Visualizing Visual Parser Executions

Gennaro Costagliola, Mattia De Rosa
Dipartimento di Informatica, University of Salerno, Fisciano, SA, Italy
{gencos, matderosa}@unisa.it

Mark Minas
Universität der Bundeswehr München, Neubiberg, Germany
mark.minas@unibw.de

Abstract

Parsing of visual structures like diagrams and graphs is more complicated than parsing strings. This is so because visual structures are inherently more complex than strings, because visual grammars are more difficult to write than string grammars, and because the algorithms for parsing visual structures are usually more complicated than for parsing strings. The developer of a visual parser, therefore, needs more tool support than a developer of a string parser. In fact, developing and debugging a visual parser without proper visualization of the parsing process is very challenging.

This paper describes a visualization approach that arose from this need. Its main focus is on the interaction of the developer with the visualization tool in order to explore the execution process of the parser. It has evolved from experiences with developing and debugging parsers by applying different visual parsing methods. In order to better describe it we introduce a concrete example.

Keywords: *visual parsing, graph parsing, parser visualization.*

1. Introduction

Parsing strings with respect to a grammar is well-known and well-understood since some 50 years [18]. Every compiler of a (textual) programming language uses a string parser to analyze the syntactic structure of its input and to control its translation into other formats, in particular machine language or intermediate representations. Parser generators make building string parsers a simple task [26, 23, 14]. A significant part of the research on visual languages in the last 25 years [5] has focused on the study of their semantics and the possibility to specify them in a formal way, also for use in visual programming languages. Researchers have therefore tried to analyze them using approaches similar to those used for strings, but with less success than in the string domain [22]. One reason is the obvious fact that parsing strings is much easier than visual parsing. All established string parsing techniques take advantage of the linear structure of strings, in particular of substrings of the input. This is apparent for top-down and bottom-up parsers using LL and LR parsing, which process the input string from left to right, i.e., analyze prefixes of the input. Even table-based parsers like Cocke-Younger-Kasami parsers [33] depend on the strings' linear structure although they do not process input strings from left to right. Instead, they construct nonterminals for arbitrary substrings (and not just prefixes) of the input string, starting with substrings of length one and eventually for the entire input, if it is valid. Substrings are easily represented by just two numbers, e.g., start and length. Parsing errors can be easily communicated to the user that way. This task is more complicated when a visual parser fails; it must then visualize those parts of the input that have already been processed when the error occurred. The situation becomes even more complicated when an implementor develops a parser, even when using a parser generator. Understanding the flow of execution of a visual parser and its consequent validation without proper visualization of the parser's progress and its data structures is then tedious, other than very challenging.

The use of visual structures like graphs have recently gained some importance in the field of natural language processing (NLP) where the meaning of sentences is represented by graphs [11]. Their syntactic structure is defined by grammars, and (graph) parsers are used for analyzing them. Several approaches are used in this context, and some tools have emerged [10]. However, similar to the situation in earlier VL research, appropriate parser visualization techniques and tools are yet missing.

This paper extends [6] by describing the analysis-based approach that led to the development of prototypical tools

DOI reference number: 10.18293/DMSVIVA20-013

for properly visualizing the execution of visual parsers. We first describe the need for such tools which became apparent when the authors developed visual parsers. Based on these needs and experiences with first visualization prototypes, we reconsidered the problem and identified the primary use cases. We generalized these results by deriving requirements that parser visualization tools should fulfill in order to effectively and efficiently support the realization of visual parsers. A general parser visualization architecture has been developed from these requirements and realized in two independent prototypical tools.

The rest of the paper is structured as follows: we start by presenting our running example describing the basic concepts of Visual Generalized LR (VGLR) parsing and the semantic representation of natural language sentences in Section 2, then, in Section 3 we outline the primary use cases of a visualization tool for visual parsers and derive the visualization requirements in Section 4. In Section 5 we describe the proposed parser visualization architecture used to implement our prototypes and related work in Section 6. Section 7 concludes the paper.

2. The application example: VGLR parsing and NLP

The proposed visualization and exploration approach can be used with many different types of visual parsers, either top-down or bottom-up. In fact, it has already been used with two different VGLR parser approaches based on *[contextual] hyperedge replacement grammars* ([C]HRGs) [13, 9, 24] and *extended positional grammars* [7, 8].

In this paper, the running example is based on a VGLR parser built for a CHRG from the domain of natural language processing. In the following, the basic concepts of VGLR parsing are given, followed by a brief description of the natural language representations used as input sentences.

The Generalized LR (GLR) parsing algorithm [32, 28] extends the well-known LR parsing algorithm [18] to ambiguous string grammars and Visual GLR (VGLR) parsing algorithm extends GLR parsing to the case of graph languages.

We assume that readers are familiar with the standard LR parsing algorithm, which analyzes an input string from left to right, maintains a stack of states through the shift/reduce actions, and produces a single parse tree if the input string is valid. In order to handle nondeterminism, a generalized parser works on multiple stacks at the same time and produces multiple parse trees, one for each interpretation. A GLR parser is able to do this efficiently by storing the stacks in a so-called *graph-structured stack* (GSS; see Fig. 5 for an example) and packing the resulting parse trees in a *packed parse forest* (see Fig. 6 for an example). A GSS is a particular directed acyclic graph representing each individual stack as a path from some top-most state to the unique initial state. There are three main operations that can be performed on a GSS: splitting, combining and local ambiguity packing. Each time the parser faces two conflicting actions (shift/reduce or reduce/reduce) the current stack top is *split* to accommodate two new branches in the graph. Whenever a new stack top, resulting from a shift action, happens to be equal to an already existing stack top, they are *combined* into one node. A local ambiguity packing is the operation of merging two equal branches. This happens when the same fragment of the input can be reduced to the same nonterminal in different ways. The goal of these operations is to maximize the sharing of the common parts of the multiple stacks. In fact, working on the GSS instead of on a set of complete copies of different stacks does not only save space, but also time: instead of repeating the same operations on separated stacks that have common parts, the parser has to perform them only once.

By following the same idea, a packed parse forest stores the many parse trees produced by analyzing an ambiguous input by sharing all of their common subtrees. The relation between a GSS and its corresponding packed parse forest is given by the fact that each edge in the GSS corresponds to a vertex of the forest and the subtree rooted in it. In this way, a vertex (called packed vertex) in the forest may be root of distinct subtrees corresponding to the same shared GSS branch (due to local ambiguity packing).

When dealing with non-linear sequences such as graphs or other types of diagrams, a *visual GLR* (VGLR) parser must also deal with the fact that there is no a priori reading sequence of visual tokens. This may force a VGLR parser to pursue different reading sequences in parallel while it performs the search process. This has consequences as follows:

- Each parse stack corresponds to a specific subset of visual tokens that have been read already. Hence, the parser must store, for each stack separately, which visual tokens have been read.

  Sets of stacks are stored as a GSS like in GLR parsers. Each GSS node corresponds to a state. Additionally, each GSS node keeps track of the set of visual tokens that have been read so far. Note that GSS nodes may be shared only if both their states and their sets of visual tokens coincide.

- VGLR parsers cannot process their sets of stacks in rounds. When a stack is obtained by executing a *shift* action, the parser must not wait until the same visual token has been read in all the other stacks (as done for GLR parsing); they may read other tokens first. As a consequence, a VGLR parser needs different strategies from that adopted by GLR parsing to control the order in which stacks are processed. Strategies are beyond

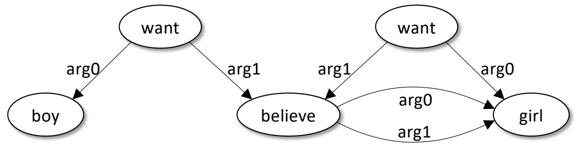


Figure 1. AMR graph for "*The boy wants the girl to believe in herself and this is what the girl wants, too.*"

the scope of this paper, but visualizations of parser executions must make them explicit to the user.

In order to describe our visualization approach we apply the VGLR parser to the graph language of *abstract meaning representations* (AMR). AMRs have been proposed as a semantic representation of English sentences. Each AMR is a directed graph with labelled nodes and edges, which represent concepts and their relations, respectively. For instance, the semantics of the sentence "*The boy wants the girl to believe in herself and this is what the girl wants, too*", taken from [11], can be represented by the AMR shown in Fig. 1. A description of AMRs is beyond the scope of this paper; details can be found in [1, 20], and a corpus of AMRs is described in [3].

3. Use Cases

The authors have realized several visual parsers and visual parser generator [7, 8, 9, 13, 24]. Developing VGLR parsers is challenging because one has to coordinate several non-trivial data structures like the input, the GSS, and the parse forest. It turned out that it is almost impossible to develop such parsers without proper visualization of all these data structures. In the following, we briefly describe the primary uses cases of visualization tools when developing VGLR parsers. We then elaborate on the requirements derived from these use cases in Section 4.

The main use case is inspecting the different data structures after a parsing error occurred. The error can either be due to an invalid input (syntax error) or to an error in the parser code or both. Only by inspecting the input, the GSS, and the parser forest at the time of the error and running the last steps that led to the error can help the parser implementor to detect the possible causes. Since the number of nodes in the GSS and the parse forest can be very high (sometimes more than 100) and many concurrent stack tops and tree roots may be present, it would be almost impossible to accomplish the task without proper visualization. Even though the number of nodes may make the data structure visualizations difficult to read, running back and forth the last actions preceding the error can help the implementor to individuate and zoom on specific nodes of the structures being modified.

Visualizing the input and those input tokens that have already been inspected by the parser is another use case. In contrast to textual input, visual input has in general no self-evident ordering of input tokens. The parser has, rather, to identify a correct parsing sequence, and it turned out that visualizing this information makes debugging VGLR parsers a lot easier.

Yet another use case is for the parser user to see all the possible syntactic interpretations of its input by looking at the parse forest visualization. Each of the contained parse trees corresponds to individual stacks within the GSS and possibly different parsing sequences through the input. It turned out to be manageable for the implementor to comprehend the correspondence of all these data structures with the help of proper visualizations.

4. Parser Visualization Requirements

The use cases outlined in the previous section allowed to derive requirements on a parser visualizer (in the following called just *visualizer*). It must essentially provide visual representations of the parser's data structures that change over time during its execution. In order to allow the implementor to analyze and validate these data structures, she must be able to stop the execution, to continue it, to watch it in single-step mode, to go backwards in time, i.e., to retrace the parser's steps, etc. In other words, the visualizer must provide control over the parser execution quite similar to a program debugger.

4.1. Granularity levels of execution control

A well known feature of program debuggers is that they allow to run a program step by step on quite different levels of granularity. At the lowest level, they allow one to stop after each statement or instruction. On a higher level, they can "step over" a procedure call, i.e., they stop automatically when the procedure call terminates. The user can choose the appropriate level of granularity freely. Similarly, a parser execution visualizer should be able to show parser executions at different levels of granularity and to allow one to easily pass from one level to the other. Furthermore, in order to better fit the mental model that a parser/language implementor has of the parser execution, each level should be programmable, in the sense that she should be allowed to define the operations included in a level. In the following, we use the action and step granularity levels.

In general, the action level visualizes the results of the execution of each parser action. As mentioned above, to better represent the user needs, a parser action may be split in more refined actions: as an example, the reduce action of a bottom-up parser may be split in “deletion” of the reduced path and “addition” of the new goto state. This gives a better understanding on how the reduction process is performed and on which states. As a further example, not represented here, in the case of visual parsers that use relations to navigate the input, the shift action may further be split in “move the cursor” to point to the next input symbol and “shift the pointed token”.

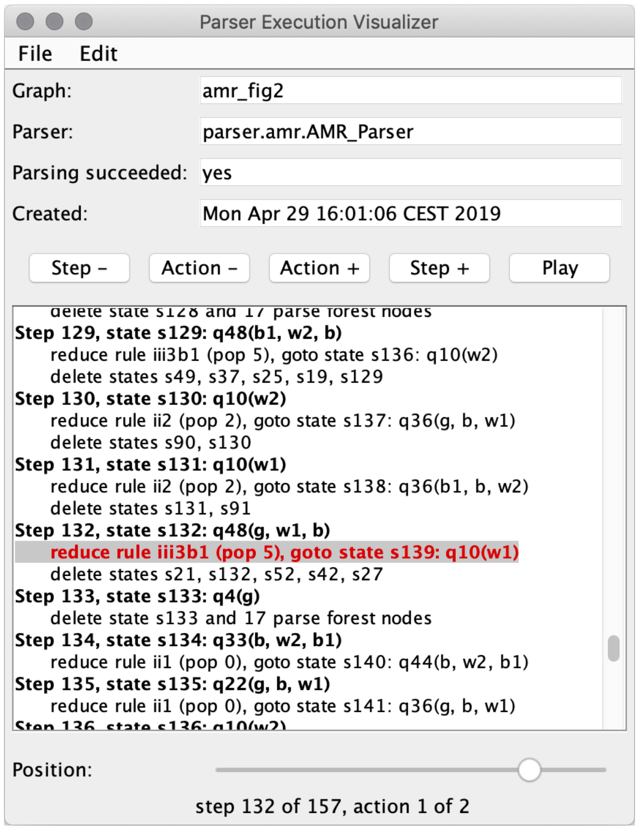
A step is usually the highest granularity level and it can vary depending on the particular strategy of execution adopted by the parser being implemented. It is usually used to synchronize the actions of the parser. As an example, in Tomita’s parser the execution is synchronized by the input tokens. As a consequence, each step includes either all the shift actions to visit a new input token or all the possible reductions that can be applied on a token. In our case, a step includes the actions to be executed on a particular top state.

4.2. Execution control

A program debugger lets the user control the execution by showing her the program source code in which she can set breakpoints and in which the line of code is highlighted where the program has been stopped. The visualizer should offer a similar view that shows the sequence of parser operations on the selected level of granularity. Fig. 2 shows an example used in our prototype visualizing the execution of the AMR graph parser analyzing the graph shown in Fig. 1. The lines in bold-face are steps where the parser operates on a specific node of the GSS, which corresponds to a state, before it continues working on the next node of the GSS. The sequence of actions composing a step are shown in normal font below a bold-face step. The action where the parser execution has been stopped is highlighted in red. Here it is a reduce action that pops five states off the stack, starting at the current state $s_{132} : q_{48}(q, w_1, b)$ and which will then perform a goto to state $s_{139} : q_{10}(w_1)$. Again, details of the parser and its states are beyond the scope of the paper. Note that the action that follows below the highlighted line will then remove the states that have been popped off.

Note also that such a view is in fact different from the program source code of a program debugger in the sense that it does not show the parser program, but rather a trace of the parser execution on a specific level of granularity. In fact, it must be an a posteriori trace taken from a previous parser execution. Otherwise, the view could not show the steps and actions following the one where the execution has been stopped. We require (see Sect. 5) that the parser stores the trace in a log file, which is read later by the visualizer, a technique which is also well-known from the analysis of parallel programs [21].

The user of the visualizer must be able to run the parser, or rather retrace its steps from the log. She must be able to execute the parser actions step by step on different granularity levels, and must be able to go back and forth in time, realized by buttons in Fig. 2. Pushing it shall start a continuous animation visualizing the progress of the parser in the data views discussed in the next section. Furthermore, clicking on an action in the trace should fast-forward (or fast-backward, resp.) the visualization to this action.[1]



Graph: amr_fig2
Parser: parser.amr.AMR_Parser
Parsing succeeded: yes
Created: Mon Apr 29 16:01:06 CEST 2019

```
    delete state s128 and 17 parse forest nodes
Step 129, state s129: q48(b1, w2, b)
    reduce rule iii3b1 (pop 5), goto state s136: q10(w2)
    delete states s49, s37, s25, s19, s129
Step 130, state s130: q10(w2)
    reduce rule ii2 (pop 2), goto state s137: q36(g, b, w1)
    delete states s90, s130
Step 131, state s131: q10(w1)
    reduce rule ii2 (pop 2), goto state s138: q36(b1, b, w2)
    delete states s131, s91
Step 132, state s132: q48(g, w1, b)
    reduce rule iii3b1 (pop 5), goto state s139: q10(w1)
    delete states s21, s132, s52, s42, s27
Step 133, state s133: q4(g)
    delete state s133 and 17 parse forest nodes
Step 134, state s134: q33(b, w2, b1)
    reduce rule ii1 (pop 0), goto state s140: q44(b, w2, b1)
Step 135, state s135: q22(g, b, w1)
    reduce rule ii1 (pop 0), goto state s141: q36(g, b, w1)
Step 136, state s136: q10(w2)
```

step 132 of 157, action 1 of 2

Figure 2. Execution control of the parser execution visualizer prototype.

4.3. Data views

In order to allow the user to understand what is going on in a parser execution, different aspects of the parser must be visualized. Apparently, its main data structures must be shown in a way that match the user’s mental model. For our running example using a VGLR parser, the visualizer must at least show the current GSS, the parse forest, and the parser input, as discussed in the following. These data structures are connected. For instance, leaves of the parse forest correspond to visual tokens of the parser input, and each edge in the GSS refers to a parse forest node. Visu-

[1]Two screencasts that demonstrate the user interactions with the prototype can be seen at [37].

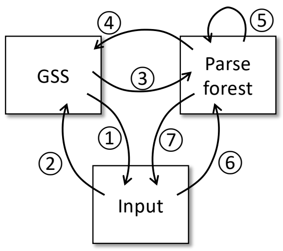


Figure 3. Different data views. Arrows indicate visual feedback triggered by user interaction described in the text.

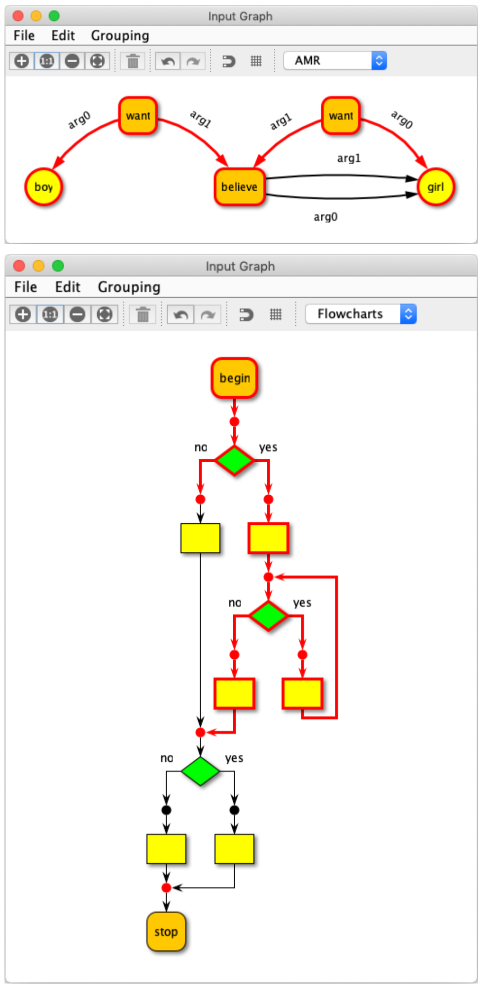


Figure 4. Input view of an AMR graph and a flowchart.

alizing all these aspects and their interconnections in a single view would produce just clutter. Instead, we suggest to provide separate views and to visualize interconnections between them by means of user interaction and visual feedback [30]. Fig. 3 symbolizes the three views that the visualizer at least should provide; arrows represent user interactions in one view and the corresponding visual feedback in a different or the same view. Arrow 7, for instance, includes highlighting a visual token in the input view when the corresponding parse forest leaf is clicked.

In the following, we first describe the requirements on the three different data views and afterwards their interconnection by means of visual feedback triggered by user interaction.

4.3.1. Input view

The parser must appropriately choose in which sequence it reads the visual tokens of the input, and it may be forced to choose different reading sequences in the same execution if the input is ambiguous, e.g., the AMR graph shown in Fig. 1. The visualizer, therefore, must provide an *input view* that shows the parser input and indicates which of its visual tokens have already been read in the current state of the execution, and which have not been read yet.

Visualizing the parser input is more complicated than showing the input of a string parser: Whereas a string can be simply shown as text, there is no uniform representation for all visual languages that can be analyzed by a visual parser. The input view, therefore, must be highly customizable, its visual representation should match the representation of the visual language. Fig. 4 shows two screenshots of our prototype. The upper screenshot depicts the AMR graph of Fig. 1, the lower one a flowchart in the process of being parsed. Visual tokens that have already been read by the parser are highlighted in red. More details are described below.

Of course, different parsers are used to analyze AMR graphs and flowcharts, but they use a common input format with less information than the concrete diagram. In general, the input format may in fact be just a kind of graph (as in our case), which does not contain any information on the layout of its visual tokens. The input view, therefore, must be customizable in the way how elements of the common input format shall be represented. Furthermore, it shall offer some automatic and manual layout functionality like in standard visual editors, but without the ability to modify the parser input semantically. Our prototype allows to switch between different customizations using the combo box in the window toolbar.

4.3.2. GSS view

The visualizer must show the main data structure of the parser. For Earley-parser or a Cocke-Younger-Kasami-based parser, it would be a table used for dynamic programming. In our VGLR example, however, this is the GSS, which shall be shown as a plain DAG as in Fig. 5.

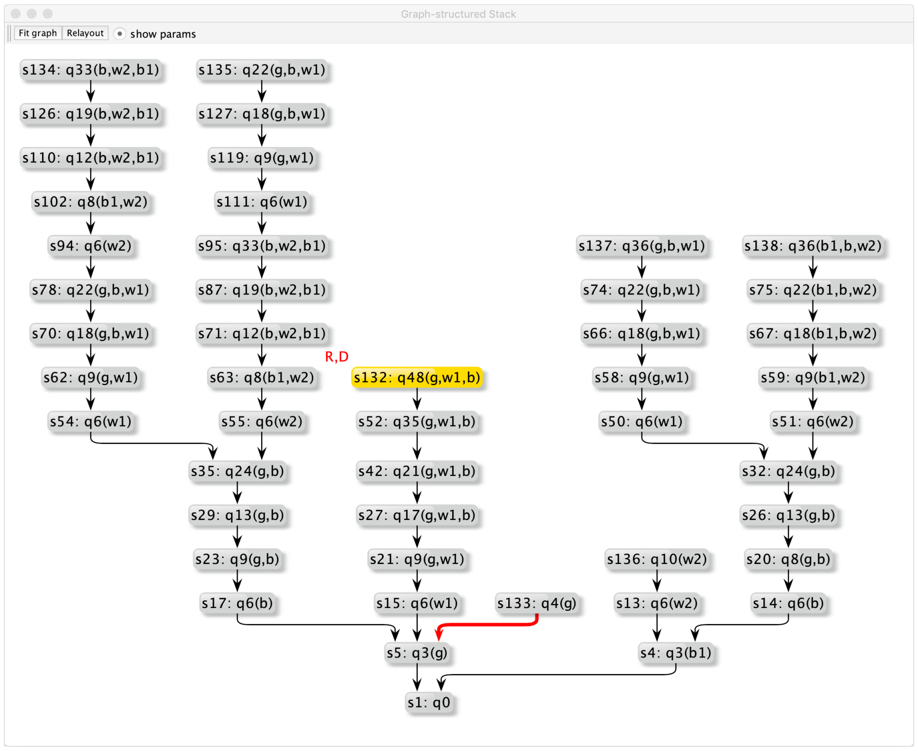


Figure 5. GSS view corresponding to Fig. 2.

The GSS changes with the parser execution, i.e., GSS nodes and edges are added and deleted. Moreover, GSSs can grow rather big as Fig. 5 shows, which can make it difficult for the user to follow these modifications. The GSS view shall provide some general features that allow the user to easily recognize any modification. First, the layout of the modified GSS should be computed from the old one incrementally in order to preserve the user's mental map [27]. Second, changes to the layout should not happen abruptly. Instead, they should be visualized in an animated way so that the user has time to see the changes happen. And finally, the view shall indicate the GSS node ("working node") that is currently processed by the parser, i.e., where changes happen, shown in orange in Fig. 5. Note that this GSS node is the same as the one indicated in the trace view of execution control (Fig. 2).

Moreover, the GSS view should also add further information about the current working node, which informs the user about the changes that will happen to this node. Possible changes are triggered by the actions within the step processing this node, i.e., shift, reduce, accept, and delete as described before. The initial letters of these actions are used here to mark the working node, here R and D, which corresponds to the actions of step 132 as shown in the log view (Fig. 2).

The highlighted edge in Fig. 5 is a visual feedback after selecting a parse forest node described in Sec. 4.3.4.

4.3.3. Parse forest view

The parse forest represents the syntactic structure of the parse input processed so far during the parser execution. It is the final parse forest, i.e., a packed form of the set of all parse trees of the input, if the parser terminates successfully, and it is empty if the parser fails. The parse forest is usually a DAG if sub-trees are shared in order to save space. And in general, it consists of several unconnected components as long as the input has not been analyzed completely yet. In fact, each edge of the GSS refers to a unique parse forest node. This interconnection of GSS and parse forest shall be communicated to the user by means of user interaction and visual feedback described in the next section.

Fig. 6 shows a part (note the scrollbar at the bottom) of the parse forest in the execution state shown in Fig. 2. Terminal parse forest nodes are drawn in yellow, nonterminal ones in green. Note also the nodes *b*, *g*, and *b*1 drawn in faded red; they are so-called contextual nodes which are a specific feature of contextual hyperedge replacement grammars used in our example (see Sec. 2 and [11]). Their incoming and outgoing arrows represent where these nodes have been created and where they are used as contextual nodes in the parse forest. Again, details are beyond the scope of this paper.

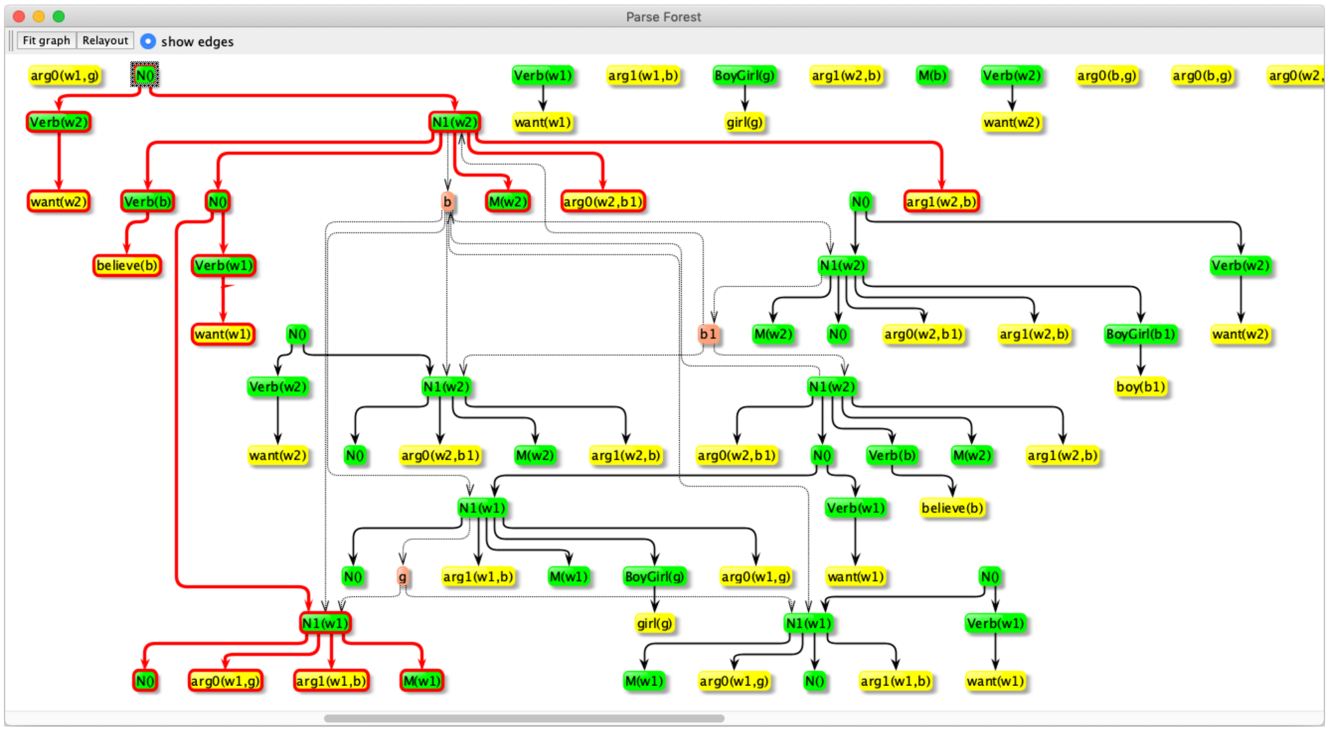


Figure 6. Parse forest created by the VGLR parser corresponding to Fig. 2.

Like the GSS, the parse forest changes with the parser execution. New parse forest nodes are added when edges are added to the GSS, possibly adding edges to child nodes in the parse forest (see the screencasts at [37]), and entire subgraphs of the parse forest may be deleted when a parse fails. The latter does not necessarily mean that the whole parser execution fails; it may be just one dead end in the search carried out by the parser. Like the GSS view and in order to prevent user confusion, the parse forest view must provide an automatic layout that allows to preserve the user's mental map.

Fig. 7 shows the complete parse forest after the parser eventually accepted the AMR graph of Fig. 1. It represents in fact two parse trees and uses local ambiguity packing (see Sec. 2) in order to save space: two different nonterminals $N_1(b)$ with different sub-DAGs are contained within a gray node, whose parent nonterminal $N(b)$ can select either of the two nodes $N_1(b)$ as a child, resulting in two different parse trees. One parse tree corresponds to the sentence "*The boy wants the girl to believe in herself and this is what the girl wants, too*", the other to the semantically equivalent "*The girl wants to believe in herself, and the boy wants the girl to believe in herself, too.*"

The nodes and edges of the parse forests of Figures 6 and 7 highlighted in red are visual feedbacks after selecting the top-most highlighted node $N()$ and $N_1(b)$, respectively, and is described in the next section.

4.3.4. User interaction and visual feedback

We are now going to describe the visual feedback triggered by selecting components in one of the views. The numbers of the following items correspond to the numbers used in Fig. 3.

① Whenever a GSS node becomes the current working node or if the user selects a GSS node in the GSS view, all visual tokens that have already been read in this parser state shall be highlighted. The nodes and edges of the AMR graph in Fig. 4 drawn in red have been read in the current state $s_{132} : q_{48}(q, w_1, b)$ in Fig. 5.

② When the user selects a visual token in the input view, all GSS nodes that have already read this token shall be highlighted in the GSS view.

③ Each edge of the GSS refers to a top-most node of the parse forest. If the user selects an edge in the GSS view, this node and its complete sub-DAG shall be highlighted in the parse forest view.

④ is in fact the opposite of ③: When a user selects a top-most node in the parse forest node, the edge of the GSS that refers to this parse forest node is selected in the GSS view. The edge in Fig. 5 highlighted in red is in fact the visual feedback for the selection of the top-most node $N()$ in the parse forest view (Fig. 6).

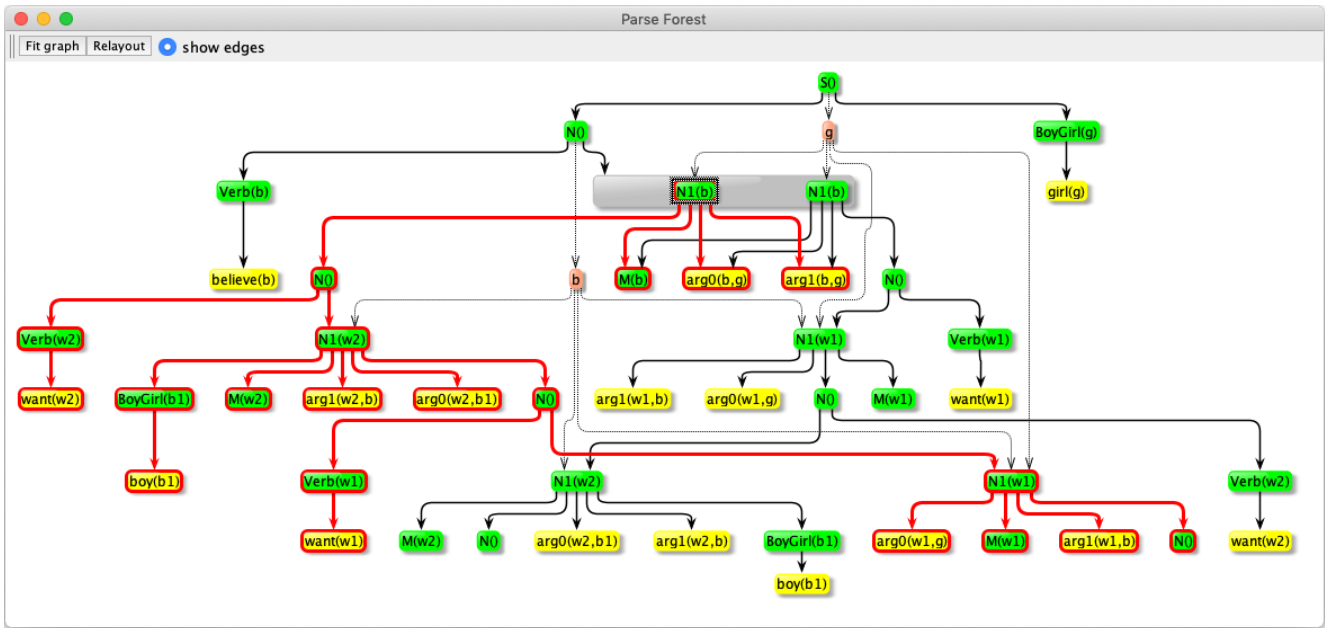


Figure 7. Final parse forest of the AMR graph shown in Fig. 1.

⑤ And whenever a node is selected in the parse forest node, all nodes and edges of its entire sub-DAG are highlighted, too. That is the reason for the other highlighted nodes and edges in Figures 6 and 7 selecting $N()$ or $N_1(b)$, respectively. This does in fact not visualize a connection between different data-views, but allows the user to recognize more easily which nodes of a parse forest belong to a sub-DAG. Otherwise, it would be rather tedious to see which parse forest nodes belong to either of the two nodes $N_1(b)$ within the gray packed node in Fig. 7.

⑥ When the user selects a visual token in the input view, the corresponding terminal parse forest nodes shall be highlighted in the parse forest view. Note that, according to ②, selecting a visual token also highlights all GSS nodes that have read the visual token.

⑦ When a parse forest node is selected, which highlights its entire sub-DAG according to ⑤, all visual tokens that correspond to terminal parse forest nodes in this sub-DAG shall be highlighted in the input view.

5. Parser Visualization Architecture

Fig. 8 shows the proposed architecture of the parser execution visualizer as it has also been realized in our prototype. Orange rectangles, yellow rounded rectangles, and green parallelograms represent data, processes, and UI components, respectively, arrows represent data flow.

We suggest that the visualizer does not visualize the parser during its execution. We rather suggest that the parser is instrumented so that it writes a trace of its actions into a log file, which is read later by the visualizer. This approach allows the visualizer to easily go back and forth in time. Moreover, the same parser execution can be visualized arbitrarily often, even if the parser runs nondeterministically [21]. We also assume that the log file contains the encoded parser input so that it can be visualized in the *input view*.

The *Log Reader* reads the file and internally stores the *parse trace* which consists of the *parser input* and the sequence of *parser events*, e.g., actions like shift and reduce. They are shown to the user in the *input view* (see Fig. 4) and the *log view* (see Fig. 2), respectively.

The *execution control process* reads the parse events forward and backward, controlled by the *execution control user interface* (see Fig. 2). This process keeps track of the current event, which is also highlighted in the *log view*. And if the user selects an action in the *log view*, execution control fast-forwards to the corresponding parser state. It maintains the parser state by means of the *parser data structures* based on the parsed events that have happened since the beginning of the parse trace. There are no uniform parser data structures, they rather depend on the specific parser type. In our example, they consist of the GSS and the parse forest. These data structures are visualized in the corresponding *data structure views* (here GSS view, Fig. 5, and parse forest view, Figs. 6 and 7) using some layouting facility. The *user interaction & feedback handler* reacts to pointing and

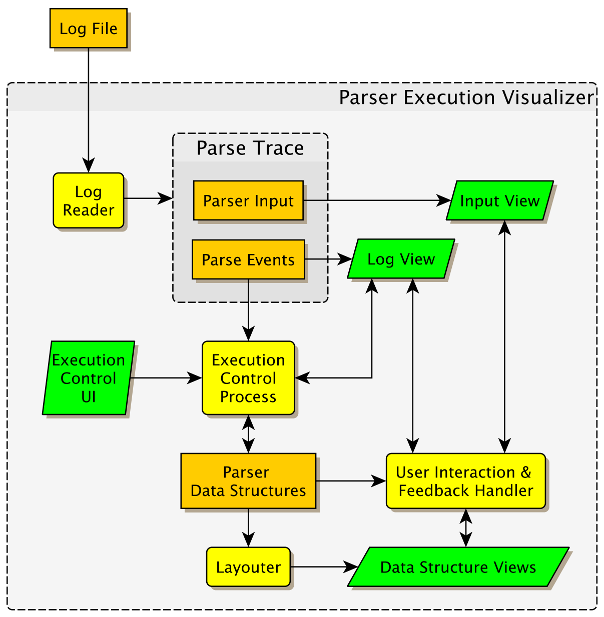


Figure 8. Parser execution visualizer architecture.

clicking in the different views and triggers visual feedback as described in Sec. 4.3.4.

6. Related Work

Data structure and algorithm visualizations have been studied for more than 35 years [31, 4, 17], and now many web resources exist implementing visualizations and animations of almost all the most common data structures and algorithms based on them, respectively [34, 35, 36]. However, lately, the research in this field has shrunk considerably. Most of the important papers are in the range from 1980 to 2000 and the applications have been basically two: visual debugging [25] and teaching and learning [29]. Among the still currently developed tools is JSAV [17], a JavaScript algorithm visualization library that is meant to support the development of general algorithm visualizations for online learning material.

Parser visualization tools either visualize the process of generating a parser from a grammar like LLparse and LRparse [2]. Or they visualize parser execution like PAT [12, 15] which has been used for the visualization and statistical comparison of various GLR parsers. Among other tools we can cite [16] for visualizing lexical generation processes and [19] that is an educational tool for visualizing compiling techniques based on deterministic parsers.

Our prototypes also visualize parser execution and, hence, are most closely related to the latter category. However, we are not aware of any tool that also allows to visualize the execution of visual parsers.

7. Conclusions

In this paper we have illustrated the requirements and the architecture of a parser visualizer system while using an example from the field of Natural Language Processing. In particular, we have discussed several visualization techniques that have proven useful in practice and then generalized the results by elicitation of visualization requirements that parser visualization tools should fulfill in order to effectively and efficiently support the realization of visual parsers.

A system based on the proposed specification is able to support a VGLR parser/language implementor at various levels of granularity and, as a side effect, it may also be used to help teachers to visualize the bottom-up parsing of a specific input when applied to simple grammars.

Two instances of the prototypical parser visualizer based on different VGLR parsing approaches exist following the guidelines and architecture presented in this paper. Even though the two instances have been specialized to the specific approach the needs to gain the maximum insight in the parser execution behavior resulted to be the same.[2] Because of the problem complexity, the use of a parser visualizer gave a huge contribution to the development of each phase of the two VGLRs: automatic generation of the VGLR parser from a grammar specification, execution of the generated parser and parsing of several languages including the example shown here. Another outcome of our parser visualizer architecture is that it allows for the visual comparison of the execution of different parsers obtained either by modified versions of the same approach or by different approaches, for the analysis of their differences and/or similarities at different level of granularity and for the discovery of specific parser behaviors.

References

[1] L. Banarescu, C. Bonial, S. Cai, M. Georgescu, K. Griffitt, U. Hermjakob, K. Knight, P. Koehn, M. Palmer, and N. Schneider. Abstract meaning representation for sembanking. In *Proc. 7th Linguistic Annotation Workshop and Interoperability with Discourse*, pages 178–186, Sofia, Bulgaria, Aug. 2013. Assoc. for Computational Linguistics.

[2] S. A. Blythe, M. C. James, and S. H. Rodger. Llparse and lrparse: Visual and interactive tools for parsing. *SIGCSE Bull.*, 26(1):208–212, Mar. 1994.

[2]For completeness, some screenshots of the other instance not described here can be found at [37].

[3] F. Braune, D. Bauer, and K. Knight. Mapping between English strings and reentrant semantic graphs. In *Proc. of the Ninth Int. Conf. on Language Resources and Evaluation (LREC'14)*, pages 4493–4498, Reykjavik, Iceland, May 2014. European Language Resources Association (ELRA).

[4] M. H. Brown and R. Sedgewick. Techniques for algorithm animation. *IEEE Software*, 2(01):28–39, Jan. 1985.

[5] G. Costagliola, M. De Rosa, V. Fuccella, and S. Perna. Visual languages: A graphical review. *Information Visualization*, 17(4):335–350, 2018.

[6] G. Costagliola, M. De Rosa, and M. Minas. Visual parsing and parser visualization. In *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 243 –247. IEEE Computer Society, Oct 2019.

[7] G. Costagliola, V. Deufemia, and G. Polese. A framework for modeling and implementing visual notations with applications to software engineering. *ACM Trans. Softw. Eng. Methodol.*, 13(4):431–487, Oct. 2004.

[8] G. Costagliola, V. Deufemia, G. Polese, and M. Risi. Building syntax-aware editors for visual languages. *J. Visual Lang. Comput.*, 16(6):508 – 540, 2005. Selected papers from Visual Languages and Formal Methods 2004 (VLFM '04).

[9] F. Drewes, B. Hoffmann, and M. Minas. Extending predictive shift-reduce parsing to contextual hyperedge replacement grammars. In E. Guerra and F. Orejas, editors, *Graph Transformation: 12th Int. Conf., ICGT 2019, Held as Part of STAF 2019, Proc.*, volume 11629 of *LNCS*, 2019.

[10] F. Drewes, B. Hoffmann, and M. Minas. Formalization and correctness of predictive shift-reduce parsers for graph grammars based on hyperedge replacement. *J. Log. Algebr. Methods*, 104:303–341, April 2019. Preprint available at arXiv:1812.11927 [cs.FL].

[11] F. Drewes and A. Jonsson. Contextual hyperedge replacement grammars for abstract meaning representations. In *13th Intl. Workshop on Tree-Adjoining Grammar and Related Formalisms (TAG+13)*, pages 102–111, 2017.

[12] G. R. Economopoulos. *Generalized LR parsing algorithms*. PhD thesis, Royal Holloway, Univ. of London, UK, 2006.

[13] B. Hoffmann and M. Minas. Generalized predictive shift-reduce parsing for hyperedge replacement graph grammars. In C. Martín-Vide, A. Okhotin, and D. Shapira, editors, *Language and Automata Theory and Applications, 13th Int. Conf., LATA 2019, Proc.*, volume 11417 of *LNCS*, pages 233–245, 2019.

[14] S. C. Johnson et al. *Yacc: Yet another compiler-compiler*, volume 32. Bell Laboratories Murray Hill, NJ, 1975.

[15] A. Johnstone, E. Scott, and G. Economopoulos. The grammar tool box: A case study comparing GLR parsing algorithms. *Electronic Notes in Theoretical Computer Science*, 110:97–113, 12 2004.

[16] A. Jorgensen, G. R. Economopoulos, and B. Fischer. VLex: visualizing a lexical analyzer generator - tool demonstration. In *Language Descriptions, Tools and Applications, LDTA 2011. Proc.*, page 12, 2011.

[17] V. Karavirta and C. A. Shaffer. JSAV: The JavaScript algorithm visualization library. In *Proc. of the 18th ACM Conf. on Innovation and Technology in Computer Science Education*, ITiCSE '13, page 159–164, 2013.

[18] D. E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607 – 639, 1965.

[19] N. Krebs and L. Schmitz. Jaccie: a Java-based compiler-compiler for generating, visualizing and debugging compiler components. *Sci. Comput. Program.*, 79:101–115, 2014.

[20] I. Langkilde and K. Knight. Generation that exploits corpus-based statistical knowledge. In *Proc. 36th Annual Meeting of the Association for Computational Linguistics and 17th Int. Conf. on Computational Linguistics, Volume 1*, pages 704–710. Assoc. for Computational Linguistics, Aug. 1998.

[21] T. J. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. Comput.*, 36(4):471–482, Apr. 1987.

[22] K. Marriott and B. Meyer, editors. *Visual Language Theory*. Springer-Verlag New York, Inc., New York, NY, USA, 1998.

[23] S. McPeak and G. C. Necula. Elkhound: A fast, practical GLR parser generator. In E. Duesterwald, editor, *Compiler Construction*, pages 73–88. Springer Berlin Heidelberg, 2004.

[24] M. Minas. Speeding up Generalized PSR parsers by memoization techniques. In R. Echahed and D. Plump, editors, *Proc. 10th Int. Workshop on Graph Computation Models (GCM 2019)*, volume 309 of *Electronic Proceedings in Theoretical Computer Science*, pages 71–86, 2019.

[25] S. Mukherjea and J. T. Stasko. Toward visual debugging: Integrating algorithm animation capabilities within a source level debugger. *ACM Trans. Comput.-Hum. Interact.*, 1:215–244, 1994.

[26] T. J. Parr and R. W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.

[27] H. C. Purchase and A. Samra. Extremes are better: Investigating mental map preservation in dynamic graphs. In G. Stapleton, J. Howse, and J. Lee, editors, *Diagrammatic Representation and Inference*, volume 5223 of *LNCS*, pages 60–73, Berlin, Heidelberg, 2008.

[28] E. Scott and A. Johnstone. Right nulled GLR parsers. *ACM Trans. Program. Lang. Syst.*, 28:577–618, 07 2006.

[29] C. A. Shaffer, M. L. Cooper, A. J. D. Alon, M. Akbar, M. Stewart, S. Ponce, and S. H. Edwards. Algorithm visualization: The state of the field. *Trans. Comput. Educ.*, 10(3):9:1–9:22, Aug. 2010.

[30] B. Shneiderman, C. Plaisant, M. Cohen, S. Jacobs, N. Elmqvist, and N. Diakopoulos. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Pearson, 6th edition, 2016.

[31] J. T. Stasko. Simplifying algorithm animation with Tango. In *Proceedings of the 1990 IEEE Workshop on Visual Languages*, pages 1–6, Oct 1990.

[32] M. Tomita, editor. *Generalized LR Parsing*, volume 1. Springer US, 1991.

[33] D. H. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189 – 208, 1967.

[34] `http://www.algomation.com`.

[35] `https://visualgo.net`.

[36] `https://www.cs.usfca.edu/~galles/visualization/`.

[37] `http://cluelab.di.unisa.it/parser_execution_visualizer/`.