Predictive Shift-Reduce Parsing for Hyperedge Replacement Grammars

Frank Drewes[1], Berthold Hoffmann[2], and Mark Minas[3] (✉)

[1] Umeå universitet, Sweden
drewes@cs.umu.se
[2] Universität Bremen, Germany
hof@informatik.uni-bremen.de
[3] Universität der Bundeswehr München, Germany
mark.minas@unibw.de

Abstract. Graph languages defined by hyperedge replacement grammars can be NP-complete. We study predictive shift-reduce (PSR) parsing for a subclass of these grammars, which generalizes the concepts of $SLR(1)$ string parsing to graphs. PSR parsers run in linear space and time. In comparison to the predictive top-down (PTD) parsers recently developed by the authors, PSR parsing is more efficient and more general, while the required grammar analysis is easier than for PTD parsing.

Keywords: hyperedge replacement grammar, graph parsing, grammar analysis

1 Introduction

"It is well known that hyperedge replacement (HR, see [11]) can generate NP-complete graph languages [1]. In other words, even for fixed HR languages parsing is hard. Moreover, even if restrictions are employed that guarantee L to be in P, the degree of the polynomial usually depends on L; see [16].[4] Only under rather strong restrictions the problem is known to become solvable in cubic time [21,5]." This quote is from our paper [8] on predictive top-down (PTD) parsing, an extension of $SLL(1)$ string parsing [17] to HR graph grammars [11]. The parser generator has been extended to the contextual HR grammars devised in [7,6]; it approximates Parikh images of auxiliary grammars in order to determine whether a grammar is PTD-parsable [9], and generates parsers that run in quadratic time, and in many cases in linear time.

Here we devise—somewhat complementary—efficient bottom-up parsers for HR grammars, called *predictive shift-reduce (PSR) parsers*, which extend $SLR(1)$ parsers [4], a member of the $LR(k)$ family of deterministic bottom-up parsers for context-free grammars [15]. We describe how PSR parsers work and how they can be constructed, and relate them to $SLR(1)$ string and PTD graph parsers.

In Sect. 2 we recall basic notions of HR grammars. We sketch $SLR(1)$ string parsing in Sect. 3 and describe in Sect. 4 how it can be lifted to PSR parsing

[4] This result has been exploited for parsing natural language in the system *Bolinas* [2].

of graphs with HR grammars. Then, in Sect. 5, we describe how HR grammars can be analysed for being PSR-parsable. Sect. 6 is devoted to the discussion of related work. Further work is outlined in Sect. 7.

2 Hyperedge Replacement Grammars

We let $\mathbb{N}$ denote the non-negative integers. A^* denotes the set of all finite sequences over a set A; the empty sequence is denoted by ε, the length of a sequence α by $|\alpha|$. For a function $f\colon A \to B$, its extension $f^*\colon A^* \to B^*$ to sequences is defined by $f^*(a_1 \cdots a_n) = f(a_1) \cdots f(a_n)$, for all $a_1, \ldots, a_n \in A$, $n \geqslant 0$.

We consider an alphabet Σ of symbols for labeling edges that comes with an *arity* function $\mathit{arity}\colon \Sigma \to \mathbb{N}$. The subset $\mathcal{N} \subseteq \Sigma$ is the set of *nonterminal labels*.

An *edge-labeled hypergraph* $G = \langle \dot{G}, \bar{G}, \mathit{att}_G, \ell_G \rangle$ *over* Σ (a *graph*, for short) consists of disjoint finite sets $\dot{G}$ of *nodes* and $\bar{G}$ of *hyperedges* (*edges*, for short) respectively, a function $\mathit{att}_G\colon \bar{G} \to \dot{G}^*$ that *attaches* sequences of nodes to edges, and a *labeling* function $\ell_G\colon \bar{G} \to \Sigma$ so that $|\mathit{att}_G(e)| = \mathit{arity}(\ell_G(e))$ for every edge $e \in \bar{G}$. Edges are said to be *nonterminal* if they carry a nonterminal label, and *terminal* otherwise; the set of all graphs over Σ is denoted by $\mathcal{G}_\Sigma$. A *handle graph* G for $A \in \mathcal{N}$ consists of just one edge x and $k = \mathit{arity}(A)$ pairwise distinct nodes $n_1, \ldots, n_k$ such that $\ell_G(x) = A$ and $\mathit{att}_G(x) = n_1 \ldots n_k$.

Given graphs G and H, a *morphism* $m\colon G \to H$ is a pair $m = \langle \dot{m}, \bar{m} \rangle$ of functions $\dot{m}\colon \dot{G} \to \dot{H}$ and $\bar{m}\colon \bar{G} \to \bar{H}$ that preserves labels and attachments: $\ell_H \circ \bar{m} = \ell_G$, and $\mathit{att}_H \circ \bar{m} = \dot{m}^* \circ \mathit{att}_G$ (where "$\circ$" denotes function composition). A morphism $m\colon G \to H$ is *injective* and *surjective* if both $\dot{m}$ and $\bar{m}$ have the respective property. If m is injective and surjective, it makes G and H *isomorphic*. We do not distinguish between isomorphic graphs.

Definition 1 (HR Rule). A *hyperedge replacement rule* (*rule*, for short) $r = L \to R$ consists of graphs L and R over Σ such that the *left-hand side* L is a handle graph with $\dot{L} \subseteq \dot{R}$.

Let r be a rule as above, and consider some graph G. An injective morphism $m\colon L \to G$ is called a *matching* for r in G. The *replacement* of $m(L)$ by R is then given as the graph H, which is obtained from the disjoint union of G and R by removing the single edge in $m(\bar{L})$ and identifying, for every node $v \in \dot{L}$, the nodes $m(v) \in \dot{G}$ and $v \in \dot{R}$. We then write $G \Rightarrow_{r,m} H$ (or just $G \Rightarrow_r H$) and say that H is *derived* from G by r.

The notion of rules introduced above gives rise to the class of HR grammars.

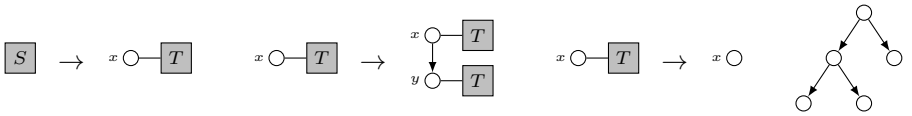
Definition 2 (HR Grammar [11]). A *hyperedge-replacement grammar* (*HR grammar*, for short) is a triple $\Gamma = \langle \Sigma, \mathcal{R}, Z \rangle$ consisting of a finite labeling alphabet Σ, a finite set $\mathcal{R}$ of rules, and a start graph $Z \in \mathcal{G}_\Sigma$.

We write $G \Rightarrow_{\mathcal{R}} H$ if $G \Rightarrow_{r,m} H$ for some rule $r \in \mathcal{R}$ and a matching $m\colon L \to G$, and denote the transitive-reflexive closure of $\Rightarrow_{\mathcal{R}}$ by $\Rightarrow^*_{\mathcal{R}}$. The language generated by Γ is given by $\mathcal{L}(\Gamma) = \{G \in \mathcal{G}_{\Sigma \setminus \mathcal{N}} \mid Z \Rightarrow^*_{\mathcal{R}} G\}$.

Without loss of generality, we assume that the start graph Z consists of a single edge labeled with a symbol $S \in \mathcal{N}$ of arity 0, that it is the left-hand side of just one rule, and that it does not occur in any right-hand side.

Graphs are drawn as in Example 1. Circles represent nodes, and boxes of different shapes represent edges. The box of an edge contains its label, and is connected to the circles of its attached nodes by lines; these lines are ordered clockwise around the edge, starting to its left. Terminal edges with two attached nodes are usually drawn as arrows from their first to their second attached node, and the edge label is ascribed to that arrow (but omitted if there is just one label, as in Example 1 below). In rules, identifiers like "x" at nodes identify corresponding nodes on the left-hand and right-hand sides.

Example 1. With a start graph as assumed above, the HR grammar below derives n-ary trees, like the graph on the right:



3 Shift-Reduce Parsing of Strings

Our shift-reduce parser for HR grammars borrows and extends concepts known from the family of context-free $LR(k)$ parsers [15], which is why we recall these concepts first. As context-free grammars, shift-reduce parsing, and in particular $LR(k)$ parsing appear in every textbook on compiler construction, we discuss these matters just at hand of an example.

Example 2. The *Dyck language* of matching nested parentheses consists of strings over the symbols "[" and "]"; it can be defined by a context-free string grammar with four rules $\mathcal{D} = \{S \to T, T \to [\,B\,], B \to T\,B, B \to \varepsilon\}$, to which we refer by the numbers 0 to 3; S, T, and B are nonterminals, and ε denotes the empty string. Starting with the string consisting only of S, the rules are applied to strings of nonterminals and terminals, by replacing an occurrence of their left-hand side by their right-hand side; this is done repeatedly until all nonterminals have been replaced. So we can derive a word of the Dyck language:

$$S \underset{0}{\Rightarrow} T \underset{1}{\Rightarrow} [B] \underset{2}{\Rightarrow} [TB] \underset{3}{\Rightarrow} [T] \underset{1}{\Rightarrow} [[B]] \underset{3}{\Rightarrow} [[\,]] \tag{1}$$

A context-free *parser* checks whether a string like "[[ ]]" belongs to the language of a grammar, and constructs a derivation as above if this is the case. A parser is modeled by a stack automaton that reads an input string from left to right, and uses a stack for remembering its actions. In a (general) shift-reduce parser, a configuration can be represented as $\alpha \centerdot w$, where w is the unconsumed input, a terminal string, and α is the stack, consisting of the nonterminal and terminal symbols that have been parsed so far. (The rightmost symbol is the "top".) The parser has its name from the two kinds of actions it performs (where α is an arbitrary string of symbols, and w an arbitrary terminal string):

– *Shift* consumes the first terminal symbol a of the input, and pushes it onto the stack. Our parser does shifts for parentheses:

$$\alpha\,\centerdot\,[w \vdash \alpha[\,\centerdot\,w \qquad \alpha\,\centerdot\,]w \vdash \alpha]\,\centerdot\,w$$

– *Reduce* pops the right-hand side symbols of a production from the stack, and pushes its left-hand side onto it. Our parser has the reductions (for symbol sequences α and terminal symbol sequences w):

$$T\,\centerdot \vdash_0 S\,\centerdot \qquad \alpha[B]\,\centerdot\,w \vdash_1 \alpha T\,\centerdot\,w \qquad \alpha TB\,\centerdot\,w \vdash_2 \alpha B\,\centerdot\,w \qquad \alpha\,\centerdot\,w \vdash_3 \alpha B\,\centerdot\,w$$

If the rule is the start rule, and α and w are empty, the parser terminates, and *accepts* the word, as in the first reduction.

A successful *parse* of a string w is a sequence of shift and reduce actions starting from the *initial configuration* $\varepsilon\,\centerdot\,w$ to the *accepting configuration* $S\,\centerdot\,\varepsilon$, as below:

$$\centerdot[[]] \vdash [\centerdot[]] \vdash [[\centerdot]] \underset{3}{\vdash} [[B\centerdot]] \vdash [[B]\centerdot] \underset{1}{\vdash} [T\centerdot] \underset{3}{\vdash} [TB\centerdot] \underset{2}{\vdash} [B\centerdot] \vdash [B]\centerdot \underset{1}{\vdash} T\centerdot \underset{0}{\vdash} S\centerdot$$

The reduction steps of a successful parse, read in reverse, yield a rightmost derivation of "[[]]" from S, in this case the one in (1) above.

This parser is *nondeterministic*: E.g., in the configuration "$[TB\centerdot]$", the following actions are possible:

1. a reduction with rule $B \to T\,B$, leading to the configuration $[B\centerdot]$;
2. a reduction with rule $B \to \varepsilon$, leading to the configuration $[TBB\centerdot]$; or
3. a shift of the symbol "]", leading to the configuration $[TB]\centerdot$.

Only action 1 will lead to the successful parse above. After action 2, further reduction is impossible, even after a subsequent shift of the unconsumed "]"; after action 3, no further action is possible. In such situations, the parser must *backtrack*, i.e., undo actions and try alternative ones, until it can accept the word, or fails altogether.

Since backtracking is inefficient, shift-reduce parsers are extended by two concepts so that they can predict which action in a configuration will lead to a successful parse:

– A *lookahead* of $k > 0$ input symbols helps to decide for an action. In the situation sketched above, the reductions (1) and (2) should only be done if the first input symbol is "]", which is the only terminal symbol that may follow B in the derivations with the grammar.
– A *characteristic finite automaton* (CFA) controls the order in which actions are performed; in the configuration $\alpha[T\,B\centerdot]$, the CFA should indicate that rule $B \to T\,B$ shall be reduced, not rule $B \to \varepsilon$.

Different lengths of lookahead, and several notions of CFAs can be used to construct a predictive shift-reduce parser. The most general one is Knuth's $LR(k)$ method [15]; here we just consider the simplest case of DeRemer's $SLR(k)$ parser [4], namely for a single symbol of lookahead, i.e., $k = 1$.

The transition diagram of the CFA $A_{\mathcal{D}}$ for the Dyck language is shown in Fig. 1. It is constructed as follows. The nodes q_0 to q_6 define its *states*, which are characterized by sets of so-called items; an *item* is a rule with a dot between the symbols on the right-hand side; e.g., state q_3 is characterized by the single item $T \to [B\,\centerdot\,]$; in this state, the parser has recognized the symbols [and B of rule $T \to [B]$, but not the closing parenthesis. The transitions $q \xrightarrow{x} q'$ define the successor state q' of a state q after recognizing a symbol x.

The start state q_0 is described by the item $S \to \centerdot\, T$, which is called a *kernel item*. Since recognizing the nonterminal T implies to recognize the rule of T, $T \to \centerdot\,[B]$ is the *closure item* of this state. The symbols appearing right of the dot in state q_0 can be recognized next; so, state q_0 has two transitions: under the nonterminal T to state q_1 with the kernel item $S \to T\,\centerdot$ (T is being read), and under the terminal "[" to state q_2 with the kernel item $T \to [\,\centerdot\, B]$. State q_2 has closure items $B \to \centerdot\,\varepsilon$ and $B \to \centerdot\, TB$, and the latter item has a further closure item $T \to \centerdot\,[B]$. While state q_1 has no transitions (nothing more needs to be recognized), state q_2 has three successor states, under the nonterminals B, T, and the terminal "[". The transition under "[" loops on state q_2. The remaining states and transitions are determined analogously.

The stack of the $SLR(1)$ parser is extended to contain an alternating sequence of states and symbols, e.g., "$q_0[q_2[q_2Tq_4Tq_4$", which record a path $q_0 \xrightarrow{[} q_2 \xrightarrow{[} q_2 \xrightarrow{T} q_4 \xrightarrow{T} q_4$ in its CFA $A_{\mathcal{D}}$, starting in the initial state. The actions of the parser are determined by its actual (topmost) state, and are modified wrt. those of the nondeterministic parser as follows:

– *Shift* consumes the first terminal a of the input, and pushes it onto the stack, with the successor state q' so that $q \xrightarrow{a} q'$ is in $A_{\mathcal{D}}$. For our grammar, and $i \in \{0, 2, 4\}$:

$$\alpha q_3\,\centerdot\,]w \vdash \alpha q_3]q_5\,\centerdot\, w \qquad \alpha q_i\,\centerdot\,[w \vdash \alpha q_i[q_2\,\centerdot\, w$$

– *Reduce* pops the right-hand side of a production $A \to \beta$ (and the intermediate states) from the stack, but only if the lookahead—the first input

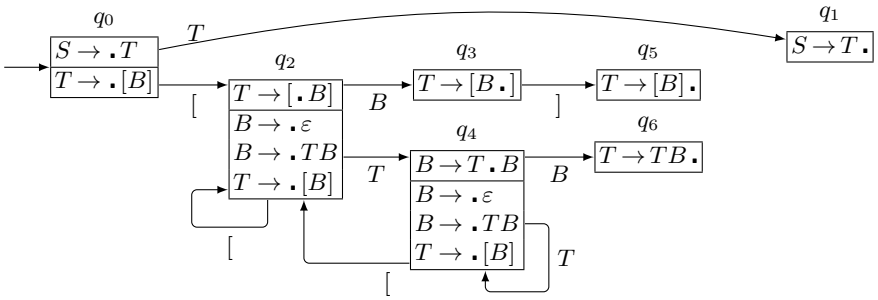


Fig. 1. $SLR(1)$ automaton $A_{\mathcal{D}}$ of the Dyck grammar

symbol—may follow A in derivations, and pushes the left-hand side A, and the successor state q' so that $q \xrightarrow{A} q'$. If $A = S$ and the input is empty, the parser accepts the word. For our grammar:

$$\begin{aligned} q_0 T q_1 \centerdot \vdash_0 S \qquad & \alpha q_0[q_2 B q_3]q_5 \centerdot w \vdash_1 \alpha q_0 T q_1 \centerdot w \\ \alpha q_2[q_2 B q_3]q_5 \centerdot w \vdash_1 \alpha q_2 T q_4 \centerdot w \qquad & \alpha q_4[q_2 B q_3]q_5 \centerdot w \vdash_1 \alpha q_4 T q_4 \centerdot w \\ \alpha q_2 T q_4 B q_6 \centerdot]w \vdash_2 \alpha q_2 B q_3 \centerdot]w \qquad & \alpha q_4 T q_4 B q_6 \centerdot]w \vdash_2 \alpha q_4 B q_6 \centerdot]w \\ \alpha q_2 \centerdot]w \vdash_3 \alpha q_2 B q_3 \centerdot]w \qquad & \alpha q_4 \centerdot]w \vdash_3 \alpha q_4 B q_6 \centerdot]w \end{aligned}$$

The CFA may reveal conflicts for predictive parsing:

- If a state allows to shift some terminal a, and a reduction under the lookahead a, this is a *shift-reduce conflict*.
- If a state allows reductions of different productions under the same lookahead symbol, this is a *reduce-reduce conflict*.

Whenever the automaton is conflict-free, the $SLR(1)$ parser exists, and is deterministic. The automaton $A_{\mathcal{D}}$ for the Dyck language is conflict-free: In states q_2 and q_4, rule $B \to \varepsilon$ can be reduced if the input begins with the only follower symbol "]" of B, which is not in conflict with the shift transitions from these states under "[".

A deterministic parse with the $SLR(1)$ parser for $\mathcal{D}$ is as follows:

$$\begin{aligned} q_0 \centerdot [[\,]] &\vdash q_0[q_2 \centerdot [\,]] \quad \vdash q_0[q_2[q_2 \centerdot]] \quad \underset{3}{\vdash} q_0[q_2[q_2 B q_3 \centerdot]] \vdash q_0[q_2[q_2 B q_3]q_5 \centerdot] \\ &\underset{1}{\vdash} q_0[q_2 T q_4 \centerdot] \underset{3}{\vdash} q_0[q_2 T q_4 B q_6 \centerdot] \underset{2}{\vdash} q_0[q_2 B q_3 \centerdot] \quad \vdash q_0[q_2 B q_3]q_5 \centerdot \\ &\underset{1}{\vdash} q_0 T q_1 \centerdot \quad \underset{0}{\vdash} S \text{ (accept)} \end{aligned}$$

4 Predictive Shift-Reduce Parsing for HR Grammars

We shall now transfer the basic ideas of shift-reduce parsing to HR grammars. First we define a textual notation for graphs and HR rules. A graph G can be represented as a pair $u_G = \langle s, \dot{G} \rangle$, called *(graph) clause* of G, where s is a sequence of (*edge*) *literals* $a(x_1, \ldots, x_k)$, one for every edge $e \in \bar{G}$ with $\ell_G(e) = a$ and $att_G(e) = x_1 \ldots x_k$. When writing down u_G, we omit $\dot{G}$ and write just s if $\dot{G}$ is clear from the context. For an HR rule $L \to R$, the *rule clause* is $u_L \to u_R$, with $\dot{L} \subseteq \dot{R}$.

While the order of literals in a graph clause is irrelevant, we shall process the literals on the right-hand side of a rule clause in the given order.[5]

Example 3 (Tree Rule Clauses and Tree Clauses). The rules of the tree grammar in Example 1 are represented by the clauses

$$S() \to T(x) \qquad T(y) \to T(y)\, edge(y, z)\, T(z) \qquad T(y) \to \varepsilon$$

[5] We assume that this order is provided with the HR grammar. Finding an appropriate order for PSR parsing automatically can be done by dataflow analysis, but is outside the scope of this paper.

We shall refer to them by r_1, r_2, r_3. The empty sequence ε in the last rule is a short-hand for the clause $\langle \varepsilon, \{y\} \rangle$. One of the possible clauses representing the graph in Example 1 is "$\mathit{edge}(1,2)\ \mathit{edge}(1,3)\ \mathit{edge}(2,4)\ \mathit{edge}(2,5)$".

We will use this simple example to demonstrate the basic ideas of PSR parsing.

The PSR graph parser will use configurations $\alpha \centerdot w$, and rely on a CFA for its control, just like an $SLR(1)$ parser. However, instead of just symbols, the configurations will consist of literals, and something has to be done in order to properly determine the nodes of these literals in the host graph. This makes the construction more complicated.

If we disregard for a moment the assignment of host graph nodes to the literals, the states of the CFA are defined as sets of items, i.e., of rule clauses with a dot at some place in their right-hand side. Consider the kernel item $T(y) \to T(y)\ \mathit{edge}(y,z) \centerdot T(z)$ as an example. It has closure items of the form $T(y) \to \centerdot\, T(y) \mathit{edge}(y,z)\, T(z)$ and $T(y) \to \centerdot$. However, we have to take care of the node names: Since the closure is built according to the literal $T(z)$, the y in the closure items is actually the z of the kernel item, and their z is a node not in the kernel item at all. This has to be reflected in the closure items, without causing name clashes. Our method will be the following: First we distinguish those nodes in the kernel items to which nodes of the host graph will have been assigned when the state is entered. These are called the *parameters* of the state. In the present state – let us call it q_2 – the parameters will correspond to y and z since the literals to the left of the dot are already on the stack in this state. First we replace y and z by parameter names, say $\boldsymbol{x}$ and $\boldsymbol{y}$, in the kernel item. Then we rename the nodes on the left-hand side of the closure items according to the kernel literal causing the closure, i.e., we replace y by $\boldsymbol{y}$ in the closure items. The remaining node names are preserved – they correspond to nodes that have not yet been assigned any host graph nodes and are thus not parameters.[6] We now call this state $q_2(\boldsymbol{x}, \boldsymbol{y})$ to indicate that $\boldsymbol{x}$ and $\boldsymbol{y}$ are its formal parameters which have to be instantiated by concrete host graph nodes whenever the parser enters the state.

State $q_2(\boldsymbol{x}, \boldsymbol{y})$ now gets a transition under the literal $T(\boldsymbol{y})$ to a state, let us call it $q_3(\boldsymbol{x}, \boldsymbol{y})$, which has two kernel items

$$T(\boldsymbol{x}) \to T(\boldsymbol{x})\ \mathit{edge}(\boldsymbol{x}, \boldsymbol{y})\, T(\boldsymbol{y}) \centerdot \text{ and } T(\boldsymbol{y}) \to T(\boldsymbol{y}) \centerdot \mathit{edge}(\boldsymbol{y}, z)\, T(z).$$

This state also gets $\boldsymbol{x}$ and $\boldsymbol{y}$ as formal parameters. For the transition, we specify which nodes of $q_2(\boldsymbol{x}, \boldsymbol{y})$ define the parameters in $q_3(\boldsymbol{x}, \boldsymbol{y})$, writing it in the form of a "call" $q_3(\boldsymbol{x}, \boldsymbol{y})$. (Note that $\boldsymbol{x}$ and $\boldsymbol{y}$ are the parameters in state $q_2(\boldsymbol{x}, \boldsymbol{y})$ that are thus transferred to the (equally named) formal parameters of $q_3(\boldsymbol{x}, \boldsymbol{y})$.)

The second item of $q_3(\boldsymbol{x}, \boldsymbol{y})$ causes a transition under $\mathit{edge}(\boldsymbol{y}, z)$ to a state that would have a kernel item $T(\boldsymbol{y}) \to T(\boldsymbol{y}) \mathit{edge}(\boldsymbol{y}, \boldsymbol{z}) \centerdot T(\boldsymbol{z})$, where the actual parameter $\boldsymbol{z}$ is determined by the shift. However, this kernel item equals the one

[6] In general, we may have to introduce fresh names for non-parameter nodes in the closure items as well in order to avoid name clashes, but this is not necessary in the present example.

of $q_2(\boldsymbol{x}, \boldsymbol{y})$ up to the names of formal parameters so that we identify these states, and "call" $q_2(\boldsymbol{x}, \boldsymbol{y})$, specifying its actual parameters by writing $q_2(\boldsymbol{y}, z)$ on the transition. In general, a transition is thus defined by a literal, and by a call that defines the actual parameters, thereby passing nodes from one state to the other.

Fig. 2 shows the CFA for the tree grammar, built according to these considerations. (Note that $q_2(\boldsymbol{x}, \boldsymbol{y})$ and $q_3(\boldsymbol{x}, \boldsymbol{y})$ are as discussed above.) A special case arises in the start state $q_0(\boldsymbol{x})$. In order to work without backtracking, parsing has to start at nodes of the start rule that can be uniquely determined before parsing starts. In our example, the node u of the host graph that corresponds to x in the start item $S() \to \boldsymbol{.}\, T(x)$ must be determined, and assigned to the formal parameter $\boldsymbol{x}$ of $q_0(\boldsymbol{x})$ before running the parser. This is done by a procedure devised in [9, Sect. 4], computing the possible incidences of all nodes created by a grammar; only if the start node u can be distinguished from all other nodes generated by the grammar, predictive parsing is possible. In our example, u is the unique node in the input graph that has no incoming edges, i.e., the root of the tree. (If the input graph has more than one root, or no root at all, it cannot be a tree, and parsing fails immediately.) So the start item is renamed to $S() \to \boldsymbol{.}\, T(\boldsymbol{x})$, and $q_0(\boldsymbol{x})$ is entered with the call $q_0(u)$.

Fig. 4 shows moves of a PSR parser that accept the tree of Fig. 3 in state q_1^1 that indicates a reduction with the start rule. We are using a compact form to denote concrete instances of states (i.e., with actual parameters being assigned to them): for a state $q(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k)$ and an assignment $\mu\colon \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k\} \to \dot{G}$ we let q^μ denote $q(\mu(\boldsymbol{x}_1), \ldots, \mu(\boldsymbol{x}_k))$. Moreover, in Fig. 4 we just denote μ by a list

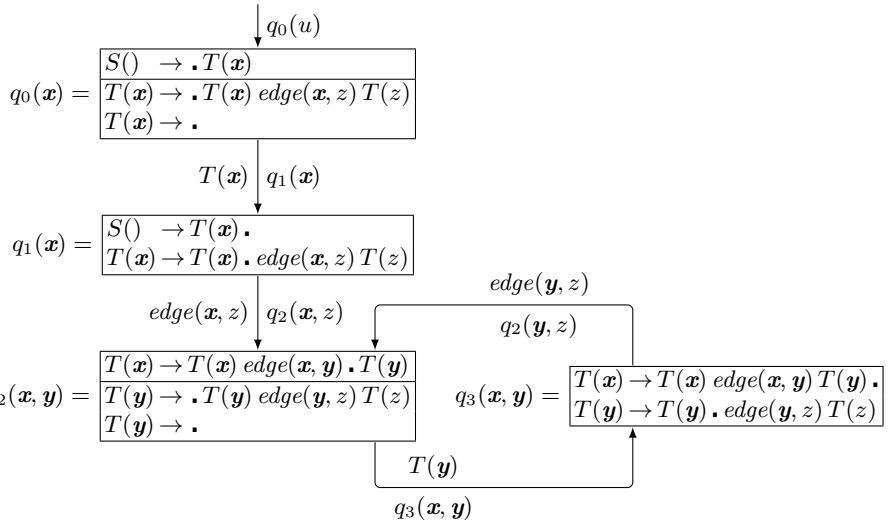


Fig. 2. The PSR CFA for the tree grammar Example 3

of nodes, i.e., q_0^1 denotes q_0^μ where $\mu(\boldsymbol{x}) = 1$. We use a similar shorthand for literals, e.g., e^{12} and T^1 abbreviate literals $edge(1,2)$ and $T(1)$, respectively.

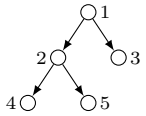


Fig. 3. An input tree

$$
\begin{aligned}
& \underline{q_0^1} \,.\, e^{12}e^{13}e^{24}e^{25} \\
\vdash_3 \quad & q_0^1T^1\underline{q_1^1} \,.\, e^{12}e^{13}e^{24}e^{25} \\
\vdash \quad & q_0^1T^1q_1^1e^{12}\underline{q_2^{12}} \,.\, e^{13}e^{24}e^{25} \\
\vdash_3 \quad & q_0^1T^1q_1^1e^{12}q_2^{12}T^2\underline{q_3^{12}} \,.\, e^{13}e^{24}e^{25} \\
\vdash \quad & q_0^1T^1q_1^1e^{12}q_2^{12}T^2q_3^{12}e^{24}\underline{q_2^{24}} \,.\, e^{13}e^{25} \\
\vdash_3 \quad & q_0^1T^1q_1^1e^{12}\underline{q_2^{12}T^2q_3^{12}e^{24}q_2^{24}T^4q_3^{24}} \,.\, e^{13}e^{25} \\
\vdash_2 \quad & \underline{q_0^1T^1q_1^1e^{12}q_2^{12}T^2q_3^{12}} \,.\, e^{13}e^{25} \\
\vdash \quad & q_0^1T^1q_1^1e^{12}q_2^{12}T^2q_3^{12}e^{25}\underline{q_2^{25}} \,.\, e^{13} \\
\vdash_3 \quad & q_0^1T^1q_1^1e^{12}\underline{q_2^{12}T^2q_3^{12}e^{25}q_2^{25}T^5q_3^{25}} \,.\, e^{13} \\
\vdash_2 \quad & \underline{q_0^1T^1q_1^1e^{12}q_2^{12}T^2q_3^{12}} \,.\, e^{13} \\
\vdash_2 \quad & q_0^1T^1\underline{q_1^1} \,.\, e^{13} \\
\vdash \quad & q_0^1T^1q_1^1e^{13}\underline{q_2^{13}} \,. \\
\vdash_3 \quad & \underline{q_0^1T^1q_1^1e^{13}q_2^{13}T^3q_3^{13}} \,. \\
\vdash_2 \quad & q_0^1T^1q_1^1 \,.
\end{aligned}
$$

Fig. 4. Moves of a PSR parser recognizing Fig. 3; places where reductions occur are underlined

The operations of the PSR parser work as follows on a host graph G:

Shift. Let the CFA contain a transition from state $p(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k)$ to state $q(\boldsymbol{y}_1, \ldots, \boldsymbol{y}_l)$ labeled by the terminal edge literal $e(v_1, \ldots, v_m)$ and the call $q(u_1, \ldots, u_l)$, and consider a concrete instance p^μ of $p(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k)$. Then there is a *shift* from p^μ to q^ν if

1. μ can be extended to the non-parameter nodes among $v_1, \ldots, v_m$, yielding an assignment μ' such that $e(\mu'(v_1), \ldots, \mu'(v_m))$ is a hitherto unconsumed edge literal of G (which is thus consumed) and
2. ν is defined by setting $\nu(\boldsymbol{y}_i) = \mu'(u_i)$ for $i = 1, \ldots, l$.

The shift then pushes the consumed edge $e(\mu'(v_1), \ldots, \mu'(v_m))$ and q^ν onto the stack.

Reduce. Let the topmost stack element be the state $q_k^{\nu_k}$, and assume that it contains an item of the form $A(w) \to s_1(w_1) \cdots s_k(w_k)\,\boldsymbol{.}$. Thus, $w, w_1, \ldots, w_k$ are sequences of formal parameters of q_k and the stack is of the form

$$\cdots p^\mu s_1(w_1')\, q_1^{\nu_1} \cdots s_k(w_k') q_k^{\nu_k}$$

where $w_i' = \nu_k^*(w_i)$ for all i.

The CFA would then allow a transition from p^μ to consume the instantiated left-hand side $A(\nu_k^*(w))$. Let q^ν be the concrete target state of this transition.

Then a *reduction* can be performed by popping $s_1(w'_1)q_1^{\nu_1} \cdots s_k(w'_k)q_k^{\nu_k}$ from the stack and pushing $A(\nu_k^*(w))$ and q^ν onto it.

Note that the parser has to choose the next action in states like q_1^ν or q_3^ν, which allow for a shift, but also for a reduction with rule r_1 and r_2, respectively. The parser predicts the next step by inspecting the unconsumed edges. We will discuss this in the next section. Note also that the PSR shift step differs from its $SLR(1)$ counterpart in an important detail: while the string parser just reads the uniquely determined next symbol of the input word, the graph parser must choose an appropriate edge. There may be states with several outgoing shift transitions (which do not occur in the present example), and the PSR parser has to choose between them. The parser, when it has selected a specific shift transition, may even have to choose between several edges to shift. E.g., in Fig. 4 the second step could shift the edge $edge(1, 3)$ instead of $edge(1, 2)$. We will discuss this problem below when considering shift-shift as well as other conflicts and the *free edge choice* property.

5 Predictive Shift-Reduce Parsability

A CFA can be constructed for every HR grammar; the general procedure works essentially as described above. However, one must usually deal with items whose left-hand sides still contain nodes which have not yet been located in the host graph. (Such items do not occur in the tree example.) We ignore this issue here due to space restrictions and refer to [8]. We shall now outline the three criteria for an HR grammar to be PSR-parsable (or just PSR for short):

1. *Neighbor-determined start nodes:* Prior to actual parsing, one must be able to determine the nodes where parsing will start. This has already been examined in [8,9] and is not considered in this paper.
2. *Conflict-freeness:* An $SLR(1)$ parser can predict the next action iff its CFA is conflict-free (cf. Sect. 3). We define a similar concept for PSR parsing in the following.
3. *Free edge choice:* The parser, when it has selected a specific shift transition, may have to choose between several edges matching the edge pattern, as described above. A grammar has the *free edge choice* property if the parser can always choose freely between these edges. There are sufficient conditions for this property that can be effectively tested when testing for conflict-freeness. This has already been examined in [8]; so we do not discuss it here.

We shall now define conflict-freeness in PSR parsing similar to $SLR(1)$ parsing so that conflict-freeness implies that the PSR parser can always predict the next action. A graph parser, different from a string parser, must choose the next edge to be consumed from a set of appropriate unconsumed edges. It depends on the next action of the parser which edges are appropriate. We define a conflict as the situation that an unconsumed edge is appropriate for an action, but could be consumed also if another action was chosen. Conflict-freeness just means that

there are no conflicts. Obviously, conflict-freeness then allows to always predict the correct action.

We now discuss how to identify host edges that are appropriate for the action caused by an item. For this purpose, let us first define items in PSR parsing more formally: An item $I = \langle L \to \alpha \centerdot \beta \mid P \rangle$ consists of an HR rule $L \to \alpha\beta$ in clause representation with a dot indicating a position in the right-hand side and the set P of parameters, i.e., those nodes in the item to which we have already assigned nodes in the host graph. These host nodes are not yet known when constructing the CFA and the PSR parser, but we can interpret parameters as abstract host nodes. A "real" host node assigned to a parameter during parsing is mapped to the corresponding abstract node. All other host nodes are mapped to a special abstract node $-$. Edges of the host graph are mapped to abstract edges being attached to abstract nodes, i.e., $P \cup \{-\}$, and each abstract edge can be represented by an *abstract (edge) literal* in the usual way. Note that the number of different abstract literals is finite because $P \cup \{-\}$ is finite.

Consider any valid host graph in $\mathcal{L}(\Gamma)$, represented by clause γ derived by the derivation $S = \alpha_1 \Rightarrow \cdots \Rightarrow \alpha_n = \gamma$. We assume that the ordering of edge literals is preserved in each derivation step. We then select any mapping of nodes in γ to abstract nodes $P \cup \{-\}$ such that no node in P is the image of two different host nodes. Edge literals are mapped to the corresponding abstract literals. The resulting sequence of literals can then be viewed as a derivation in a context-free string grammar $\Gamma(P)$ that can be effectively constructed from Γ in the same way as described in [9, Sect. 4]; details are omitted here because of space restrictions. $\Gamma(P)$ has the nice property that we can use this context-free string grammar instead of Γ to inspect conflicts. This is shown in the following.

Consider an item $I = \langle L \to \alpha \centerdot \beta \mid P \rangle$. Each edge literal $e = l(n_1, \ldots, n_k)$ has the corresponding abstract literal $\mathit{abstr}_P(e) = l(m_1, \ldots, m_k)$ where $m_i = n_i$ if $n_i \in P$, and $m_i = -$ otherwise, for $1 \leqslant i \leqslant k$. Let us now determine all host edges, represented by their abstract literals, which can be consumed next if the action caused by this item is selected. The host edge consumed next must have the abstract literal $\mathit{First}_P(\beta) := \mathit{abstr}_P(e)$ if I is a shift item, i.e., β starts with a terminal literal e. If I, however, causes a reduction, i.e., $\beta = \varepsilon$, we can make use of $\Gamma(P)$. Any host edge consumed next must correspond to an abstract literal that is a follower of the abstract literal of L in $\Gamma(P)$. This is exactly the same concept as it is used for $SLR(1)$ parsing and indicated in Sect. 3. Let us use the notion $\mathit{Follow}_P(L)$ for this set of followers.

As an example, consider state $q_3(\boldsymbol{x}, \boldsymbol{y})$ in Fig. 2 with its items $\langle L_i \to \alpha_i \centerdot \beta_i \mid P_i \rangle$, $i = 1, 2$, with $P_1 = \{x, y\}$ and $P_2 = \{y\}$. For the first item, one can compute

$$\mathit{Follow}_{P_1}(L_1) = \mathit{Follow}_{P_1}(T(x)) = \{\mathit{edge}(x, -), \mathit{edge}(-, -), \varepsilon\},$$

i.e., the host edge consumed next must either be an edge between the host node assigned to x and a node different from the ones assigned to x or y, or it must be a completely unrelated edge wrt. to the host nodes assigned to x or y, or all edges of the host edge have been completely consumed, indicated by ε.

For the second item, one can compute

$$\mathit{First}_{P_2}(\beta_2) = \mathit{abstr}_{P_2}(\mathit{edge}(y, z)) = \mathit{edge}(y, -),$$

i.e., the host edge under which the shift step is taken is an edge between the host node assigned to y and a node different from the ones assigned to x or y.

As $\mathit{First}_{P_2}(\beta_2)$ is not in $\mathit{Follow}_{P_1}(L_1)$ the edge under which the shift step is taken cannot be consumed next when the reduce step is taken instead. This condition is sufficient to avoid a conflict in $SLR(1)$ parsing, but this is not the case for PSR parsing. Because host edges are not consumed in a fixed sequence, $\mathit{edge}(y, -)$ might be consumed later when the reduce step is taken. We must actually compute the set of all abstract literals that may follow $\mathit{abstr}_{P_1}(L)$ immediately, or later in $\Gamma(P)$. Let us denote the set of all such abstract literals $\mathit{Follow}^*_P(L)$, whose computation from $\Gamma(P)$ is straightforward. In this example, one can see that

$$\mathit{Follow}^*_{P_1}(L_1) = \{\mathit{edge}(x, -), \mathit{edge}(-, -), \varepsilon\}.$$

We conclude that, if the parser can find a host edge matching $\mathit{edge}(y, -)$, this edge can never be consumed if the reduce step is taken; the parser must select the shift step. If it cannot find such a host edge, it must select the reduce step.

Note that the test for a conflict (which we have not yet defined formally) is not symmetric: we have just checked whether $\mathit{First}_{P_2}(\beta_2) \notin \mathit{Follow}^*_{P_1}(L_1)$. But we could also check whether any abstract literal in $\mathit{Follow}_{P_1}(L_1)$, i.e., if the reduce step is taken, can be consumed later when the shift step is taken instead. Let us denote the set of these abstract literals as $\mathit{Follow}^*_{P_2}(L_2, \beta_2)$, which can be computed using $\Gamma(P)$ in a straightforward way. In the tree example, it is

$$\mathit{Follow}^*_{P_2}(L_2, \beta_2) = \{\mathit{edge}(y, -), \mathit{edge}(-, -)\},$$

i.e., $\mathit{edge}(-, -) \in \mathit{Follow}_{P_1}(L_1) \cap \mathit{Follow}^*_{P_2}(L_2, \beta_2)$. Thus the parser cannot predict the next step by just checking the existence of host edges matching abstract literals in $\mathit{Follow}_{P_1}(L_1)$. But this is insignificant, because it can predict the correct action based on the other test (in the "opposite direction").

Analogous arguments apply if two different shift actions or two different reduction actions are possible in a state. However, there are no such states in the tree example.

We are now ready to define conflicting items in PSR parsing. In order to simplify the definition, we refer to sets $\mathit{Follow}_P(I)$ and $\mathit{Follow}^*_P(I)$ for an item $I = \langle L \to \alpha \centerdot \beta \mid P \rangle$. If I is a shift item, we define

$$\mathit{Follow}_P(I) := \{\mathit{First}_P(\beta)\} \text{ and } \mathit{Follow}^*_P(I) := \mathit{Follow}^*_P(L, \beta).$$

If I is a reduce item, we define

$$\mathit{Follow}_P(I) := \mathit{Follow}_P(L) \text{ and } \mathit{Follow}^*_P(I) := \mathit{Follow}^*_P(L).$$

Definition 3 (Conflicting items). Let I_1 and I_2 be two items with sets P_1 and P_2 of parameters, respectively. I_1 and I_2 are in *conflict* iff

$$\mathit{Follow}_P(I_1) \cap \mathit{Follow}^*_P(I_2) \neq \varnothing \wedge \mathit{Follow}_P(I_2) \cap \mathit{Follow}^*_P(I_1) \neq \varnothing$$

where $P = P_1 \cap P_2$. The conflict is called a *shift-shift*, *shift-reduce*, or *reduce-reduce conflict* depending on the actions caused by I_1 and I_2.

The above considerations make clear that the parser can predict the next action correctly if all states of its CFA are conflict-free. They also make clear that the parser has to consider only a fixed number of abstract edge literals in any state to choose the next action, and the host edge to shift if a shift is chosen. However, each abstract literal may match several host edges. But proper preprocessing of the host graph allows to find an (arbitrary, because of the free edge choice property) unconsumed host edge in constant time. This preprocessing is linear in the size of the graph (in space and time). Because the number of actions of the parser is linear in the size of the input graph, it follows that PSR parsing is linear in the size of the host graph.

The *Grappa* tool implemented by the third author generates PSR parsers based on the construction of the PSR CFA and the analysis of the three criteria outlined above. Table 1 summarizes test results for some HR grammars. The columns under "Grammar" indicate the size of the grammar in terms of the maximal arity of nonterminals (A), number of nonterminals (N), number of terminals (T) and number of rules (R). The columns under "CFA" indicate the size of the generated CFA in terms of the number of states (S), the overall number of items (I) and the number of transitions (T). The number of conflicts in the CFA are shown in the columns below "Conflicts" that report of shift-shift (S/S), shift-reduce (S/R) and reduce-reduce conflicts (R/R). Note that the grammars without any conflicts are PSR, the others are not. The columns under "Analysis" report on the time in milliseconds needed for creating the CFA (CFA) and checking for conflicts (Confl.), on a MacBook Pro 2013 (2,7 GHz Intel Core i7, Java 1.8.0 and Scala 2.12.1).[7]

Table 1. Test results of some HR grammars.

Example	Grammar				CFA			Conflicts			Analysis [ms]	
	A	N	T	R	S	I	T	S/S	S/R	R/R	CFA	Confl.
Trees (Example 3)	1	2	1	3	4	10	4	–	–	–	93	38
$a^n b^n c^n$ [8]	4	3	3	5	14	22	14	–	–	–	130	31
Nassi-Shneiderman diagrams [19]	4	3	3	6	12	78	59	–	–	–	233	76
Palindromes (Corollary 1)	2	2	2	7	12	32	19	–	–	–	142	36
Arithmetic expressions	2	4	5	7	12	34	22	–	–	–	137	49
Series-parallel graphs	2	2	1	4	7	63	32	12	4	–	179	61
Structured flowcharts	2	3	4	6	14	75	50	–	4	–	212	81

6 Comparison with Related Work

PSR parsing can be compared with $SLR(1)$ string parsing if we define the representation of strings as graphs, and of context-free grammars as HR grammars.

The *string graph* $w^\bullet$ of a string $w = a_1 \cdots a_n \in A^*$ (with $n \geqslant 0$) consists (in clausal form) of n edge literals $a_i(x_{i-1}, x_i)$ over $n+1$ distinct nodes $x_0, \ldots, x_n$. (The empty string ε is represented by an isolated node.) The HR rule of a context-free rule $A \to \alpha$ (where $A \in \mathcal{N}$ and $\alpha \in \Sigma^*$) is $\mathcal{A}^\bullet \to \alpha^\bullet$. For the purpose of this section, we represent an ε-rule $A \to \varepsilon$ by a rule that maps both nodes of $A^\bullet$ to the only node in $\varepsilon^\bullet$. Such rules are called "*merging*" in [8]. Context-free grammars and HR grammars can be *cleaned*, i.e., transformed into equivalent grammars without ε-rules and merging rules, respectively; however, they may loose their $SLL(1)$ and PTD property, respectively.

The *string graph grammar* of a context-free grammar G with a finite set $\mathcal{P}$ of rules and a start symbol S is the HR grammar $G^\bullet = (\Sigma, \mathcal{P}^\bullet, S^\bullet)$ with the rules $\mathcal{P}^\bullet = \{A^\bullet \to \alpha^\bullet \mid \mathcal{A} \to \alpha \in \mathcal{P}\}$. It is easy to see that the HR language of $G^\bullet$ is $\mathcal{L}(G^\bullet) = \{w^\bullet \mid w \in \mathcal{L}(G)\}$.

The following can easily be shown by inspection of the automata of string and HR grammars.

Proposition 1. *For an $SLR(1)$ grammar without ε-rules, the string graph grammar is PSR.*

This allows to establish the expected relation between PTD and PSR string graph grammars.

Theorem 1. *The clean version of a PTD string graph grammar is PSR.*

Proof. The main result of [12] states that the ε-cleaned version $\tilde{G}$ of an $SLL(1)$ grammar G is $SLR(1)$. This result can be lifted to string graph grammars as follows: By [8, Thm. 2], the string graph grammar $G^\bullet$ is PTD since G is $SLL(1)$. The string graph grammar $\tilde{G}^\bullet$ is the clean version of $G^\bullet$. Since $\tilde{G}$ is $SLR(1)$, $\tilde{G}^\bullet$ is PSR by Proposition 1. □

The inclusion of $SLR(1)$ grammars is proper, as is the inclusion of $SLL(1)$ grammars in PTD grammars.

Corollary 1. *There are context-free languages that cannot be generated with an $SLR(1)$ grammar, but have a PSR string graph grammar.*

Proof. The language of *palindromes* over $V = \{a, b\}$, i.e., all words which read the same backward as forward, can be generated by the unambiguous grammar with rules $S \to P$ and $P \to a \mid aa \mid aPa \mid b \mid bb \mid bPb$. Since the language cannot be recognized by a deterministic stack automaton [20, Prop. 5.10], this language neither has an $LL(k)$ parser, nor an $LR(k)$ parser. However, the grammar is PTD by [8, Theorem 2], and ε-free so that it is PSR by Thm. 1. □

For graph languages beyond string graphs, a comparison of PTD and PSR appears to be difficult: On the one hand, the tree grammar in Example 3 is left-recursive, and not PTD. (However, moving the *edge*-literal to the front in rule r_2 makes the grammar PTD.) On the other hand, PTD grammars with merging rules are not PSR, and it will be rather difficult to lift the main result of [12] to the general case of graph languages.

For early related work on efficient parsing algorithms, we quote from the conclusions of [8]: "Related work on parsing includes precedence graph grammars based on node replacement [10,14]. These parsers are linear, but fail for some PTD-parsable languages, e.g. the trees in Example 1. According to our knowledge, early attempts to implement *LR*-like graph parsers [18] have never been completed. Positional grammars [3] are used to specify visual languages, but can also describe certain HR grammars. They can be parsed in an LR-like fashion, but many decisions are deferred until the parser is actually executed. The CYK-style parsers for unrestricted HR grammars (plus edge-embedding rules) implemented in DiaGen [19] work for practical languages, although their worst-case complexity is exponential."

7 Conclusions

We have devised a predictive shift-reduce (PSR) parsing algorithm for HR grammars, along the lines of $SLR(1)$ string parsing. For string graphs, PSR is strictly more powerful than $SLR(1)$ and predictive top-down (PTD) parsing [8]. Checking PSR-parsability is complicated enough, but easier than for PTD, as we do not need to consider HR rules that merge nodes of their left-hand sides. PSR parsers also work more efficiently than PTD parsers, namely in linear vs. quadratic time. The reader is encouraged to download the *Grappa* generator of PTD and PSR parsers and to conduct own experiments.[7]

Like PTD, PSR parsing can be lifted to contextual HR grammars [6,7], a class of graph grammars that is more relevant for the practical definition of graph languages. This remains as part of future work. We will also study whether the specification of priorities for rules, as in the `yacc` parser generator [13], can be lifted to PSR parsing. However, an extension of PSR corresponding to $SLR(k)$ or $LR(k)$ may be computationally too difficult. A still open challenge is to find a HR (or contextual HR) language that has a PSR parser, but no PTD parser. The corresponding example for $LL(k)$ and $LR(k)$ string languages exploits that strings are always parsed from left to right—the palindrome example shows that this is not the case for PTD and PSR parsers.

References

1. I. Aalbersberg, A. Ehrenfeucht, and G. Rozenberg. On the membership problem for regular DNLC grammars. *Discrete Applied Mathematics*, 13:79–85, 1986.

[7] The *Grappa* tool is available at www.unibw.de/inf2/grappa; the examples mentioned in Table 1 can be found there as well.

2. D. Chiang, J. Andreas, D. Bauer, K. M. Hermann, B. Jones, and K. Knight. Parsing graphs with hyperedge replacement grammars. In *Proc. 51st Ann. Meeting of the Assoc. for Computational Linguistic (Vol. 1: Long Papers)*, pages 924–932, 2013.
3. G. Costagliola, A. De Lucia, S. Orefice, and G. Tortora. A parsing methodology for the implementation of visual systems. *IEEE Trans. Softw. Eng.*, 23:777–799, 1997.
4. F. L. DeRemer. Simple LR(k) grammars. *Comm. ACM*, 14(7):453–460, 1971.
5. F. Drewes. Recognising k–connected hypergraphs in cubic time. *Theoretical Computer Science*, 109:83–122, 1993.
6. F. Drewes and B. Hoffmann. Contextual hyperedge replacement. *Acta Informatica*, 52:497–524, 2015.
7. F. Drewes, B. Hoffmann, and M. Minas. Contextual hyperedge replacement. In *Proc. Applications of Graph Transformation with Industrial Relevance (AGTIVE'11)*, volume 7233 of *LNCS*, pages 182–197, 2012.
8. F. Drewes, B. Hoffmann, and M. Minas. Predictive top-down parsing for hyperedge replacement grammars. In *Proc. 8th Intl. Conf. on Graph Transformation (ICGT 2015)*, volume 9151 of *LNCS*, pages 19–34, 2015.
9. F. Drewes, B. Hoffmann, and M. Minas. Approximating Parikh images for generating deterministic graph parsers. In *Software Technologies: Applications and Foundations - STAF 2016 Collocated Workshops: DataMod, GCM, HOFM, MELO, SEMS, VeryComp, Vienna, Austria, July 4-8, 2016, Revised Selected Papers*, volume 9946 of *LNCS*, pages 112–128, 2016.
10. R. Franck. A class of linearly parsable graph grammars. *Acta Informatica*, 10(2):175–201, 1978.
11. A. Habel. *Hyperedge Replacement: Grammars and Languages*, volume 643 of *LNCS*. 1992.
12. B. Hoffmann. Cleaned SLL(1) grammars are SLR(1). Technical Report 17-1, Studiengang Informatik, Universität Bremen, 2017. `http://www.informatik.uni-bremen.de/~hof/papers/sllr.pdf`.
13. S. C. Johnson. Yacc: Yet another compiler-compiler. Computer Science Technical Report 32, AT&T Bell Laboratories, 1975.
14. M. Kaul. Practical applications of precedence graph grammars. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, *Graph-Grammars and Their Application to Computer Science*, volume 291 of *LNCS*, pages 326–342, 1986.
15. D. E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607 – 639, 1965.
16. C. Lautemann. The complexity of graph languages generated by hyperedge replacement. *Acta Informatica*, 27:399–421, 1990.
17. P. M. Lewis II and R. E. Stearns. Syntax-directed transduction. *J. ACM*, 15(3):465–488, 1968.
18. H. J. Ludwigs. A LR-like analyzer algorithm for graphs. In R. Wilhelm, editor, *GI - 10. Jahrestagung, Saarbrücken, 30. September - 2. Oktober 1980, Proceedings*, volume 33 of *Informatik-Fachberichte*, pages 321–335, 1980.
19. M. Minas. Diagram editing with hypergraph parser support. In *Proc. 1997 IEEE Symposium on Visual Languages (VL'97), Capri, Italy*, pages 226–233, 1997.
20. S. Sippu and E. Soisalon-Soininen. *Parsing Theroy I: Languages and Parsing*, volume 15 of *EATCS Monographs in Theoretical Computer Science*. 1988.
21. W. Vogler. Recognizing edge replacement graph languages in cubic time. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proc. Fourth Intl. Workshop on Graph Grammars and Their Application to Comp. Sci.*, volume 532 of *LNCS*, pages 676–687. 1991.