# Generating Efficient Predictive Shift-Reduce Parsers for Hyperedge Replacement Grammars

Berthold Hoffmann[1] and Mark Minas[2] (✉)

[1] Universität Bremen, Germany
hof@informatik.uni-bremen.de
[2] Universität der Bundeswehr München, Germany
mark.minas@unibw.de

**Abstract.** Predictive shift-reduce (PSR) parsing for a subclass of hyperedge replacement graph grammars has recently been devised by Frank Drewes and the authors [6]. This paper describes in detail how efficient PSR parsers are generated with the *Grappa* parser generator implemented by Mark Minas. Measurements confirm that the generated parsers run in linear time.

**Keywords:** hyperedge replacement grammar, graph parsing, parser generator

## 1 Introduction

Since the processing of diagram languages with computers becomes more and more common, the question whether a diagram adheres to the rules of such a language gets more and more important. If the rules of a language go beyond validity wrt. a metamodel, the notion of grammars gains relevance. Here, graph grammars are a natural candidate, in particular if they are context-free, like those defined by hyperedge replacement (HR) [10]. Unfortunately, general HR parsers, like the adaptation of the Cocke-Younger-Kasami (CYK) parser to graphs [13], do not scale to graphs of the size used in modern applications, e.g., in model transformation. So it is worthwhile to identify subclasses of HR grammars that have efficient parsers. After devising *predictive top down parsing* (PTD) [4], Frank Drewes and the authors have recently proposed *predictive shift-reduce parsing* (PSR) [6], its bottom-up counterpart, which lifts *SLR*(1) string parsing to graphs. Now Mark Minas has completed his implementation of *Grappa*, a generator for PTD and PSR parsers.[3]

In order to keep the paper self-contained, we start with a brief account of HR grammars in Section 2, introduce PSR parsing in Section 3, and sketch conflict analysis in Section 4. (More details can be found in [6].) Then we describe the implementation of efficient PSR parsers generated by *Grappa* in Section 5. Evaluation of their efficiency in Section 6, also in comparison to PTD and CYK parsers, confirm that they run in linear time. Finally we point out some future work, in Section 7.

[3] *Grappa* is available at www.unibw.de/inf2/grappa.

## 2 Hyperedge Replacement Grammars

We use the wellknown relation between graphs and logic [1] to define graphs and hyperedge replacement grammars.

**Definition 1 (Graph).** Let $\Sigma$ be a vocabulary of *symbols* that comes with an *arity function* $arity\colon \Sigma \to \mathbb{N}$, and let $X$ be an infinite set of *variables*. We assume that $\Sigma$ is the disjoint union of *nonterminals* $N$ and *terminals* $T$.

A *literal* $e = \ell(x_1, \ldots, x_k)$ consists of a symbol $\ell \in \Sigma$ and $k = arity(\ell)$ variables $x_1, \ldots, x_k$ from $X$. A *graph* is a sequence $G = e_1 \ldots e_n$ of literals. With $\Sigma(G)$ and $X(G)$ we denote the *symbols* and *variables* occurring in $G$, respectively.

$X(G)$ represents the *nodes* of $G$, and a literal $e = \ell(x_1, \ldots, x_k)$ represents a *hyperedge* (*edge*, for short) that has the *label* $\ell$ and is attached to the nodes $x_1, \ldots, x_k$. If necessary, isolated nodes can be represented by a literal $\nu(x)$, where $\nu$ is a *ficticious node symbol* with $arity(\nu) = 1$. Multiple occurrences of literals represent parallel edges.

**Definition 2 (HR Grammar).** A pair $r = (L, R)$ of graphs is a *hyperedge replacement rule* (*rule* for short) if its *left-hand side* $L$ consists of a single nonterminal literal and if its *right-hand side* $R$ satisfies $X(L) \subseteq X(R)$; we usually denote a rule as $r = L \to R$.

An injective function $\varrho\colon X \to X$ is a *renaming*; $G^\varrho$ denotes the graph obtained by replacing all variables in $G$ according to $\varrho$.

Consider a graph $G$ and a rule $r$ as above. A renaming $\mu\colon X \to X$ *matches* $r$ to the $i$th literal of $G$ if $L^\mu = e_i$ for some $1 \leqslant i \leqslant n$ and $X(G) \cap X(R^\mu) \subseteq X(L^\mu)$. A match $\mu$ of $r$ *rewrites* $G$ to the graph $H = e_1 \ldots e_{i-1} R^\mu e_{i+1} \ldots e_n$. This is denoted as $G \Rightarrow_{r,\mu} H$, or just as $G \Rightarrow_r H$. We write $G \Rightarrow_\mathcal{R} H$ if $G \Rightarrow_r H$ for some rule $r$ taken from a finite set $\mathcal{R}$ of rules, and denote the transitive closure of this relation by $\Rightarrow^*_\mathcal{R}$, as usual.

A *hyperedge replacement grammar* $\Gamma = (\Sigma, T, \mathcal{R}, Z)$ (*HR grammar* for short) consists of a finite set $\mathcal{R}$ of rules over $\Sigma$, and of a start graph $Z$ with $\Sigma(Z) = S \in N$ and $X(Z) = \varnothing$. $\Gamma$ generates the language $\mathcal{L}(\Gamma) = \{G \mid Z \Rightarrow^*_\mathcal{R} G, \Sigma(G) \subseteq T\}$.

*Example 1 (Nested Triangles).* Consider nonterminals $S$ and $\blacktriangle$ and the terminal $\triangle$. We use $\ell^{x_1 \cdots x_k}$ as a a shorthand for literals $\ell(x_1, \ldots, x_k)$. (Here $\varepsilon$ denotes the empty variable sequence.) Then the rules

$$S^\varepsilon \to \blacktriangle^{xyz} \qquad \blacktriangle^{xyz} \to \triangle^{xuv}\, \triangle^{uyw}\, \triangle^{vwz}\, \blacktriangle^{uwv} \qquad \blacktriangle^{xyz} \to \triangle^{xyz}$$

(which are numbered 1, 2, and 3) generate a nested triangle:

$$\begin{aligned} S^\varepsilon \underset{1}{\Rightarrow} \blacktriangle^{123} \underset{2}{\Rightarrow} \triangle^{145}\, \triangle^{426}\, \triangle^{563}\, \blacktriangle^{465} &\underset{2}{\Rightarrow} \triangle^{145}\, \triangle^{426}\, \triangle^{563}\, \triangle^{478}\, \triangle^{769}\, \triangle^{895}\, \blacktriangle^{798} \\ &\underset{3}{\Rightarrow} \triangle^{145}\, \triangle^{426}\, \triangle^{563}\, \triangle^{478}\, \triangle^{769}\, \triangle^{895}\, \triangle^{798} \end{aligned}$$

(In Fig. 1, the graphs of this derivation are drawn as diagrams.)

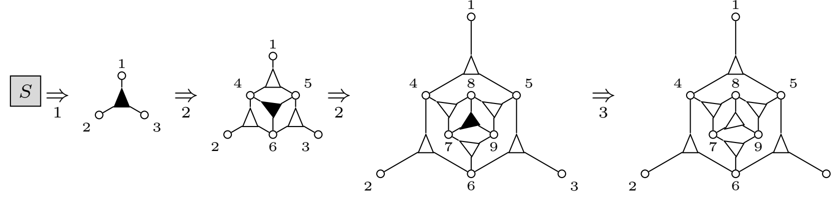



**Fig. 1.** Diagrams of a derivation of a nested triangles. Circles represent nodes, boxes and triangles represent edges of triangle graphs, which are connected to their attached nodes by lines; these lines are ordered clockwise around the edge, starting at the sharper edge of the triangle.

## 3 Predictive Shift-Reduce Parsing for HR Grammars

A parser attempts to construct a derivation for a given input according to some grammar. In our case, graphs shall be parsed according to a HR grammar. A bottom-up parser constructs the derivation by an operation called *reduction*: the right-hand side of a rule is matched in the input graph, and replaced by its left-hand side. A bottom-up parser for the nested triangles in Example 1 may reduce every triangle $\vartriangle^{abc}$ according to rule 3. However, only a single reduction, of the "central" triangle, will lead to a successful chain of reductions reaching the start graph $S^{\varepsilon}$. Cocke-Younger-Kasami parsers use this idea (after transforming grammars into Chomsky normal form). Even if this works for small graphs with up to hundred edges [13], it does not scale to bigger graphs. See our evaluation in Sect. 6 below.

PSR parsers borrow an idea of context-free bottom-up string parsers: they consume edges of a graph in exactly the order in which they would be constructed by a derivation. An operation, called *shift*, puts terminal edges onto a stack, to be considered for reduction later. A PSR parser will reduce rule 3 only once, after all other edges have been shifted; further reductions of rule 2, and finally of rule 1, may then lead to a successful parse.

A predictive bottom-up shift-reduce parser uses a *characteristic finite automaton* (CFA) to control its actions. We describe its construction at hand of the running example. The states of the CFA are defined as sets of *items*, which are rules where a dot indicates how far the right-hand side has been shifted onto the stack. Consider the item $\blacktriangle^{xyz} \rightarrow \vartriangle^{xuv} \vartriangle^{uyw} \vartriangle^{vwz} \centerdot \blacktriangle^{uwv}$ of rule 2: Here the parser has shifted all terminals, but not the nonterminal. This item will constitute a *kernel item* of some state of the CFA, say $q_3$. All variables $x, y, z, u, v, w$ of the rule are known in this situation. So we consider them as *parameters* of the state, and denote it as $q_3(x, y, z, u, v, w)$. Before the missing $\blacktriangle^{uwv}$ can be shifted, the parser must recursively parse rule 2 or 3. So the items $\blacktriangle^{uwv} \rightarrow \centerdot \vartriangle^{urs} \vartriangle^{rwt} \vartriangle^{stv} \blacktriangle^{rts}$ and $\blacktriangle^{uwv} \rightarrow \centerdot \vartriangle^{uwv}$ are added to $q_3$ as *closure items*. The dots at the start of these items indicate that nothing of these rules has been shifted in this state. We have to rename variables in order to avoid name clashes with the kernel item.

Like every CFA state, $q_3$ has transitions under every symbol appearing after the dot in some of its items. A transition under $\triangle^{urs}$ leads from $q_3$ to a state with the kernel item $\blacktriangle^{uwv} \to \triangle^{urs} \,.\, \triangle^{rwt}\, \triangle^{stv}\, \blacktriangle^{rts}$. No closure items arise in this state since the dot is in front of a terminal. This state would be denoted as $q'_1(u,w,v,r,s)$, but if there is already a state $q_1(x,y,z,u,v)$ that is equal to $q'_1$ up to variable names, we redirect the transition to this state and write a "call" $q_1(u,w,v,r,s)$ on the transition to specify how parameters should be passed along. Another transition, under $\triangle^{uwv}$, leads to a state with kernel item $\blacktriangle^{uwv} \to \triangle^{uwv} \,.$, say $q_5^{uwv}$. This transition matches a terminal where all nodes are known; so it differs from that under $\triangle^{urs}$ that has to match two nodes $r$ and $s$ to hitherto unconsumed nodes. Finally, a transition under $\blacktriangle^{uwv}$ leads from $q_3$ to a state, say $q_4(x,y,z,u,v,w)$ with the kernel item $\blacktriangle^{xyz} \to \triangle^{xuv}\, \triangle^{uyw}\, \triangle^{vwz}\, \blacktriangle^{uwv} \,.$.

A special case arises in the start state $q_0$. In order to work without backtracking, some nodes of the start rule must be uniquely determined in the input graph before parsing starts. In our example, all nodes $x, y, z$ match the unique nodes $a, b, c$ that are attached to just one edge, with their first, second, and third attachment, respectively. If the input graph does not have exactly three nodes like that, it cannot be a nested triangle, and parsing fails immediately. Otherwise the start state is called with $q_0(a,b,c)$. Unique start nodes can be determined by a procedure devised in [5, Sect. 4], which computes the possible incidences of all nodes created by a grammar.

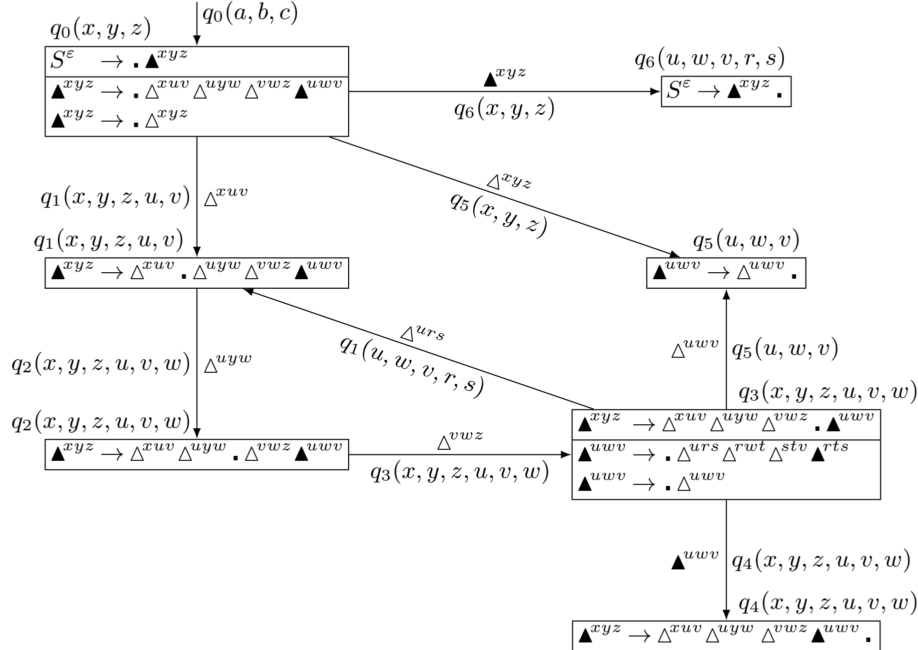



**Fig. 2.** The characteristic finite automaton for nested triangles

*Example 2 (A CFA for Nested Triangles).* In Fig. 2 we show the transition diagram of the CFA of Example 1. The states $q_1, q_3, q_4, q_5$ are as discussed above.

A PSR parser pushes concrete states and transitions of its CFA onto a stack while it performs transitions. In the states and transitions stored on the stack, variables in the abstract states and transitions of the CFA are replaced by concrete nodes matching them in the input graph.

The topmost stack entry, a state, determines the next action of the parser. This may be a shift under a terminal literal, or a reduction of some rule. (Transitions under nonterminal literals are handled as final part of a reduction.) The actions for some topmost state $q$ are as follows:

***Shift:*** If $q$ calls for a terminal transition under some literal $\ell(x_1, \ldots, x_k)$, lookup the concrete nodes of $q$ matching some of these variables, and match the edge $\ell$ with an edge $e = \ell(v_1, \ldots, v_k)$ in the host graph. Push $e$ onto the stack, remove it from the input, and push the target state, replacing variables with the concrete nodes determined by $q$ and $e$.

***Reduce:*** If $q$ calls for a reduction of some rule $r$, pop all literals of the right-hand side of $r$ from the stack, with their corresponding states. The state that is on top has a transition under the left-hand side of $r$; push the left-hand side and its target state, replacing the variables with the nodes determined by the popped states. If the rule $r$ is the start rule, and the input graph is empty, *accept* the input as a graph of the language.

Note that the parser has to choose the next action in states that allow for different shifts and/or reductions; in our example, $q_0$ and $q_3$ allow two shifts, see Fig. 2. The parser predicts the next step by inspecting the unconsumed edges. We will discuss in the next section how the conditions to be used for inspection are computed. In the example, the shifts according to rule 3 (in states $q_0$ and $q_3$) have to be chosen if and only if the input is empty.

Even if a particular shift transition has been chosen, a PSR parser may still have to choose between different edges matching the literal. (Such a situation does not occur in our example (but with the trees in [6, Sect. 4]). In such a case, the PSR parser generator has to make sure that the *free edge choice* property holds, i.e., that any of the matching edges can be chosen without changing the result of the parser.

*Example 3 (A PSR Parse for Nested Triangles).* A parse of the graph derived in Example 1 is shown in Fig. 3. We write the parameters of states as exponents, just as for literals.

## 4 Conflict Analysis

A CFA can be constructed for every HR grammar; the general procedure works essentially as described above. In this paper, we focus on the implementation of parsers for HR grammars that are PSR-parsable. Criteria for an HR grammar to

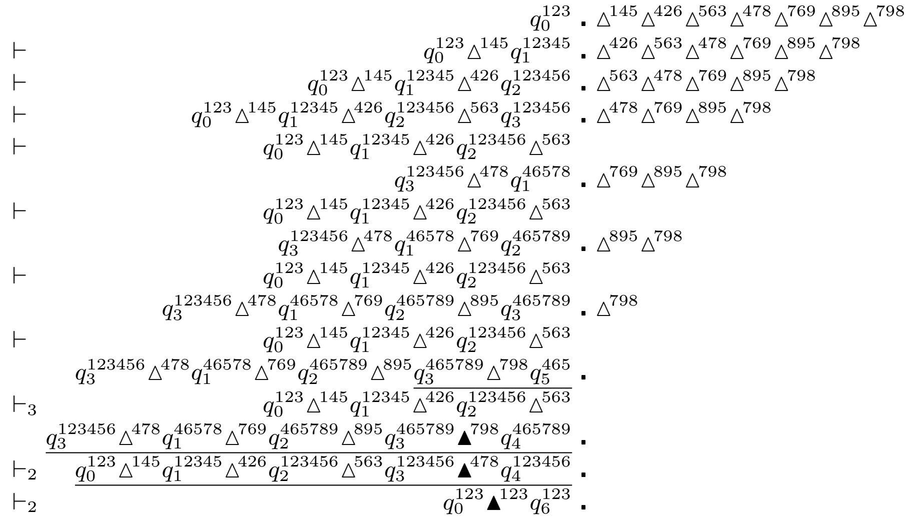


**Fig. 3.** A PSR parse for the triangle derived in Example 1

be PSR-parsable have been discussed in [6]. In particular, such a grammar must be *conflict-free*. In the following, we roughly recall this concept since it is needed for the implementation of efficient PSR parsers, which is described in Sect. 5.

A graph parser must choose the next edge to be consumed from a set of appropriate unconsumed edges. We define a *conflict* as a situation where an unconsumed edge is appropriate for one action, but could be consumed also if another action was chosen. Obviously, the parser can always predict the correct action if the grammar is free of conflicts.

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 $I = \langle L \to \bar{R} \centerdot R \mid P \rangle$ consists of a rule $L \to \bar{R}R \in \mathcal{R}$ with a dot indicating a position in the right-hand side, and of the set $P$ of parameters, i.e., those nodes in the item which do already have matching nodes in the host graph. These host nodes are not yet known when we construct 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 $G$ in $\mathcal{L}(\Gamma)$, generated by the derivation $S = G_1 \Rightarrow \cdots \Rightarrow G_n = G$. We assume that the ordering of edge literals is preserved in each derivation step. We then select any mapping of nodes in $G$ 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 $\Gamma$ in the same way as described in [5, 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 $\Gamma$ to inspect conflicts. This is shown in the following.

Consider an item $I = \langle L \to \bar{R} \centerdot R \mid P \rangle$. 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 $\mathit{First}_P(R) := \mathit{abstr}_P(e)$ if $I$ is a shift item, i.e., $R$ starts with a terminal literal $e$. If $I$, however, causes a reduction, i.e., $R = \varepsilon$, 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 $\Gamma(P)$. We refer to [6] for a discussion of the general case. Here, we discuss the concept at hand of our running example.

As an example, consider state $q_3(x, y, z, u, v, w)$ in Fig. 2 with its items $I_i = \langle L_i \to \bar{R}_i \centerdot R_i \mid P_i \rangle$, $i = 1, 2, 3$, with $P_1 = \{x, y, z, u, v, w\}$ and $P_2 = P_3 = \{u, v, w\}$. For the second item, one can compute

$$\mathit{First}_{P_2}(R_2) = \mathit{abstr}_{P_2}(\triangle^{urs}) = \triangle^{u--},$$

i.e., the shifted edge in this step (called *shift step 1* in the following) must be a triangle edge being attached to the host node assigned to $u$ with its first arm, and nodes that have not yet been consumed by the parser with its other arms.

For the third item, one can compute

$$\mathit{First}_{P_3}(R_3) = \mathit{abstr}_{P_3}(\triangle^{uwv}) = \triangle^{uwv},$$

i.e., the shifted edge in this step (called *shift step 2* in the following) must be a $\triangle$ edge being attached to host nodes that are assigned to $u$, $w$, and $v$, respectively.

The parser needs a criterion for deciding the correct step when it has reached $q_3(x, y, z, u, v, w)$. It is clear that shift step 1 must be taken when the host graph contains an unconsumed edge matching $\triangle^{u--}$, but not $\triangle^{uwv}$, and shift step 2 if it contains an unconsumed edge matching $\triangle^{uwv}$, but not $\triangle^{u--}$. However, the parser would be unable to decide the next step if the host graph contained two unconsumed edges matching $\triangle^{u--}$ and $\triangle^{uwv}$, respectively. Conflict analysis makes sure that this situation, called *shift-shift conflict*, cannot occur here. This is outlined in the following.

Let us assume that this conflicting situation occurs, i.e., $\triangle^{uwv}$ may follow later when shift step 1 is taken, or $\triangle^{u--}$ may follow later when shift step 2 is taken. Conflict analysis, therefore, must compute from $\Gamma(P)$ the (finite) set of all abstract edges that may follow when either shift step is taken. Let us denote this set as $\mathit{Follow}^*_P(I_i)$, $i = 2, 3$. Its computation is straightforward and well-known

from string grammars. In our example

$$Follow^*_{P_2}(I_2) = \{\triangle^{u--}, \triangle^{-w-}, \triangle^{--v}, \triangle^{---}\}$$
$$Follow^*_{P_3}(I_3) = \{\triangle^{uwv}\}$$

Of course, $First_{P_i}(R_i) \in Follow^*_{P_i}(I_i)$ for $i = 2, 3$. As one can see, $\triangle^{u--} \notin Follow^*_{P_3}(I_3)$ and $\triangle^{uwv} \notin Follow^*_{P_2}(I_2)$, i.e., such a shift-shift conflict cannot occur, and the parser can decide in state $q_3$ which step shall be taken, by checking whether there exists a yet unconsumed edge matching $\triangle^{u--}$ or $\triangle^{uwv}$.

Similar arguments apply when the parser has to decide between a shift and a reduce step and between two reduce steps, potentially causing *shift-reduce* or *reduce-reduce conflicts* as discussed in [6]. But these situations do not occur in the CFA of our running example.

## 5 Efficient Implementation of PSR Parsers

We shall now describe how PSR parsers can be implemented efficiently so that their runtime is linear in the size of the input graph. We shall first describe the implementation at hand of our running example, and then the general procedure.

In the following, we assume that nodes and edges are represented by separate data structures. Each edge keeps track of its label and all attached nodes. We will discuss later what information must be stored at node objects to make parsing efficient.

The implementation of the parser outlined in Sect. 3 is rather straightforward: The parsing stack described in Sect. 3 holds (terminal) host edges as well as nonterminal edges produced by reduce steps, and CFA states with their parameters bound to host nodes that have already been consumed by the parser. In Example 3, we have represented such a state $q_i(x_1, \ldots, x_k)$ together with its binding match $\mu\colon \{x_1, \ldots, x_k\} \to X(G)$ by $q_i^{\mu(x_1)\ldots\mu(x_k)}$. In the implementation, we represent each state just by its number $i$, and its binding by an array *params* of host nodes such that $params[j] = \mu(x_j)$ for each $j = 1, \ldots, k$. And, instead of just a single stack, we shall use three stacks: a *stateStack* of state numbers, a *paramStack* of node arrays representing binding matches, and an *edgeStack* of (terminal) host edges and nonterminal edges produced by reduce steps. The elements stored in *stateStack* and in *paramStack* correspond to each other; each corresponding pair represents a state $q_i^{\mu(x_1)\ldots\mu(x_k)}$ with a binding match $\mu$. The parser is then implemented as the procedure *parse* shown in Fig. 4. The start nodes, which have been determined before parsing begins (see Sect. 3), represented by an array *startNodes*, are passed as a parameter. The parser initializes its stacks with the start state, which we assume to have number 0, together with the binding match defined by the start nodes. The actual parsing actions are implemented in procedures $action_i$, one for each CFA state $q_i(x_1, \ldots, x_k)$. It is their task to operate on the stacks and to terminate the seemingly infinite loop.

Fig. 5 shows the *action*-procedure for the accept state $q_6(x, y, z)$ of the nested triangle CFA (Fig. 2); the parser terminates with success iff all nodes and edges

```
procedure parse (startNodes: array of Node)
    push 0 on stateStack;
    push startNodes on paramStack;
    while true do
        i ← top of stateStack;
        call action_i
    end
end
```

**Fig. 4.** The parsing procedure

```
procedure action_6
    if all edges and nodes
        have been consumed
    then
        stop with success
    end;
    stop with error
end
```

**Fig. 5.** Action for state $q_6$

of the host graph have been consumed when this state is reached. The action procedures for states $q_3(x, y, z, u, v, w)$ and $q_5(x, y, z)$ are shown in Fig. 6. Procedure *action*$_3$ must check which of the two shift transitions leaving $q_3$ must be taken. The third transition leaving $q_3$, labeled with nonterminal edge $\blacktriangle^{uwv}$, is implemented in procedure *goto*$_3$ and described later. Procedure *action*$_3$ first tries to find a yet unconsumed edge of the host graph that corresponds to $\triangle^{urs}$, where $u$ is a parameter node, i.e., bound to a host node that is stored at position 4 of the current parameter array, whereas $r$ and $s$ must correspond to host nodes that have not yet been consumed. Such a host edge is looked for in lines 6–7. Grammar analysis shows that a host graph in the language of nested triangles cannot contain more than one edge like that. (We will discuss later how the parser generator can do so.) The parser, therefore, looks whether it finds any such edge and, if successful, stores it in $e$. The parser takes the corresponding shift transition to $q_1(u, w, v, r, s)$ if such an edge exists and if its other two connected nodes have not been consumed before (line 8). The shift step marks the identified edge $e$ and nodes $\beta$ and $\gamma$ as consumed and computes the parameter array of the next state $q_1(u, w, v, r, s)$. (Details on the corresponding data structures are discussed later.) The host nodes corresponding to $u$, $w$, and $v$ are already known from the current parameter array, but $r$ and $s$ (at positions 4 and 5 of the array) are the nodes $\beta$ and $\gamma$ visited by edge $e$. The procedure then returns, and the parsing loop can continue with the next iteration.

Procedure *action*$_3$ checks the other shift transition if the test in line 8 fails. Lines 19–28 are similar to lines 8–17. Note that the parser need not look for another edge than the one found in lines 6–7 once such an edge has been found. Finally, the parser must stop with an error if the test in line 19 also fails, because a valid host graph must contain an edge satisfying one of the two conditions.

Procedure *action*$_5$ shows the implementation of the reduce step to be taken in state $q_5(x, y, z)$. Lines 54–55 create the nonterminal edge corresponding to $\blacktriangle^{xyz}$ produced by the reduce step (see Fig. 2). Lines 56–57 pop the elements corresponding to the right-hand side of the rule from the stacks, i.e., the CFA returns to state $q_i$ (line 58) where a transition labelled with the newly created edge $e$ must be taken (line 59). Such a *goto*-procedure for state $q_3(x, y, z, u, v, w)$ is shown in lines 31–48. Lines 37–38 check whether the parameter edge matches the one defining the transition. This is actually not necessary here where we

```
1  procedure action_3
2      params ← top of paramStack;
3      u ← params[4]; v ← params[5];
4      w ← params[6];
5      /* shift △^{urs} */
6      e ← any unconsumed edge △^{αβγ}
7              in the host graph with α = u;
8      if e exists and β, γ are unconsumed
9      then
10         mark e, β, and γ as consumed;
11         nextParams ←
12                 new array {u, w, v, β, γ};
13         push e on edgeStack;
14         push 1 on stateStack;
15         push nextParams on paramStack;
16         return
17     end;
18     /* shift △^{uwv} */
19     if e exists and β = w and γ = v
20     then
21         mark e as consumed;
22         nextParams ←
23                 new array {u, w, v};
24         push e on edgeStack;
25         push 5 on stateStack;
26         push nextParams on paramStack;
27         return
28     end;
29     stop with error
30 end

31 procedure goto_3(e : Edge)
32     params ← top of paramStack;
33     x ← params[1]; y ← params[2];
34     z ← params[3]; u ← params[4];
35     v ← params[5]; w ← params[6];
36     /* ▲^{uwv} */
37     if e has label ▲ and
38         visits nodes (u, w, v)
39     then
40         nextParams ←
41                 new array {x, y, z, u, v, w};
42         push e on edgeStack;
43         push 4 on stateStack;
44         push nextParams on paramStack;
45         return
46     end;
47     stop with error
48 end

49 procedure action_5
50     params ← top of paramStack;
51     x ← params[1]; y ← params[2];
52     z ← params[3];
53     /* reduce rule 2 */
54     e ← new edge with label ▲ and
55             visiting nodes (x, y, z);
56     pop 4 elements from edgeStack,
57             stateStack, and paramStack;
58     i ← top of stateStack;
59     call goto_i(e)
60 end
```

**Fig. 6.** Some procedures implementing the PSR parser for nested triangles.

have just a single transition with a nonterminal edge; but in general, there may be several leaving transitions with different nonterminal edges, and the *goto*-procedure must select the correct one. Lines 40–44 move the parser into the next state $q_4(x, y, z, u, v, w)$.

The other *action* and *goto* procedures are similar to the presented ones. Let us now consider the general case. Each state of a CFA provides a set of operations which can be *shift*, *reduce*, *accept*, or *goto* operations. *Shift* and *goto* operations correspond to transitions to other states; *shift* transitions are labelled by terminal edge literals and *goto* transitions by nonterminal edge literals. The latter are easily implemented by *goto* procedures similar to $goto_3$ in Fig. 6. Each *goto* procedure must check a fixed number of different cases, which can be performed in constant time. The *action* procedures are responsible for choosing among the

*shift*, *reduce*, and *accept* operations provided by the corresponding states. For each of these operations, the parser generator must identify a condition that controls when the parser shall select its operation to be executed next. In the example, procedure $action_3$ selects the shift over $\triangle^{wrs}$ iff the condition in line 8 is satisfied, and a shift over $\triangle^{uwv}$ if the condition in line 8 is not satisfied, but the one in line 19. Moreover, the parser must be able to efficiently check these conditions, i.e., in constant time.

The conditions can be easily derived from the conflict analysis described in the previous section. Conflict analysis determines, for each item of a state, a finite characterization of terminal edges of the host graph that must be consumed next (for *shift* steps) or that may be consumed in later steps (for *reduce* steps). There are no conflicts if an HR grammar is PSR, i.e., the parser can use these conditions to always correctly select the next operation.

Now, how do these conditions look like? Each one is an abstract literal characterizing (yet unconsumed) terminal edges of the host graph determining their label and some of their nodes, whereas the other attached nodes are yet unconsumed. A naïve procedure for searching for such an edge would be to iterate over all unconsumed edges and to select an edge that has attached nodes as specified. This would take linear time instead of the required constant time; proper preprocessing of the host graph prior to parsing is necessary. The parser generator knows about all abstract literals that are used in any condition. The idea is to preprocess the host graph so that each abstract literal corresponds to a data structure of all unconsumed edges that match the abstract literal, and to update the corresponding data structures whenever an edge is consumed. The parser, when searching for an unconsumed edge matching an abstract literal, then just has to look into the corresponding data structure. Hash tables are an appropriate data structures for this purpose. For each abstract literal $a = l(m_1, \ldots, m_k)$, we assign a hash table to label $l$. Prior to parsing, each edge $e$ matching this literal is added to this hash table. More specifically, all nodes attached to $e$ and being determined by the abstract literal define a tuple $Key_a(e)$ of nodes. $Key_a(e)$ is mapped to a list in the hash table; this list contains $e$ and (when the complete graph has been preprocessed) all edges that have the same key $Key_a(e)$. Searching for an edge being attached to some predefined nodes specified by an abstract literal then consists of just computing the corresponding key and looking up the mapped list in the appropriate hash table.

The speed of looking up this list is constant on average because the hash table is fixed after preprocessing; only the contents of the list are modified when edges are consumed. This can be done in constant time, too, when lists are implemented by doubly linked lists and each edge keeps track of the list nodes of all lists in which the edge is stored. Because the number of abstract literals is fixed for a grammar, all these data structures (hash table, lists, and keeping track of list nodes) require linear space in the size of the host graph if each hash table size is chosen proportional to the number of all edges. Moreover, setting up these data structures requires, on average, linear time in the size of the host graph.

However, looking up unconsumed edges matching certain host nodes specified by an abstract literal can be simplified in many cases (for instance in our running example of nested triangles) so that hash tables become obsolete in many cases, or altogether: Grammar analysis may reveal, for a state $q_i(x_1, \ldots, x_k)$ of the CFA, a terminal edge label $l \in T$, a parameter node $x_i$ and an "arm" $j$, $1 \leqslant j \leqslant \mathit{arity}(l)$, that a host graph cannot have more than one unconsumed edge with label $l$ and being attached to $\mu(x_i)$ with its $j$-th arm when the parser has reached $q_i^\mu$. If an abstract literal used for edge lookup refers to such a node, called *determining node* in the following, one can use just this node as a key in the hash table. However, this makes the hash table unnecessary: Instead of mapping a node to a list of attached edges, one can simply keep this list in the node data structure. Such a list is just a plain association list which must be maintained when consuming edges.

Grammar analysis can identify determining nodes by using the same techniques as for conflict analysis, which computes sets of abstract edges that may follow during parsing. If this computation shows that a parameter node is attached to a shift edge, but to no other edge with the same label and using the same arm later in the derivation process, one can conclude that such a parameter node is determining for the corresponding edge label and arm. Amazingly, experiments with many PSR grammars have shown that almost all of them can be processed without any hash table, i.e., run in linear time even in the worst case and not only on average.

Finally note that PSR parsers not only process syntactically correct graphs in linear time, but also erroneous graphs. This is so because each step of the parser still takes constant time (at least on average), and the number of steps linearly depends on the graph size.

## 6 Evaluation of Generated PSR Parsers

In order to demonstrate that PSR parsing is linear in the size of the host graph, we have conducted some experiments with some example HR grammars. For each grammar, we generated a PSR as well as a PTD parser using *Grappa*, and also a CYK parser using *DiaGen*.[4] We then measured parsing time for input graphs of different size for each of these parsers. The results can be found at www.unibw.de/inf2/grappa; here, we present the results for our running example of nested triangles and also for Nassi-Shneiderman diagrams [12].

Each triangle graph consists, for some positive integer $n$, of $3n$ nodes and $3n - 2$ edges. Fig. 7a shows the runtime of the PSR and PTD parsers when processing triangle graphs with varying value $n$. Runtime has been measured on a MacBook Pro 2013, 2,7 GHz Intel Core i7, Java 1.8.0, and is shown in milliseconds on the $y$-axis while $n$ is shown on the $x$-axis. Note the apparent linear behavior of the PSR parser and the, slightly slower, PTD parser. Fig. 7b shows the corresponding diagram for the CYK parser. Note that the runtime of

[4] Homepage: www.unibw.de/inf2/DiaGen

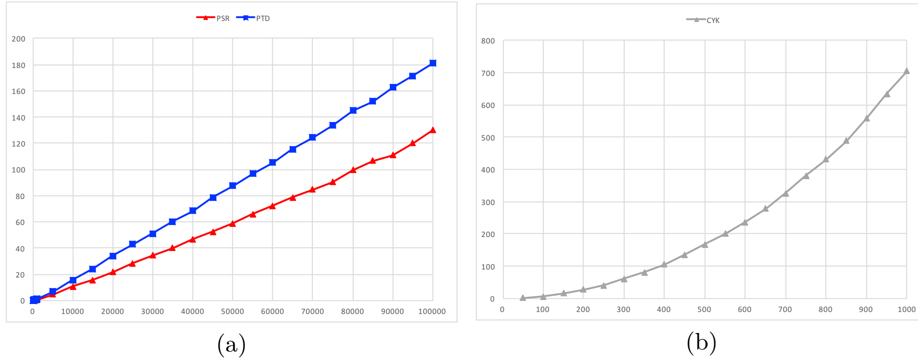



**Fig. 7.** Runtime (in milliseconds) of the PSR as well as the PTD parser (a) and the CYK parser (b) for nested triangles. Note that the scales in (a) and (b) differ.

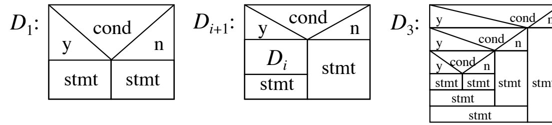



**Fig. 8.** Nassi-Shneiderman diagrams $D_i$, $i = 1, 2, 3, \ldots$.

the CYK parser is not linear in the size of the triangle graph. Note also that PTD parsing and, in particular, PSR parsing is, by several orders of magnitude, faster than CYK parsing. For instance, the CYK parser needs 700ms to parse a triangle graph with $n = 1000$ whereas the PTD parser needs just 0.97ms, and the PSR parser just 0.44ms.

We also conducted experiments with the more complicated language of Nassi-Shneiderman diagrams that represent structured programs with conditional statements and while loops. Fig. 8 shows such diagrams. Each diagram can be modelled by a graph where statement, condition, and while blocks are represented by edges of type *stmt*, *cond*, and *while*, respectively. Diagram $D_1$ in Fig. 8, for instance, is represented by a graph $\mathit{cond}^{abcd}\,\mathit{stmt}^{cefg}\,\mathit{stmt}^{edgh}$. The language of all *Nassi-Shneiderman graphs* is defined by an HR grammar with the following rules:

$$\begin{aligned} S^{\varepsilon} &\to \mathit{NSD}^{xyuv} \\ \mathit{NSD}^{xyuv} &\to \mathit{NSD}^{xyrs}\,\mathit{Stmt}^{rsuv} \;\mid\; \mathit{Stmt}^{xyuv} \\ \mathit{Stmt}^{xyuv} &\to \mathit{stmt}^{xyuv} \;\mid\; \mathit{cond}^{xyrs}\,\mathit{NSD}^{rmun}\,\mathit{NSD}^{msnv} \;\mid\; \mathit{while}^{xyrsut}\,\mathit{NSD}^{rstv} \end{aligned}$$

We use the shorthand notation $L \to R_1 \mid R_2$ to represent rules $L \to R_1$ and $L \to R_2$ with the same left-hand side.

Runtime of the different parsers has been measured for Nassi-Shneiderman graphs $D_n$ with varying values $n$. Fig. 8 recursively defines these graphs $D_i$ for $i = 1, 2, 3, \ldots$ and also shows $D_3$ as an example. Each diagram $D_i$ consists of $2 + 6i$ nodes and $3i$ edges.

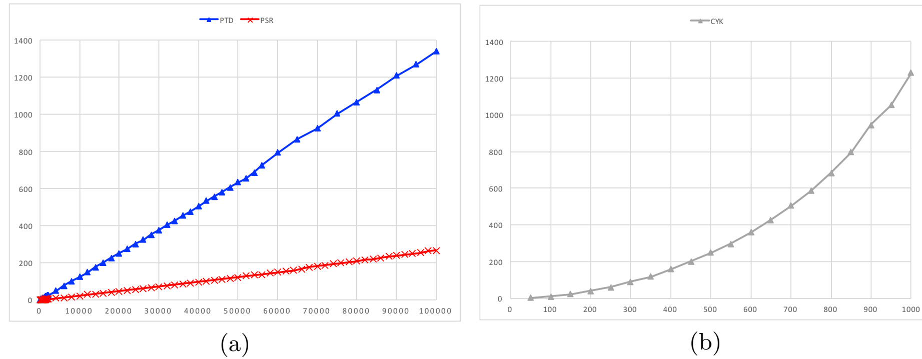



**Fig. 9.** Runtime (in milliseconds) of the PSR as well as the PTD parser (a) and the CYK parser (b) for Nassi-Shneiderman graphs built as shown in Fig. 8. Note that the scales in (a) and (b) differ.

$$
\begin{aligned}
S^{\varepsilon} &\to \mathit{Tree}^{xy} \\
\mathit{Tree}^{xy} &\to \mathit{pair}^{xy} \;\mid \\
&\phantom{\to} \mathit{Child}^{xyu}\ \mathit{Tree}^{xy} \\
\mathit{Child}^{xyu} &\to \mathit{edge}^{xyuv}\ \mathit{Tree}^{uv}\ \mathit{Next}^{xyu} \\
\mathit{Next}^{xyu} &\to \mathit{Child}^{xyu} \;\mid \\
&\phantom{\to} \varepsilon
\end{aligned}
$$

**Fig. 10.** Blowball graph grammar.

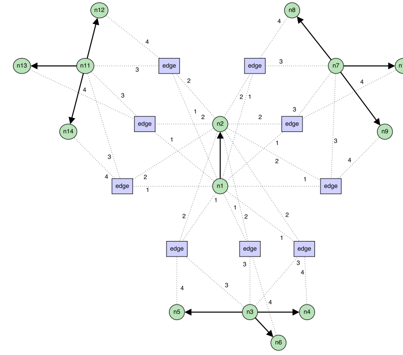
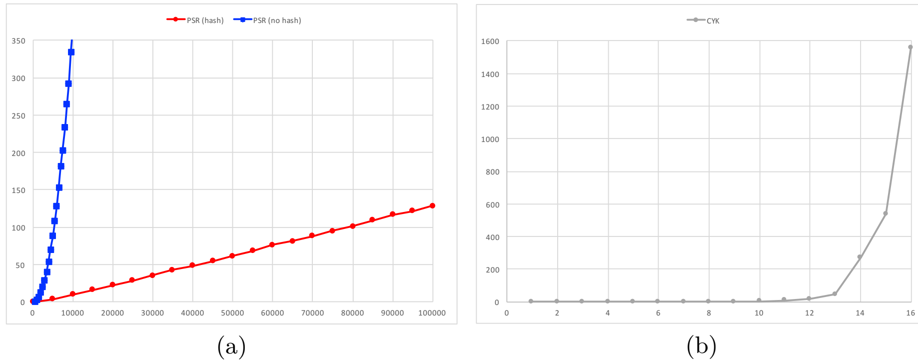


**Fig. 11.** Blowball graph $B_{10}$.

Fig. 9a shows the runtime of the PSR and the PTD parser for graphs $D_n$ with $n$ being shown on the $x$-axis and the runtime in milliseconds on the $y$-axis. Fig. 9b shows the corresponding diagram for the CYK parser. The PSR parser and the CYK parser have been generated from the HR grammar presented above. For generating the PTD parser, a slightly modified grammar with *merging rules* [4] had to be used because the presented grammar is not PTD.

Note that the runtime of the PSR parser and the slower PTD parser is linear in the size of the input graph whereas the runtime of the CYK parser is not linear. Note again that the scales in the diagrams shown in Fig. 9a and b differ and that PTD parsing and, in particular, PSR parsing is, by several orders of magnitude, faster than CYK parsing. For instance, the CYK parser needs 1.2s to parse $D_{1000}$ whereas the PTD parser needs just 12ms, and the PSR parser just 1.0ms.

The PSR parsers for triangle and Nassi-Shneiderman graphs make use of determining nodes and, therefore, do not require hash tables to obtain linear




(a) (b)

**Fig. 12.** Runtime (in milliseconds) of the PSR parser (a) with hash tables (faster) and without hash tables (slower) and the CYK parser (b) for blowball graphs $B_n$. Note that the scales in (a) and (b) differ.

parsing time. In order to demonstrate the speed-up produced by hash tables, we constructed an HR grammar (see Fig. 10), called *blowball grammar* because of the shapes of its graphs. Its PSR parser must perform some edge look-ups without determining nodes. *Grappa* has been used to generate two versions of a PSR parser: Version *PSR (hash)* uses hash tables to speed up these edge look-ups, whereas version *PSR (no hash)* iterates over lists of candidates instead. Moreover, a PTD and a CYK parser have been generated. For the experiments, we considered blowball graphs $B_n$, $n \geqslant 1$, like $B_{10}$ shown in Fig. 11: $B_n$ consists of $n$ *pair* edges (represented by arrows in Fig. 11), one in the center and the rest forming stars where the number of edges in each star is as close to the number of stars as possible. Runtime of the different parsers has been measured for these graphs $B_n$ with varying values $n$. Fig. 12a shows the results of the two PSR parsers. The *PSR (no hash)* parser has quadratic parsing time and is much slower than the *PSR (hash)* parser with linear parsing time. For instance, *PSR (no hash)* needs 360ms to parse $B_{10000}$, whereas *PSR (hash)* needs just 10ms. Parsing time of the PTD parser is similar to the *PSR (no hash)* parser and is not shown here. Fig. 12b shows the results of the CYK parser, which is again by several orders of magnitude slower than the other parsers. For instance, the CYK parser needs 1.6s to parse $B_{16}$ whereas the PTD parser needs just 9$\mu$s, and the PSR parsers (both versions) just 5$\mu$s.

## 7 Conclusions

We have described the implementation of efficient predictive shift-reduce parsers for HR grammars that can be automatically generated by the *Grappa* parser generator. Measurements for the generated parsers confirm that they run in linear time, as postulated in [6]. In that paper, we have established some relationship between HR grammars generating string graphs: PSR parsing turned out to be a true extension of De Remer's *SLR*(1) parsing and also of PTD parsing [4].

Earlier, now abandoned work on predictive graph parsers [9,11] has been based on fairly restricted subclasses of node replacement grammars [8] and on edge precedence relations.

Like PTD parsing, PSR parsing can be lifted to contextual HR grammars [2,3], a class of graph grammars that is more relevant for the practical definition of graph languages. This remains as part of future work. Moreover, it might be worthwhile to extend PSR to the more powerful Earley-style parsers that use a more general kind of control automaton, and pursue several goals in parallel [7].

## References


1. B. Courcelle and J. Engelfriet. *Graph Structure and Monadic Second-Order Logic – A Language-Theoretic Approach*, volume 138 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, 2012.
2. F. Drewes and B. Hoffmann. Contextual hyperedge replacement. *Acta Informatica*, 52:497–524, 2015.
3. F. Drewes, B. Hoffmann, and M. Minas. Contextual hyperedge replacement. In A. Schürr, D. Varró, and G. Varró, editors, *Applications of Graph Transformation with Industrial Relevance (AGTIVE'11)*, LNCS 7233, pages 182–197. 2012.
4. F. Drewes, B. Hoffmann, and M. Minas. Predictive top-down parsing for hyperedge replacement grammars. In F. Parisi-Presicce and B. Westfechtel, editors, *Graph Transformation - 8th Int. Conf., ICGT 2015. Proceedings*, LNCS 9151, pages 19–34. 2015.
5. F. Drewes, B. Hoffmann, and M. Minas. Approximating Parikh images for generating deterministic graph parsers. In P. Milazzo, D. Varró, and M. Wimmer, editors, *Software Technologies: Applications and Foundations - STAF 2016 Collocated Workshops: DataMod, GCM, HOFM, MELO, SEMS, VeryComp, Vienna, Austria, July 4-8, 2016, Revised Selected Papers*, LNCS 9946, pages 112–128. 2016.
6. F. Drewes, B. Hoffmann, and M. Minas. Predictive shift-reduce parsing for hyperedge replacement grammars. In J. de Lara and D. Plump, editors, *Graph Transformation - 10th Int. Conf., ICGT 2017. Proceedings*, LNCS. 2017. To appear.
7. J. Earley and H. Sturgis. A formalism for translator interactions. *Comm. of the ACM*, 13(10):607–617, 1970.
8. J. Engelfriet and G. Rozenberg. Node replacement graph grammars. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. I: Foundations*, chapter 1, pages 1–94. World Scientific, Singapore, 1997.
9. R. Franck. A class of linearly parsable graph grammars. *Acta Informatica*, 10(2):175–201, 1978.
10. A. Habel. *Hyperedge Replacement: Grammars and Languages*. LNCS 643. 1992.
11. 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*, LNCS 291, pages 326–342, 1986.
12. M. Minas. Diagram editing with hypergraph parser support. In *Proc. 1997 IEEE Symp. on Visual Languages (VL'97), Capri, Italy*, pages 226–233. IEEE Computer Society Press, 1997.
13. M. Minas. Concepts and realization of a diagram editor generator based on hypergraph transformation. *Science of Computer Programming*, 44(2):157–180, 2002.