CEUR-WS.org/Vol-2756/paper_17.pdf

A Generalized LR (1) Parser for Extended Context-Free Grammars*

Angelo Borsotti, Luca Breveglieri,
Stefano Crespi Reghizzi, and Angelo Morzenti

Politecnico di Milano, Piazza Leonardo Da Vinci 32, 20133 Milano, Italy
angelo.borsotti@mail.polimi.it,
{luca.breveglieri, stefano.crespireghizzi, angelo.morzenti}@polimi.it

Keywords: LR parsing, Tomita Generalized parser, extended CFG, EBNF

1 Background and objectives

A parser checks if a text is syntactically correct and returns its syntax tree(s). Parsers for complex grammars are produced by a generator. Knuth's LR (k) shift-reduce algorithm [4] applies to any deterministic Context-Free (CF) language, i.e., recognized by a deterministic push-down automaton (DPDA). Knuth's construction produces a DFA, to be called *pilot* for lack of an established name, which drives the DPDA by defining its moves. The LR (1) parsing is very efficient for simple computer languages. Yet it is rarely suitable to more complex computer languages and is totally inadequate for natural language processing.

Tomita's *Generalized* LR (1) parser (GLR) [7], later variously optimized [3], is linear-time for LR (1) grammar rules and degrades to polynomial-time if there are non-LR (1) rules. A few GLR parsers are used for unrestricted CF grammars in different domains, from natural language processing to compilers.

We address a practically motivated shortcoming in the GLR parsers: how to accept an *Extended CF* (ECF) grammar that has regular expressions (RE) in the right-hand sides (rhs) of the rules. Such REs may use union, concatenation and Kleene star. In fact, the syntax of a computer language is more conveniently expressed by means of ECF rules than by "pure" CF rules. Moreover, an ECF rule is neatly visualized through the graph of the DFA that recognizes the regular language of the rhs of the rule. A collection of such DFAs is called a *Transition Net* (TN) or syntax chart. We rely on TNs to formalize ECF languages and to build parsers. Such parsers do directly process languages specified by ECF or TN definitions, instead of requiring a preliminary conversion from ECF to CF.

The best known LR (1) parser generators, e.g., YACC/bison, do not support ECF grammars. However a recent theoretical extension [1,2], named ELR (1) parsing, is used here as a starting point to extend the GLR (1) method [7] to ECF and TN. We refer to [6] for a survey of the development of GLR (1) techniques and for a presentation of a state-of-the-art parser. Here it suffices to say that GLR

* Copyright © 2020 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

operates as a deterministic shift-reduce LR (1) parser, until an LR (1) conflict occurs. We recall that the memory tape of an LR (1) parser is a pushdown stack, where the parser states (*p-states*) are pushed by shift moves and popped by reduction moves. In principle, to cope with grammars with LR (1) conflicts, the deterministic LR (1) PDA should be turned into a nondeterministic one (NPDA). Yet simulating a NPDA by backtracking would be inefficient. Thus a different approach was pioneered by Lang [5] and developed by Tomita: a data-structure called *Graph-Structured Stack* (GSS) is used to subsume a multiplicity of pushdown stacks. Then, when an LR (1) conflict occurs, GLR traces all the possible moves in its GSS. During parsing the GSS evolves and represents all the possible successful move sequences. To compactly store the syntax tree(s) of the source text, GLR uses another data-structure called *Shared Packed Parse Forest* (SPPF), which we also use in our parser and do not have to describe.

To sum up, our goal was (*i*) to develop a new GLR parser generator that inputs ECF grammars or transition nets and outputs a parser that manages directly and efficiently the syntax structures of ECF grammars, and (*ii*) to experimentally compare the performances of our generated parser against those of existing GLR parsers. To do so, we have augmented the (deterministic) ELR (1) parser [1] so as to manage shift-reduce and reduce-reduce conflicts. Accordingly, we name GELR (1) this new *Generalized* ELR parser for unrestricted ECF grammars. We had to solve some technical problems to get a pilot suitable for the new situation and to adjust the GSS data-structure. We also had to carefully examine the issue of ambiguity, especially for the infinite degree ambiguity caused by an RE that applies a Kleene star to a language containing the empty string.

This extended abstract outlines: the GELR (1) parser generation algorithm and its differences from the deterministic ELR (1) case, the axiomatic definition of the GSS structure, and the correctness proof. The parser generator for ECF grammars and the parser itself are available as an interactive Javascript tool, see https://github.com/FLC-project/GELR, while an optimized Java implementation suitable for large grammars and long texts is under testing.

2 Definitions and constructions

We assume familiarity with the concepts of DFA, RE, CF grammar and language, and LR (1) parsing. The *terminal* alphabet is Σ and the empty word is ε. A character "$\dashv$" (called end-of-text) always marks the end of an input word.

Definition 1. *An Extended CF grammar (ECF) G is a 4-tuple (Σ, V_N, P, S), where V_N is the nonterminal alphabet, P is the set of rules and $S \in V_N$ is the axiom. A grammar symbol is an element of $V = \Sigma \cup V_N$. For each symbol $A \in V_N$, there is a unique rule $(A \to \alpha) \in P$. The right part α of a rule is an* RE *over V, which uses concatenation, union, Kleene star and cross, i.e., "$*$" and "$+$", the empty string ε, and parentheses. The language defined by α is called the* regular language associated *with A and is denoted R_A or $R(\alpha)$. For a (non-extended) CF grammar, the right part of any rule has the form $\alpha = \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$, with $\alpha_i \in V^*$.* □

We later assume that every grammar contains by default the *special rule* $S' \to S \dashv$, where the symbol S' acts as axiom and does not occur in the other rules.

Definition 2. *For an* ECF *grammar G, an immediate derivation is a binary relation $\Rightarrow$ over V^*, s.t. $z \Rightarrow z'$ if $z = u\,A\,v$, $z' = u\,w\,v$, $(A \to \alpha) \in P$, $w \in R(\alpha)$, with $z, z', u, v, w \in V^*$. The reflexive-transitive closure of $\Rightarrow$ is $\overset{*}{\Rightarrow}$. The language of G from A is $L(G,\, A) = \{\; x \in \Sigma^* \mid \quad A \underset{G}{\overset{*}{\Rightarrow}} x \;\}$. The language of G is $L(G) = L(G,\, S)$.* □

Definition 3. *A Transition Net (*TN*) $\mathcal{M}$ is a finite collection of* DFAs *M_A called machines, i.e., $\mathcal{M} = \{\; M_S,\; M_A,\; M_B,\; \ldots \;\}$, and machine M_S is the starting one. Each machine $M_A \in \mathcal{M}$ is a 5-tuple $M_A = (\, V,\; Q_A,\; \delta_A,\; 0_A,\; F_A \,)$. The input alphabet is $V = \Sigma \cup V_N$. The state set Q of $\mathcal{M}$ is the union $Q = \bigcup_{M_A \,\in\, \mathcal{M}} Q_A$. Without loss of generality, by assuming $Q_A \cap Q_B = \emptyset$ for any two machines $M_A \neq M_B$, we define the transition function (or graph) of a* TN *as $\delta = \bigcup_{M_A \,\in\, \mathcal{M}} \delta_A$, and we write δ instead of δ_A with no confusion. The regular language recognized by a machine M_A starting from a state q_A is $R\,(M_A,\, q_A) = \{\; x \in V^* \mid \quad q_A \in Q_A \;\wedge\; \delta\,(q_A,\, x) \in F_A \;\}$. We say that a net $\mathcal{M}$ is structurally equivalent to a grammar $G = (\Sigma,\, V_N,\, P,\, S)$ if for each rule $(A \to \alpha) \in P$ the identity $R\,(M_A,\, 0_A) = R_A$ holds. Then, the CF language recognized by a net starting from a machine M_A, with $A \in V_N$, is defined as $L\,(M_A,\, 0_A) = L\,(G,\, A)$, and the language defined by the net is $L\,(\mathcal{M}) = L\,(M_S,\, 0_S) = L\,(G)$.* □

Since the TNs we consider are structurally equivalent to reduced ECF grammars, every machine M_A is trim, recursively reachable from M_S, and $L\;(M_A,\; 0_A) \neq \emptyset$. As there is a one-to-one correspondence between a derivation of a string $x \in L\,(G)$ and an accepting computation on $\mathcal{M}$, the notion of syntax tree of x applies to language $L\,(\mathcal{M})$ as well. An *indexed syntax tree* is the tree of $x = x_0 \,\ldots\, x_{n-1}$, where each internal node X is labeled by a nonterminal and by an index pair $\langle i,\, j \rangle$ such that *frontier*$\,(X) = x_i \,\ldots\, x_{j-1}$ if $i < j$, or *frontier*$\,(X) = \varepsilon$ if $i = j$.

Given a TN, we build a DFA *pilot* that controls the parser. The construction mirrors the one for a conflict-free (deterministic) TN [1,2]. Each pilot state (p-state) is a set of *items*, but some extra information for managing shift-reduce or other conflict types is added to the items. Since we deal with DFAs, Knuth's dotted rule notation for items is replaced by machine state names, e.g., $A \to a^+\;(\,A \,\bullet\, B\,)^*$ is replaced by the state name 2_A of machine M_A, see Fig. 1.

Definition 4. *For a* TN *$\mathcal{M} = \{\; M_S,\; M_A,\; \ldots \;\}$, a GELR$\,(1)$ item is a 3-tuple:*

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

An item set I is a finite collection $I = \{\; \langle q_1,\, \pi_1,\, \lambda_1 \rangle,\; \langle q_2,\, \pi_2,\, \lambda_2 \rangle,\; \ldots \;\} \neq \emptyset$ of items. For any two items $\gamma_1 = \langle q,\, \pi_1,\, \lambda_1 \rangle$ and $\gamma_2 = \langle q,\, \pi_2,\, \lambda_2 \rangle$ with an identical state q, differing in their look-ahead π and/or pointer set λ, the coalesce operation yields a unique item $\gamma = \langle q,\, \pi_1 \,\cup\, \pi_2,\, \lambda_1 \,\cup\, \lambda_2 \rangle$. After coalescing in all the possible ways the items of a set I, the state component is a key that uniquely identifies the items in I. □

The state and look-ahead components of an item have exactly the same definition as in the deterministic case [1]. We recall that in all the LR (1) approaches, as well as in all the derived ones, an item set I represents a p-state.

Example (in Fig. 1). We show a non-deterministic ECF grammar and a structurally equivalent TN of minimal DFAs, built by well-known algorithms. Each p-state I contains items as of Def. 4. The pilot has many conflicts and one of them is outlined (see Fig. 1). The pointer set λ is new w.r.t. LR (1) items. It is needed to implement the reduction operation correctly and efficiently, by linking on a path (of unbounded length) the states visited in the current machine. The parser scans an input string, e.g., $a\,a\,a\,b$, and produces a *Graph-Structured Stack* (GSS), which compactly represents the push-down stacks of all the possible analysis threads. Our GSS is similar to [6], in turn based on Tomita's GSS, but differs in that ECF syntax trees are unranked, i.e., a node may have unboundedly many child nodes. Here is a new axiomatic GSS definition.

Definition 5. *Take a string $x = x_0 \dots x_{n-1}$ (with $n \geq 1$) ended by $\dashv$, i.e., either $n = 1$ and $x = x_0 = \dashv$, or $n \geq 2$ and $x_{n-1} = \dashv$. Take a* TN *$\mathcal{M}$ with pilot $\mathcal{P} = (\Sigma \cup V_N,\, R,\, \vartheta,\, I_0,\, -)$. The Graph-Structured Stack (GSS) of x is a directed labeled graph $\Gamma = (W,\, E)$, with a node set $W \subseteq \{\, 0,\, \dots,\, n\,\} \times R$ and arcs labeled by $V = \Sigma \cup V_N$ (including $\dashv$), i.e., an arc set $E \subseteq W \times (\Sigma \cup V_N) \times W$. Graph Γ must satisfy* (1-5)*:*

1. *The* initial node *of Γ is $v_0 = \langle 0,\, I_0 \rangle$, where I_0 is the initial state of the pilot.*
2. ***If it holds** (i) $\langle k,\, I \rangle \in W$ and (ii) $\vartheta\,(I,\, x_k) = J$ (with $0 \leq k < n$), **then** $\langle k{+}1,\, J \rangle \in W$ and $\big\langle\, \langle k,\, I \rangle,\, x_k,\, \langle k+1,\, J \rangle \,\big\rangle \in E$, called a* terminal arc*.*
3. ***If it holds** (i) $\langle h,\, I \rangle \in W$, (ii) I includes an item with machine state 0_A ($A \in V_N$), (iii) $\exists\, H \in R$ that is a reduction p-state for A, (iv) Γ has a path $\langle h,\, I \rangle \longrightarrow \langle k,\, H \rangle$ of length $k - h \geq 0$, and (v) $\vartheta\,(I,\, A) = J$, **then** $\langle k,\, J \rangle \in W$ and $\big\langle\, \langle h,\, I \rangle,\, A,\, \langle k,\, J \rangle \,\big\rangle \in E$, called a* nonterminal arc *(if $h = k$ then $I = H$, 0_A is final, the reduction is null).*
4. *Graph Γ has no node or arc other than those specified by the previous clauses.*
5. *The final or* accepting node *of Γ is $\langle n,\, I_f \rangle$, where I_f is the pilot state (p-state) that includes an item with machine state $2_{\&}$, which is the final state of the machine for the special (axiomatic) rule $S' \to S \dashv$. The final node may be missing.*

String x is said to be accepted by the GSS iff graph Γ includes a final node. □

3 Correctness, complexity and experimentation

Here we sketch the correctness proof of *acceptance*. The proof is based on the fact that the GSS includes information that encodes all and only the valid derivation(s) of an input string. A *derivation-encoding GSS subgraph* (DE-GSS-S) is a subgraph of a GSS, that encodes exactly one derivation.

Definition 6. *Take an input string $x \in L\,(\mathcal{M})$, ended by $\dashv$ as in Def.* 5*, and consider its indexed syntax tree. For each internal node $A_{h,\,k}$ that has a left-to-right sequence of child nodes σ, the* DE-GSS-S *is a subgraph of Γ that includes:*

– *a path $\langle h,\, I \rangle \longrightarrow \langle k,\, H \rangle$ labeled by σ', for some p-states I and H; consequently the pilot also has a path $I \longrightarrow H$ labeled by σ'*
– *an arc $\langle h,\, I \rangle \xrightarrow{A} \langle k,\, J \rangle$, where $J = \vartheta\,(I,\, A)$, except for the special sequence $S_{0,\,n-1} \dashv$ that matches the special rule $S' \to S \dashv$* □

Lemma 1. *A* GSS *has a* DE-GSS-S *iff it includes a path $\langle 0,\, I_0 \rangle \xrightarrow{S} \big\langle n - 1,\, \vartheta\,(I_0,\, S) \big\rangle \xrightarrow{\dashv} \big\langle n,\, \vartheta\,\big(\vartheta\,(I_0,\, S),\, \dashv\,\big) \big\rangle$, where I_0 is the initial pilot state.* □

Lemma 2. *For every* DE-GSS-S *there exists a derivation* $S \overset{*}{\Rightarrow} x$. $\square$

Lemma 3. *For every derivation* $S \overset{*}{\Rightarrow} x$ *there exists a* DE-GSS-S. $\square$

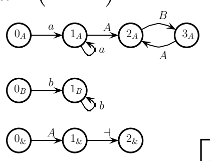
Theorem 1 (correctness). *A string* $x \in \Sigma^*$ *is accepted iff* $x \in L(\mathcal{M})$. $\square$

Proof. The three lemmas above and a definition provide the arguments:

$x \in L(\mathcal{M}) \Leftrightarrow \exists$ a deriv. $S \overset{*}{\Rightarrow} x \Leftrightarrow$ by Lem. 2, 3 $\exists$ DE-GSS-S for $x \Leftrightarrow$ by Lem. 1 the final node $\left\langle n,\, \vartheta\left(\vartheta\left(I_0,\, S\right),\, \dashv\right)\right\rangle \in \Gamma \Leftrightarrow$ by Def. 5 x is accepted $\square$

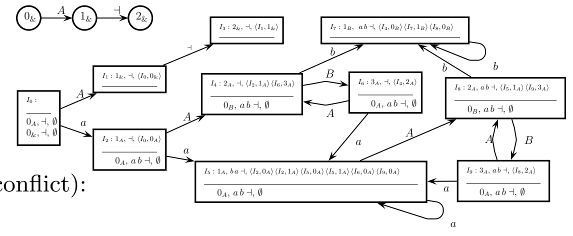
Experimentation. We have developed two tools. An interactive one is available at https://github.com/FLC-project/GELR. It has a graphical interface for small grammars and TNs, and has been instrumental to choose and validate different ideas while designing our algorithms. The optimized GELR (1) tool is under testing and has successfully generated efficient parsers for large grammars, e.g., C++ and HTML. We hope that a compared experimentation against [3] will qualify GELR (1) as a valid competitor of existing algorithms, for ECF parsing.

Grammar: $A \to a^+ \ (\ A\ B\ )^*$ $\quad B \to b^*$ $\quad$ (axiom rule $S_{\&} \to A \dashv$)

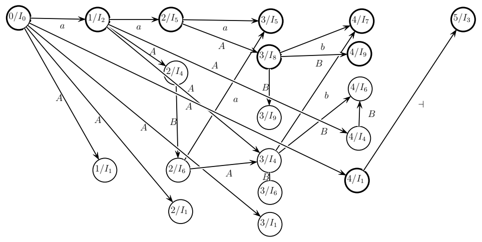


Transition net, where the final states are 1_A, 3_A, 0_B, 1_B and $2_{\&}$:

GELR (1) pilot
(with shift-reduce conflict):

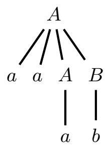


GSS of the ambiguous string $a\,a\,a\,b$ (it contains the GSS subgraph at the bottom):



Two derivations (of 5), the syntax tree of the first and its GSS subgraph:

$A \Rightarrow aaAB \Rightarrow aaaB \Rightarrow aaab$ $\quad$ and $\quad$ $A \Rightarrow aABAB \Rightarrow aaBAB \Rightarrow aa\varepsilon AB \overset{2}{\Rightarrow} aaab$



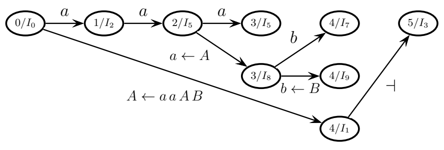


Fig. 1. These graphs are drawn by our tool. The pilot has a few LR (1) conflicts. One of them (SR) is in the p-state I_9: shift to I_5 on input a and reduction to A on look-ahead a (state 3_A is final). The pair $\langle I_8,\, 2_A \rangle$ of the reduction item $q,\ \pi,\ \lambda = 3_A,\ a\,b \dashv,\ \langle I_8,\, 2_A \rangle$ in I_9 (see Def. 4) points to item $2_A,\ a\,b \dashv,\ \ldots$ in the predecessor p-state I_8.

References

1. A. Borsotti, L. Breveglieri, S. Crespi Reghizzi, and A. Morzenti. Fast deterministic parsers for transition networks. *Acta Inf.*, 55(7):547–574, 2018.
2. S. Crespi Reghizzi, L. Breveglieri, and A. Morzenti. *Formal Languages and Compilation, Third Edition*. Texts in Computer Science. Springer, 2019.
3. A. Johnstone, E. Scott, and G. Economopoulos. Evaluating GLR parsing algorithms. *Sci. Comput. Program.*, 61(3):228–244, 2006.
4. D. Knuth. On the translation of languages from left to right. *Inform. and Contr.*, 8:607–639, 1965.
5. B. Lang. Deterministic techniques for efficient non-deterministic parsers. In J. Loeckx, editor, *Automata, Languages and Programming, 2nd Colloquium, University of Saarbrücken, Germany, July 29 - August 2, 1974, Proceedings*, volume 14 of *Lecture Notes in Computer Science*, pages 255–269. Springer, 1974.
6. E. Scott and A. Johnstone. Right nulled GLR parsers. *ACM Trans. Program. Lang. Syst.*, 28(4):577–618, 2006.
7. M. Tomita. *Efficient parsing for natural language: a fast algorithm for practical systems*. Kluwer, Boston, 1986.