standard we include it to introduce the notation later needed. It adds a new p-state at a time, possibly coalesces the new one with an existing one that is p-equivalent, and adds the related new arc. It terminates when nothing has changed in the last iteration. Alg. 1 always terminates since for a given TN the number of possible p-states is bounded, and the inner loop either adds a new p-state or modifies an existing p-state by coalescing, which results in a new p-state containing more items.

Example 3 (pilot graph). By applying the algorithm to the TN of Fig. 1, we obtain the pilot graph in Fig. 4 (running example). □

It is known that a GLR parser performs as a deterministic LR one until it enters a conflictual p-state, from where different computations depart, which will be represented in the graph structured stack (GSS); the GSS subsumes all the possible stack configurations generated by those computations.

To prepare for the discussion of the GSS construction phase, we introduce a way for directly representing the p-states traversed by the parser, on the syntax tree of the input string.

Definition 7 (p-state-labeled indexed syntax tree – LIST). For each indexed syntax tree (IST) of a string $x \in L\,(\mathcal{M})$, define the *p-state-labeled indexed syntax tree* (LIST) of x as follows:

- to each node t of the IST, except for the root node S', attach two p-states as subscript, respectively called the *left* and *right p-states* of t
- the left p-state of node S, which is the left child node of the derivation $S' \Rightarrow S\ \dashv$, is the initial p-state I_0
- for any non-leaf node $t \neq S'$ that has a sequence of $k \geq 1$ child nodes $u_1\ \ldots\ u_k$:

[4] In Sect. 5 we describe an optimization that applies to a p-state I and allows to perform a reduction to nonterminal A when the p-state I contains an item with a machine state that is final or *right-nullable* (see Def. 3) for machine M_A.

Algorithm 1: Computation of the GELR pilot.

Input: a machine net $\mathcal{M}$
Output: the pilot of $\mathcal{M}$

```
// vars:  p-state sets R' (current) and R (stored), pilot ϑ
```
$I_0 := \mathit{closure}\,(\,\{\,\langle\, 0_S,\, \{\,\dashv\,\},\, \emptyset\,\rangle\}\,)$ `// compute initial p-state` I_0
$R' := \{\, I_0 \,\}$ `// initialize current p-state set` R' `with` I_0
do `// update p-state set and transition function`
  $R := R'$ `// store current p-state set` R' `into` R
  `// compute new p-states and arcs if any`
  foreach (p-state $I \in R$ and symbol $X \in V$) **do**
    $I' := \mathit{closure}\,(\vartheta\,(I,\, X))$ `// shift to (new) p-state` I'
    `// verify if there is a pilot transition from` I `by` X
    if $(I' \neq \emptyset)$ **then** `// transition exists`
      `// verify if there is a p-state` J `p-equivalent to` I'
      if $(\exists\, J \in R' \quad p_eq\,(J,\, I'))$ **then** `// coalesce and update`
        $J' := \mathit{coalesce}\,(J \cup I')$
        in the p-state set R', replace p-state J with p-state J'
        add arc $I \xrightarrow{X} J'$ to the graph of ϑ
      else `// just update`
        add p-state I' to p-state set R'
        add arc $I \xrightarrow{X} I'$ to the graph of ϑ
while $(R' \neq R)$ `// repeat if p-state set` R' `has changed`

- the left p-state of the leftmost child node u_1 is the same as the left p-state of the parent node t
- $\forall\, i \in [2 \ldots k]$ the left p-state of the child node u_i is the same as the right p-state of the (preceding) sibling node u_{i-1}

- for each tree node t that has a left p-state I, the right p-state J is as follows:
  - if $t \in V$ then $J = \vartheta\,(I,\, t)$
  - if $t = \varepsilon$, i.e., t is a leaf node representing the empty string, then $J = I$ □

Example 4 (p-state labeled indexed syntax tree). The LIST of the string $a\,a\,a\,b\,\dashv$ of the running example (Fig. 4) is shown in Fig. 5. □

We prove that for any indexed syntax tree, the corresponding p-labeled indexed syntax tree, as in Def. 7, always exists.

Lemma 1 (correctness of LIST − Def. 7). *The definition of p-state-labeled indexed syntax tree is well founded: for every tree node $t \in V$ that has a p-state I as its left p-state, there exists a p-state $J \in R$ such that $J = \vartheta\,(I,\, t)$.* □

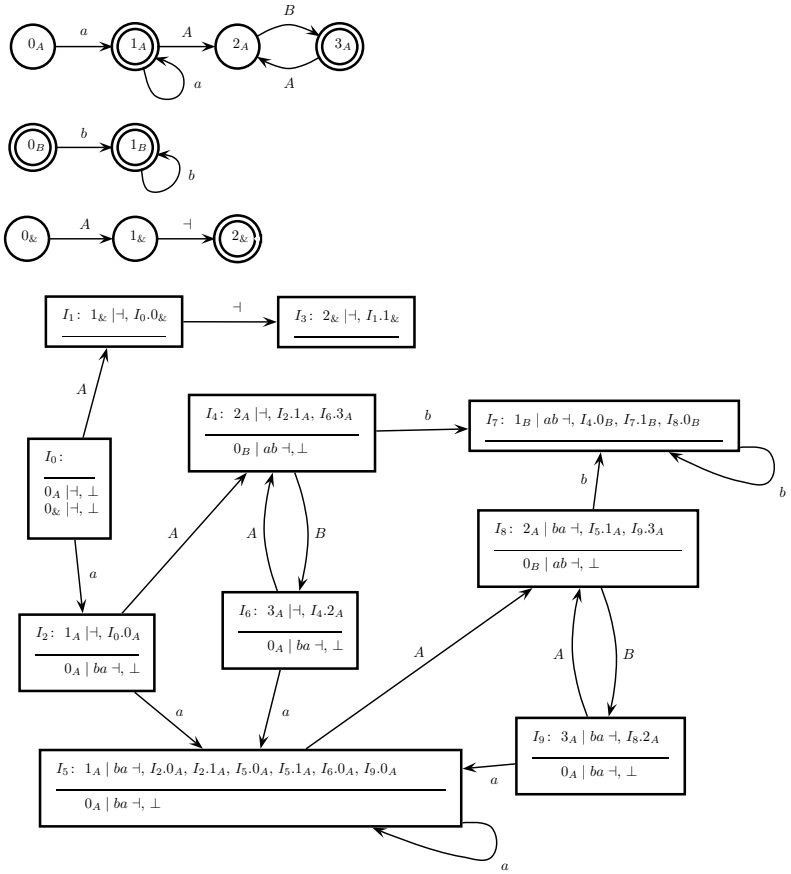


Figure 4: The GELR (1) pilot for the TN reproduced from Fig. 1 (running example).

Proof of Lemma (1). Consider any internal node t of the syntax tree, and the sequence of the child nodes of t:

1. If $t = S'$ then the child nodes are S and $\dashv$. The right p-state of S is $I_1 = \vartheta\,(I_0, S)$, which is also the left p-state of $\dashv$, the right p-state of which is $I_2 = \vartheta\,(I_1, \dashv)$; both p-states I_1 and I_2 are present in every pilot graph.

2. Every internal node other than the root node is a nonterminal $A \neq S'$. Let I be the left p-state of A and let σ be the sequence of the child nodes of A. We consider the case where $\sigma = u_1 \ldots u_k$, with $k \geq 1$ and $u_i \in V$ for all $i \in [1 \ldots k]$, because if node A has one child node ε then the left and right p-states of the leaf node ε are equal to I, according to the last clause of Def. 7. From the definition of derivation and from the pilot construction algorithm (Alg. 1), p-state I includes an item with machine state 0_A (as a result of the application of the closure function) and machine M_A includes

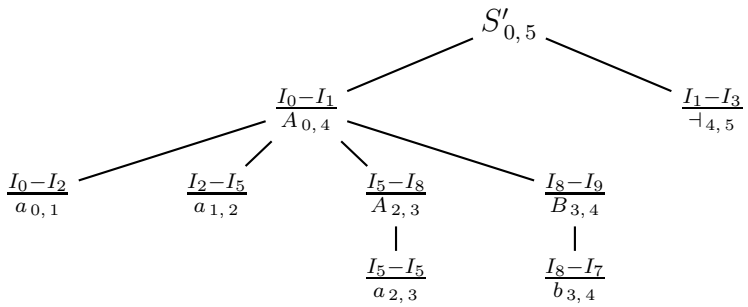


Figure 5: LIST for the derivation $S' \Rightarrow A \dashv \Rightarrow a\, a\, A\, B \dashv \Rightarrow a\, a\, a\, B \dashv \Rightarrow a\, a\, a\, b \dashv$. The subtree rooted at $A_{0,4}$ corresponds to tree n. 1 in Fig. 2 and the root S' to the special rule $S' \to A \dashv$.

a path $0_A \xrightarrow{u_1} i_1 \ldots \xrightarrow{u_k} i_k$, where $i_j \in Q_A$ for all $j \in [1 \ldots k]$ and $i_k \in F_A$. Therefore, the pilot includes a corresponding path $I \xrightarrow{u_1} I_1 \ldots \xrightarrow{u_k} I_k$ where, for all $j \in [1 \ldots k]$, p-state I_j includes an item with machine state i_j. Furthermore, if $u_j \in V_N$ then p-state I_{j-1}, which is also the left p-state of node u_j, includes an item with machine state 0_{u_j} (from the application of the closure function) and the same p-state I_{j-1} is the left p-state of the leftmost child node of u_j.

Since the reasoning at point (2) above applies to every non-leaf tree node and to the child nodes thereof, the left and right p-states are well defined for every (non-root) node of the syntax tree. □

3.3. Graph-structured stack

Given an input string, the parser scans it from left to right and produces a graph, called *graph-structured stack* (GSS), that compactly represents the stacks built by all the possible computations. Our graph is similar to the GSS described in [15], in turn based on earlier GSS definitions; it just differs from it in that a syntax tree of an EBNF grammar may contain nodes of unbounded arity. Since a GSS is the established data-structure created by a GLR parser, we do not explain how it is constructed, and we formally define the properties of our GSS model, which later will allow us to prove the correctness of our parser.

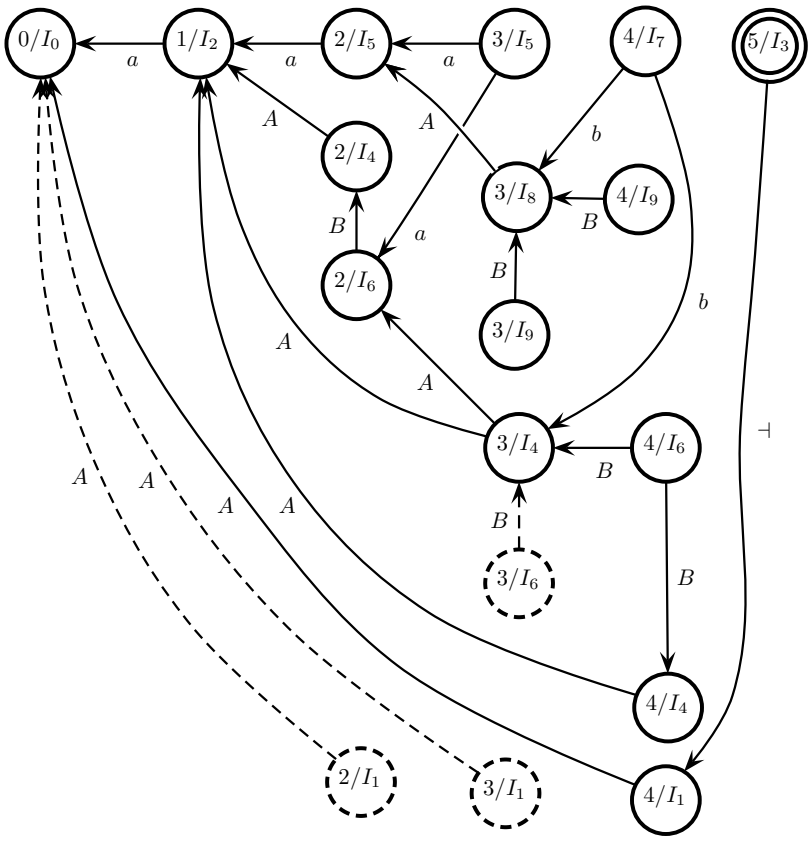


Figure 6: The GSS for the input string $a\,a\,a\,b$, to be compared with the pilot of the running example shown in Fig. 4. Notice that the dashed nodes and arcs do not contribute to any syntax tree for the string (see Fig. 8); all of them can be canceled from the GSS.

Axiomatic definition of GSS.

Definition 8 (graph-structured stack – GSS). Let $x = a_0 \ldots a_{n-1} \in \Sigma^+$, with $n \geq 1$, be an end-marked input string, with $a_0 \ldots a_{n-2} \in \big(\Sigma \setminus \{ \dashv \}\big)^*$; notice that the case $x = a_0$ with $a_0 = \dashv$ is possible. Also let $\mathcal{M}$ be a TN with pilot $\mathcal{P} = \langle V, R, \vartheta, I_0, - \rangle$, where "–" means don't care.

Then the *graph-structured stack* (GSS) of x is a directed labeled graph $\Gamma = \langle W, E \rangle$, where the node set is $W \subseteq \{ 0, \ldots, n \} \times R$ and the arcs are labeled by the input alphabet V of the TN (including the end-mark $\dashv$), i.e., set $E \subseteq W \times V \times W$. Graph Γ is defined by the following clauses:

1. **If** $\langle h, I \rangle \in W$ **and** $\vartheta(I, a_h) = J$, with $h \neq n$, **then** $\langle h+1, J \rangle \in W$ **and** set E includes the arc $\langle h, I \rangle \xleftarrow{a_h} \langle h+1, J \rangle$, called a *terminal arc.*

2. **If** $\langle\, h,\, I \,\rangle \in W$ **and** p-state I includes an item with machine state 0_A for a nonterminal $A \in V_N$ **and** p-state I is a reduction one for A, **then** $\langle\, k,\, J \,\rangle \in W$ **and** set E includes the arc $\langle\, h,\, I \,\rangle \overset{A}{\longleftarrow} \langle\, h,\, J \,\rangle$, called a *nullable nonterminal arc*.

3. **If** $\langle\, h,\, I \,\rangle \in W$ **and** p-state I includes an item with machine state 0_A for a nonterminal $A \in V_N$ **and** there exists a reduction p-state $H \in R$ for A **and** graph Γ has a path $\langle\, h,\, I \,\rangle \longleftarrow \langle\, k,\, H \,\rangle$ of length ≥ 1 that corresponds to a (reversed) accepting path of machine M_A **and** $\vartheta\,(I,\, A) = J$, **then** $\langle\, k,\, J \,\rangle \in W$ **and** set E includes the arc $\langle\, h,\, I \,\rangle \overset{A}{\longleftarrow} \langle\, k,\, J \,\rangle$, called a *nonterminal arc*.

4. The *initial node* of graph Γ is $\langle\, 0,\, I_0 \,\rangle$, where I_0 is the initial p-state of $\mathcal{P}$.

5. The *final* (or *accepting*) *node* of Γ, if present in W, is $\langle\, n,\, I_f \,\rangle$, where I_f is the p-state that includes the item with the final state of the machine for the special rule $S' \to S \dashv$ (in the figures this state is denoted $2_{\&}$, double lined).

6. Graph Γ includes no node or arc other than those from the clauses above. □

To illustrate, in Fig. 6 we show the GSS of the string $a\,a\,a\,b$. For further reference, Fig. 2 shows the five trees of string $a\,a\,a\,b$, while Fig. 7 and Fig. 8 respectively show the corresponding ISTs and LISTs.

Remark 2. Notice that for each node of a LIST, denoted as $[I\ t_{h,k}\ J]$, where I and h are the left p-state and index, while J and k are the right ones, the GSS includes an arc $\langle\, h,\, I \,\rangle \overset{t}{\longleftarrow} \langle\, k,\, J \,\rangle$. For instance, the node $[I_2\ A_{1,3}\ I_4]$, which is present in the LISTs for the syntax trees n. 2 and n. 3, corresponds to the GSS arc $\langle\, 1,\, I_2 \,\rangle \overset{A}{\longleftarrow} \langle\, 3,\, I_4 \,\rangle$; in the same LISTs, the node $[I_4\ B_{3,4}\ I_6]$ corresponds to the GSS arc $\langle\, 3,\, I_4 \,\rangle \overset{B}{\longleftarrow} \langle\, 4,\, I_6 \,\rangle$. □

3.4. Construction of the forest

To represent the syntax tree(s) of the input, along with the GSS the parser constructs another graph, called *Shared Packed Parse Forest* (SPPF). If two or more trees share a subtree, the latter is represented only once in the SPPF. Slightly different SPPFs are used in the literature (see the discussion in [15]), which differ in the compression (degree of subtree sharing) they support.

A SPPF is a *directed graph* that subsumes the syntax trees of a valid string. SPPF nodes are of two types, called *plain* and *packed*. A plain node encodes a tree node, and an SPPF edge encodes a parent-child relationship between tree nodes. A packed node is linked by edges to the alternative trees of the same (sub)string. To understand the SPPF representation of a valid string $x = a_0 \ldots a_{n-1}$ (with $n \geq 1$), it is necessary to keep in mind the GSS of x, from which the forest derives.

In more detail, each SPPF node carries as information a 3-tuple with two integer indexes that identify a (sub)string of x, and with a terminal or nonterminal symbol, as follows:

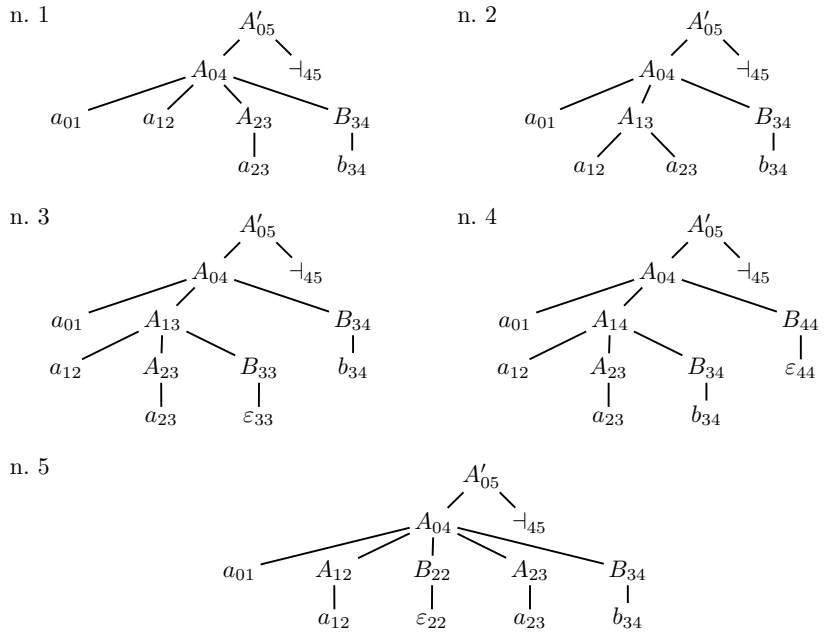


Figure 7: The five indexed syntax trees (ISTs) of string $a\,a\,a\,b$.

plain round node tagged as $(h.a.k)$ or $(h.A.k)$, the two cases being referred to as terminal and nonterminal subtype

packed rectangular node tagged as $[h.A.k]$, only of nonterminal subtype

where $a \in \Sigma$, $A \in V_N$ (but $A \neq S'$) and $0 \leq h \leq k \leq n-1$. We clarify the relationship between an SPPF and the trees of string x:

– a plain node encodes either a terminal a_h (i.e., a leaf node) of x, or a nonterminal A (i.e., an inner possibly nullable node) that generates a substring $a_h \ldots a_{k-1}$ of x (or ε if $h = k$)

– a packed node represents all the alternatives, for the same nonterminal A, that generate the same substring $a_h \ldots a_{k-1}$ of x (or ε if $h = k$)

Notice that, for a fixed 3-tuple $\langle\, h,\, a,\, h+1 \,\rangle$ of terminal subtype, the forest may contain only one plain node tagged $(h.a.h+1)$. Instead, for a fixed 3-tuple $\langle\, h,\, A,\, k \,\rangle$ of nonterminal subtype, the forest may contain: (*i*) one or more plain nodes, all of which are tagged $(h.A.k)$ and are distinguished by their sequence of child nodes; and (*ii*) at most one packed node tagged $[h.A.k]$.

Example 5 (shared packed parse forest – SPPF). For the string $a\,a\,a\,b$ of the running example (Fig. 4), the SPPF shown in Fig. 9 is obtained from the GSS of Fig. 6. The string has ambiguity degree 5 and its five labeled indexed syntax trees are depicted in Fig. 8.

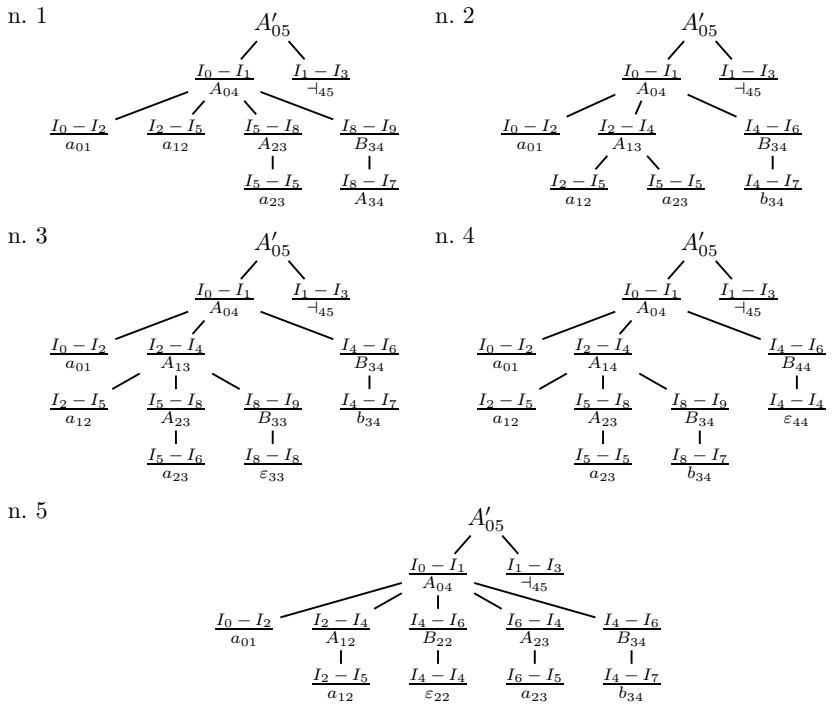


Figure 8: The five labeled indexed syntax trees (LISTs) of string $a\,a\,a\,b$.

The topmost SPPF packed (rectangular) node, of type $[0.A.4]$, has four plain nodes appended, all of them of type $(0.A.4)$, which cover, from left to right, the five trees below:

– n. 1

– n. 2 and n. 3, combined by means of the packed node $[1.A.3]$, which represents a logical *or*

– n. 5 and n. 4

In the forest graph shown in Fig. 10, we focus on the trees n. 2 and n. 3, which are highlighted by drawing their edges with a dashed (red) line and a dotted (blue) line, respectively. These two trees have identical first and last subtrees of the root A_{04} (in the LISTs we ignore the special node A'_{05}), therefore the SPPF represents such subtrees only once, as shown by the two-colored edges $(0.A.4) \rightarrow (0.a.1)$ and $(0.A.4) \rightarrow (3.B.4)$. The packed node $[1.A.3]$ has two outgoing edges corresponding to the following alternatives for the subtree rooted at A_{13}:

– the *dashed* edge $[1.A.3] \rightarrow (1.A.3)$ points to the SPPF representation of that subtree for LIST n. 2

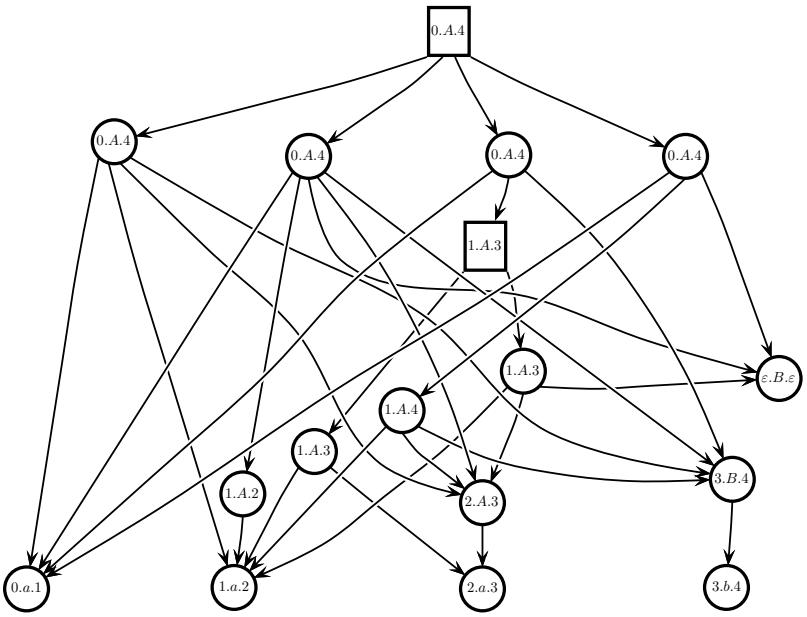


Figure 9: The SPPF of the input string $a\,a\,a\,b$, which has the GSS shown in Fig. 6.

– the *dotted* edge $[1.A.3] \rightarrow (1.A.3)$ points to the SPPF representation for LIST n. 3

The root node A_{13} of the latter subtree has, from the left, three child nodes respectively labeled a_{12}, A_{23} and B_{33}, which correspond to the GSS path $\langle\, 1,\, I_2 \,\rangle \overset{a}{\leftarrow} \langle\, 2,\, I_5 \,\rangle \overset{A}{\leftarrow} \langle\, 3,\, I_8 \,\rangle \overset{B}{\leftarrow} \langle\, 3,\, I_9 \,\rangle$.

The third child node, i.e., B_{33}, represents a nulled B nonterminal, i.e., the reduction $\varepsilon \rightsquigarrow B$; it is encoded in the SPPF as $(\varepsilon.B.\varepsilon)$. This node originates from the statements 2 and 4 of Alg. 2, with $\langle\, h,\, I \rangle = \langle\, k,\, H \,\rangle = \langle\, 3,\, I_8 \,\rangle$, where p-state I_8 is a reduction one for B and leads to a null reduction.

The SPPF packed (rectangular) node $[1.A.3]$, with two nodes appended, both of type $(1.A.3)$, matches the presence of two paths in the GSS, namely, $\langle\, 1,\, I_2 \,\rangle \overset{a}{\leftarrow} \langle\, 2,\, I_5 \,\rangle \overset{a}{\leftarrow} \langle\, 3,\, I_5 \,\rangle$ and $\langle\, 1,\, I_2 \,\rangle \overset{a}{\leftarrow} \langle\, 2,\, I_5 \,\rangle \overset{A}{\leftarrow} \langle\, 3,\, I_8 \,\rangle \overset{B}{\leftarrow} \langle\, 3,\, I_9 \,\rangle$, respectively corresponding to the reductions $a\,a \rightsquigarrow A$ and $a\,A\,B \rightsquigarrow A$. □

For completeness, a procedure for constructing the SPPF is sketched in Alg. 2. The procedure needs a mapping from GSS arcs (terminal and nonterminal) to SPPF nodes, called *associate*. The associate mapping provides a way for ordering from left to right the child nodes of a plain nonterminal node. Instead, the child nodes of a packed node are unordered, as they just encode alternative derivations from the same nonterminal.

Alg. 2 creates the plain nodes of the SPPF, associates one or more GSS arcs

Algorithm 2: Construction of the Shared Packed Parse Forest (SPPF).

Input: the GSS $\langle W,\ E \rangle$ of a string x

Output: the SPPF of the string x

`// initialize the SPPF with all terminal leaf nodes`

1 **foreach** (set ν of all the terminal arcs $\big(\ \langle h, I\rangle \xleftarrow{a} \langle h+1, J\rangle\ \big) \in E$ that have equal parameters a and h) **do**

  make a node tagged $(h.a.h+1)$ and *associate* to it each arc in set ν

`// add to the SPPF all null nonterminal leaf nodes`

2 **foreach** (set ν of all the nonterminal arcs $\big(\ \langle h, I\rangle \xleftarrow{A} \langle h, J\rangle\ \big) \in E$ that have equal parameters A and h, and their p-state I is a reduction one for A) **do**

  make a node tagged $(h.A.h)$ and *associate* to it each arc in set ν

`// add to the SPPF all nonterminal inner nodes`

3 **foreach** (nonterminal A and set π of all the paths $\big(\ \langle h, I\rangle \longleftarrow \langle k, H\rangle\ \big) \subseteq E$ that have equal parameters h and k, correspond to a (reversed) accepting path of machine M_A, and each of their arcs is (already) associated to a node) **do**

  let ν be the set of all the nonterminal arcs $\big(\ \langle h, I\rangle \xleftarrow{A} \langle k, J\rangle\ \big) \in E$

  `// by Remark 2 set` E `contains one such arc for each` I

  foreach (sequence χ of the nodes that are associated to the arcs of a path in π, ordered right-to-left as the arcs) **do**

    make a node tagged $(h.A.k)$; *append* to it, left-to-right, the (reversed) sequence χ^R of child nodes (make a new edge for each appended child); and *associate* to it each arc in set ν

  `// mapping associate may have become multi-valued`

  if (there are nodes equally tagged $(h.A.k)$, not all of which are grouped yet under (as children of) one packed node) **then**

    group those nodes under one packed node $[h.A.k]$; *switch* to this node all the edges that enter those; and *reassociate* to it the arcs of those (the grouped nodes remain disassociated)

  `// mapping associate has become again one-valued`

`// simplify the nullable nodes of the forest`

4 *change* the parameter h of all the nodes $(h.A.h)$ and $[h.A.h]$ into ε

`// unify the null leaf nodes of the forest`

5 *merge* together all the leaf nodes $(\varepsilon.A.\varepsilon)$ with equal nonterminal A

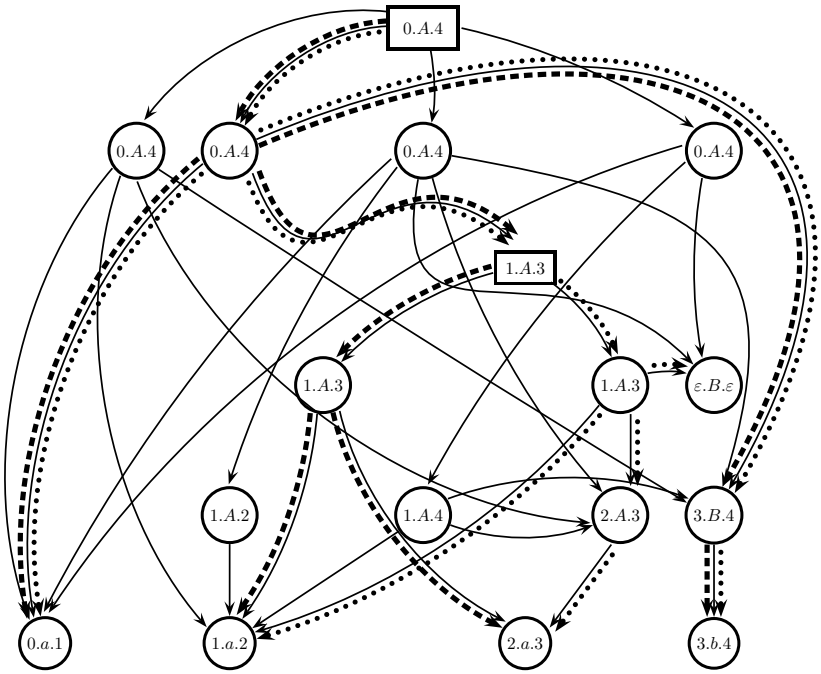


Figure 10: SPPF with trees 2 and 3, resp. drawn with dashed and dotted lines.

to a node, and connects the nodes with SPPF edges (to avoid confusion, the terms "edge" and "arc" refer to the SPPF and GSS, respectively). Furthermore, it may restructure the SPPF when a new packed node is introduced or a new child node is appended to an existing packed node. To ensure that a plain node is created only after all its child nodes are already present in the SPPF, i.e., to proceed in a bottom-up way, the procedure uses the associate mapping; the mapping is constructed together with the SPPF. The three main *foreach* loops 1, 2 and 3 of Alg. 2 match the items 1, 2 and 3 of Def. 8.

Notice that for simplicity the features of the tool that are needed to represent infinitely ambiguous strings are not present in Alg. 2 and are briefly described in Sect. 3.5.

Grammar G, TN *and* GELR(1) *pilot:* $A \to a^+ \; (\; A \; B \;)^* \;\mid\; \varepsilon \qquad B \to b^*$

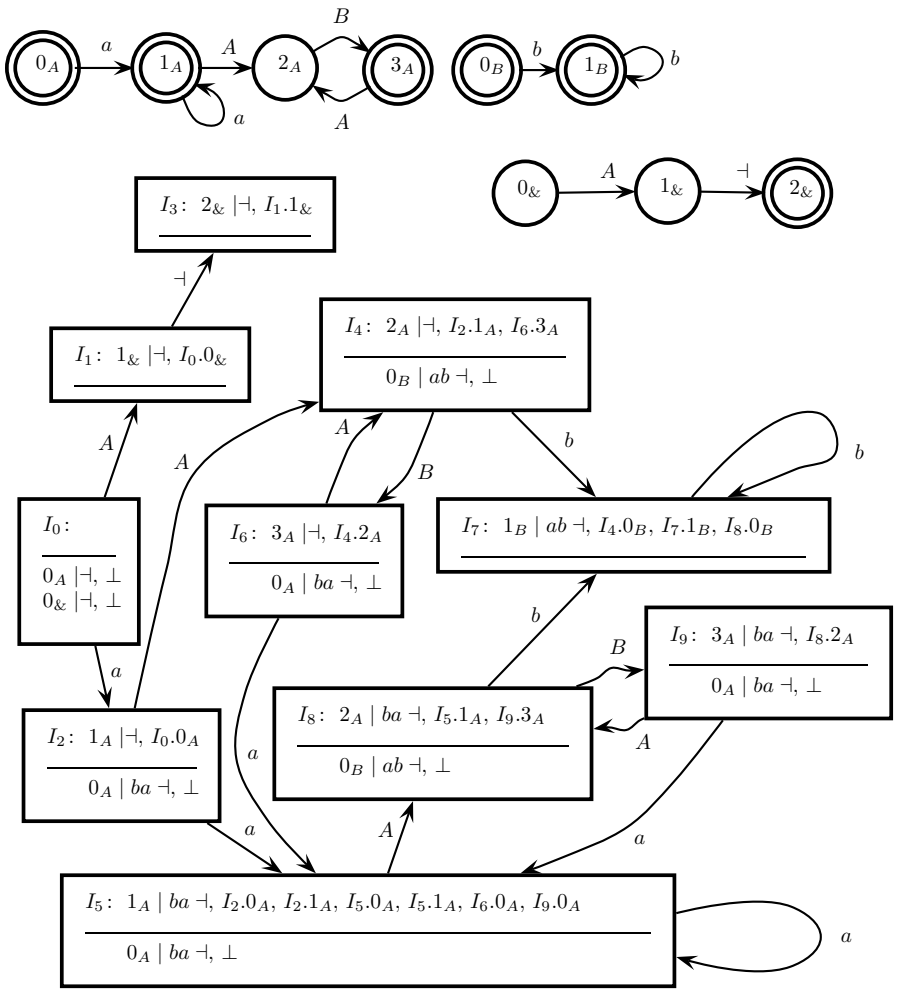


Figure 11: An ∞-ambiguous grammar of type 1 with its transition net and GELR(1) pilot.

3.5. Infinite ambiguity

We say that a string that admits unboundedly many distinct syntax trees is infinitely ambiguous (∞-ambiguous). We first recall how such an ambiguity may arise in an EBNF grammar and transition net, then we show how to represent ∞-ambiguity in a finite SPPF forest, either explicitly or by a finite sample of ambiguous trees. Two types of ∞-ambiguity are possible:

type 1 (*iteration of a nullable language*) If $\varepsilon \in X \subseteq \Sigma^*$, then language X^+ (hence also X^*) is ∞-ambiguous. An example is the language of nonterminal A in Fig. 11.

type 2 (*cyclic (recursive) derivation*) A cyclic derivation has the form $Y \overset{k}{\Rightarrow} Y$, with $k \geq 1$, and any terminal string y such that $Y \overset{+}{\Rightarrow} y$ is ∞-ambiguous. An example is in Fig. 13, with $A \overset{2}{\Rightarrow} A$ and $A \Rightarrow a$.

Notice that the same string may involve ambiguities of both types. Since type 2 already occurs in the CF grammars and is well described in the literature, we mostly focus on type 1, which may only occur in the EBNF grammars that use iterators in their rules.

We approximately represent the ∞-ambiguous strings of type 1 by including in the SPPF a finite number of trees. We draw with dashed lines the edges from a parent node to the child nodes that compose a sequence of nulled nonterminals, which sequence can be iterated for arbitrarily many times. Such sequences typically label cycles in the GSS, the nodes of which are laid in a column at the same string position. Such cycles can be iterated for an unbounded number of times to generate syntax trees for a unique ∞-ambiguous string. Our algorithm and the derived tool deal with ∞-ambiguity: the output of the parser represents as an SPPF the structure of an ∞-ambiguous string. Next, we describe such a representation by means of examples.

Example 6 (∞-ambiguity of type 1). The grammar G with these rules $A \to a^+ \, (\, A \, B \,)^* \mid \varepsilon$ and $B \to b^*$, where both nonterminals A and B are nullable, is a variant of the running example (Fig. 4) and is ∞-ambiguous of type 1: since $\varepsilon \in L(A \; B)$, then $L\big((A \; B)^*\big)$ is ∞-ambiguous. Fig. 11 shows the transition net and pilot for G, and Fig. 12 shows the GSS and SPPF for string $a\,b$.

The SPPF in Fig. 12 includes the packed node $[0.A.2]$, which has four appended (plain) nodes $(0.A.2)$ that encode as many possible syntax trees. All such nodes, but the third one from the left, exhibit one or more (ordered) pairs of dashed edges that point to their immediate descendants; each pair is evidenced by a dotted (red) bidirectional arrow. In the GSS, such edge pairs correspond to the cycles labeled by the nullable nonterminals A and B. Specifically, in the second subtree from the left, the two pairs correspond to the two GSS loops: (*i*) $\langle 1, I_4 \rangle \overset{B}{\leftarrow} \langle 1, I_6 \rangle \overset{A}{\leftarrow} \langle 1, I_4 \rangle$, leftmost pair, position 1 after reading symbol a; and (*ii*) $\langle 2, I_6 \rangle \overset{A}{\leftarrow} \langle 2, I_4 \rangle \overset{B}{\leftarrow} \langle 2, I_6 \rangle$, rightmost pair, position 2 after reading symbol b. □

Our tool deals with ∞-ambiguity of type 1 by refining the condition of the *foreach* loop 3 of Alg. 2, in order to ensure that each cycle in a path of the GSS is skipped or traversed exactly once.

An ∞-ambiguous string of type 2 is represented, in the same way as in [15], by means of an SPPF graph with cycles that include both packed nodes and plain nodes, as exemplified next.

Example 7 (∞-ambiguity of type 2). The grammar G with rules $A \to B \mid a$ and $B \to A \mid b$ generates the strings a and b that exhibit ∞-ambiguity of type 2, because of the circular derivations including $\cdots A \Rightarrow B \Rightarrow A \cdots$. Fig. 13

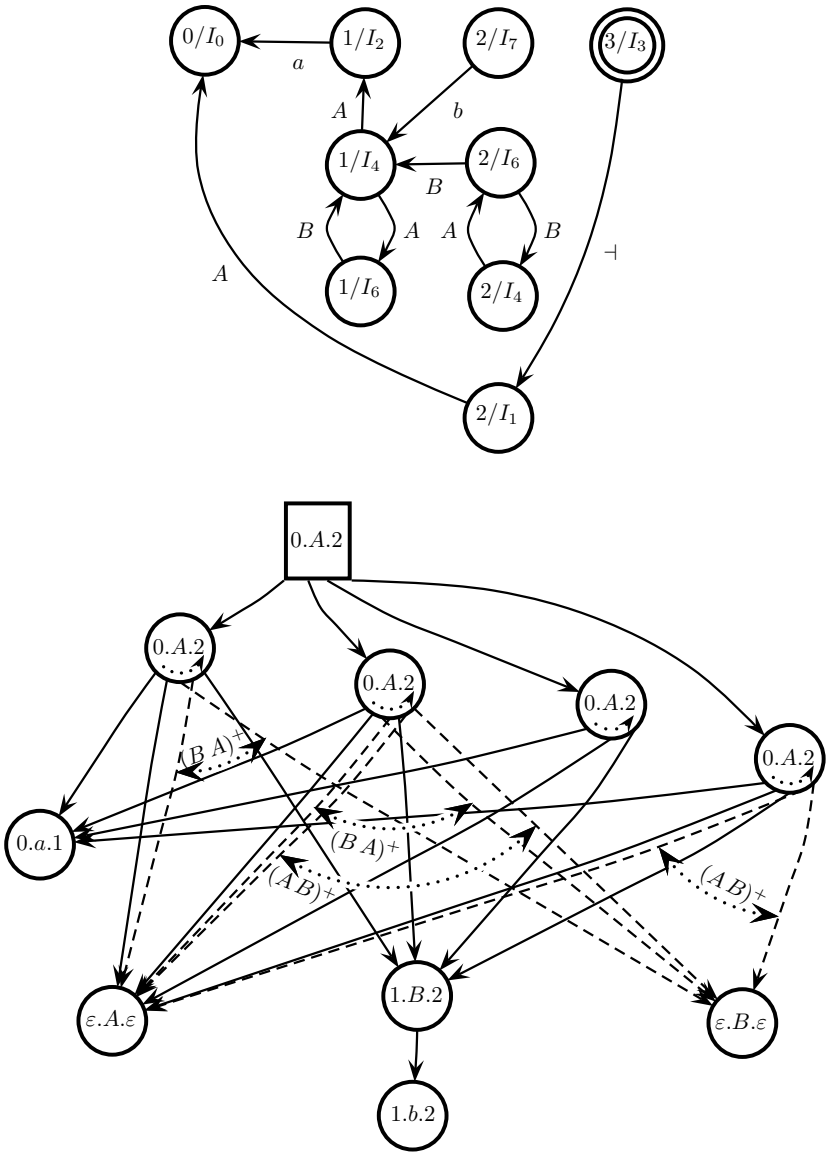
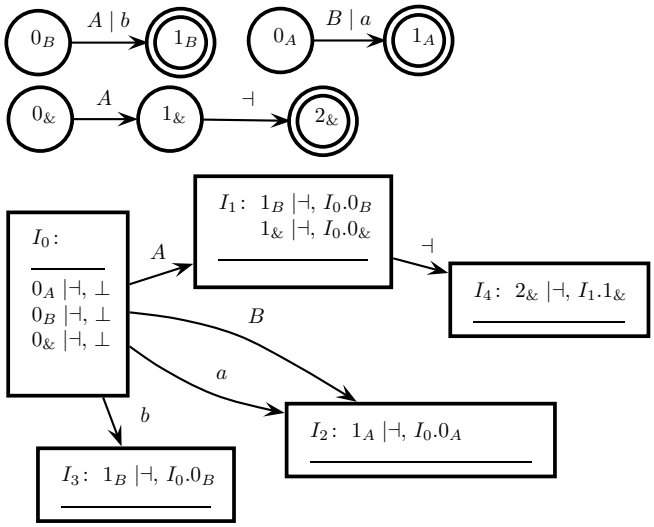


Figure 12: The GSS (top) and SPPF (bottom) of the ∞-ambiguous string $a\,b$ (type 1). Their TN and pilot are shown in Fig. 11. The orientation inside each of the four plain nodes ($0.A.2$) indicates the ordering of their child nodes. Each bidirectional dotted arrow evidences a pair of nullable nonterminals, and carries as label the RE of a nullable substring that can be repeated arbitrarily many times.

(a) *Grammar* G, TN *and* GLR (1) *pilot:* $A \to B \mid a \quad B \to A \mid b$



(b) GSS *and* SPPF *of string* a

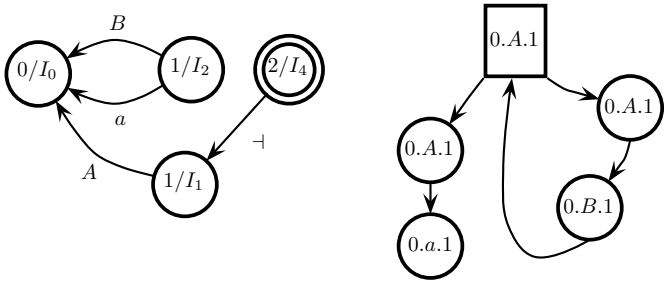


Figure 13: Part (a) – an ∞-ambiguous grammar of type 2 with its transition net and GELR pilot. Part (b) – GSS and SPPF of string a.

shows the transition net and pilot of G, plus the GSS and SPPF of string a. Notice the SPPF cycle $[0.A.1] \to (0.A.1) \to (0.B.1) \to [0.A.1]$, which allows a compact representation of an infinite number of syntax trees for string a. □

Remark 3. We have described the SPPF of the ambiguous strings, yet it would be out of scope to discuss how one or more syntax trees are selected by the subsequent phases of a compiler. Selecting a tree out of many ones for the same string is referred to as *disambiguation*. In this paper we do not discuss disambiguation criteria.

4. Parser correctness

We need a projection, denoted by $|_V$, that deletes the two indices of an IST node label, e.g., the projection $A_{1,3}|_V$ is A. Projection $|_V$ is naturally extended to the *sequences* of IST node labels: if $\sigma = a_{0,1}\ A_{1,3}\ B_{3,4}$, then $\sigma|_V = a\,A\,B$. Next we focus on a subgraph of the GSS graph, which has the property of representing a derivation or – which is essentially the same – an indexed tree.

Definition 9 (derivation-encoding GSS subgraph – DE-GSS-S). Given $x \in L(\mathcal{M})$, with $|\,x\,| \geq 1$ and x ending with $\dashv$, and given the GSS of x, consider an IST T of x. For each internal node $A_{h,k}$ of T, for brevity let σ be the left-to-right sequence of child nodes. The DE-GSS-S of T includes, for each internal node $A_{h,k}$ of T:

1. path $\langle\, h,\, I \,\rangle \xleftarrow{\sigma|_V} \langle\, k,\, H \,\rangle$ for some p-states I and H; consequently the pilot has a path $I \xrightarrow{\sigma|_V} H$
2. arc $\langle\, h,\, I \,\rangle \xleftarrow{A} \langle\, k,\, J \,\rangle$, where $J = \vartheta\,(I,\, A)$ except for the root node $S'_{0,|\,x\,|}$ □

Lemma 2 (DE-GSS-S and path). *A* GSS *contains a derivation-encoding* GSS *subgraph if, and only if, it includes a path:*

$$\langle\, 0,\, I_0 \,\rangle \xleftarrow{S} \langle\, n-1,\, \vartheta\,(I_0,\, S) \,\rangle \xleftarrow{\dashv} \left\langle\, n,\, \vartheta\left(\vartheta\,(I_0,\, S),\, \dashv\right) \right\rangle$$

where p-state I_0 is the initial one. □

Proof of Lemma (2). It follows from Remark 2 and Def. 9, and from the fact that the pilot has exactly one path labeled by $S\,\dashv$. □

Example 8 (DE-GSS-S subgraph). For the (ambiguous) string $a\,a\,a\,b\,\dashv$ of the running example (Fig. 4), with the GSS of Fig. 6, we show in Fig. 14 the GSS subgraph of the derivation corresponding to the IST and LIST n. 3 in Fig. 7 and Fig. 8 (the syntax tree is drawn with dotted (blue) lines in Fig. 10). □

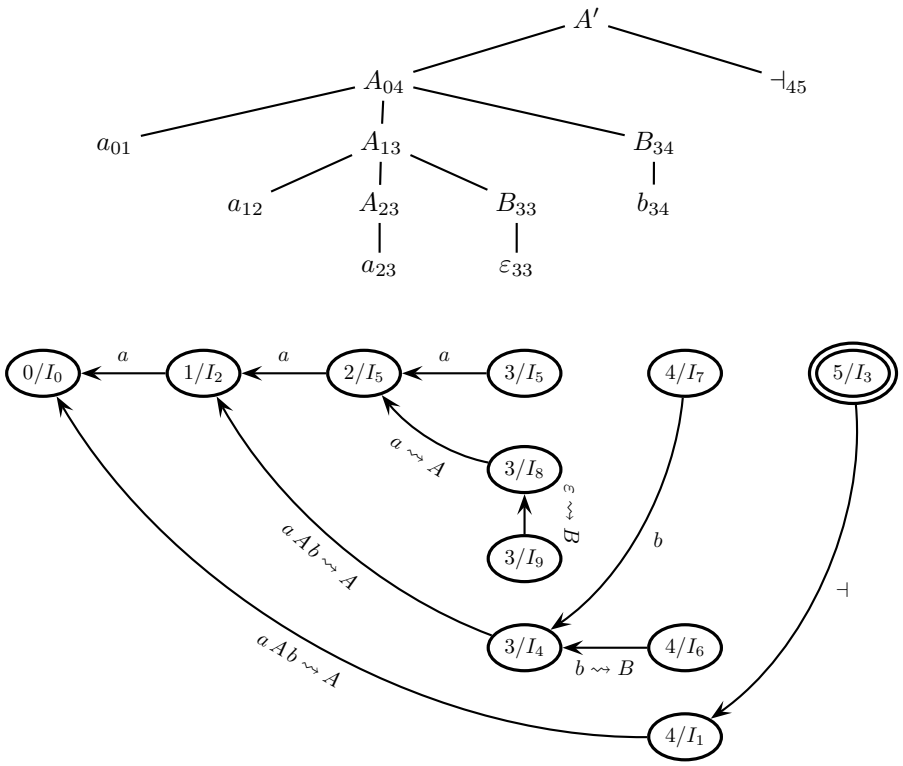
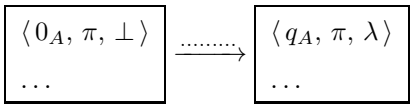


Figure 14: Top – indexed syntax tree for string $a\,a\,a\,b$ (same as tree n. 3 of Fig. 7). Bottom – a derivation-encoding GSS subgraph for the same string.

From Lemma 1 we recall (*i*) that every accepting path in a machine M_A of the TN is included in the pilot graph, and (*ii*) that every pilot path of the form:



from a p-state including an item with state 0_A to a reduction p-state for A (state q_A is final), includes an *item path*, i.e., a pointer chain as of Def. 4 and Eq. (1). This path traverses items with states belonging to M_A, starting from 0_A and ending in q_A, and touches the states of an accepting path of M_A.

Lemma 3 (DE-GSS-S and derivation – I). *For every derivation-encoding* GSS *subgraph with associated input string x, there is a derivation $S \overset{*}{\underset{G}{\Longrightarrow}} x$.* □

Proof of Lemma (3). From Def. 9 and clause (3) of Def. 8 (or from clause (2) for the nullable nonterminals), any path in the derivation encoding GSS subgraph (henceforth *subgraph* for brevity) of type $\langle\, h,\, I \,\rangle \longleftarrow \langle\, k,\, H \,\rangle$, where p-state I includes an item with machine state 0_A, with $A \in V_N$, and p-state H is

a reduction one for the same nonterminal A, is labeled by a sequence of grammar symbols that label a (reversed) accepting path of machine M_A. Thus such a sequence corresponds to the left-to-right sequence of child nodes of a node of type A. Consequently, the nonterminal symbol A of the arc $\langle h, I \rangle \xleftarrow{A} \langle k, J \rangle$, where $J = \vartheta(I, A)$, is also included in the subgraph due to clause (3) of Def. 8 (or to clause 2), and corresponds to the parent node of type A. Furthermore, the arcs of the subgraph that are of type $\langle h, I \rangle \xleftarrow{a_h} \langle h+1, J \rangle$, as of clause (1) of Def. 8, correspond to the (non-null) leaf nodes of the syntax tree, which nodes in turn match the terminal elements of the string. Altogether, the above nodes and arcs of the subgraph identify a derivation of the input string x, according to Def. 2. □

Lemma 4 (DE-GSS-S and derivation – II). *For every derivation $S \overset{*}{\Rightarrow} x$, there exists a derivation-encoding* GSS *subgraph.* □

Proof of Lemma (4). To the labeled indexed syntax tree that matches the derivation $S \overset{*}{\Rightarrow} x$, apply the next clause to generate a set of nodes and a set of edges: *for every node t of the p-state labeled IST, denoted by $[I\ t_{h,k}\ J]$, where I and h are the left p-state and index, while J and k are the right ones, the set of nodes includes $\langle h, I \rangle$ and $\langle k, J \rangle$, and the set of arcs includes $\langle h, I \rangle \xleftarrow{t} \langle k, J \rangle$.* Then, from Def.s 7, 8 and 9, the resulting sets of nodes and arcs constitute a GSS subgraph that encodes the derivation. □

Definition 10 (GSS-based string acceptance). The string x is *accepted by the* GSS if, and only if, the GSS includes a final node. □

Theorem 1 (parsing correctness). *A string $x \in \Sigma^*$ is accepted by the* GSS *if, and only if, it holds $x \in L(\mathrm{TN})$.*

Proof of Theorem (1). String $x \in L(\mathcal{M}) \Leftrightarrow \exists$ derivation $S \overset{*}{\Rightarrow} x \Leftrightarrow \exists$ DE-GSS-S for string x (by Lemmas 3, 4) $\Leftrightarrow$ final node $\langle n, \vartheta(\vartheta(I_0, S), \dashv) \rangle \in$ GSS (by Lemma 2) $\Leftrightarrow$ string x is accepted (by Def. 10). □

We finish the section with a short remark on the parsing complexity. Our parser constructs the GSS and SPPF data-structures, which are the dominant parsing phases, by essentially the same algorithms as the Tomita-like parsers. Thus its asymptotic complexity can differ only by a constant factor from that of other Tomita-like parsers. This is confirmed by the experimental measures in Sect. 5.

5. Implementation and experimental results

We have developed two implementations of the algorithm: the former is a web tool (written in HTML and Javascript) that provides a visual representation of the GSS and of the SPPF, while the latter (written in Java) is an optimized version of the former. Both tools are available on the GitHub repository at

`https://github.com/FLC-project/GELR`. The former tool is meant for small grammars and for experimenting the algorithm. The latter tool is targeted at large grammars and texts, and is competitive for performance measurements and real applications. The former is based on the RNGLR algorithm [15], modified to process EBNF grammars and the transition nets that represent the grammar rules. The latter is also based on the RNGLR algorithm, but it uses the scheme described below (see Sect. 5.1), so as to optimize parse time and memory footprint. We refer to the latter as the GELR (*generator*), and all the measurements we report are for GELR, or for versions thereof that differ in the optimizations applied.

For comparison with GELR, we have implemented generators for LR and RNGLR parsers, both for BNF grammars, together with an EBNF-to-BNF converter that automatically transforms into BNF the grammar rules containing extended constructs. All the parsers used in comparative evaluations are implemented in the same programming language, on the same computing platform and operating system. Furthermore, the parsers have the same optimization level, so as to provide time and memory measurements unaffected by external factors, thus they exclusively compare the net algorithms as such.

Since the purpose of experimenting is to collect practically relevant measurements, we have chosen a set of modern languages in common use and, for each of them, a large corpus of texts. Tab. 1 lists the grammars and corpora we used. The languages endowed with official EBNF grammars were left unchanged (possibly with a format adapted to comply with ISO-EBNF). The cases where changes were needed are the following: Java-lr is a Java grammar modified by us so as to have the LALR property, thus allowing deterministic parsing (see below); and XML is an example scheme of XML, manually transformed into an EBNF grammar. The official definition of HTML is in the SGML style,[5] and is intended not for parsing but for validating an implementation; therefore we had to translate it into EBNF. Notice that a different translation, i.e., into BNF, was used to measure the performance of RNGLR, to be compared with our GELR parser. For the grammars, it would be inappropriate to express the (production) rule size in bytes, since the size depends on the length of the nonterminal names and of the keywords, which in no way affects the parsing time. Instead, we have chosen to count the number of occurrences of nonterminal symbols and tokens, i.e., lexemes, plus the number of occurrences of the RE meta-symbols, i.e., Kleene star and cross, and also of the parentheses.

All the text corpora we use are taken from public sources. Notice that the text size is expressed as a number of tokens instead of characters, as in this work we are not concerned with lexical analysis. More precisely, all the measurements presented include only the parsing time and are cleared from the lexing time.

Unfortunately, there is no guarantee that corpora for different languages exercise the language constructs to a similar coverage degree. This has the annoying consequence that the performances measured for two different languages

[5] It is known as a *Transitional DTD*: `https://www.w3.org/TR/html4/sgml/loosedtd.html`.

Table 1: Grammars and text corpora employed in our measurements. The grammar size is the number of occurrences of terminal and nonterminal symbols and of RE metasymbols. The text size is the number of tokens.

grammar	*grammar size* occurrences of symbols	*text name and source*	*text size* in tokens
Java-lr	1,095	Oracle Java 8 sources	1,290,147
Java	1,320	Oracle Java 8 sources	1,290,147
C	912	Apache HTTP Server	5,758,663
C++	1,990	Apache C++ Std. Lib. 4.2.1	2,648,384
Freepascal	2,005	Freepascal compiler	1,006,575
HTML	3,037	Apache HTTP doc.	351,784
JSON	86	Data.gov datasets	8,943,757
XML	53	XML data example	109,227
Go	1,112	Go source distribution	10,702,999

may diverge not only because of the inherent language differences, but also because of the difference in language coverage. The reader is thus warned not to hastily draw conclusions about the comparative performance of the same parser on different languages. All the measurements have been carried out on an AMD 64 X2 4200+ 2.2 GHz, with Windows 10.

5.1. GELR *program*

We briefly describe the relevant design choices, starting from the representation used for the GSS graph. Then we discuss the optimizations we have applied over the basic GLR algorithm. We refer to the layout of a GSS graph into vertical columns and horizontal layers. To illustrate, in Fig. 6 the input string $a\,a\,a\,b\,\dashv$ has length 5 and the number of columns is $5 + 1 = 6$. In particular, the second character a is associated to the column appended to node $2/I_5$, which contains four layers naturally represented by the ordered set $\langle\, 2/I_5,\, 2/I_4,\, 2/I_6,\, 2/I_3 \,\rangle$.

The GSS nodes are processed sequentially by GELR until none remains. Processing an existing GSS node may (*i*) add a new node to the set for the next column as a result of a terminal shift, and (*ii*) add new nodes and arcs to the current set as a result of the reductions and of their nonterminal shifts.

We use a *worklist* to keep the new arcs created. When the algorithm produces a new arc that goes into a node that has already been processed, an entry is added to a reduction worklist, which will be examined after analyzing all the nodes of the current set (thus possibly producing further nodes to be processed). When no more nodes remain to be processed, nor entries in the reduction worklist, the next input symbol is read, and a new column and a set are initiated. Termination occurs successfully upon reading the end-of-text symbol, or unsuccessfully when no column can be initiated. When a reduction is concluded,

the parser builds a fragment of the SPPF forest. The fragment contains the derivation corresponding to the reduction, and together with similar fragments it will compose the final forest.

5.1.1. Optimizations applied in GELR

Right-nullable reductions. We introduce the right-nullable reductions as described in the RNGLR parser of [15], applied to the states of the TN (see Def. 3). Any p-state containing an item with a TN machine state q that is connected to a final state by a path exclusively labelled by nullable nonterminals, is a reduction p-state and allows for a *right-nullable reduction*. When such a p-state I is entered, the parser performs a reduction, the derivation of which is completed with all the possible right-nullable tails, thus matching all the right-nullable paths from q to a final state.

The gain obtained from the right-nullable reductions is substantial: each GSS set is processed in one pass, thus avoiding the cost of reprocessing the nodes until a fixed point is reached. On the other hand, this optimization has the side effect of increasing the pilot size. We later report on the resulting increase of size and speedup.

Fixed-arity reductions. We recall that a reduction $X_1\, X_2\, \ldots\, X_r \rightsquigarrow X$ has arity r. In a BNF grammar the arity of all the reductions is bounded, whereas in an EBNF one the arity is unbounded when the reduction involves a rule that uses a Kleene star or cross operator. Consider now a reduction such that all the pointer paths from the reduction items in the reduction p-state to the initial item have the same length. We say that such a reduction has a *fixed-arity*. In GELR, to save some reduction overhead, fixed-arity reductions are identified ahead of parsing time. They are represented by items without pointers; this in turn reduces the number of pilot states and simplifies the visit of the reduction paths, since the path length is known.

Variable-arity reductions. A regular expression construct that uses a Kleene star or cross is called a *group*. A reduction for a group involves a variable-arity path, which traverses an unbounded number of TN states. In the GSS, such a path is performed along a path of variable length. The path end is reached by following a chain of items, beginning from the one that started the reduction and hopping to its predecessor by following a pointer, until an item is found that has no predecessor. In the GSS graph, a group with a nullable argument is represented by a cycle. To reduce the overhead when visiting the reduction paths, a check for the cycles in a GSS is performed only when the GSS is known to be cyclic, a property that can be detected at GSS construction time when forward arcs are added.

LALR (1) *look-ahead sets.* It is well known that the LALR (1) method overapproximates the LR (1) look-ahead sets, and that it was introduced to reduce the number of p-states in the years when the main memory was a scarce resource [6]. But even today, for some languages the size of an LR pilot may

cause an intolerable slowdown in the parser generation phase on stock hardware: in particular, this was the case for HTML. Therefore, we decided to try the LALR method for all the languages and parsers. Then, after ascertaining that the overhead caused by the precision loss in the LALR look-ahead sets is negligible, we replaced LR sets with LALR sets in all the parsers.

Deterministic isles. It is well known that context-free grammars may contain LR subgrammars; this occurs quite often in the grammars of the actual programming languages, where ambiguity may concern a limited number of constructs. We exploit this fact by applying an optimization similar to those described in [1] for the Earley parsers and in [13] for the GLR Elkhound parser, but with suitable modifications because our method uses machine states rather than marked grammar rules.

The existence of LR subgrammars is reflected in the pilot automaton, which includes maximal connected sub-graphs consisting of conflict-free p-states, i.e., states the items and transitions of which do not raise any conflict. Such pilot regions are called *deterministic isles*, and can be identified statically and employed to guide the GELR parser. When the GELR parser enters an isle, it can execute shift and reduce actions in a deterministic fashion – like the ELR parser of [3] – until a reduction is performed, followed by a nonterminal shift to a node outside the isle; during such deterministic phases the parsing takes a linear time. Exploiting the deterministic isles also allows the GELR parser to use a pushdown stack when executing a potentially large sequence of shift and reduce operations, therefore pruning significant branches of the GSS graph and reducing the overall memory need.

Coding optimizations. We mention also some programming optimizations. Each pointer from an item to a predecessor item is represented as a simple integer, to simplify the visit of paths when making reductions. By using integers, during a reduction the predecessor items are directly addressed. This is faster than using a pair $\langle$ *p-state*, *machine state* $\rangle$ as a pointer (see Def. 4), as it is done in the non-optimized algorithm. The nodes of the GSS are represented as entries in block arrays instead of Java objects, to reduce parsing time (by avoiding garbage collections) as well as memory footprint. Last, to search for GSS nodes and SPPF nodes, we use auto-aging hash tables.

5.1.2. Measurements and experimental results

Pilot size. Tab. 2 shows the size and the generation time for the pilots of the grammars listed in Tab. 1. The pilot size is reported as a number of p-states.

Table 2: Pilot size (number of p-states) and pilot generation time. Columns: 1: LALR pilot; 2: the pilot described in [3] with pointers only for variable-arity reduction operations; 3: ELR1+ with LALR (1) look-ahead instead of LR (1); 4: percentage (%) of ELALR1 states that belong to isles; 5: ELALR1 without look-ahead for fixed-arity reduction isles; 6: percentage (%) of GELR states that belong to isles; 7: pilot generation time for GELR, in seconds (s).

grammar	[1]LALR	[2]ELR1+	[3]ELALR1	[4]*isle*	[5]GELR	[6]*isle*	[7]*time*
Java-lr	624	3,113	1,882	99.84	1,027	99.71	7.0
Java	744	3,732	2,270	56.08	1,332	40.02	10.5
C	483	2,604	1,286	45.26	849	22.61	2.2
C++	1,072	24,740	3,688	46.50	2,696	38.32	17.5
Freepascal	1,116	16,044	2,381	58.71	1,272	33.41	11.1
HTML	1,732	105,816	2,054	55	1,890	52	9.4
Json	46	68	47	93.62	36	91.67	0.1
XML	30	23	24	87.50	24	87.50	0.1
Go	608	3,524	1,537	47.82	766	37.99	1.5

The reader may look back to the grammar size in Tab. 1. The *isle* column lists as a percentage the number of deterministic states that make up the deterministic isles. As shown, the number of states of the ELALR1 pilots is considerably smaller than that of the corresponding ELR1+ ones; in particular, the ELR1+ tables for HTML are too large for a practical use. The last column is the pilot generation time in seconds. The values refer to those of the pilot used by the parser, i.e., ELALR1 with fixed-arity reduction.

Table 3: Parsing speed and memory occupation of GELR – left with all optimizations and right without deterministic isles.

	GELR *wholly optimized*		GELR *without isles*	
	token / ms	byte / token	token / ms	byte / token
Java-lr	1,414.32	61.2	370.10	285.8
Java	412.45	74.4	279.17	343.6
C	90.71	199.8	81.89	746.4
C++	204.52	41.9	143.27	426.8
Freepascal	394.79	139.1	242.36	340.7
HTML	832.80	17.2	518.32	153.4
Json	1,417.72	1.2	391.96	95.6
XML	1,523.31	2.6	627.38	53.0
Go	143.48	99.9	102.87	327.0

Parsing speed and memory footprint. The results are listed in Tab. 3 and, as said, they are net from the time for lexical analysis and I/O operations on texts. The parse speed is the average over the entire corpus (specified in Tab. 1), i.e., the total number of tokens per total parsing time; similarly for the memory footprint. It is fair to say that the parse speed is computed primarily to permit a comparison for different computer languages as well as between algorithms optimized or not; clearly the absolute datum depends on the computer used and is less interesting.

We discuss some relevant figures. As expected, the deterministic grammars (Java-lr, Json, XML) are faster and less wasteful of memory. We have warned that differing performances for different languages may depend more on the differences between corpora than on the intrinsic differences between languages, and we discuss one such case in Tab. 3. Two grammars of similar sizes (Java $1,320$ symbols versus Go $1,112$ symbols) for languages intended for similar applications may significantly differ in parsing performance: the parsing speed of Java is almost three times more than that of Go, and its footprint is less than 3/4 of Go. The lower speed for the Go language is mostly due to some unusual files in the corpus.[6] Those files contain very long tuples ended by a comma, which define Unicode data. There is nothing equivalent in the Java corpus. If such files are removed from the corpus, the speeds of Java and Go are quite comparable.

Table 4: Percentage of parser actions performed in GLR mode – the remaining actions are in LR mode, i.e., in isle mode.

grammar	*actions* (GLR mode)
Java-lr	0.1 %
Java	62.2 %
C	94.0 %
C++	80.5 %
Freepascal	18.6 %
HTML	96.9 %
JSON	0.0 %
XML	0.0 %
Go	70.2 %

In Tab. 3 (right) we report also the performance degradation when the very profitable deterministic isle optimization is disabled. As the table shows, the gain from the exploitation of deterministic isles ranges from nothing to a speed

[6] Such files are: tables9.0.0.go, tables10.0.0.go, tables11.0.0.go and tables12.0.0.go.

increase of 260 % and to a memory reduction of 400 %. In fact, it is not surprising that the closer a grammar is to being deterministic, the larger is the performance loss caused by disabling such an optimization.

We add a quantitative justification for the differences in the parse speed figures (Tab. 3), by comparing in Tab. 4 the number of parser actions (meaning terminal and nonterminal shifts) that a parser makes when operating in different modes, namely the general GLR mode versus the LR mode, i.e., the isle mode. The above figures confirm the intuition that the percentage of GLR actions is higher in the test cases where the parse speed is lower, see C with 94.0 % versus C++ with 80.5 % as an example.

Linear-time parse. In order to verify that GELR preserves the linear-time complexity of LR and ELR parsers, we have chosen an LR grammar and we have measured the parse speed (in *tokens* per *millisecond*) for texts of increasing length. In Fig. 15 we report the results for language JSON. The parse speed is fairly constant, if we discard the penalty for initializing the parser, which disadvantages the short texts and vanishes on the larger ones. This experiment

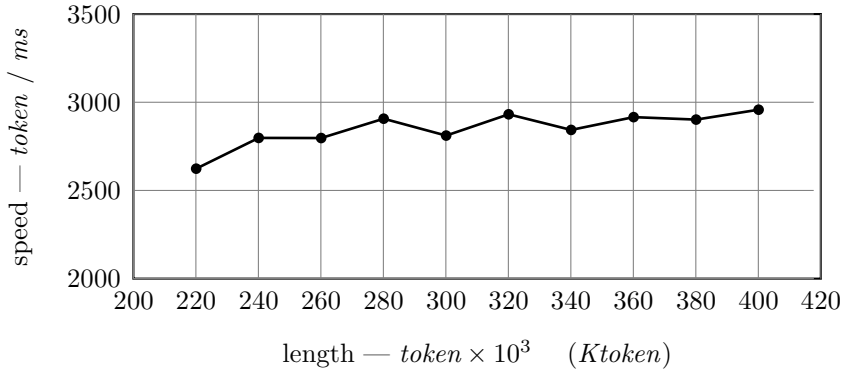


Figure 15: Parsing speed of GELR for language JSON.

witnesses that GELR has the linear-time behavior of an LR parser when the grammar is LR (or better the grammar part activated by the input).

Comparison with other parsers. As said, we have implemented an LR parser and an RNGLR parser to measure their comparative performance against GELR. Clearly, the comparison with LR is only possible for LR grammars, thus we have chosen Java-lr and JSON as representative cases. All these parsers use an LALR pilot instead of an LR one. The results are listed in Tab. 5. Needless to say, the figures for all parsers exclude the time spent in the lexer and in performing I/O. Comparing the speed of the LR and GELR parsers, for Java-lr the slowdown of GELR versus LR is only 4.8 % for a grammar that is (almost) LR. For JSON, the superiority of GELR over LALR is likely to be due to the higher processing efficiency of star reductions over recursive reductions. The import of such a result is that GELR can be used with a negligible penalty, at worst, whenever

Table 5: Parsing speed of GELR and of two classical algorithms.

	GELR token / ms	RNGLR token / ms	LALR token / ms
Java-lr	1,414.32	1,098.69	1,510.09
Java	412.45	340.86	—
C	90.71	40.38	—
C++	204.52	129.54	—
Freepascal	394.70	689.04	—
HTML	832.80	595.22	—
JSON	1,417.72	1,237.12	1,252.45
XML	1,523.31	1,415.74	—
Go	143.48	338.38	—

texts generated by an LR grammar must be analyzed. All the more, using GELR eliminates the need of changing a grammar to make it amenable to LR parsing: such an operation is not simple and is quite error prone, and it may cause annoying differences with respect to the official grammar.

Moving to the comparison with the RNGLR parser, there are both cases when the GELR parser is faster, and conversely. A similar situation occurs for the memory footprint. We only discuss the parsing speed.

- The GELR parser is faster in all the cases when a grammar rule contains a Kleene star construct K that is followed by a token that does not belong to K, e.g., $X \to K\, c$ and $K \to a\,(b\,a)^*$. At each step, GELR knows if the argument of the group is ended, and then performs shift actions until it finds its end, thus producing a forest with a node that has all the elements matching the group as child nodes. On the other hand, RNGLR relies on BNF recursive rules instead of the Kleene construct, thus it has to build a forest that contains a node for each iteration of the group.

- Instead, GELR is slower if the Kleene construct is followed by a token that can belong to it, e.g., like in $X \to K\,(b \mid \varepsilon)\,c$ and $K \to a\,(b\,a)^*$. Then it is impossible for GELR to know at each step if the star group has ended, and a new fragment of forest has to be constructed for it (just in the case it has really ended). Such a fragment is then trashed each time a new iteration is matched. On the other hand, RNGLR can build the fragment by relying on the previous one, i.e., by creating a node that contains the current iteration and a link to the previous one as a child node.

A situation similar to the latter occurs in the grammars that contain nested Kleene constructs, thus allowing for very ambiguous derivations. This happens in particular for the language Go, due to the presence of those large files

mentioned in a previous footnote. Other than in such cases, the speed difference between the two parsers of GLR type is not significant. To sum up, each representation of the syntax tree, i.e., a shallow derivation for EBNF iterative rules versus a deeper recursive derivation for BNF rules, has its strengths and weaknesses.

Lately, a recent publication [9] unexpectedly offered us an opportunity for checking the sensitivity of our measurements with respect to the choice of the benchmark. In that work, a large web search engine coded in Java[7] is used as a benchmark. We found out that the parsing speed of GELR for the new benchmark is 444 *token / ms*, which is consistent with, and even slightly faster than, the value 412.15 *token / ms* reported in Tab. 5.

6. Conclusion

This research proves that a Generalized LR (or Tomita style) parser / parser generator can operate directly and efficiently on context-free grammars extended with regular expressions, i.e., on the EBNF grammars used in most language reference manuals to define computer languages. The novel approach of our GELR parser consists of a deterministic shift-reduce method for transition nets (TN). Such a deterministic parser is then generalized to deal with LR conflicts, in accordance with Tomita's approach, thus obtaining a general parser for any EBNF grammar.

The measurements carried out over large benchmarks that involve modern computer languages, give evidence that there is no performance penalty from operating on EBNF grammars. Thus one can reap the benefits of directly operating on the official language grammars without the inconvenience of a conversion into BNF.

We have applied both known and (partly) new optimizations, and we have measured the incremental speed-up brought by individual optimizations, for each computer language and benchmark. In this way, the measures can be viewed as a sort of score of how well each language syntax is suitable for parser optimization.

Future researches are possible in two directions: algorithmic developments and new applications, and we mention a few ideas. More optimizations are also worth experimenting, e.g., the chain-rule (renaming-rule) reduction.

Since GELR only exposes the ambiguity due to the BNF syntactic rules, while the structure and ambiguity of the REs is cleared by the shallow derivations, the GELR algorithm could be revised so as to take into account also the syntactic structure and the ambiguity of the regular expressions that occur in the EBNF rules. In other words, such a development would involve a combination of Tomita's approach with an RE parsing algorithm such as [4].

An ambitious development we are also considering consists of extending the parsing algorithm into a substring parser, with the aim of analyzing incomplete

[7] See `https://github.com/elastic/elasticsearch`.

or erroneous texts. Many important applications rely on such capabilities, such as rendering an HTML page by a browser.

Last, the startling performance variations between the GELR parsers for grammars of different computer languages, has prompted us to wonder whether a GLR parser might also be used as a tool for objectively evaluating the quality of published syntactic definitions, with respect to the presence of ambiguity and to the percentage of determinism (deterministic isles).

References

[1] J. Aycock and A. Borsotti. Early action in an Earley parser. *Acta Inf.*, 46(8):549–559, Oct. 2009.

[2] J. Aycock and R. Horspool. Practical Earley parsing. *Comp. Jou.*, 45(6):620–630, 2002.

[3] A. Borsotti, L. Breveglieri, S. Crespi Reghizzi, and A. Morzenti. Fast deterministic parsers for transition networks. *Acta Inf.*, 55(7):547–574, 2018.

[4] A. Borsotti, L. Breveglieri, S. Crespi Reghizzi, and A. Morzenti. A deterministic parsing algorithm for ambiguous regular expressions. *Acta Inf.*, 2020.

[5] S. Crespi Reghizzi, L. Breveglieri, and A. Morzenti. *Formal Languages and Compilation, Third Edition*. Texts in Computer Science. Springer, 2019.

[6] F. DeRemer and T. Pennello. Efficient computation of LALR(1) look-ahead sets. *ACM Trans. on Prog. Lang. and Sys.*, 4(4):615–649, Oct. 1982.

[7] M. Harrison. *Introduction to formal language theory*. Addison Wesley, Reading, MA, 1978.

[8] K. Hemerik. Towards a taxonomy for ECFG and RRPG parsing. In *Language and Automata Theory and Applications (LATA)*, volume 5457 of *LNCS*, pages 410–421. Springer, 2009.

[9] G. Herman. Faster general parsing through context-free memoization. In *Programming Language Design and Implementation (PLDI)*, pages 1022–1035. ACM, 2020.

[10] A. Johnstone, E. Scott, and G. Economopoulos. The grammar tool box: A case study comparing GLR parsing algorithms. *Electron. Notes Theor. Comput. Sci.*, 110:97–113, 2004.

[11] A. Johnstone, E. Scott, and G. Economopoulos. Evaluating GLR parsing algorithms. *Sci. of Comp. Prog.*, 61(3):228–244, 2006.

[12] D. Knuth. On the translation of languages from left to right. *Inform. and Contr.*, 8:607–639, 1965.

[13] S. McPeak and G. C. Necula. Elkhound: A fast, practical GLR parser generator. In *Compiler Construction (CC)*, volume 2985 of *LNCS*, pages 73–88. Springer, 2004.

[14] P. W. Purdom and C. A. Brown. Parsing extended LR(k) grammars. *Acta Inf.*, 15:115–127, 1981.

[15] E. Scott and A. Johnstone. Right nulled GLR parsers. *ACM Trans. on Prog. Lang. and Sys.*, 28(4):577–618, 2006.

[16] E. Scott and A. Johnstone. GLL syntax analysers for EBNF grammars. *Sci. of Comp. Prog.*, 166:120–145, 2018.

[17] M. Tomita. *Efficient parsing for natural language: a fast algorithm for practical systems.* Kluwer, Boston, 1986.