Predictive Top-Down Parsing for Hyperedge Replacement Grammars

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

[1] Umeå Universitet, Umeå, Sweden
drewes@cs.umu.se
[2] DFKI Bremen and Universität Bremen, Bremen, Germany
hof@informatik.uni-bremen.de
[3] Universität der Bundeswehr München, Neubiberg, Germany
Mark.Minas@unibw.de

Abstract. Graph languages defined by hyperedge replacement grammars can be NP-complete. We invent predictive top-down (PTD) parsers for a subclass of these grammars, similar to recursive descent parsers for string languages. The focus of this paper lies on the grammar analysis that computes neighbor edges of nonterminals, in analogy to the first and follow symbols used in SLL(1) parsing. The analysis checks whether a grammar is PTD parsable and yields all information for generating a parser that runs in linear space and quadratic time.

1 Introduction

It is well known that hyperedge replacement (HR, see [8]) 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 [11].[1] Only under rather strong restrictions the problem is known to become solvable in cubic time [4,17]. In this paper, we develop a parsing technique, called predictive top-down (PDT) parsing, which extends the *SLL*(1) string parsers of [12] to HR grammars and yields parsers that run in quadratic time, and in many cases in linear time. Of course, not all grammars are suitable for PDT parsing. As the requirements are not easy to check, an algorithm for the structural analysis of a given grammar is developed. This analysis is the focus of the present paper. It determines whether the grammar is PDT parsable and, if so, constructs a PDT parser. The basic idea is to determine the edges that can potentially be neighbors of the attached nodes of nonterminals. This information is computed approximatively by solving equations on semilinear sets of edge literals. It determines at which nodes of the input graph the parser has to start, and by which rule a nonterminal has to be expanded in a particular situation.

[1] Lautemann's result has been exploited for parsing natural language in the system *Bolinas* [2].

F. Parisi-Presicce and B. Westfechtel (Eds.): ICGT 2015, LNCS 9151, pp. 19–34, 2015.
DOI: 10.1007/978-3-319-21145-9_2

The remainder of this paper is structured as follows. In Sect. 2 we give the basic definitions of HR grammars. In Sect. 3, we recall $SLL(1)$ parsing and sketch what is needed to extend it to HR grammars. In Sect. 4 we introduce predictive top-down parsers for HR grammars, prove that they have quadratic time complexity, and show that they can indeed be considered as extensions of $SLL(1)$ parsers of string grammars. In Sect. 5, we sketch the analysis of HR grammars in order to determine whether a grammar is PTD parsable or not, and discuss how complex the analysis is. Further work is outlined in Sect. 6.

2 Hyperedge Replacement Grammars

In this paper, $\mathbb{N}$ denotes all 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 string w by $|w|$. 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 $a_i \in A$, $1 \leq i \leq n$, $n \geq 0$.

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

A *labeled hypergraph* $G = \langle \dot{G}, \bar{G}, att_G, \ell_G \rangle$ 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 $att_G\colon \bar{G} \to \dot{G}^*$ that *attaches* sequences of nodes to edges, and a *labeling* function $\ell_G\colon \bar{G} \to \Sigma$ so that $|att_G(e)| = arity(\ell_G(e))$ 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 $\mathcal{G}_\Sigma$. $G - e$ shall denote the subgraph of a graph G obtained by removing an edge $e \in \bar{G}$. A *handle graph* G for $A \in \mathcal{N}$ consists of just one edge x and pairwise distinct nodes $n_1, \ldots, n_{arity(A)}$ such that $\ell_G(x) = A$ and $att_G(x) = n_1 \ldots n_{arity(A)}$.

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 $att_H \circ \bar{m} = \dot{m}^* \circ att_G$ (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) $r = (L, R, \tilde{m})$ consists of graphs L and R over Σ such that the *left-hand side* L is a handle graph, and $\tilde{m}\colon (L - x) \to R$ is a morphism from the discrete graph $L - x$ to the *right-hand side* R. We call r a *merging rule* if $\tilde{m}$ is not injective.

Let r be a rule as above, and consider some graph G. A morphism $m\colon L \to G$ is called a *matching* for r in G. The *replacement* of $m(x)$ by R (via m) is then given as the graph H, which is obtained from the disjoint union of $G - m(x)$ and R by identifying, for every node $v \in \dot{L}$, the nodes $m(v) \in \dot{G}$ and $\tilde{m}(v) \in \dot{R}$. 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.

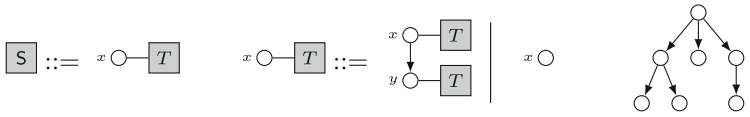
Definition 2 (HR Grammar [8]). A *hyperedge-replacement grammar* (*HR grammar*, for short) is a triple $\Gamma = \langle \Sigma, \mathcal{R}, Z \rangle$ consisting of a finite labeling alphabet Σ, a finite set $\mathcal{R}$ of rules, and a start graph $Z \in \mathcal{G}_\Sigma$.

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, and if not mentioned otherwise, we assume that the start graph Z is a handle graph for a nonterminal label $S \in \mathcal{N}$ with arity 0. We furthermore assume that S is not used in the right-hand side of any rule.

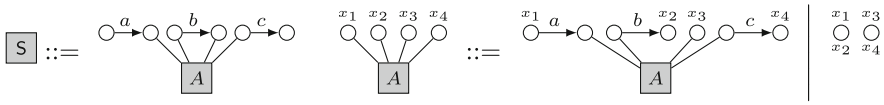
Graphs are drawn as in Examples 1 and 2. Circles represent nodes, and boxes of different shapes represent edges. The box of an edge contains its label, and is connected to the circles of its attached nodes by lines; these lines are ordered clockwise around the edge, starting to its left. Terminal edges 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_i" at nodes identify corresponding nodes on the left-hand and right-hand sides; in merging rules, several identifiers may be ascribed to a node on the right-hand side.

Example 1 (HR Grammars for Trees). With start symbol S, the HR grammar below derives n-ary trees like the graph on the right:



Strings $w = a_1 \cdots a_n \in A^*$ can be uniquely represented by *string graphs* consisting of $n+1$ nodes $x_0, \ldots, x_n$ and n binary edges $e_1, \ldots, e_n$ where e_i is labeled with a_i and connects x_{i-1} to x_i for $1 \le i \le n$. (The empty string ε with $n = 0$ is represented by an isolated node.)

Example 2 (HR Grammar for $a^n b^n c^n$). The language of string graphs given by the well-known non-context-free language $a^n b^n c^n$ (for $n > 0$) is generated by



3 Predictive Top-Down Parsing: From Strings to Graphs

We discuss how the idea of top-down parsing can be transferred from string grammars to HR grammars.

Example 3 (SLL(1)-Parsing for Tree Strings). The Dyck language of balanced parentheses can be generated by the context-free string grammar with four rules

$$S ::= T \quad T ::= (B) \quad B ::= T\,B \mid \varepsilon$$

The strings of the language generated by this grammar correspond to trees. For instance, the string "(((())())(()))" corresponds to the tree shown in Example 1.

Rules can be considered as abstract definitions of top-down parsers: Nonterminals act as a procedures that *expand* their rules, by *matching* terminal symbols (i.e., comparing them with the next input symbol, and consuming it in case of success) and by calls to procedures of other nonterminals. So the parser for T expands "(B)" by matching "(", calling the parser for B, and matching ")". It fails as soon as a match or one of the called parsers fails. If a nonterminal has alternative rules, *backtracking* is used to find the first alternative that succeeds. If the parser for B fails to expand "TB", it tries to expand "ε" instead, and fails if that does not succeed either. The parser for the start symbol initializes the input, expands its rule, and succeeds if the entire string has been consumed.

$SLL(k)$ parsers [12] avoid backtracking by pre-computing $k \geq 0$ terminal symbols of *lookahead* in order to predict which alternative of a nonterminal must be expanded. For B, and $k = 1$, we obtain $\mathit{First}(T\,B) = \{(\}$ and $\mathit{First}(\varepsilon) = \{\varepsilon\}$. For a lookahead "$\varepsilon$", the *followers* of the left-hand side nonterminal have to be determined, by inspecting occurrences of that nonterminal in other rules. Since B only occurs in the rule $T ::= (B)$, we obtain $\mathit{Follow}(B) = \{)\}$. A grammar is $SLL(k)$-parsable if the lookahead allows to predict which alternative must be expanded. Our example is $SLL(1)$; we can make the parser for B predictive by adding conditions for expanding rules:

$$\begin{aligned} B ::= {} & T\,B \ \textbf{if}\ \mathit{lookahead} = (\\ \mid {} & \varepsilon \quad \textbf{if}\ \mathit{lookahead} =) \end{aligned}$$

Pre-computation makes sure that every other lookahead lets the parser fail – no backtracking is needed.

We shall now transfer the basic ideas of top-down parsing to HR grammars. While the set of edges in a graph is inherently unordered, our parsing procedure must follow a prescribed search plan. For this reason, we generally assume that the edges in the right-hand side of each rule $(L, R, \tilde{m})$ are ordered. We therefore use a convenient representation for graphs: such a graph G can be represented as a pair $u = \langle s, \dot{G}\rangle$, called *(graph) clause* of G, where s is a sequence of *edge literals* $a(x_1, \ldots, x_k)$ such that there is an edge $e \in \bar{G}$ with $\ell_G(e) = a$, and $att_G(e) = x_1 \ldots x_k$. The order of the edge literals defines the order to be used by the parsing procedure if this graph is the right-hand side of a rule.[2] We let $u^\bullet = G$ and $\dot{u} = \dot{G}$ and call u *terminal* if $u^\bullet$ contains no nonterminal edge, and a *handle clause* if $u^\bullet$ is a handle. Let $\mathcal{C}_\Sigma$ and $\mathcal{T}_\Sigma$ denote the set of all graph clauses and terminal clauses, respectively, for a given alphabet Σ. When writing down u, we usually omit $\dot{G}$ and write just s if $\dot{G}$ is given by the context. We define the concatenation uv of two graph clauses $u = \langle s_G, \dot{G}\rangle$ and $v = \langle s_H, \dot{H}\rangle$ so that it represents the union of $u^\bullet$ and $v^\bullet$: $uv = \langle s_G s_H, \dot{G} \cup \dot{H}\rangle$.

Transferring the notion of derivations to clauses u, v we write $u \Rightarrow_r v$ iff $u^\bullet \Rightarrow_r v^\bullet$, where the ordering of edge literals is preserved, i.e., there are clauses

[2] We assume that the order of edges in a right-hand side is provided with the HR grammar. Finding an appropriate order automatically is left to future work.

α, β, γ and a nonterminal edge literal N such that $u = \alpha N \beta, v = \alpha\gamma\beta$ and γ corresponds to the order of the right-hand side of r. We call such a derivation a *left-most* derivation, written $u \overset{\text{L}}{\Rightarrow}_r v$, iff α is a terminal clause.

Top-down parsing for HR grammars uses the same ideas as for string grammars. A PTD parser consists of parsing procedures; each of them expands a nonterminal edge whose attached nodes are passed as parameters. We augment the rules with conditions under which they are chosen, and use them as abstract parsers. This is illustrated in the following examples:

Example 4 (PTD Parsing for Trees). The rules of the tree grammar (Example 1) are written down as

$$S() ::= T(x) \qquad T(x) ::= \mathit{edge}(x, y)\ T(x)\ T(y) \ \mid\ \varepsilon$$

The empty sequence ε representing the right-hand side of the third rule is just a short-hand for the clause $\langle \varepsilon, \{x\} \rangle$. This can be turned into a PTD parser for trees:

$$\begin{array}{lrll} p_1: & S() & ::= T(x) & \textbf{where } \neg \mathit{edge}(-, x) \\ p_2: & T(x) & ::= \mathit{edge}(x, y)\ T(x)\ T(y) & \textbf{if } \mathit{edge}(x, -) \\ p_3: & & \mid\ \varepsilon\ . & \end{array}$$

(Here, we use **where**- and **if**-clauses whose meaning will shortly be explained.)

Example 4 exhibits a clear similarity to $SLL(1)$ parsers. However, there are four major differences:

Firstly, expanding the edges of a right-hand side means, either to call the procedure for a nonterminal edge like $T(x)$ with its nodes as parameters, or to *match* a terminal edge like $\mathit{edge}(x, y)$. The latter means to select some matching edge that is correctly attached to the nodes bound to identifiers in the edge literal. This binds identifiers that were previously unbound, e.g., y to the target node of the matched edge. Each edge and node selected that way is marked as consumed. Consumed edges must not be matched again later, and all nodes and edges must have been consumed when the parser terminates.

Note that the ordering of the right-hand side specifies a search plan. Following this plan, the parser for the tree grammar will look for the terminal $\mathit{edge}(x, y)$ before trying to parse $T(y)$ in p_2. Thus, when invoking $T(y)$, node y has already been determined. Ordering the right-hand side by $T(x)\ \mathit{edge}(x, y)\ T(y)$ would make the rule left-recursive. By the proof of Lemma 1 such a grammar is not PTD parsable.

Secondly, conditions are written down in **if**-clauses. The right-hand side of the first rule whose **if**-clause evaluates to true is expanded, i.e., all **if**-clauses of the previous rules must have evaluated to false.

If-clauses represent conditions that the yet unconsumed part of the input graph must satisfy. This is analogous to $SLL(k)$ parsers examining a prefix of the yet unconsumed substring. In Examples 4 and 5, we use graph patterns as a simplified version of the more general conditions described in Sect. 5. Graph patterns are written down as extended graph clauses in which the ordering of

the literals is considered to be irrelevant. Nodes may be identifiers referring to already bound nodes, such as x, or they may be $-$, which matches any node that has not yet been bound by an identifier. A preceding $\neg$ indicates that the edge must not be present, i.e., this specifies a negative context. In our example, a nonterminal $T(x)$ is expanded by p_2 if the node bound by x has an outgoing edge that has not yet been consumed by the parser.

The third difference is that a graph parser must autonomously identify where the processing starts. In particular, the nodes generated by start rules are unknown in the beginning. Some (or all) of them — they are called *start nodes* in the following — can be uniquely determined by graph conditions which are written down as **where**-clauses. **Where**-clauses again employ extended graph clauses in our examples. However, note the difference to **if**-clauses: **If**-clauses are used to select rule alternatives, but do not bind identifiers, whereas **where**-clauses do not select rule alternatives, but specify how to bind identifiers such that a valid match is found. In p_1, x is bound to some node without an incoming edge.

A fourth distinguishing feature is that terminal edges are consumed by selecting matching edges. However, the parser must usually choose between several edges. In p_2, this holds for $edge(x, y)$, which is matched by any yet unconsumed edge leaving x. It is clear in this simple example that it does not make a difference which edge is chosen first. Since not every HR grammar has this property, grammar analysis must check that choosing an arbitrary edge from a set is insignificant for the outcome of the parse; if the parser fails for a particular choice, it must fail for every other choice as well, thus making backtracking unnecessary.

Example 5 (PTD Parsing for $a^n b^n c^n$). By choosing an ordering for the right-hand sides of rules, we turn the grammar of Example 2 into a PTD parser:

$$
\begin{array}{lrl}
p_1: & S() & ::= a(n_1, n_2)\ A(n_2, n_3, n_4, n_5)\ b(n_3, n_4)\ c(n_5, n_6) \\
 & & \quad \textbf{where}\ a(n_1, -), \neg a(-, n_1), b(-, n_4), c(n_4, -), \\
 & & \qquad c(-, n_6), \neg c(n_6, -) \\
p_2: & A(x_1, \underline{x_2}, x_3, \underline{x_4}) & ::= a(x_1, n_1)\ A(n_1, n_2, x_3, n_3)\ b(n_2, x_2)\ c(n_3, x_4) \\
 & & \quad \textbf{if}\ a(x_1, -) \\
p_3: & & \mid\ x_2 \leftarrow x_1; x_4 \leftarrow x_3
\end{array}
$$

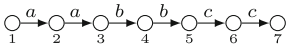
The example illustrates that it is not always possible to determine all nodes of a nonterminal edge before its procedure is called. Nonterminals may even have different *profiles*; each profile represents a certain subset of nodes that have already been determined prior to the procedure call. These nodes, called *profile nodes* in the following, are passed as parameters to the procedure. The other nodes are determined during the invocation and can be considered as "call by reference"-parameters. Therefore, different profiles require different procedures.

In our example, A has just one profile: only the first and the third node are determined when its procedure is called in p_1 or p_2; the second and the fourth node are yet unknown and must be determined by the procedure. The corresponding parameters x_2 and x_4 are underlined to illustrate this fact.

The example also demonstrates merging rules, which correspond to explicit assignments in the parser, here p_3: It is known from the A-profile that identifiers x_1 and x_3 are bound to profile nodes, and x_2 and x_4 are set accordingly by the procedure. This cannot cause any conflict because neither x_2, nor x_4 have been bound earlier, which is also known from the profile.

Figure 1 shows a trace of parsing the graph representing *aabbcc*. Each line of the trace consists of an action and the resulting bindings of identifiers to nodes after completion of the action. The currently active bindings are shown with a white background whereas the bindings of the calling procedures are shown with a gray background. Yet unbound identifiers are marked with a dash; new bindings are written in bold face. An identifier is bound to a node either by identifying start nodes (line 2), by explicit assignment (line 9), or by edge matching. An example of the latter is shown in line 3, which corresponds to edge $a(n_1, n_2)$ in p_1 of the parser. The matching edge is $a(1, 2)$ because n_1 is already bound to node 1. As a result, n_2 is bound to node 2. Note that the second and fourth parameter of the invocations of procedure A in line 4 and 7 are yet unknown. They are passed "by reference" and bound to nodes when the procedure binds their corresponding "formal parameters". For instance, the edge matching in line 10 assigns node 4 to x_2, but also to n_3. Finally note the selection of rule alternatives in line 5 and 8. The parser selects p_2 in line 5 because it finds edge $a(2, 3)$, and it selects p_3 in line 8 because there is no a-edge leaving node 3.

		S invocation						1st A invocation							2nd A inv.			
		n_1	n_2	n_3	n_4	n_5	n_6	x_1	x_2	x_3	x_4	n_1	n_2	n_3	x_1	x_2	x_3	x_4
1	call $S()$	-	-	-	-	-	-											
2	determine start nodes	**1**	-	-	**5**	-	**7**											
3	match $a(1, 2)$	1	**2**	-	5	-	7											
4	call $A(2, n_3, 5, n_5)$	1	2	-	5	-	7	**2**	-	**5**	-	-	-	-				
5	select alternative p_2	1	2	-	5	-	7	2	-	5	-	-	-	-				
6	match $a(2, 3)$	1	2	-	5	-	7	2	-	5	-	**3**	-	-				
7	call $A(3, n_2, 5, n_3)$	1	2	-	5	-	7	2	-	5	-	3	-	-	**3**	-	**5**	-
8	select alternative p_3	1	2	-	5	-	7	2	-	5	-	3	-	-	3	-	5	-
9	$x_2 \leftarrow 3, x_4 \leftarrow 5$	1	2	-	5	-	7	2	-	5	-	3	**3**	**5**	3	**3**	5	**5**
10	match $b(3, 4)$	1	2	**4**	5	-	7	2	**4**	5	-	3	3	5				
11	match $c(5, 6)$	1	2	6	5	**6**	7	2	4	5	**6**	3	3	5				
12	match $b(4, 5)$	1	2	4	5	6	7											
13	match $c(6, 7)$	1	2	4	5	6	7											

Fig. 1. A trace of parsing 

4 Predictive Top-Down Parsability

PTD parsers create left-most derivations for an input graph if such a derivation exists. More precisely, let Γ be a fixed HR grammar, z the start graph clause and

H a graph in the language of Γ. At each point of time, the parser has consumed some subgraph H' of H, represented as a terminal clause α, $\alpha^\bullet = H'$. The yet unexpanded or unmatched edges of the current stack of procedure invocations correspond to a graph clause x such that $z \overset{\text{L}}{\Rightarrow}^* \alpha x$. The clause x represents the remaining goal; parsing will continue to build $\alpha x \overset{\text{L}}{\Rightarrow}^* \alpha\beta$ such that $(\alpha\beta)^\bullet = H$.

For instance, when the A-procedure is called first in line 4 of Fig. 1, then $x = A(2, n_3, 5, n_5)\, b(n_3, 5)\, c(n_5, 7)$ since $S() \Rightarrow a(1,2)\, A(2,4,5,6)\, b(4,5)\, c(6,7)$. Note, however, that the parser was not yet able to bind n_3 and n_5. Edges $b(4,5)$ and $c(6,7)$ will be matched in lines 12 and 13.

The parser terminates successfully if $x^\bullet$ is the empty graph. Otherwise, $x = ey$ for an edge e. The parser continues with expanding e if e is nontermial, or with matching e if it is terminal:

Case 1 (e is nonterminal). The parser then calls the parsing procedure that corresponds to e with the profile nodes $P = \dot{\alpha} \cap \dot{e}$.

We use line 4 of Fig. 1 as an example again. Nodes 1, 5, and 7 have been determined as start nodes in line 2, and node 2 has been determined by matching $a(1,2)$ in line 3. Therefore, $\alpha = \langle a(1,2), \{1,2,5,7\}\rangle$. In the procedure call in line 4, only nodes 2 and 5 of $A(2, n_3, 5, n_5)$ are in $\dot{\alpha}$ and, therefore, profile nodes.

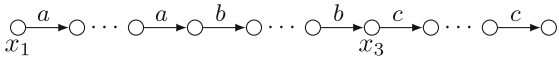
The procedure must choose a rule r that can continue the current derivation of the input graph, i.e., there must be a derivation $z \overset{\text{L}}{\Rightarrow}^* \alpha e y \overset{\text{L}}{\Rightarrow}_r \alpha u y \overset{\text{L}}{\Rightarrow}^* \alpha\beta$ where $\beta^\bullet$ is the remainder of the input graph, which we call the rest graph in the following. For a given set P of profile nodes, let $\text{Rest}(e, P, r)$ denote the set of all such rest graphs, taken over all possible input graphs in the language:

$$\text{Rest}(e,P,r) = \{\beta^\bullet \mid \exists \alpha, \beta \in \mathcal{T}_\Sigma \text{ and } u, y \in \mathcal{C}_\Sigma \text{ such that } z \overset{\text{L}}{\Rightarrow}^* \alpha e y \overset{\text{L}}{\Rightarrow}_r \alpha u y \overset{\text{L}}{\Rightarrow}^* \alpha\beta \text{ and } P = \dot{\alpha} \cap \dot{e}\}.$$

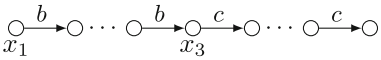
The procedure, therefore, must choose the rule r which satisfies

$$\beta^\bullet \in \text{Rest}(e, P, r). \tag{1}$$

For instance, when $A(x_1, x_2, x_3, x_4)$ has been called with profile nodes $P = \{x_1, x_3\}$ in lines 4 and 7 of Fig. 1, the parser has already consumed at least one a-edge, but no other edge. Thus, $\text{Rest}(e, P, p_2)$ consists of all graphs



with $k > 0$ a-edges and $m > k$ b-edges and as many c-edges, whereas $\text{Rest}(e, P, p_3)$ consists of all graphs



with $m > 0$ b-edges and as many c-edges.

The parsing procedure cannot predict which alternative rule has to be chosen if there are rules $r \neq r'$ that allow to continue the derivation with the same rest graph, i.e., predictive parsing requires that $\text{Rest}(e, P, r)$ and $\text{Rest}(e, P, r')$ are disjoint for all rules $r \neq r'$. Moreover, the parsing procedure needs an actual

indicator of the right choice. (1) does not result in a practical test, because it requires a parsing procedure (see Sect. 5). The **if**-clauses of the rules are supposed to be such an indicator. To make this more precise, let $\mathrm{Sel}(e, P, r)$ denote the set of all rest graphs that satisfy the **if**-clause of rule r. It is easy to see that the **if**-clauses of the rules are an actual indicator of the right choice iff the following two conditions are satisfied, because they imply $\mathrm{Rest}(e, P, r) \cap \mathrm{Rest}(e, P, r') = \emptyset$ for all rules $r \neq r'$ with the same left-hand side.

Condition 1. $\mathrm{Rest}(e, P, r) \subseteq \mathrm{Sel}(e, P, r)$ for each e, P, and each rule r.

Condition 2. $\mathrm{Sel}(e, P, r) \cap \mathrm{Sel}(e, P, r') = \emptyset$ for each e, P, and rules $r \neq r'$.

We continue our example when procedure A has been called for an edge $e = A(x_1, x_2, x_3, x_4)$ and profile nodes $P = \{x_1, x_3\}$. Instead of checking a condition that evaluates to true iff the rest graph $\beta^\bullet$ is in $\mathrm{Rest}(e, P, p_2)$ or $\mathrm{Rest}(e, P, p_3)$ when p_2 or p_3 is chosen, respectively, we simply check whether there is an a-edge leaving x_1, i.e., we check whether $\beta^\bullet \in \mathrm{Sel}(e, P, p_2)$ or $\beta^\bullet \in \mathrm{Sel}(e, P, p_3)$ where $\mathrm{Sel}(e, P, p_2)$ is the set of all graphs with an a-edge leaving x_1, and $\mathrm{Sel}(e, P, p_3)$ is its complement. Conditions 1 and 2 are obviously satisfied.

Section 5 will outline how one can construct **if**-conditions that can be checked in linear time in the size of the rest graph. The graph patterns used in Examples 4 and 5 are optimizations of such conditions that can be checked in constant time.

Case 2 (e is terminal). The parser chooses a yet unmatched edge. In doing so, it must consider its edge label and the nodes already determined by $\dot{\alpha}$. For instance, the parser in Example 4 can freely choose between any edge leaving node x when $edge(x, y)$ must be matched. This free choice must not lead to backtracking: it must not be possible that one choice results in a parser failure while a different choice results in a successful parse. More precisely, a PTD parsable HR grammar must satisfy the following condition:

Condition 3. For all derivations $z \overset{\mathrm{L}}{\Rightarrow}^* \alpha e y \overset{\mathrm{L}}{\Rightarrow}^* \alpha e \beta$ and $z \overset{\mathrm{L}}{\Rightarrow}^* \alpha e' y'$ where $y, y' \in \mathcal{C}_\Sigma$ and $\alpha, \alpha', e, e' \in \mathcal{T}_\Sigma$ such that e and e' consist of just one edge each, and $(\alpha e')^\bullet$ is a subgraph of $H = (\alpha e \beta)^\bullet$, there exists a derivation $y' \overset{\mathrm{L}}{\Rightarrow}^* \beta'$ and $H = (\alpha e' \beta')^\bullet$.

Apparently, one can create a PTD parser if one can predict correct rule alternatives and matching terminal edges does not require backtracking. This leads to the following definition:

Definition 3 (Predictive Top-Down Parsability). An HR grammar is called *predictively top-down (PTD) parsable* if one can augment rules with conditions, which can be decided in linear time, under which a rule is chosen such that Conditions 1, 2, and 3 are satisfied.

Note that Conditions 1 and 2 must be satisfied for all profiles that may occur, which depends on the start nodes that can be identified at the beginning. Profiles can be computed using a data flow analysis of the grammar rules (see Sect. 5).

Let us call a rule r *useful* if it occurs in a derivation $Z \Rightarrow^* G \Rightarrow_r G' \Rightarrow^* H$ such that H is a terminal graph. A rule r is called *useless* if it is not useful.

Lemma 1. *For every PTD parsable HR grammar without useless rules there exists a constant k such that the following holds: For every handle clause h, if $h \overset{\text{L}}{\Rightarrow}^k v$ then $v^\bullet$ contains at least one node or terminal edge not in $h^\bullet$.*

Proof. Let us assume the contrary, i.e., a PTD parsable HR grammar without useless rules and a handle clause h such that for each i, there is a clause w without terminal edges, $\dot{w} = \dot{h}$, and $h \overset{\text{L}}{\Rightarrow}^i w$. Because the set of nonterminal labels is finite and there are no useless rules, there must be a handle clause g and rules $r \neq r'$ such that $g \overset{\text{L}}{\Rightarrow}_r u \overset{\text{L}}{\Rightarrow}^* gv$ and $g \overset{\text{L}}{\Rightarrow}_{r'} x \overset{\text{L}}{\Rightarrow}^* \beta$ for some terminal clause β and a clause v without terminal edges, $\dot{v} \subseteq \dot{g}$. Therefore, there must be a derivation, where z is the start graph clause, $z \overset{\text{L}}{\Rightarrow}^* \alpha g y \overset{\text{L}}{\Rightarrow}_r \alpha u y \overset{\text{L}}{\Rightarrow}^* \alpha g v y \overset{\text{L}}{\Rightarrow}_{r'} \alpha x v y \overset{\text{L}}{\Rightarrow}^* \alpha \beta v y \overset{\text{L}}{\Rightarrow}^* \alpha\gamma$ for terminal clauses α, γ and a graph clause y. In this derivation, g must be expanded using rule r and r' with the same rest graph $\gamma^\bullet$, i.e., $\gamma^\bullet \in \text{Rest}(g, P, r) \cap \text{Rest}(g, P, r')$ with $P = \dot{\alpha} \cap \dot{g}$. Therefore, the grammar is not PTD parsable. □

In particular, the proof shows that HR grammars with left-recursive rules are not PTD parsable. In fact, by Lemma 1 the number of procedure invocations in a PTD parser depends linearly on the size of the input graph. This yields the following theorem.

Theorem 1 (Complexity of PTD Parsing). *PTD parsers have time complexity $\mathcal{O}(n^2)$ and space complexity $\mathcal{O}(n)$ where n is the size of the input graph.*

Proof. Parsers work without backtracking, and the number of procedure calls depends linearly on the size of the input graph, by Lemma 1. Each parsing procedure must choose a rule for expansion and, for this purpose, check the conditions of a fixed number of **if**-clauses, and match a fixed number of terminal edges, which has linear time complexity. Considering space complexity, each node and edge must carry a flag such that they can be marked as consumed. Moreover, the depth of recursion is linear in the size of the input graph. □

Note that this is a worst-case time complexity. If one can choose among alternative rules in constant time, which is possible for the tree-parser and the $a^n b^n c^n$-parser, time complexity is actually linear in the size of the input graph. We presume that this is the case for many parsers, but we have not yet identified the conditions under which this is the case.

Theorem 2 (Relation to $SLL(1)$-Parsing). *String-generating HR grammars for $SLL(1)$ grammars are PTD parsable, and there exist PTD parsable HR grammars for context-free string languages which are not $SLL(k)$-parsable.*

Proof. For the first statement, consider an $SLL(1)$ grammar. Then every rule can be turned into a corresponding HR rule; ε-rules $n ::= \varepsilon$ are turned into a merging rule $n(v_0, v_1) ::= v_0 = v_1$. Now consider two alternative rules $n ::= \alpha \mid \alpha'$ (where $\alpha, \alpha' \in \Sigma^*$). Since the grammar is $SLL(1)$, the sets of possible starts are disjoint for these rules, say F and F'. Then the clause $c = \{a(v_0, -) \mid a \in F\}$ and $c' = \{a(v_0, -) \mid a \in F'\}$ are such that the correct rule alternative of

$$n(v_0, v_k) ::= \ldots \textbf{ if } c \mid \ldots \textbf{ if } c'$$

can be predicted in constant time. The **where**-clause determining the start node is $\{\neg a(-, v_0) \mid a \in \Sigma\}$ which determines the start node in linear time. It is easy to see that the so defined PTD parser recognizes string graphs corresponding to those recognized by the original $SLL(1)$ grammar.

As for the second statement, palindromes over $\{a, b\}$ form a context-free language, but there is no $SLL(k)$ parser ($k \in \mathbb{N}$) because it would have to look ahead until the end of the input, which grows beyond any fixed k. However, one can easily construct a string-generating HR grammar and a PTD parser that "reads" the input simultaneously from the left and from the right. □

The possibility to have ε-rules is important for some $SLL(1)$ string grammars. While one can transform every context-free string grammar into an equivalent one without ε-rules (except a possible ε-rule as a start rule), these grammars are in general no longer $SLL(1)$, and the corresponding HR grammar is not PTD parsable. A direct translation of an $SLL(1)$ grammar with ε-rules into a PTD parsable HR grammar is possible only with merging rules. This has actually been the reason for allowing merging rules in HR grammars in this paper.

5 Grammar Analysis

Figure 2 describes all the tasks (drawn as rectangles with rounded corners) that must be performed in order to check PTD parsability of an HR grammar, called *original grammar*, and to create a PTD parser for it; some tasks depend on results (drawn as rectangles) of other tasks. Note that Fig. 2 omits the code generator, which creates the actual parser from the results of the earlier tasks.

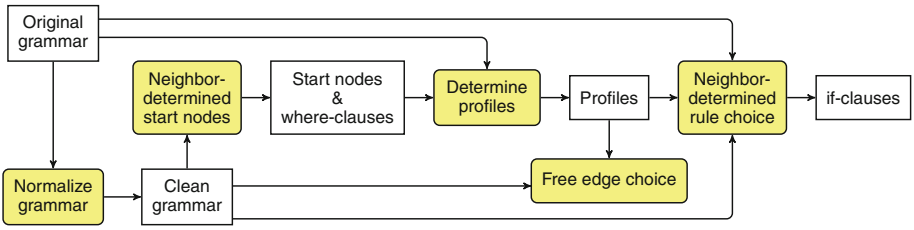


Fig. 2. Steps taken to check PTD parsability.

The first task, *Normalize grammar*, transforms the original grammar into an equivalent normalform that we call "clean", as it contains neither merging rules nor nonterminals with repetitions among their attached nodes. Although PTD parsers can deal with merging rules without any effort, as seen in Example 5, many grammar analysis tasks require a clean HR grammar. Such a task is *Neighbor-determined start nodes*; it identifies the start nodes that can be uniquely recognized in a syntactically correct graph by checking just their incident edges, and creates the **where**-clauses in the generated parser.

Profiles specify which attached nodes have already been matched to nodes in the input graph when a parsing procedure is invoked. Task *Determine profiles* computes all profiles using a data flow analysis of the grammar rules. It begins with the start nodes and continues by examining the matching of terminal edges as well as expanding nonterminal edges. A nonterminal label may actually have multiple profiles. The profiles are then used by task *Neighbor-determined rule choice* that tries to find **if**-clauses as conditions under which rules are chosen so that Conditions 1 and 2 are satisfied. Each profile gives rise to a parsing procedure. Finally, Conditions 3 is checked in task *Free edge choice*.

We now discuss details of task *Neighbor-determined rule choice* when an HR grammar Γ is given. The task tries to determine, for each rule r and each profile, an **if**-clause as a condition under which r has to be chosen in its parsing procedure. Case 1 in Sect. 4 already describes the situation. We now assume that a parsing procedure has been invoked for a specific nonterminal edge, i.e., a handle clause a, and a set P of profile nodes. The other nodes of a have not yet been determined; this is left to the procedure (see Fig. 1). We first show that the set $\mathrm{Rest}(a, P, r)$ of all rest graphs in this situation is an HR language. This is trivially true if a is the start graph clause z. Let us, therefore, assume that a does not represent the start graph and that $a \Rightarrow_r u$. A terminal graph $\beta^\bullet$ belongs to $\mathrm{Rest}(a, P, r)$ iff there is a derivation $a_0 \Rightarrow_{r_1} x_1 a_1 y_1 \Rightarrow_{r_2} \cdots \Rightarrow_{r_k} x_1 \dots x_k a_k y_k \dots y_1$ for some $k > 0$, rules $r_1, \dots, r_k$ and handle clauses $a_0, \dots, a_k$ with $a_0 = z$ and $a_k = a$ such that $u y_k \dots y_1 \Rightarrow^* \beta$, $x_1 \dots x_k \Rightarrow^* \alpha \in \mathcal{T}_\Sigma$, and $P = \dot{\alpha} \cap \dot{a}$. One can now construct an HR grammar, called *follower grammar* $\Gamma^{\mathrm{f}}(a, P, r)$, that generates $\mathrm{Rest}(a, P, r)$. Its derivations are of the form $z^{\mathrm{f}} \Rightarrow u a_k^{\mathrm{f}} \Rightarrow u y_k a_{k-1}^{\mathrm{f}} \Rightarrow \cdots \Rightarrow u y_k \dots y_1 a_0^{\mathrm{f}} \Rightarrow u y_k \dots y_1 \Rightarrow^* \beta$, starting from a start graph handle z^{f} with some new nonterminal label S' that is attached to the profile nodes P. We introduce a new nonterminal label A^{f} for each original nonterminal label A. Let each handle clause a_i have a nonterminal label A_i; handle clause a_i^{f} then has label A_i^{f} and is attached to the same nodes as a_i. $\Gamma^{\mathrm{f}}(a, P, r)$ has the rules of Γ as rule set, extended by a new rule that derives $z^{\mathrm{f}} \Rightarrow u a_k^{\mathrm{f}}$ and new rules for $a_0^{\mathrm{f}} \Rightarrow \varepsilon$ and $a_i^{\mathrm{f}} \Rightarrow y_i a_{i-1}^{\mathrm{f}}$ for $i = 1, \dots, k$.

Example 6. We illustrate the construction of a follower grammar for p_2 in Example 4. Let us assume that $a = T(x)$ and $P = \{x\}$. $\Gamma^{\mathrm{f}}(a, P, p_2)$ has the start graph $S'(x)$, its rule set consists of p_1, p_2, p_3 and the following rules:

$$\begin{aligned} S'(x) &::= \mathit{edge}(x, y)\ T(x)\ T(y)\ T^{\mathrm{f}}(x) \\ S^{\mathrm{f}}() &::= \varepsilon \\ T^{\mathrm{f}}(x) &::= S^{\mathrm{f}}() \mid T(y)\ T^{\mathrm{f}}(x) \mid T^{\mathrm{f}}(y) \end{aligned}$$

A parsing procedure can choose the rule alternative whose follower grammar has the rest graph in its language. While this question is decidable, it does not result in a practical test, because we need an actual indicator of the right choice that we can check in a given situation without presupposing a parsing procedure. Similar to $SLL(1)$, which looks ahead just one symbol, we examine unconsumed nodes and edges only within the *neighborhood* of the profile nodes P: Let u be a graph clause and $e = s(v_1, \dots, v_n)$ be an edge literal in u. The nodes v_i are

either profile nodes, $v_i \in P$, or other nodes, $v_i \in \dot{G} \setminus P$. We do not distinguish those other nodes, but map them to the "don't care" node – and define the *neighborhood literal* $nh_P(e) = s(x_1, \ldots, x_n)$, where $x_i = v_i$ if $v_i \in P$, and $x_i = -$ otherwise. The *neighborhood clause* $nh_P(u)$ is obtained by replacing each literal in u by the corresponding neighborhood literal.

It is easy to see that the set of all possible neighborhood clauses (up to permutation of its literals) in the language of $\Gamma^{\mathrm{f}}(a, P, r)$ is a context-free string language because each derivation $z^{\mathrm{f}} \Rightarrow_{r_1} g_1 \Rightarrow_{r_2} \cdots \Rightarrow_{r_k} g_k$ in $\Gamma^{\mathrm{f}}(a, P, r)$ can be transformed into a derivation $nh_P(z^{\mathrm{f}}) \Rightarrow_{p_1} nh_P(g_1) \Rightarrow_{p_2} \cdots \Rightarrow_{p_k} nh_P(g_k)$ of neighborhood clauses. The finite set of all neighborhood literals with a terminal (nonterminal) label forms the set of terminal (nonterminal) symbols. The start symbol is $nh_P(z^{\mathrm{f}})$. The resulting string grammar is called *neighborhood grammar*.

Example 7. We continue Example 6. The neighborhood grammar of $\Gamma^{\mathrm{f}}(a, P, p_2)$ has the start symbol $S'(x)$ and the following rules:

$$
\begin{array}{llll}
S'(x) ::= edge(x,-)\ T(x)\ T(-)\ T^{\mathrm{f}}(x) & & S^{\mathrm{f}}() & ::= \varepsilon \\
T(x) ::= edge(x,-)\ T(x)\ T(-) \mid \varepsilon & & T^{\mathrm{f}}(x) & ::= S^{\mathrm{f}}() \mid T(-)\ T^{\mathrm{f}}(x) \mid T^{\mathrm{f}}(-) \\
T(-) ::= edge(-,-)\ T(-)\ T(-) \mid \varepsilon & & T^{\mathrm{f}}(-) & ::= S^{\mathrm{f}}() \mid T(-)\ T^{\mathrm{f}}(-) \mid T^{\mathrm{f}}(-)
\end{array}
$$

It is well-known that Parikh images of context-free languages are semilinear [15]. Let us briefly recapitulate the necessary notions: A *Parikh mapping* $\psi\colon \Sigma^* \to \mathbb{N}^n$ for an ordered vocabulary $T = \{a_1, \ldots, a_n\} \subseteq \Sigma$ is defined by $\psi(w) = (\#_{a_1}(w), \ldots, \#_{a_n}(w))$, where $\#_{a_i}(w)$ denotes the number of occurrences of a_i in w. The *Parikh image* of a string $w \in \Sigma^*$ is the vector $\psi(w)$, and the Parikh image of a language $\mathcal{L} \subseteq \Sigma^*$ is the set of Parikh images of its elements: $\psi(\mathcal{L}) = \{\psi(w) \mid w \in \mathcal{L}\}$. A set $M \subseteq \mathbb{N}^n$ is *linear* if there are $k+1$ vectors $x_0, \ldots, x_k \in \mathbb{N}^n$ such that $M = \{x_0 + \sum_{i=1}^{k} c_i x_i \mid c_1, \ldots, c_k \in \mathbb{N}\}$.[3] We also write $M = x_0 + \{x_1, \ldots, x_k\}^*$ and call x_0 the *tip* and $\{x_1, \ldots, x_k\}$ the *span* of M. $S \subseteq \mathbb{N}^n$ is *semilinear* if it is the union of a finite number of linear sets. Parikh's theorem [15] states that $\psi(\mathcal{L})$ is semilinear for every context-free language $\mathcal{L}$.

We now view the set of terminal neighborhood literals as an ordered vocabulary and represent neighborhood clauses by their Parikh image, called the *neighborhood vector*. The set of all rest graphs, when a rule shall be chosen for a given set of profile nodes, therefore, has a semilinear set of neighborhood vectors.

Example 8. The Parikh image of the neighborhood grammar in Example 7 is the semilinear set $(1,0) + \{(1,0),(0,1)\}^*$ for the vocabulary $\{edge(x,-), edge(-,-)\}$. This means that the rest graph, when rule p_2 shall be chosen by parsing procedure $T(x)$, must contain at least one edge leaving x.

There are algorithms that, given a context-free Chomsky grammar G, create the Parikh image of $\mathcal{L}(G)$ as its semilinear set, but our experiments have shown that they are far too inefficient. Instead, we employ the following procedure that determines a useful approximation of such semilinear sets and a finite description. Details of this procedure are omitted due to space restrictions:

[3] On $\mathbb{N}^k$, sums and scalar products are defined component-wise as usual.

The neighborhood grammar is represented by an *analysis graph* that has all nonterminal and terminal symbols as well as rules as nodes; a rule $A::= x_1 \dots x_k$ has an incoming edge from A and outgoing edges to each x_i. Each node representing a terminal or nonterminal symbol can be associated with the Parikh image of the language that can be generated from the corresponding symbol. The analysis graph defines a system of equations, and the Parikh images form the least fixed-point of this system of equations. It is computed by determining the strongly connected components of the analysis graph, contracting each strongly connected component to a single node, and evaluating the obtained DAG in a bottom-up fashion. Nevertheless, we compute only approximations of the Parikh images: Instead of computing general linear sets, which have arbitrary vectors in their span, we restrict span vectors to be 0 at every position but one, where it is 1. These sets are called *simple* in the following. Approximating linear sets by simple ones considerably simplifies computations. Nevertheless, the approximation is precise in the sense that the computed simple semilinear sets and the exact sets actually have the same tips in their linear sets.

Each parsing procedure defines a handle clause a and profile nodes P by its input parameters. We can now compute, for each parsing procedure and each rule alternative, a simple semilinear set of neighborhood vectors, which contains the neighborhood vectors of all possible rest graphs as a subset. These simple semilinear sets actually define the conditions under which the corresponding rule alternative is chosen: As soon as a parsing procedure is called, one computes the neighborhood vector of the current rest graph. This can be easily done in linear time. The parsing procedure then chooses the rule alternative whose simple semilinear set contains this vector. In Examples 4 and 5, we have actually written down graph patterns in the corresponding **if**-clauses, but this is just an optimization that allows a constant-time check.

This approach makes checking Conditions 1 and 2 for PTD parsability easy: $\mathrm{Sel}(a, P, r)$ is just the set of rest graphs whose neighborhood vectors are members of the corresponding simple semilinear set. Therefore, Conditions 1 is satisfied by construction, and Conditions 2 is easy to check.

A similar approach can be used in task *Neighbor-determined start nodes*. Task *Free edge choice* also creates analysis graphs, and checks whether edges that must be matched by the corresponding parsing procedure occur as competing nodes in the analysis graph. Edges can be freely chosen if there are no competing nodes.

The table below summarizes test results of the PTD analysis of 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), and number of rules (R). Column "Profiles" shows the maximal number of profiles of nonterminals. Column "PTD" indicates whether the respective grammar is PTD parsable. In all cases the parsers actually run in linear time. The columns under "Analysis" report on the time in milliseconds that the tasks *Neighbor-determined start nodes* (SN), *Neighbor-determined rule choice* (RC), and *Free edge choice* (FC) took on a MacBook Air (2 GHz Intel Core i7, Java 1.8.0). Of course, as mentioned in the introduction, many HR languages are

not PTD parsable. In fact, this includes polynomial time parsable languages such as structured flowcharts and series-parallel graphs [8]. They require to inspect the neighborhood in unbounded depth in order to choose between rules.

	Grammar			Pro-		Analysis [ms]		
Example	A	N	R	files	PTD	SN	RC	FC
Trees (Example 1)	1	2	3	1	yes	96	19	11
$a^n b^n c^n$ (Example 2)	4	2	3	1	yes	133	25	22
Palindromes (Theorem 2)	2	2	5	1	yes	129	23	14
Arithmetic expression	2	6	9	2	yes	351	90	52
Nassi-Shneiderman diagrams [14]	4	4	7	3	yes	440	80	85
Series-parallel graphs	2	2	4	2	no	132	34	24
Structured flowcharts	2	4	7	2	no	326	60	50

6 Conclusions

We have introduced predictive top-down parsing for HR grammars, in analogy to $SLL(1)$ string parsers, and shown that these parsers are of quadratic complexity. The analysis of HR grammars for PTD parsabilty has been implemented, and evaluated with several examples, including the grammars presented in this paper.

Related work on parsing includes precedence graph grammars based on node replacement [6,10]. 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 [13] 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 [14] work for practical languages, although their worst-case complexity is exponential. It is unclear whether more general grammars, like layered graph grammars [16] can be used in practice, even with the improved parser proposed in [7].

Future work has already started: the analysis of PTD parsability can actually check the contextual HR grammars studied in [5], where the left-hand side of a rule may contain isolated ("contextual") nodes that can be used on the right-hand side. Contextual HR grammars allow to generate languages such as the set of all connected graphs and all acyclic graphs, which cannot be defined by HR grammars and are more useful for practical modeling of graph languages. See [5] for further examples, which all turn out to be PTD-parsable. We also conjecture that it is possible to handle contextual HR rules equipped with positive or negative application conditions involving path expressions, as discussed in [9], without loosing too much of the efficiency of PTD parsing. Finally, we hope that deterministic bottom-up parsers of contextual HR grammars (in analogy to $SLR(1)$ string parsing) can be developed using concepts similar to those presented in this paper.

References

1. Aalbersberg, I., Ehrenfeucht, A., Rozenberg, G.: On the membership problem for regular DNLC grammars. Discrete Appl. Math. **13**, 79–85 (1986)
2. Chiang, D., Andreas, J., Bauer, D., Hermann, K. M., Jones, B., Knight, K.: Parsing graphs with hyperedge replacement grammars. In: Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics, Sofia, Bulgaria. Long Papers, vol. 1, pp. 924–932, August 2013
3. Costagliola, G., De Lucia, A., Orefice, S., Tortora, G.: A parsing methodology for the implementation of visual systems. IEEE Trans. Softw. Eng. **23**(12), 777–799 (1997)
4. Drewes, F.: Recognising k-connected hypergraphs in cubic time. Theor. Comput. Sci. **109**, 83–122 (1993)
5. Drewes, F., Hoffmann, B.: Contextual hyperedge replacement. Acta Informatica, 31 (2015, accepted for publication). doi:10.1007/s00236-015-0223-4
6. Franck, R.: A class of linearly parsable graph grammars. Acta Informatica **10**(2), 175–201 (1978)
7. Fürst, L., Mernik, M., Mahnič, V.: Improving the graph grammar parser of Rekers and Schürr. IET Softw. **5**(2), 246–261 (2011)
8. Habel, A. (ed.): Hyperedge Replacement: Grammars and Languages. LNCS, vol. 643. Springer, Heidelberg (1992)
9. Hoffmann, B., Minas, M.: Defining models - meta models versus graph grammars. In: Proceedings of the 6th Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2010), Electronic Communications of the EASST, 29, Paphos, Cyprus (2010)
10. Kaul, M.: Practical applications of precedence graph grammars. In: Ehrig, H., Nagl, M., Rozenberg, G., Rosenfeld, A. (eds.) Graph-Grammars and Their Application to Computer Science. LNCS, vol. 291, pp. 326–342. Springer, Heidelberg (1986)
11. Lautemann, C.: The complexity of graph languages generated by hyperedge replacement. Acta Informatica **27**, 399–421 (1990)
12. Lewis II, P.M., Stearns, R.E.: Syntax-directed transduction. JACM **15**(3), 465–488 (1968)
13. Ludwigs, H.J.: A LR-like analyzer algorithm for graphs. In: Wilhelm, R. (ed.) GI - 10. Jahrestagung: Saarbrücken, 30. September - 2. Oktober 1980. Informatik-Fachberichte, vol. 33, pp. 321–335. Springer, Heidelberg (1980)
14. Minas, M.: Diagram editing with hypergraph parser support. In: Proceedings of 1997 IEEE Symposium on Visual Languages (VL 1997), Capri, Italy, pp. 226–233 (1997)
15. Parikh, R.J.: On context-free languages. JACM **13**(4), 570–581 (1966)
16. Rekers, J., Schürr, A.: Defining and parsing visual languages with layered graph grammars. J. Vis. Lang. Comput. **8**(1), 27–55 (1997)
17. Vogler, W.: Recognizing edge replacement graph languages in cubic time. In: Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) Graph Grammars and Their Application to Computer Science. LNCS, vol. 532, pp. 676–687. Springer, Heidelberg (1991)